The development of an Agent Based Critiquing System Architecture for a project management tool: *Prompter*

By Eamon Gaffney B.Sc.

School of Computer Applications

Dublin City University

Glasnevin

Dublin 9

Supervisor: Professor J. A. Moynihan

A thesis submitted for the degree of

Masters of Science

March 1999

Declaration

I hereby certify that this material, which I now submit for the assessment on the programme of study leading to the award of Master of Science in Computer Applications, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed : Eamon Gaffney Date : 19/2/99

Eamon Gaffney

Acknowledgements

I would like to thank my supervisor Prof. Tony Moynihan for all the help and guidance he has given me during my time in DCU. I would like to thank Mark Johnston for his technical guidance and Rory O' Connor and the other members of the ***Prompter*** team whom I worked with, during the ***Prompter*** project. They made my involvement in the project very enjoyable (especially the project trips).

I would also like to thank Kieran O'Sullivan and Martin Gaffney for having the patience to proof read this work. And finally to the other postgraduates and members of staff in CA for the constant distractions they created, nights out, football etc. They helped make my time there both memorable and enjoyable. And last but not least to my family and friends for helping me get this far and encouraging me along the way. Nice one.

Eamon Gaffney

Abstract

The development of an Agent Based Critiquing System Architecture for a project management tool: *Prompter*

By Eamon Gaffney B.Sc.

Since the Software Crisis was first identified in 1969 there has been a frantic scramble among practitioners to define a software engineering discipline. This has led to development of established 'best practices' in areas of software design, metrics collection, cost estimation, risk analysis etc. To date, no tool has provided software managers with integrated project management support. This is the motivation behind the ***Prompter*** tool which seeks to provide assistance for project managers in the areas of decision support and planning throughout the lifecycle of a project.

The basis of this thesis is the design and development of a component of the ***Prompter*** tool known as the Daemon architecture. The ***Prompter*** tool, is an ESPRIT project developed by a consortium of companies including Dublin City University, Catalyst Software and Objectif Technologie. Its goal is to provide decision support to the user in the field of Software Project Management.

The Daemon Architecture for which I am responsible, provides the dynamic advice or criticism to the user using intelligent agents or mini experts. The architecture had to be as open as possible, distributed, domain independent, an easily expandable knowledge base, asynchronous from the rest of ***Prompter***, have the ability to incorporate new Agent languages and finally, platform independence.

The first stage of this thesis involved the design of the architecture outlining some of its components. The second stage was the development of the architectural design into a functioning prototype operated within the ***Prompter*** tool. This is followed by a discussion of some of the more implementational issues that arose during this phase due to, design flaws, implementation languages chosen, networking problems etc.

The resulting architecture outlined in this thesis can thus be used to provide decision support in many domains and on many platforms and is easily maintainable.

Table of Contents

Figures

1. Introduction

In the last few years computers and software have increased in complexity and sophistication at an extraordinary rate. With the advent of Operating Systems that were multithreaded and network based, computers had more processing power and resources available to them than ever before. This allowed computer software to become responsible for a wide range of complex tasks, from landing the NASA space shuttle, to controlling power stations. Thus, since software has become more complex, so too has the job of managing its development. Software project managers must be aware of as much information as possible when making decisions, and sometimes this can prove overwhelming. They have no one to advise them on what decision to make. For this reason, decision support systems such as ***Prompter*** were developed. ***Prompter*** provides project managers with a better view of what is happening in their project and assists them in their decision making process.

In this thesis a specific component of ***Prompter*** that I was responsible for designing and developing is discussed. This component is called the daemon architecture and in the coming chapters, it is broken down and explained in much greater detail.

1.1 Overview of Prompter

The ***Prompter*** tool is a decision support tool for software project managers. It was developed to assist project managers, provide them with advice in their decision making process and help them assimilate best practices in the field of software project management, specifically software project planning [Prompter 97].

1.1.1 *Prompter* background

The P3 project (Project and Process ***Prompter***) ***Prompter*** was a European project funded by ESPRIT by the fourth framework programme of the European Commission as ESPRIT project 22241 and had five project partners. The main developer partners were a consortium of companies that included Catalyst Software(Dublin), Dublin City University and Objectif Technologie (Paris). Catalyst were responsible for the central component of the tool, the School of Computer Application in DCU(Dublin) were responsible for the implementation of the knowledge within the tool, and finally Objectif Technologie (Paris) were responsible for the GUI

to the tool. In addition to these partners there were a number of user partners including Schneider Electric(France) and INTRACOM (Athens Greece) who provided user feedback and comments throughout the project relating to the tool itself and the knowledge it contained.

Prompter is based on work performed by Dublin City University between the years 1987-94. The " Riskman 2 project" [MoynihanT 94] developed in 1994 provided much of the inspiration which built on work from the Riskman and the IMPW [Verbruggen 87] (Integrated Management Process Workbench). It was a critiquing system architecture within the risk management domain. It was concerned with helping a project manager to walk around a proposed software development and anticipate any major risks to which the project may be exposed [HenryW 94].

The initial development of ***Prompter*** began in September 1996 and was completed in February 1999. I joined the School of Computer Applications in DCU (Dublin City University), in the early stages of the project during the architectural design phase. I was involved in the construction and writing of the architectural design document and the detailed design document. I was also the main programmer for the individual components within the daemon architecture, the interface between the architecture and the rest of the tool and finally in the integration of intelligent agents into the system. A number of prototypes were developed which allowed me to refine the architecture and produce an improved design.

As mentioned previously, software project management and specifically software project planning is the domain of the ***Prompter*** tool. It is concerned with the entire development process. Software project managers must be aware of many things in the course of their jobs, all of which may affect the outcome. A definition of project management can be found at [ThayerR 88].

Some areas of concern to Software Project Management include [PressmanR 94]:

- measurement
- estimation
- risk analysis
- scheduling
- tracking
- control

These measurement activities allow managers to better understand the direction in which the project may be going and thus foresee potential problems. As this is an on ongoing process throughout the development of the project, it can become difficult to keep track of all aspects of the project at any one time.

1.1.2 High Level *Prompter* Architecture

This section contains an overview of the architecture of the ***Prompter*** tool. Figure 1.1 highlights the tool's three logical components:

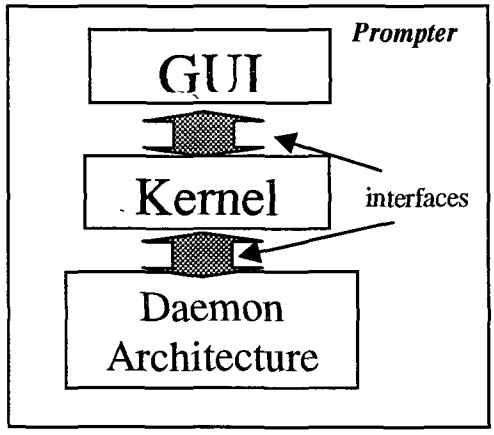

Figure 1.1 Overview of *Prompter* tool

- **The GUI** (developed by Objectif Technologie): This is the user interface to the tool and is responsible for all communication between the user and the tool.

- **The Kernel** (developed by Catalyst Software): This contains the central functionality of the tool. It is responsible for the scheduling of tasks and the processing of information. It also acts as an interface between the GUI and the daemons.

- **The Daemon Architecture** (developed by School of Computer Applications in DCU): The daemon architecture, for which I was the main developer, is a critiquing system in itself, providing the dynamic knowledge for the tool. A critiquing system critiques a user on what they have performed and illustrates how they may overcome their problem. It is similar to the idea of an expert system but is more dynamic in that it works with the user to develop a solution instead of providing suggestions. Contained within this architecture are a number of sub-components called daemons (described in full in chapter 3) which are responsible for the advice generated by the ***Prompter.*** These daemons are mini experts in the field of software

project management, which provide advice if requested or criticise what the user is performing, if a potential problem is discovered.

As can be viewed in figure 1.1, each of these components is separated by an interface. This allowed each to be designed and developed independently of the others. It also allows ***Prompter*** be an easily distributed tool. Each components can be resident on a different host on a network, and having some underlying networking language handling any information that crosses the interfaces. As a results the GUI may be situated on a number of machines in a network, with the kernel and daemon architecture situated on a high powered server in another area of the network This results in a light weight front end to the tool.

The tool itself helps guide a user through some tasks associated with the beginning of a project such as activity planning, resource allocation and cost estimation. If the user encounters some problems or feels they are unable to make a decision they can ask the tool for advice. The daemons are consulted and supply advice based on its knowledge and the information the user has entered. A good conceptual model of this process [Prompter 97] can be viewed in figure 1.2

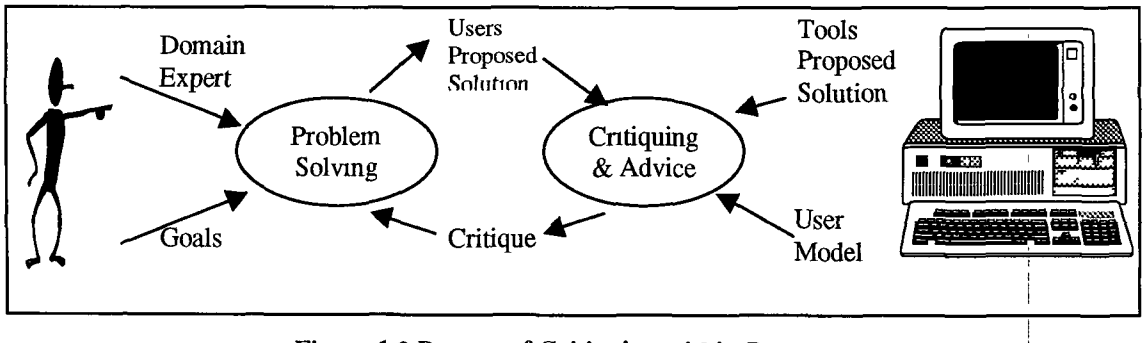

Figure 1.2 Process of Critiquing within *Prompter*

1.1.2.1 Daemon Architecture

My role within the project was to design and develop this daemon architecture from the conceptual level to a fully working component of ***Prompter***. This meant performing the initial design, the detailed design, the implementation and testing phases of the architecture.

Figure 1.3 gives an outline of the conceptual model of the daemon architecture. It is a critiquing system which retrieves information from the kernel about what the user is doing, and criticises it or provides advice on what to do next.

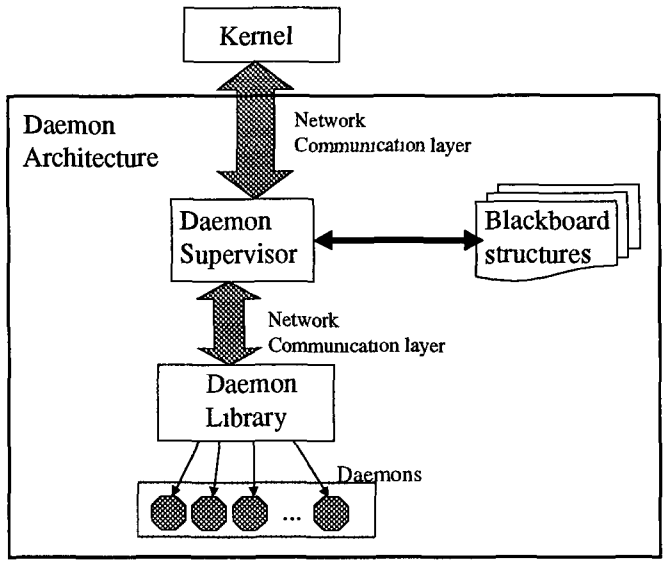

Figure 1.3 Overview of Daemon Architecture

The components making up the daemon architecture are:

- The **Daemon Supervisor**: The main controller of the architecture and is responsible for all communication between the architecture and the rest of the tool as well as the execution of the daemons.

- The **Daemons**: (intelligent agents) are the mini experts that perform the critiquing process. Each daemon is an expert in some specific area of software project management.

- The **Daemon Library**: This module is responsible for the maintenance of the daemons including their actual file storage, their versioning etc. This module is also responsible for their retrieval when they are available to execute.

- The **Blackboard**: This component helps monitor the state of all the daemons at any one time. This is by far the most complex structure in the daemon architecture. At any stage the blackboard holds the state of all the daemons in the system, this is to ensure that daemons do not execute over and over if they have not received any new information. This blackboard also holds the last piece of advice each daemon gives.

- The communication layer between the daemon supervisor and the daemon library allows the daemons to reside on a different machine to the rest of the tool if necessary.

1.2 Thesis Outline

Chapter 1 gave a brief overview of the ***Prompter*** tool and the area for which the tool was designed for. It gave some background to the ***Prompter*** project, such as who the project was funded by and previous projects that led to this tool's development. The architecture at a high level was outlined including a description of where my work fitted in. Finally, an outline of the major structure in the daemon architecture was given.

Chapter 2 offers an explanation of what a critiquing system is and its advantages over expert systems. Some critiquing systems (in development at the time of writing) are described with an overview of their architectures. The systems covered include CDMCS, Riskman, TraumTIQ and ICADS. Their architectures are outlined at a high level and their operations are discussed. From this discussion, I was able to construct a conceptual model of a critiquing system to illustrate what features are common to these systems. This model then helped me in the design of the daemon architecture.

The daemon architecture uses a concept called critiquing agents to providing advice to user and in chapter 3 this is explained. They are described and contrasted against other types of agents available. This chapter finished with a discussion on how these agents fit into the architecture and the structures required to allow them to operate.

Chapter 4 explains the detailed design of the daemon architecture. I used Object Oriented diagrams to design each component and in this chapter these diagrams are explained. This chapter also explains how each component communicates and operates with the rest of the tool and with the agents.

Chapter 5 discusses some of the implementation issues and some of the problems that were encountered during the development phase. To help understand some of these problems I thought it necessary to explain the implementational languages. There is an overview of programming in CORBA and how it affects the efficiency of the tool and an overview of Java and its advantages over other languages. Some other issues that are also dealt with include whether the choice of language was a good one, whether or not the prototype should have been multithreaded and a general evaluation of the implementation strategy is given. This chapter concludes with a discussion on what has been learned during this phase of the development.

Chapter 6 contains the main conclusions of the thesis. It discusses the various merits of the design, the weak points of the design and finishes with a discussion on the possible future work to improve the architecture.

2. An overview of Critiquing Systems.

This chapter outlines how a general model of a critiquing system was constructed and how it was used to aid in the design of the daemon architecture.

2.1 Introduction

Since I decided that the daemon architecture was to mimic a critiquing system, I needed a general model to base my design on. In this chapter I outline how I developed this model by explaining the architectures of a number of critiquing systems and highlighting their common components and how they interoperate. This general model is then explained and how it relates to the daemon architecture.

2.2 Overview of Expert Systems

An expert system is a system that employs human knowledge to solve problems that ordinarily require human expertise [TurbanE 95]. They were first developed in the 1960s with early attempts producing general-purpose problem solvers. By the 1980s they had become quite popular as a means of assisting a user with their problem solving. They remain quite popular today.

These systems provide the user with advice or plans of action on how to overcome a problem or perform a task. A definition of an expert system is as follows [RichE 91]:

> *a system which exploits a number of reasoning mechanisms for the purpose of solving a task or problem i.e. expert systems as well as expert advisory systems compute their own solutions and offer these to the user as a solution to the problem.*

They do not work with the user or examine the user's solution however.

Expert systems are generally composed of two major environments, the **consultation environment** and the **development environment**. The purpose of the development component is to provide the facility for the user to add to the knowledge base of the tool. The consultation environment assists the user in obtaining expert advice and knowledge, based on the information they inputted.

Thus, a user partakes in a question and answer session in the consultation environment, with the user providing the problem specific information to the system. From this the system

makes inferences and suggestions to the user in the form of a plan of action as to what to do next.

There are a number of problems that have been encountered in developing and using these types of systems:

- There is no guarantee that they will in fact reach a conclusion at all thus leaving the user with no advice. This renders the tool useless in some situations.
- As a result of the above point, expert systems only work well in situations with a narrow domain.
- For expert systems to work well they must be experts in their area. For this to be the case, the knowledge they have at their disposal must be collected from human experts in the same area. However many experts do not agree on several issues and thus the knowledge base is biased toward the experts consulted.
- As above, if the expert system provides a plan of action to overcome a problem there is no guarantee that it is the best way. In other words the approach of experts to situations may vary greatly.

In summary, expert systems are highly useful in many areas of problem solving. However once the domain becomes relatively large, the risk of it being unable to produce useful advice for the user becomes greater. In addition to this, the plans produced by the system are static, the user must stick to it rigidly or else the system is of no use. However a critiquing system is a more dynamic model that works with the user in developing a plan.

2.3 Overview of Critiquing Systems

A Critiquing system allow the user to work with the system to highlight problems or develop solutions to them instead of provide plans that they must stick to. For this reason it was decided to model the daemon architecture on such a structure.

It provides a criticism of a user's plan based on all knowledge at its disposal and can also advise them on different decisions. However it is up to the user to make the decision in the end. This process of critiquing is described below: [SilvermanB 92]

> *The user when using the system provides two sets of data to the system, the first set of information is the* ***problem description.*** *This data may contain the design requirements. The second set of data is the user's* ***plan of action.*** *This*

is the user's procedure for solving the problem e.g. the final plan of a building's structure or the steps involved in treating a patient in a hospital. This data is then analysed by the critiquing system to check the correctness efficiency, clarity or workability of the plan in relation to the problem and to its one suggested plan.

As stated above, the general model critiques before during and after the user's input. It provides feedback, criticism, and justification for this criticism, so the user may improve their solution. This is an iterative process and occurs several times in a project's life. It is this concept of feedback that distinguishes critiquing systems from other types of decision based systems such as expert systems and expert advisory systems. It is also an ideal model to base the daemon architecture on.

Some **advantages** critiquing systems have over expert systems [GernerA 93] are below:

- **Acceptability**: It may be easier for a user to accept a critiquing system over an expert system since the user views the system as helping their solution, rather than the expert system approach, which can be viewed as forming the plan which the user follows. Also, instead of the user taking a passive role, they are leading the critiquing system through the problem at hand, thus the system only prompts them when a problem occurs. Finally the solution produced is user centred and so individual to them.
- **Flexibility**: In general when dealing with real life situations, there is more than one solution to a specific problem, thus critiquing systems are better able to generate an acceptable solution dependent upon the user, and the knowledge base. Subjective judgement is also a real consideration which expert systems find extremely difficult to model and is yet so important in decision making.
- **Expert user**: The critiquing system sees the user as an expert with its own knowledge and beliefs and is therefore capable of making their own decisions. However the expert system provides no facility for the user to provide an input to the final solution. They can only accept it or reject it.

In the design the daemon architecture for ***Prompter,*** it was necessary for me to firstly create a conceptual model of a critiquing system to base it on. This required the surveying a number of critiquing systems available (at the time of writing), and the extraction of their common components.

Those systems studied were:

- **Riskman**: critiquing system in the Risk Management Domain
- **TraumTIQ**: System to provide critiquing to a physician dealing with trauma patients
- **ICADS**: Intelligent Critic System for Architectural Design. A system to help in all areas of building design
- **CDMCS**: Composite Design and Manufacturing critiquing system. A Critiquing system dealing with composite materials

These systems were chosen because each demonstrated a different type of critiquing system. One system provided critiquing in real time, one provided critiquing using highly mathematical methods etc. This ensure that the model I developed was not based on a specific type of critiquing system or on a specific domain.

From these systems, a number of common components were extracted:

- **Plan generator**: converts the user's data into an internal representation. This component is also responsible for generating a plan which is passed to the following component.
- **Differential Analyser**: This component compares the user's plan to the internal knowledge or the generated plan.
- **Knowledge**: the internal knowledge of the system.
- **Dialogue generator:** This generates the output advice in a more acceptable format.
- **Mini Experts**: These experts provide the critiquing process.

These headings are the most commonly used components in these systems and were taken from general critiquing system papers e.g. [SilvermanB 92].

2.4 Intelligent Critic System for Architectural Design (ICADS)

2.4.1 Introduction

The majority of information relating to this system was taken from [ChunH 97]. It is a Computer Aided Architectural Design system called ICADS for use in the area of building design. Critiquing systems are beginning to be used more frequently in this area.

This system is similar to many Intelligent Computer-Aided Architectural Design (ICAAD) systems, in that it provides the basic tools to draw plans (in this case floor plans), create 3D models of objects etc. However it also has extended capabilities with the introduction of AI techniques which provide the ability to offer advice and criticisms to the user on their designs. These AI techniques take the form of critics or as they are called within this system, **wizards**. Each critic, is an expert in a particular area of building design. These critics have the power to criticise a plan if necessary and to offer alternative solutions.

The main research focus of the ICADS project according to the designers was "*to develop a spatial representation that is rich enough to capture qualities of spatial relationships that are important in reasoning with government regulations and design principles*". The system is derived from work performed on other projects such as **EKSPRO, Janus** and **NALIC.** However ICADS goes one step further in that it also has the ability to reason out the relationships between object positioning e.g. placing a stairs in front of a door.

2.4.2 Architecture

The ICADS system architecture is illustrated in figure 2.1. It is an embedded system in that it is attached to the end of a CAD software. It is not concerned with a GUI since there are many systems available on the marketplace which already perform this task adequately.

This architecture is broken down using the above mention critiquing system components outlined in the section 2.3:

- **Plan Generator**: where the CAD information is converted into a representation that can be understood by the Critics and other modules
- **Knowledge base**: where the knowledge is stored in the tool
- **The differential analyser**: where the user's plans and the critics suggestions are compared
- **The dialogue generator**: where the results are converted into information that the user can understand

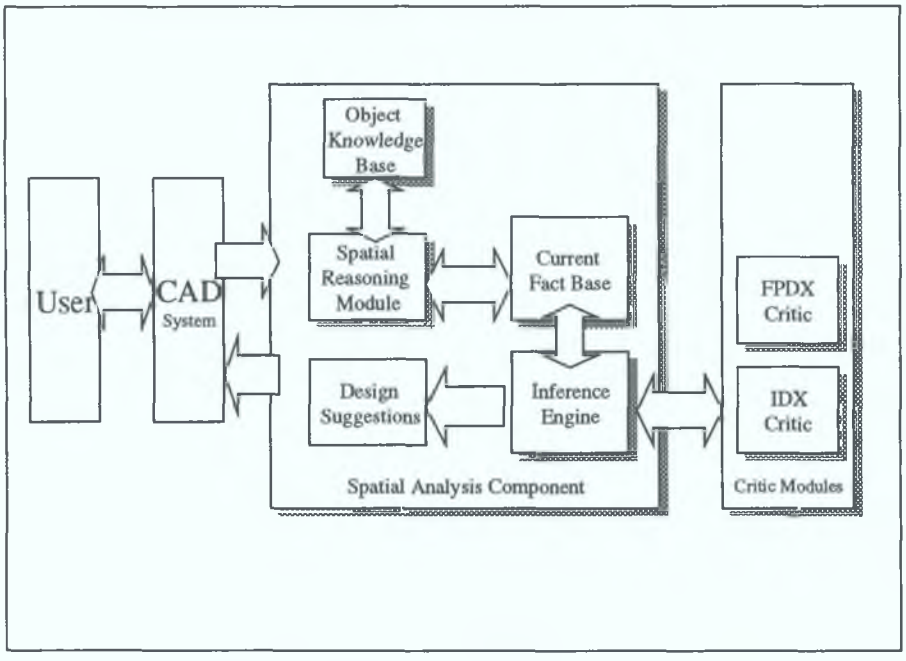

Figure 2.1 ICADS architecture

2.4.3 Plan generator: Spatial Analysis Component (SAC).

Critiquing systems use a plan generator to read in the user's data and convert it into a representation that it can understand. This information is then compared with the knowledge the system has at its disposal.

Within the ICADS architecture, there are two main components or areas of expertise, the **Spatial Analysis Component** and the **Critics Module** [ChunH 97]. The SAC is responsible for analysing the drawing or plan to produce an internal knowledge representation of it. It takes the graphical drawing from the CAD and converts it into an internal knowledge representation that the critics can understand. This allows the critics to analyse this plan.

In general, critiquing systems cannot analyse the data the user is working with i.e. pictures of the floor plan. The system must convert the user's information into data that the critiquing system can analyse. One of the components used to convert this data is the Spatial Reasoning Module.

2.4.3.1 Spatial Reasoning Module

This module is responsible for identifying all objects in the drawing that the CAD system has supplied [ChunH 97] and computing the relevant geometric information of each object. Also contained within this module is a set of ICADS spatial primitives or special rules, which analyse the data and extract spatial relationships etc. The result is a data set that fully

describes the plan the CAD has supplied. The relevant domain information about the objects is taken from the Object Knowledge Base. This stores all the static knowledge about objects. The resulting data is stored or output to the Current Fact Base.

2.4.4 Differential Analyser: Inference Engine

The critiquing system uses the "**Differential Analyser**" as the core of its intelligence. The user inputs the data set of their project (in this case the object), their types their locations etc. The Differential Analyser consults the **Critics** and the **Static Knowledge** using the problem specific information and constructs a plan of its own. This plan is then compared to the user's version. Obviously the two plans will never be identical and some leeway is required. Thus only differences beyond an acceptable threshold are criticised.

In the case of the ICADS, the inference engine is responsible for relating the information stored within the Current Fact-Base to the rules within the critics. This structure communicates with the critic modules. The critics check the designer's use of specific values and if a problem is found it will try to supply advice on how the problem may be overcome.

2.4.5 Knowledge Base

In this system there are two forms of knowledge base represented, Static and Dynamic knowledge which serve different purposes. The Static knowledge is stored in the Object Knowledge base and the dynamic knowledge is stored in the critics. These critics are represented in a rule format that takes information from the user and filters it through a number of rules that deal with different aspects of architectural design [ChunH 97].

This static information is stored in the Object Knowledge Base, which the system needs to access when analysing the plan the CAD has supplied. It is a repertoire of information on possible objects that the system can accept.

2.4.6 Mini Experts: Critics

These mini experts perform the dynamic critiquing of the user's plan. They form the second component of the critiquing process in this system. They consist of the Critic's rules and its knowledge space and are capable of analysing the data and proposing a plan or forming an intelligent criticism of it. These critics analyse the data through the inference engine and pass any advice/justification back to the SAC.

The rules within each critic are supported by a number of rules-of-thumb and basic guidelines, which have been entered into the Knowledge base. An example of a critic of this system [ChunH 97] is the Interior Design Expert (IDX) critic. This rule concerns the max and min size of various objects in a building.

The minimum width of each fire exit route is 900mm.

The minimum width of each exit door is 750mm.

The developer categorises these critics into Direction of Objects, Proximity of Objects, Spatial Relations of Rooms, Dimensions of Objects and locations of Rooms.

As a result, if the suggestion of the rule is followed, it leads to the changing of spatial co-ordinates. If the designer ignores this advice, the system must not continually prompt them about the fact.

The knowledge for these critics is encoded in rules which the inference engine can interpret and then easily apply to the information in the Current Fact-Base. The critic itself is built with the goal of acting as the designer's regulation advisor, and only informs them when one of these rules have been broken. It then provides an explanation of how the problem may be overcome. The rules themselves have different levels of importance so if a government rule is broken, the advice must be taken. However if a suggestion of a lesser priority is not taken the system must accept the decision.

2.4.7 Dialogue Generator: Design Suggestions

When a rule fires, the results are displayed in textual messages back to the designer. This is due to the fact that this system was still in development at the time of writing this thesis. The developers stated however that they planned to introduce the potential to allow the critics to modify the drawings automatically to satisfy rules. However there was little documentation on this component of the tool.

2.5 Composite Design and Manufacturing Critiquing System CDMCS

2.5.1 Introduction

The outline of this system was taken from [CDMCS 92]. It was developed by the University of Alabama and Tulsa and the "Research, Development and Engineering Centre" (RDEC) of the U.S. Army Missile Command.

It was designed to assist engineers who may not be familiar with composite technology to evaluate the advantages and disadvantages of using it for a proposed component design, and also to provide technical support to designers in the area of "productivity and engineering analysis". It is an attempt to provide a broad base of manufacturing knowledge, which provides engineers with up to date advice in the production/manufacturing discipline. The CDMCS critiques a design and supplies an account of the strengths and weaknesses of the proposed design.

A composite material can be described as a combination of two or more distinct materials differing in form or composition on a macro-scale. When developing some material, designers generally do not have a specific set of criteria for what each part of the object should be made up of. Generally they have a set of non-specific requirements e.g. the windings of a filament may not have a set value but instead have a range between two values. This is known as the Fuzzy criteria of the object i.e. the data and the evaluation criteria are "spread" or "fuzzy". Thus any number of possible combinations can lead to the solution.

2.5.2 Architecture

This system differs from the previously mentioned system, in that it utilises a more mathematical technique as part of its expertise. These mathematical models [CDMCS 92] are called **metrics**. They are used as a criterion to measure the correctness of the result of a piece of information and are grouped in levels of importance in the differential analyser. The architecture of the system can be viewed in figure 2.2 below.

The designer's data and the expert metrics are entered into the differential analyser. This compares the two plans by passing them through three levels of metrics, where the first level is the most important having a large effect on the result, the next level having a lesser effect,

and so on. The resulting evaluation is then presented to the user after some processing by the Dialogue Generator on the results.

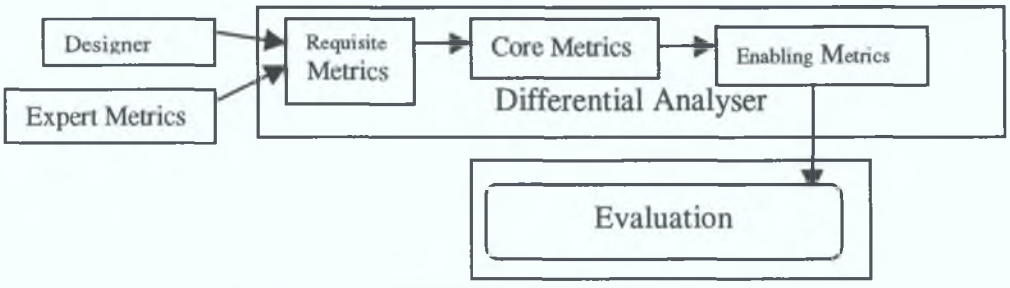

Figure 2.2 CDMCS architecture overview

2.5.3 Mini Experts: Metrics

Metrics describe the materials used and how the knowledge is related, to provide a measure of the "correctness" of a given process. They are composed of definitions of relationships between design parameters, i.e. domain data supplied by the designer's plan. These metrics are organised into a hierarchy, which is a reasoning mechanism that utilises qualitative and fuzzy knowledge, and operates on these metrics to produce the critique on the design.

Metrics are as varied as the knowledge they contain and are the core of the system. Realistic problems deal with various type of metrics, which can be qualitative, quantitative, Boolean and conditional. An example of a boolean metric can be seen below.

If **Hole** is yes, THEN
Tolerance / Wall-Thickness >10%

The result of each metric is then considered i.e. how important the result of a specific metric is to the entire design. If it is not that critical then it may be ignored. The metrics are classified into three levels of importance [CDMCS 92]:

- **Requisite**: These metrics in general have only two possible outcomes and represent conditions that must occur in order for the design to be realistically considered. If a metric is not satisfied then the design is rejected. These metrics contain the most important information in the system.
- **Core**: The next level of importance. It is essential that they are to some degree, accepted. The degree of satisfaction directly effects the outcome. It is at this point that the notion of the fuzzy criteria associated with the metric is introduced.
- **Enabling**: The lowest level of metrics. They alter the basic correctness of the design but only in very small ways. These metrics are not essential to the success or the failure of the design.

2.5.4 Knowledge Base

Figure 2.3 was taken from the project overview of all the stages of composite product/process design. It is important to note that this hierarchy was still under development at the time of writing this thesis. The diagram shows a number of metrics and how they fit together. Some examples of the metrics are given below [CDMCS 92]. This taxonomy is displayed to show the complexity of the knowledge that must be represented. Each box is a metric that may not appear that important on its own but when a number of these metrics are collected together the systems can become quite powerful. There also exists a mechanism which operates on the structure to produce an overall state of the system. This mechanism is called **Satisfaction.**

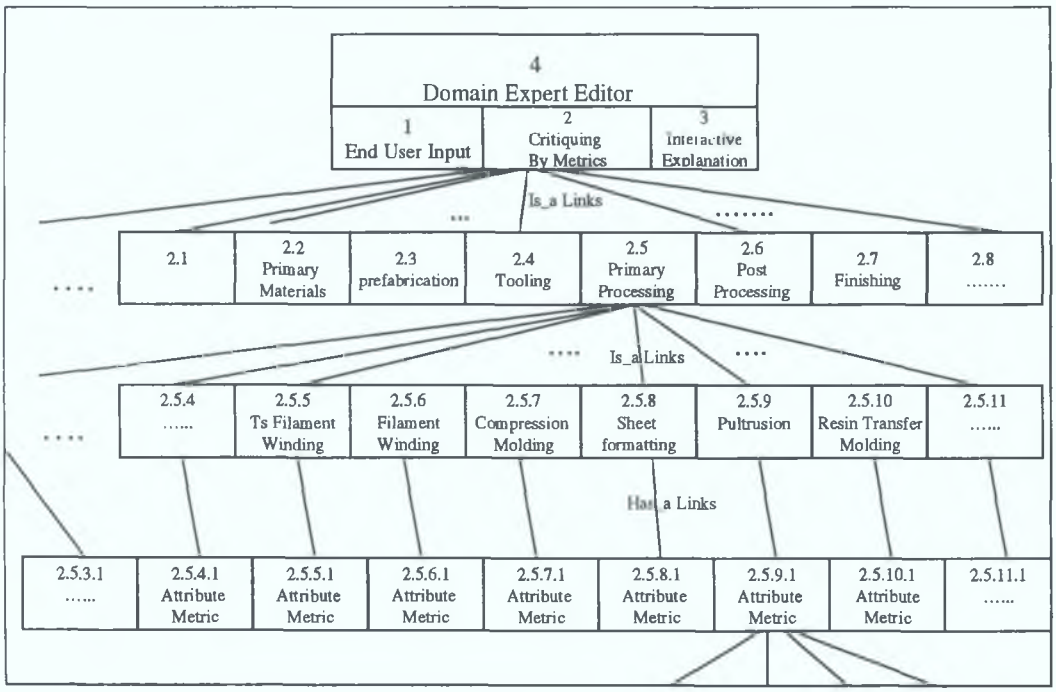

Figure 2.3 The CSMCS taxonomy

2.5.5 Differential Analyser: Satisfaction

The key element adapted with this system for the critiquing of a design is the concept of "Satisfaction". *Satisfaction is the measure of the degree to which some data item is contained in a metric.* It is a function of both the input data and the value given by the designers to the metric. Thus since there are three levels of metrics, there are three levels of satisfaction. For the core metrics and enabling metrics the satisfiability is determined by probability, since these are simple 0 or 1 outputs. An example of one of these formulas is as follows.

$$S_i = \iint_{Surface} W(x,y) P_x(x) P_y(y) M(x,y) dxdy$$

Where W is the unitizing weighting function, P is the density function and M is the metric satisfaction function. The output of this function in graph format is illustrated in figure 2.4 with the area of satisfaction being the shaded area or also known as the area of variable

acceptability. After the value has been acquired it is possible to evaluate the importance of this satisfaction in terms of the importance of the particular metric.

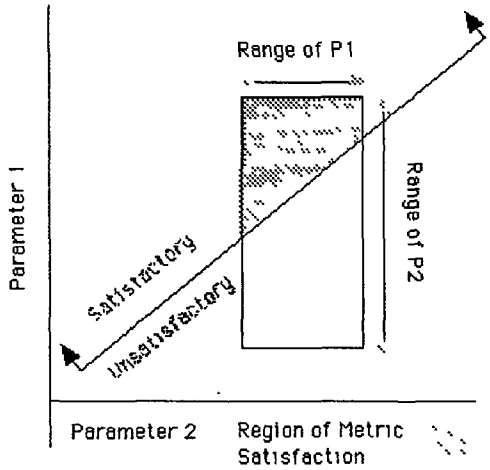

Figure 2.4 Graph showing the area of satisfaction of a metric

These functions are a form of thresholding. A breakpoint is determined for the metric, which represents the point at which a metric is assumed to become unlikely to be satisfied. Each metric has its own breakpoint which the developers determine.

Thus metrics are capable of evaluating information against the knowledge they have at their disposal and from this, produce a result. This result is then converted into useful advice for the user. This is performed in the dialog generator.

2.5.6 Dialog Generator: Aggregation

This is the final stage in the critiquing process. The purpose of Aggregation is to collect the satisfiability values received from all three levels of metrics to compute one overall value. The method of aggregation is defined as the sum of the averages of the core and enabling metric fuzzy function evaluations. This value is mapped to the success or failure state. There the results are transformed into useful advice and returned to the user through a GUI.

2.6 Critiquing Trauma Management Plans On-Line: TraumaTIQ

2.6.1 Introduction

The information concerning TraumaTIQ came from a number of sources [GernerA 93] [GernerA 94]. It is an extension to the **TraumAID** system, which is a decision support system for the delivery of trauma care during the initial definitive phase of patient management. Although TraumaTIQ is not part of this system it is responsible for interpreting the proposed actions in the context of the current state of the patient and thus producing a critique of these actions.

The TraumAID project was developed by the University of Pennsylvania to assist a physician during an initial "definitive management phase of patients with severe injuries". At the very core of this system, data about the patient's condition is monitored and advice or criticism provided to the physician about the plan of action. Again the planner must schedule all the above actions and procedures so as to produce a plan for the physician to consider.

This system differs from many others including those discussed already, in that most systems perform critiquing during an off-line consultation session. The user in this situation will not have the time to be sitting at the screen. Their attention is only drawn to the output. Thus for the system to be kept up to date requires constant attention and the amount of time available for executing actions is also limited. These requirements lead to a system that is **task centred** rather than system centred.

2.6.2 Architecture

An overview of the system is given in figure 2.5. The critiquing process in the TraumaTIQ system can be activated in a number of ways e.g. when either the physician orders some actions to be performed or new information or advice upon which route to take is requested.

The system is divided into three main components:

- **Plan Recognition**: which allows the system to infer the likely goals of the physician's plan in relation to the patient's condition.
- **Plan Evaluation**: identifies any flaws or potential problems within the physician's plan and any large deviations between its plan and the physicians.
- **Dialog Generation**: Translates all the results found into useful information that the physician can use. It must be prioritised so the most critical information comes first.

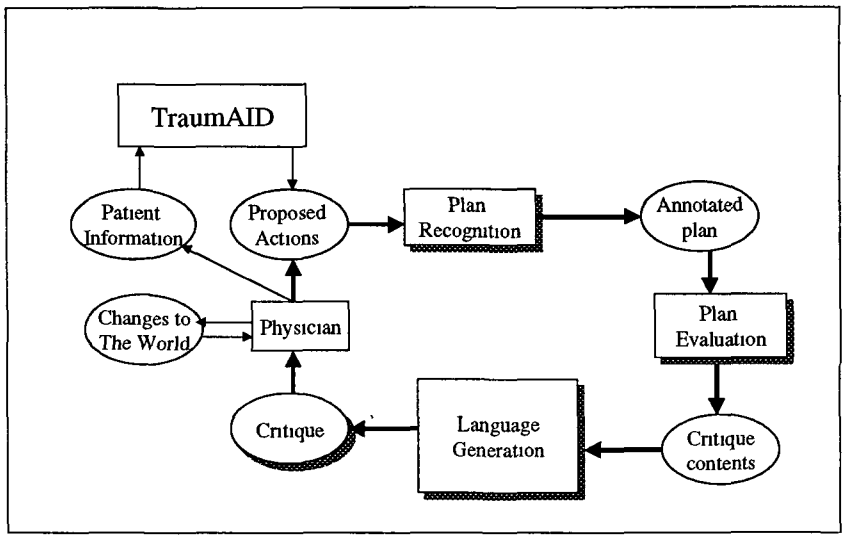

Figure 2.5 The Architecture of the TraumaTIQ system

The critiquing process is triggered whenever a new piece of relevant information is made available to the system. The cyclic appearance of the architecture is to ensure that as new information becomes available, the entire system is updated. The resulting critiques are generated based on the complete set of orders that are pending at any one time.

2.6.3 Plan Generator: Plan Recognition

This system uses information about the situation in which the plan is being developed, in order to *infer* the plan the physician will most likely use. This decision will be based upon the plan that finishes and the minimum number of unsatisfied goals at its end. The reason for this is the user cannot be treated as a co-operative provider of information but what can be observed "through the keyhole" is actions that have been performed and orders the physician has placed for actions.

An advantage of this system is that it uses a form of knowledge representation known as **contextual knowledge** along with **basic domain principles** to guide the search for an explanation plan. Thus if the system has a goal it considers relevant, it tries to use a number of principles outlined in the following algorithm as an explanation for the physician's proposed actions.

IF an action is ordered by the physician **THEN**

check if it is part of TraumAID's recommended plan as a means of satisfying a goal

IF action is in the recommended plan **THEN**

add the action to the representation of the physician's plan

ELSE

DO determine whether there is a relevant goal that might address what the action involves

IF any goals that might lead to the action are present in the TraumAID's set of active goals **THEN**
assume that the action is being performed to address the goal
ELSE IF there is no relevant goal to explain why the physician has ordered it **THEN**
check whether any of the possible goals motivating the action are part of a currently active diagnostic strategy
IF no relevant goal or strategy is found **THEN**
add action to the representation of the physician's plan with no goal attached
IF the system only knows of one goal that would lead to performing the action **THEN**
assume the action is addressing the goal.

2.6.4 Differential Analyser: Plan Evaluation

The plan evaluation detects flaws or mistakes, misconceptions and disagreements with the physician's plan in comparison to the suggested one. This can be performed using two approaches called the **differential** and the **analytical** approaches.

The **differential approach** compares the physician's plan with another plan which could be a broadly acceptable course of action to solve the problem. So if it is unhappy with the plan suggested, it will have at least one other plan to recommend.

The **analytical approach** is described as a workspace of possible plans within which a solution is more or less acceptable. It allows the system to deal with domains, where variability and subjectivity are introduced into the decision making process i.e. since it is capable of generating its own solutions, it can operate on problems where the domains are too complex or constrained to be solved using simple decision rules.

Thus the differential evaluation, explaining why its solution is the correct way to approach the problem, and the analytical constraints, generating explanations as to what is wrong with the users plan, are combined. By comparing the model of the physician's plan with the plan developed by TraumAID, TraumaTIQ is capable of recognising four different types of discrepancies [GernerA 94]:

- **Omission**: if the physician is ignoring some specific goal.
- **Commission**: if the physician orders some procedure, which the system does not feel is constructive or useful
- **Procedure Choice**: when the physician had ordered a procedure to tackle a goal or problem which the system considers the wrong choice and that better procedures exist.
- **Scheduling**: The system considers the physician is addressing lesser problems when more pressing ones exist.

2.6.5 Knowledge

As outlined in the above algorithm in section 2.6.3, the knowledge base is stored in an hierarchical structure through which the system navigates. This knowledge contains a set of plans on how to deal with different situations. The system has a set of guidelines to interpret orders that do not correspond to its knowledge of possible plans.

Knowledge is then used, after the differential analyser has finished, to filter the output, so that non-trivial errors will be critiqued, using the magnitude of the different types of errors that have occurred. This method of filtering is implemented by separating errors into one of three categories: Tolerable, Non-Critical, Critical. This knowledge is stored in an error taxonomy This taxonomy classifies these errors by their potential impact on the patient's outcome. Using this knowledge it calculates an expected disutility value for each error classification. This information is then given to the dialog generator as feedback to the user.

2.6.6 Dialog Generator: Critique Generation

This component involves the generation of advice to the user from the information retrieved from the Plan Evaluation stage. It is considered to be one of the most important stages since, if the advice is not presented correctly then the physician may become confused or start to ignore the system altogether. It is separated into two stages:

- **Strategic generation** involves determining the content and structure of the output. The output itself is driven by the plan evaluation stage, information from omission errors, commission, procedure choice and scheduling errors. Depending on the level of urgency of these errors each piece of advice will be given an INFORM or WARN flag. In addition to this the system must supply justification information to support this criticism.

- **Tactical generation,** on the other hand, is related to displaying the critique to the physician. It was considered that individual tokens could relate to a string such as:

  Close_Chest_Wound $\Longrightarrow$ " closing the chest wound"

  However this does not allow these tokens to fit into a lot of varied sentences. To improve the quality of the output, a more general semantic decomposition of these concepts must be available. This representation together with an appropriate grammar and lexicon, is used to generate sentences. Combinatory Categorical Grammar as this is called is a functional head-driven, top-down approach to tactical grammar [SilvermanB 92].

2.7 A Critiquing System Architecture in the Risk Management Domain: Riskman 2

2.7.1 Introduction

The information for Riskman 2 was taken from the following sources, [PowerJ 94] [HenryW 94]. It was developed by the School of Computer Applications in Dublin City University (Ireland) to provide decision support for Software Risk Management. Riskman 2 was developed as a research project funded by the Irish National Software Directorate in order to develop a Risk management case tool. It is based on previous tools such as Riskman 1 and IMPW [Verbruggen 87]. Its goal was to help a project manager "walk-around" a proposed software development project and to help them anticipate any major risks to which the project might be exposed. Riskman 2 was one of the initial tools that acted as a starting point for the P3 project. ***Prompter*** is a development on the Riskman concept into the wider field of Software Project management.

Similar to the previous systems, the user enters information dealing with their project which passes to a collection of mini-experts or daemons in the field of risk which analyses this information. The system stores the results or advice in a structure called a Blackboard. This advice takes the form of text, which is embedded in the daemon and provides the user with ideas about reducing the risk factor of their suggested design. These daemons also supply justification for the advice if the user requires it.

2.7.2 Architecture

The basic architecture of the Riskman2 is based upon the lessons learnt form the Riskman1 project, which was a rule-based system. The architecture for Riskman2 is both flexible and could readily be enhanced. The architecture [PowerJ 94] can be viewed in figure 2.6:

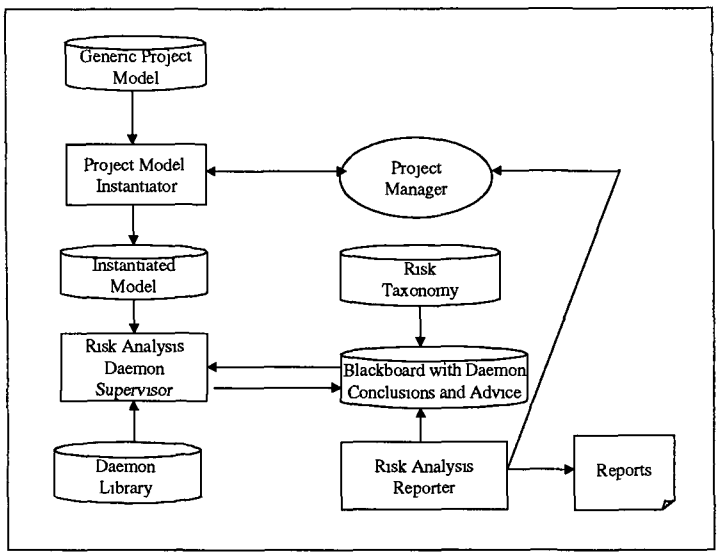

Figure 2.6 Architecture of Riskman2

2.7.3 Knowledge: Risk Taxonomy

The project Risk Taxonomy is a description of the various types of risk a project can be exposed to. The Riskman2 project categorises its taxonomy into five groups. One of these categories, Technical Risk, is illustrated below in figure 2.7 to show how the knowledge is represented. This class can be broken into a number of subclasses.

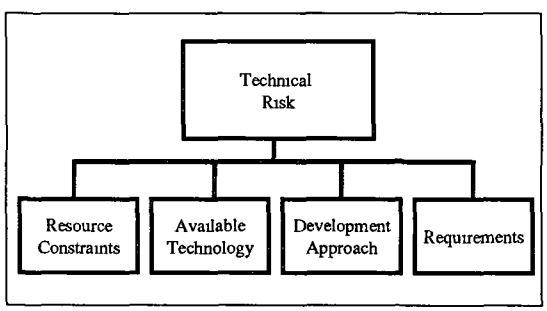

Figure 2.7 Subclasses of the Technical Risk class

This representation allows the easy addition of new categories to the knowledge base. The differential analyser would then consult this taxonomy, when the user presented information to the tool.

2.7.4 Differential Analyser: Risk Analyser

This is the heart of the tool where the user's decisions are analysed and critiqued. To perform this task there are a number of components such as the Daemon Library, the Blackboard, the Risk Taxonomy, and the Daemons themselves.

2.7.4.1 Mini Experts: Daemons

A daemon is defined as an individual expert in some specific area, of Risk. They hold all the knowledge about their specific area. For each class of the Risk taxonomy illustrated above, there exists at least one mini expert. When Riskman2 analyses a project's risk it lets the daemons examine the user's project information as well as the outputs from other daemons. They compare this information with what is stored in their knowledge base, and criticise it if necessary.

These daemons are controlled by an Inference Engine (IE) situated in the daemon library. This IE passes the user's data into the daemons and processes any advice they produce.

2.5.4.2 Blackboard

A blackboard is a problem-solving model, used for the purpose of allowing a number of mechanisms to communicate [EngelmoreR 88]. The Riskman2 developers described it as *a system, which uses multiple independent knowledge sources to analyse different aspects of complex problems.* These independent knowledge sources are the daemons in the system. For every element in the above taxonomy there exists a daemon which is the expert in that area, and for every daemon there exists an area for it in the blackboard. A partial taxonomy structure in the blackboard structure can be viewed in figure 2.8 below.

On the blackboard the specific areas, drivers and respective factors of the taxonomy are represented. Each daemon contributes its information or advice to the common workspace and at that point other daemons may take it and use it to generate more information i.e. blackboards are composed of solutions from component solutions [HenryW 94].

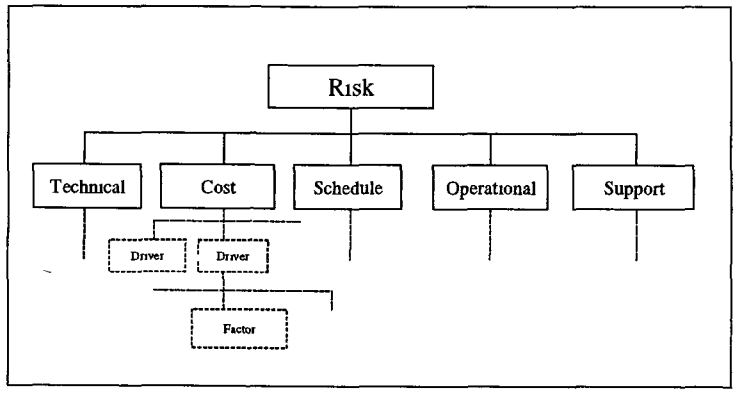

Figure 2.8 Blackboard structure for Riskman2

A blackboard however must have some form of organisation imposed upon it. These daemons cannot write to it anywhere, or the result would be chaotic. The blackboard in the Riskman2

project is hierarchically organised into various levels of analysis. Information associated with objects on one level serves as input to a set of knowledge sources which place new information in other levels. When a daemon has performed its calculations it posts its results to the appropriate place in the blackboard. Every daemon has its own allocated space or slot. Thus the daemons will know where to look if they require information.

2.7.4.2 Daemon Library

This component is responsible for the storage and upkeep of the daemons themselves and to act as an inference engine to the daemons. An inference engine can be defined as a mechanism for manipulating rules from the knowledge base and drawing conclusions and inferences from them with respect to the data from the project. The inference engine chosen for this system is a forward chaining production rule system. The rules take the form of simple *IF* statements which provide the bulk of the domain dependent knowledge in most expert and critiquing systems.

These rules are contained in a daemon or mini-expert system, and are parsed by combining the rules in the daemon with the data from the user to produce a result.

2.7.5 Dialog Generator: Risk Analyser Reporter

When daemons finish their execution, the Risk Analyser Reporter delivers the final report to the user in a concise and clear manner. The developers specified that this output should contain a number of factors e.g. Risk Area, Nature of the risk, justification for the concern, plan to remove the risk, etc.

There are two mechanisms used to produce the output, breath-first and depth-first traversal.

- **Breath first Traversal**: this is traversal of the blackboard hierarchy as a breath first level to provide the user with an overview of all the risks that are of concern in all areas of risk.
- **Depth-first Traversal** – if the user requires a more detailed explanation of a particular area then the search continues down that area of the blackboard only, resulting in all the risk concerns that led to the conclusion.

2.8 General model of a Critiquing System

2.8.1 Introduction

From the above-mentioned systems it was possible to see that they have a number of concepts in common. In this section a general structure for a critiquing system is illustrated. This model acted as a base in the design of the daemons and the daemon architecture.

The general model of a critiquing system involves a user inputting data and allowing the system to provide feedback, criticism, and justification for this criticism to the user. In addition to this, many systems come with static advice built into a repository of some kind. This advice is used during critiquing and also before the user begins. Thus, most critiquing systems have a knowledge base, a mechanism for generating a plan, a component for comparing these plans and finally a mechanism for turning results into advice.

2.8.2 General Model

The general model I developed is illustrated in figure 2.9.

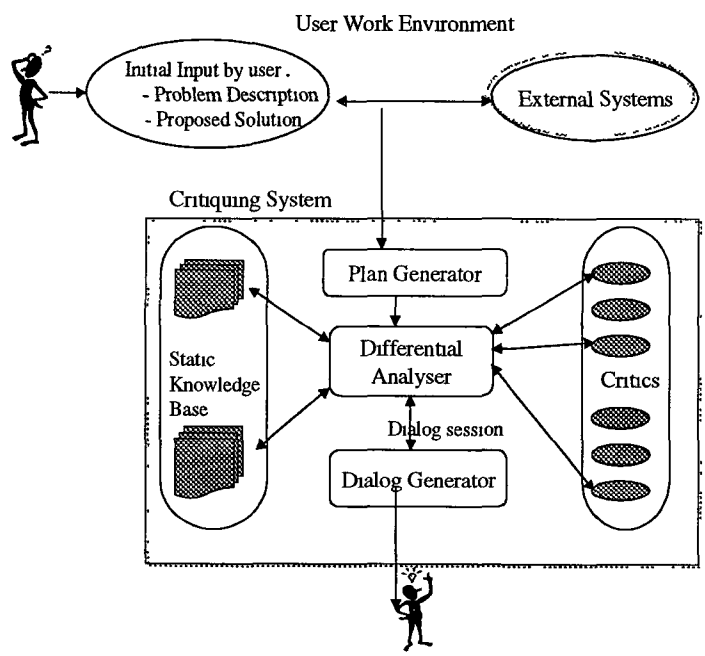

Figure 2.9 Overview of a Critiquing system

2.8.2.1 Plan Generator

The first stage of most critiquing systems is the "**Plan Generator**" which causes the system to develop its own plan. A plan is generated from the user's information and passed to the Differential Analyser. Some systems use their knowledge base to generate a default plan

while others have a more dynamic plan generator. In the ICADS system, the plan generator is the SAC component, which converts the user's data into an internal representation and does not involve the system generating its own plan. However, in the TraumaTIQ system there is a specific plan generator which creates its own plan separate to that of the user.

2.8.2.2 Differential Analyser

The critiquing system uses the "**Differential Analyser**" as the core to its intelligence. They input data relating to their project along with other relevant data from other systems involved within the project environment. This allows the plan generator to generate its own plan of action. The two plans (the user plan and the system generated plan) are then fed into the differential analyser.

In general there are two mechanisms for creating a differential analyser which are conceptually the same. In the first mechanism two plans are passed to the differential analyser. It compares them by consulting the **Critics** and the **Static Knowledge** to decide which plan is better, as is the case with TraumaTIQ. The second mechanism is to give the differential analyser the problem specific information (the user plan) only and compare the plan against the knowledge of the critics and the static knowledge as is the case with ICADS or Riskman.

However there are rarely two plans exactly the same and so some leeway is must be given. Thus only differences beyond an acceptable threshold are criticised. This is known as fuzzy logic (used in the CDMCS systems).

2.8.2.3 Dialog Generator

The differences are presented to the "**Dialog Generator**" which converts them into information useful to the user. This information usually contains justification for the criticism and advice on how the user plan may be changed to incorporate these changes. The user considers this information and if required a dialog session is started whereby the system and the user "discuss" the results.

As a result of this "discussion" property, critiquing systems can complement problem-solving systems with or without full knowledge of the problem space. They can provide criticism on areas in which they have expertise and since the user is also considered an expert, the solution is based upon a larger knowledge base than if the system had to solve the problem on its own.

2.8.3 Knowledge

The above statement about world knowledge hits upon one problem. Critiquing systems are capable of critiquing without full knowledge and as a result the problem of guaranteeing the correctness of the knowledge becomes a problem. How much information does the system require before it can give correct advice? Silverman [SilvermanB 92] suggested a four-point framework to reduce the chances of error occurring in the knowledge base. If any of these tests fail then the body of knowledge is considered unsound. This plan is outlined below

- **Clarity Test**: All statements must be clear and unambiguous as they are easier to falsify and therefore more testable. A statement is considered to say more, the more falsifiable it is.
- **Coherence Test**: This tests the logical structure of statements and whether the result omits knowledge about the problem.
- **Correspondence Test**: This test concerns the agreement of statements with reality i.e. if the body of knowledge reflects the real world in relation to the problem at hand. This test fails if one or more elements of the problem space are not represented within the knowledge base.
- **Workability Test**: This test is to see if it is possible for the body of knowledge to lead to a description of the problem and to check that there are no omissions. Silverman states that critiquing is not a "sermon" but instead a case of verification and validation of the body of knowledge. It requires a mutual exchange of viewpoints between the human and the system.

These four points lead Silverman to the following principle [SilvermanB 92]:
"One can criticise knowledge in terms of its clarity, coherence, correspondence and workability but if one of these tests fail to discredit the knowledge then the others are still free to do so ... The principle is one of critiquing the credibility of the user task results, rather than proving the correctness of it".

Although this knowledge is a very important factor in critiquing systems, a number of other considerations must be taken into account before the user is presented with advice. For example: What does a critiquing system do when it runs into conflict with the user? Should it argue its case? Critiquing systems must interact with the user so that if it presents advice in the wrong way it can be rejected. It is often not just the content of a critique but the way in which the critique is delivered, or the author's reaction to it, which can determine its successful usage.

Gerard Fisher has made active critiquing the central focus of his research [FischerG 93]. **Active critiquing** can be described as the monitoring of designer actions and the active interruption of the design process to point out errors, or suggest guidance [EckertC 95]. Fisher showed that passive critics of the user's design were not requested early enough and resulted in mistakes early in the design, which were costly to fix. Eckert [EckertC 95] suggests a form of critic known as Collaborative Critics for professional designers. It is a combination of the two forms of critics where the system suggests advice when deemed necessary but in addition to this the user can request advice.

2.8.4 Critics

Within this knowledge base, the static rules and standards are contained. This information can be very useful before the user even starts. After this the more general information must be stored in the form of simple rules such as

IF condition1 THEN
statement1
ELSE IF condition2 THEN
statement2
END

If condition1 is true execute statement1 else if condition2 is true then execute statement2.
Although there are more complex mechanisms for developing critics the above mentioned mechanism still seems the most popular. Judging by the aforementioned critiquing systems i.e. the ICADS, TarumaTIQ and Riskman, they all represent their critics this way, as they are simple to construct and verify. However since their implementation generally depends on the domain the language used, the complexity of the knowledge etc. their structure cannot be generalised.

2.9 Summary

In this chapter a number of critiquing systems were described for the purposes of developing a general model. The chapter began with an overview of an expert system which was then contrasted with a critiquing system. This was followed by a description of the critiquing systems used in the survey.

The first architecture was the ICADS system. This system demonstrates a model that fits in well with the general critiquing system model. It demonstrates the use of separate critics that specialise in a particular area of building design.

The next system discussed was the CDMCS system. This system was included in the review because of its divergence from the normal critiquing system design in that instead of critics it uses mathematical functions to model knowledge. However, even though it is different to the previous system it still conforms to the general model.

TraumaTIQ was included because it operates in a real time environment. It deals with real world situations in real time. It is a task centred system and must be able to adapt to situations that occur outside of its control.

Riskman was the final system. It attempted to model a critiquing system for Software Risk management. It was included because ***Prompter*** is a development on some of the its ideas.

One problem that was encountered during the creation of this study was the difficulty in acquiring detailed information about commercial systems. This was due to the fact that most of the critiquing systems designed today are produced for commercial use and not research, as a result the design documents remain classified to the project.

In the following chapter, the mechanism by which the daemon architecture provides critiquing is explained i.e. Intelligent Agents. It is shown how the general model developed here was used to develop an Intelligent Critiquing Agent and the structures that were necessary to integrate them.

3. Agents

The objective of this chapter is to explain the concept of software agents and the various types available. Following on from this I describe how I combined the idea of an agent with the concept of a critiquing system, to produce a number of agents that are capable of providing decision support within an implemented prototype of the ***Prompter*** tool.

3.1 Introduction

With the advent of the Internet, the availability of knowledge contained within many different types of system such as expert systems, databases etc. became much more accessible. It, as a result, became more difficult to produce systems capable of accessing, incorporating or simply taking advantage of this knowledge due to compatibility problems, various conflicting standards and security issues. Thus the concept of the open architecture was born. These systems were designed to be easily distributed, platform independent, easily expandable, domain independent etc. They could interact with knowledge bases or other systems of various types.

One of my goals was to make the daemon architecture as open as possible and one mechanism of achieving this was, through the use of intelligent agents. Agents work on behalf of the user or system, performing tasks such as information retrieval or database querying etc.

Open agent systems mediate between different types of programs and generate problem-oriented solutions. In some situations agents reduce network traffic, provide efficient means of overcoming the problem of incorporating legacy systems, and most importantly, they have the ability to operate asynchronously and autonomously of the process that created them thus helping developers construct more robust and fault tolerant systems.[LangeD 98]

The agent concept was adapted for the ***Prompter*** tool as a mechanism of storing and organising the knowledge of the tool. Each agent contains information relating to a specific area of software project planning and can execute on its own, offer advice when necessary and provide an easy mechanism for the addition or deletion of agents from the knowledge base. They also allow the architecture to remain distinct from the domain it represents.

3.2 Agents

The term Agent has been used for quite some time now without people fully understanding what it is that an agent is expected to do. There is ongoing research into finding an agreed definition for it. There are also those that do not support the use of agents and are sceptical of their problem solving ability [PetrieC 97].

There have been attempts at a definition for agents. The examples below show how each definition leans towards a specific type of agent and also illustrates how vague the language used:

- Intelligent Software Agents can be defined [CroftD 97]as software agents that use AI in the pursuit of the goals of its clients.
- A mobile agent is an active object that can move both data and functionality to multiple places within a distributed system. [FarleyS 97]
- An agent is a computational entity [Broadcom 97] which:
  - acts on behalf of other entities in an autonomous fashion
  - performs its actions with some level of proactivity and/or reactiveness
  - exhibit some level of the key attributes of learning, co-operation and mobility.

All these definitions describe an autonomous goal orientated entity that operates asynchronously and may communicate with the user as well as with other agents with the purpose of helping the user. However these definitions are quite vague due to the variety of areas in which agents are found and the various tasks they perform. A better description of an agent is given in Danny B. Lange's paper [LangeD 98], an agent is a software object:

- situated within an execution environment
- possessing all, of the following properties:

  reactive – senses change in the environment and acts according to those changes

  autonomous – has control over its own actions;

  goal driven – it is proactive;

  temporally continuous – is continuously executing.

  - And possibly possessing any of these orthogonal properties:

    communicative – able to communicate with other agents;

    mobile – can travel from host to host

    learning – adapts accordingly to previous experience

    believable – appears believable for the end user.

This definition allows us to pick out the specific properties that our agents can posses and even though this definition is at a lower level, it is in no way closed. Thus for the purpose of the ***Prompter*** tool the definition of an agent is [Prompter 97]:

> *... a fully encapsulated program (entity) which is capable of autonomous asynchronous behaviour in some environment for a specific purpose (goal). They have a knowledge base that allows them to manipulate information and also have the ability to communicate with the Prompter tool.*

These agents are reactive to a certain extent within the daemon architecture. They react when information relating to them changes. They are autonomous i.e. execute on their own, they are independent of the architecture itself and are also goal driven. There is no reason for these types of agent to be mobile. However the architecture is open to the incorporation of these types of agents if the situation ever arose.

3.2.1 How Agents differ from programs

There is a tendency to see agents as nothing more than a buzz word. In this section I will argue that this tendency is misguided and that agents are more sophisticated independent entities.

Agents are based on the concept of reactivity in that they are not scheduled for execution by the system but execute when they have something to contribute to the user. They have their own thread of control and execution environment, which is not the same as programs or methods. Within the ***Prompter*** prototype this is also true. Each agent is autonomous and operates asynchronously of the system. Each is a mini expert which supplies information to the user when advantageous. An agent is programmed with its own individual knowledge base, from which it gathers information when supplying advice to the user.

Programs on the other hand are less reactive [Broadcom 97]. They are specifically scheduled or called from code to perform a particular task. They only respond to what interface designers call direct manipulation. Nothing happens unless a person gives the command. Functions and methods only execute when called. The only information they have is what is supplied to them through variables. When they finish, the results are passed back to the system. It is a very client/server oriented approach. Although the code execution may not be static, it still conforms to the

client/server approach. They are not pre-emptive. They do not suggest their availability to execute and as a result are not temporally continuous or autonomous.

3.3 Types of Agents

Since there is no hard and fast definition of what an agent should look like, the result has been the production of a number of different agent types. Up until now a lot of agent implementation and development has been focused on the area of the Internet and agent mobility. Some of these agents are outlined below.

- **Mobile agents** – These are the most common form of agents. They have the ability to move from machine to machine and shuffle their code and state with them when they move. So when an agent is operating on a heavily laden machine it may be advantageous to move to a different, less burdened machine. These types of agents are generally used as a method of information gathering, travelling from server to server possibly querying many databases etc. This is where the whole area of agent trust and security becomes important.
- **Distributed Agents** – These types of agents are mainly used to reduce load balancing and can be distributed over a number of computers or processors on a network. This allows agents requiring large amounts of processor time to search for a free processor or to be allocated a dedicated processor, thus reducing the overall execution time of the system.
- **Multi-Agents** – This category of agents is used when dealing with relatively large tasks or goals. If a task is submitted by the client it can be beneficial for it to be broken up into a number of subtasks which can then be handled by a number of specialised agents. These agents then report back to their supervisory agent, which analyses their outputs and reports the results back.
- **Collaborative Agents** – these agents interact with each other in a similar way to multiple agents, however the concept of an agent's autonomy is weakened. Here agents work together to produce an output. They share knowledge about the situation and work as a team.
- **Social Agents** – (or anthropomorphism) involves the collaboration between humans and agents. Some agents are being developed which can present themselves as human-like creations to improve how humans interact with them [MaesP 97].

3.4 Agents as Critics – Daemons

Now that the various types of agents have been discussed, it is possible to describe the agents I developed within the ***Prompter*** tool. These agents have a number of properties similar to those presented above but also have some that are unique to themselves. Each agent or daemon as they are called in ***Prompter*** is a mini expert which monitors user actions and provides advice on possible alternatives or potential problems that may lie ahead. This advice takes the form of a criticism of the user's work, as in a critiquing system, and a justification for its conclusions.

3.4.1 Agent's Internal Structure

I developed a common architecture for the critiquing daemons (see figure 3.1). Contained within it are many of the components outlined in the general model developed in the previous chapter such as a plan generator, differential analyser, dialogue generator etc.

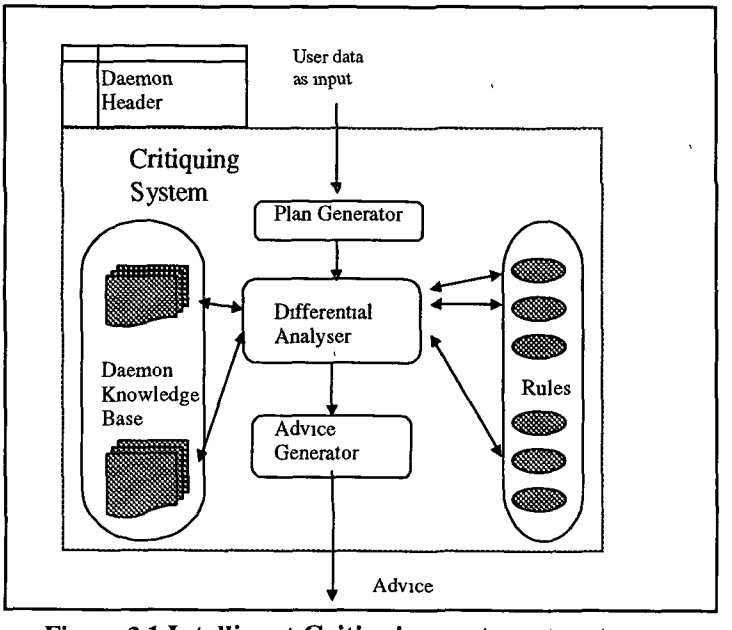

Figure 3.1 Intelligent Critiquing system structure

The identification information relating to a specific agent is stored in a structure called the daemon header. This header identifies the agent to the tool and contains information relating to the area the agent is an expert in, what version number it is and other related information. It also makes the job of sorting, storing and searching for agents much easier.

The core of the agent contains a number of rules that interpret the information given to it by the inference engine. Here it performs the differential analysis on the user's data by comparing it to its own static knowledge base. Its output is then passed to its dialogue generator

This output is passed through a dialog generator which produces the advice for the user. I decided that the best mechanism of encapsulating advice was using HTML as it allows the introduction of bullet points, text manipulation etc. This advice is passed back to an inference engine which is encased within the daemon architecture.

3.4.2 Criticism

This advice allows the user to be alerted to a potential problem within their project. I divided it into a number of sections:

- **Critique:** a dynamic criticism of what the user has done this may be accompanied by advice on how to overcome the problem.
- **Background:** This provides some generic advice that may help the user know why the problem has been highlighted.
- **Justification** How the advice was generated. This tells the user why the advice is given.
- **Bibliography:** A list of reading material that the user may consult if unsure or confused. This is a list of papers, conference proceedings, books etc. that may help the user.

3.4.3 Agent Communication

There is no point in collecting a number of experts into a room if there is no organisation imposed on them. There would be chaos and the advice supplied to the user would be irrelevant, inconsistent and most likely out of date or supplied to the user at the wrong time. There are several motivating factors behind why groups of agents need to be co-ordinated outlined in [Broadcom 97]. Thus I had to impose some form of architecture around the agents. I had a choice of two basic concepts of approaching the design which I outlined below[GeneserethM 94].

- **Direct communication** In direct communication, agents handle their own communication. An agent can communicate directly with another agent. See figure 3.2. The advantage here being, agents can communicate without the assumption of data structures being available in the system itself. It allows a greater separation of the agents from the tool thus making them more autonomous. It does not rely on the existence or capabilities of other programs. The

language KQML [FininT 92] was designed specifically to allow agents to easily communicate and pass information to each other.

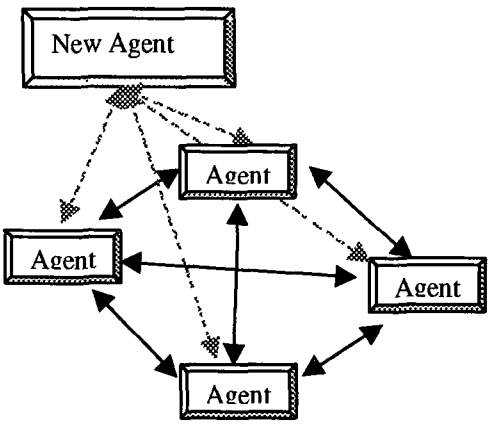

Figure 3.2 Direct Agent Communication

However the drawback with this is that once the number of agents increases, so too does the complexity and the resulting cost [GeneserethM 94]. As in figure 3.2 if a new agent is introduced the number of connections grows by a factor of N (N = number of agents). A requirement of the ***Prompter*** tool was the ability to add to the knowledge base i.e to add new agents. Thus if this architecture was chosen it would result in a complicated integration period.

- **Assisted coordination** This architecture differs from the above in that agents rely on special system programs to achieve coordination. It introduces a more controlled environment over which agents execute. Figure 3.3 shows the representation of four agents, all dealing with different areas of risk. If the Risk Agent has information relevant to other agents, it can broadcast it using a structure called a facilitator. The level of complexity is much less with this architecture compared to the previous one. If a new agent is to be introduced this requires only one new connection.

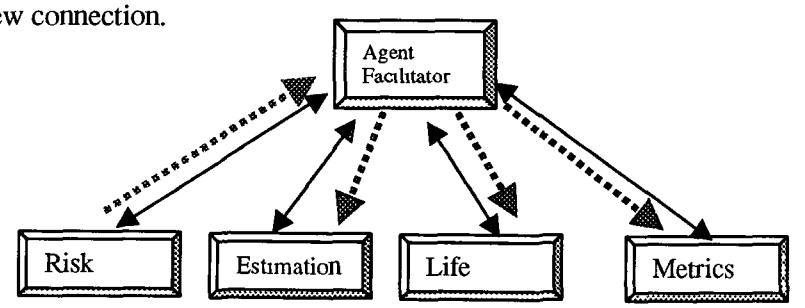

Figure 3.3 Assisted Agent Coordination

In contrast to the competing concept, this architecture allows agents to become more integrated into the system and not as isolated. A facilitator is mechanism for performing this organisation [GeneserethM 94]. The facilitator is the controller of the agents. It forwards information to them and takes advice from them. They surrender their autonomy in some way to this facilitator, which now controls to some extent their execution. This design allows the agents to better express their requirements so the facilitator can be more discriminating in routing messages.

I chose the later of the two architectures for the daemon architecture for the following reasons:

- In general this mechanism is more efficient as the amount of communication is reduced. In direct communication, if the number of agents is large the cost of broadcasting a job is very high.
- it allowed agents to be grouped together into a hierarchical structure, similar to the taxonomy they had to represent. The facilitator itself may be similar to an agent with a facilitator controlling it.
- the knowledge base has to be easily expandable meaning a quick and simple process to add new agents. Since this architecture only needs the introduction of one new connection for a new agent, it was the best option.
- Since ***Prompter*** is a marketable tool other issues such as efficiency and multi-threading had to be also taken into account. The overhead incurred if direct communication was chosen would be higher since the communication and expandability factor would be much greater. This would slow the tool down and also make it to difficult to regulate the flow of advice from agents to the user. Assisted coordination is easier since the facilitator controls all the agents under it. Thus the tool only has to deal with the facilitators and never sees the underlying agents.

Hence all the critiquing agents in the daemon architecture are organised into an **Assisted co-ordination** structure. If the facilitator sees that an agent can execute it informs the relevant inference engine to perform the task. Internally the agent interprets the information, compares it to its knowledge base and constructs some worthwhile advice for the user. The mechanism chosen to simulate a facilitator within the daemon architecture is called a Blackboard similar to that used for Riskman2 (section 2.5.4.2).

3.4.4 Blackboard within the Agent Architecture

As described above the decision taken was to design the architecture using the Assisted co-ordination approach. Thus the agents and facilitators are monitored by a controlling structure called a blackboard which acts as the master facilitator for all the agents.

A blackboard as described in the previous chapter is a complex problem-solving model prescribing the organisation of knowledge, data and problem-solving behaviour within the overall organisation. Put more simply, one might imagine an analogue of a group of experts in a room as in figure 3.4. The only mechanism they have of communicating with each other is through writing what they think on a blackboard. The other experts can read what has been written.

Figure 3.4 The blackboard

Since they can see the current state of the solution they can use it to develop and write their own suggestion. This results in experts working together to provide some solution. However for this structure to be viable some mechanism of control must be enforced.

The blackboard I developed is responsible for ensuring that order is maintained within the agents architecture. It ensures information is kept up to date and the system is informed when advice is received from the agents. Each agent is given its own area of the board to write its advice to as was the case with Riskman 2. However if necessary it can read the outputs from other agents or inform an agents if other dependent agents have produced new outputs or if some new information has been received from the user.

In the daemon architecture, the blackboard acts as a high level method of agent communication It does not merely communicate simple information at a low level from agents to agents, but instead communicates highly detailed advice. The blackboard model thus contains two basic components.

- *The knowledge sources* - The knowledge needed to solve the problem is partitioned into knowledge sources, which are kept separate and independent. In ***Prompters*** case, these are the agents themselves.
- *The BB data structure* - The problem-solving state is kept in a global database. Knowledge sources produce changes to the blackboard, which lead incrementally to a solution to the problem. Communication and interaction among the knowledge sources takes place solely through the blackboard.

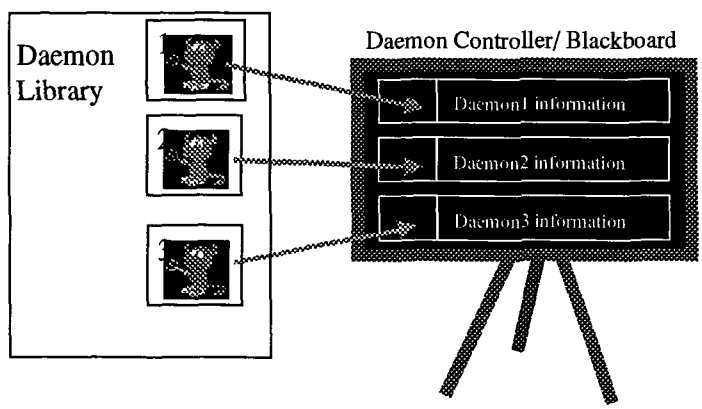

Figure 3.5 Overview of blackboard

The operation of the blackboard systems within the architecture is as follows: knowledge sources (daemons) respond opportunistically to changes in the blackboard. The expert sees some change on the blackboard and responds by performing analysis on the data given. It then writes its conclusions to the blackboard which can be read by other agents.

3.4.5 Knowledge Representation for agents

An agent draws its intelligence from its knowledge base. There are many mechanisms for representing knowledge in an agent, however these mechanisms are, majority of time, specific to the knowledge that must represent.

The knowledge base for the ***Prompter*** tool was constructed by dividing up the main area of software project planning into categories. Within each of these, subcategories were introduced. This led to the taxonomy for the agent knowledge base as illustrated in figure 3.6.

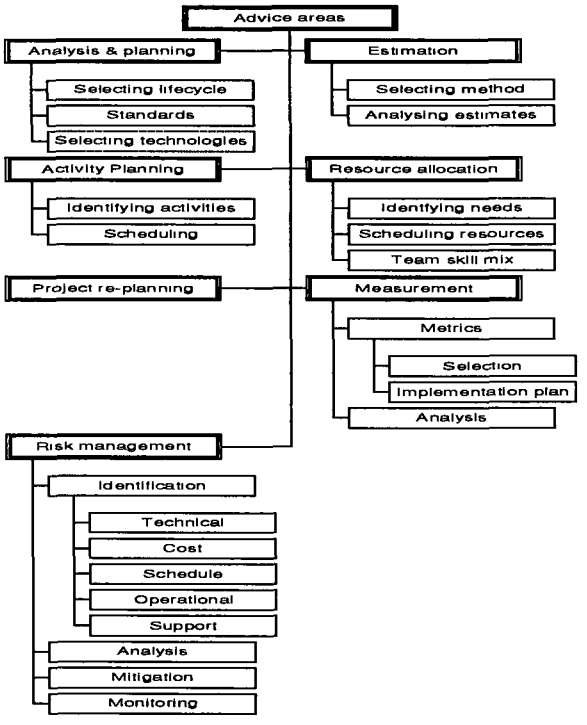

Figure 3.6 *Prompter* Taxonomy

The simplest way of representing this taxonomy in the daemon architecture was by creating an individual daemon for each category. Thus for the area of Risk identification there were a total of five daemons created. Note that it would also have been possible to place all the categories of Risk identification into one agent or on a bigger scale placing all the risk areas in total Risk daemon. However this reduces the level of expandability e.g. if a new agent is produced for the area of support under Risk identification then the whole agent would have to be changed since this area would only be part of the daemon. If each category has its own separate agent, the category can be easily substituted.

3.5 Summary

The objective of this chapter was to introduce the reader to the concept of agents highlighting the type of agents developed for the ***Prompter*** tool.

A description of what an agent is and the various types available was given. The critiquing agents I developed were then explained and how they differ from the general model of an agent. The architecture that was developed to organise them and integrate them into the daemon architecture was then elaborated upon.

Following on from this, the knowledge of the daemon architecture was outlined, highlighting how it was broken down into groupings of agents.

In the following chapter, the daemon architecture is examined at a more detailed level. Each component is explained using object diagrams and a detailed explanation of some of the more interesting protocols is given.

4. Detailed Design

4.1 Introduction

This chapter is concerned with the detailed design of the daemon architecture of ***Prompter***. The Daemon Supervisor, the Blackboard, the Daemon Library the daemons and their knowledge. An outline of the knowledge engineering process that was used to develop these daemons is also given are all explained. Firstly however, some of the standards that were adhered to during the development of the architecture are introduced.

4.2 Standards

The main reason for coding standards is the maintainability of the code. It is summed up in the following quote [AmblerS 97]:

> *"Coding standards for java are important because they lead to greater consistency within your code .... Greater consistency leads to code that is easier to understand, which in turn means it's easier to develop and maintain. This reduces the overall cost of the application."*

Standards make it easier to share code and for others to maintain it. The Java coding standards outlined in [AmblerS 97] were adhered to for the implementation of the daemon architecture of ***Prompter***.

Another standard adhered to during the project was the Object Orientated Analysis (OOA) specification [RumbaughJ 91]. This is a semiformal specification technique for the OO paradigm. There are currently over 40 different techniques for performing OOA, and new techniques are put forward on a regular basis. The most popular techniques are: OMT, UML and Booch's technique. However, most techniques are largely equivalent and consist of three basic steps:

- **Class modelling** - Determine the classes and their attributes. Then determine the interrelationships between the classes. Present this information in the form of a diagram - termed a 'Class Model'.
- **Dynamic modelling** - Determine the actions performed by or to each class or subclass. Present this information in a diagram - termed a 'Dynamic Model'.

- **Functional modelling** - Determine how the various results are computed by the various products. Present this in the form of a diagram - termed a 'Functional Model'.

The choice of which particular method of OOA to employ for a given project is usually arbitrary and linked to the experience or preference of the system designer, or dictated by outside influences. With this in mind, OMT was chosen as the OOA method for the design of the ***Prompter***. Within this thesis all class diagrams, event diagrams and state diagrams conform to this standard. A brief discussion of OMT is given below. However for a good description to the OMT methodology, see [RumbaughJ 91] or [MartinL 96]

4.2.1 Object Model Technique (OMT)

OMT describes the structure of objects and illustrates their identities and relationships to other objects, their attributes and their methods. An outline is given to help the reader better understand the class diagrams in the daemon architecture.

The basic structure used in all OMT diagrams is the class structure. The class structure corresponds to a class in OO languages and is defined as [RumbaughJ 91]:
"A schema, pattern, or template for describing many possible instances of data. A class diagram describes an object class".

An example of a class can be viewed in figure 4.3. Contained within a class are the various attributes that make up the class. Following on from this are the methods or operations which perform tasks or operate on these attributes.

There are numerous mechanisms of linking classes together however only those used in the OMT diagrams to follow, are explained. The most basic link is known as the association link and is illustrated in figure 4.3. An association link, is a line between two classes indicating a relationship between them, the relationship is always displayed beside the line. These links come in a number of formats, the simplest being a line between two classes, indicating the classes are related in some way by a one to one relationship. The next format is a link with a dot at one end. This is known as a one to many relationship and means that one class (the end without the dot) is linked to one or more of the corresponding class (as illustrated in figure 4.3) by the relationship defined.

The next mechanism of creating a relationship between classes is called the Aggregation relationship and takes the appearance of a diamond in the line. It is defined as relating an assembled class to a component class. It is the "part-off" relationship and indicates that one class is associated with an object by making up part of it i.e. the class is part-off the associated class. An example of this can be view in figure 4.7.

4.3 Rationale behind Architecture

Before the architecture is introduced it is important to illustrate some of the reasons/rationale or requirements behind it.

Platform Independence – It was decided the ***Prompter*** tool, including the daemon architecture, was to be a platform independent tool, meaning it had to be capable of running on any type of machine using any type of operation system. Within the architecture there had to be no machine specific protocols or features such as the ability to write directly to COM ports etc. Thus a project wide decision was made to implement the tool using the Java Language (see Chapter 5 for a detailed description). This placed a number of constraints on the design of the architecture. It had to be performed in an Object Oriented fashion to allow ease of implementation in Java which it is itself OO. Also Java's ability to allow only one level of inheritance constrained the class level design further.

Distributed – It was also decided that the ***Prompter*** tool was to be a distributed system. The decision was made to divide the tool into three main components the daemon architecture, the GUI and the kernel. Within the daemon architecture component, I introduced an extra level of distribution at the Daemon Library component. It was also decided that CORBA was to act as the distributed layer to bind all the components together. It utilises interfaces to allow components to communicate, meaning that the design had to incorporate these properties.

Independence – Since each component was to be distributed in the architecture, it meant that each component had to be independent and asynchronous of the others, so structures such as buffers and threading issues had to be considered in the design phase. Also, the introduction of CORBA allowed the system to be divided up into a client-server system (The concept of CORBA and its client-server structure is developed further in Chapter 5). This allowed the design of each component to be more client server oriented as well, thus increasing the independence of each.

Expandability – One of the major concerns within the daemon architecture was the ability to allow the simple integration of new daemons or agents into the knowledge base thus requiring the architecture to be as open as possible. The design had to have a minimum amount of coupling between the architecture and the daemons themselves i.e. I had to ensured that if new daemons were added, their integration into the knowledge taxonomy was not difficult. I also had to provide the facility to allow daemons written in different languages to be added thus allowing the tool to be capable of keeping up with new languages and concepts.

4.4 Architecture

As mentioned in the Chapter 1 (section 1.1.2) the architecture of ***Prompter*** is broken up into three main components as viewed in figure 4.1, with the daemon architecture broken up into four components: the Daemon Supervisor, the Blackboard, the Daemon Library and the Daemons.

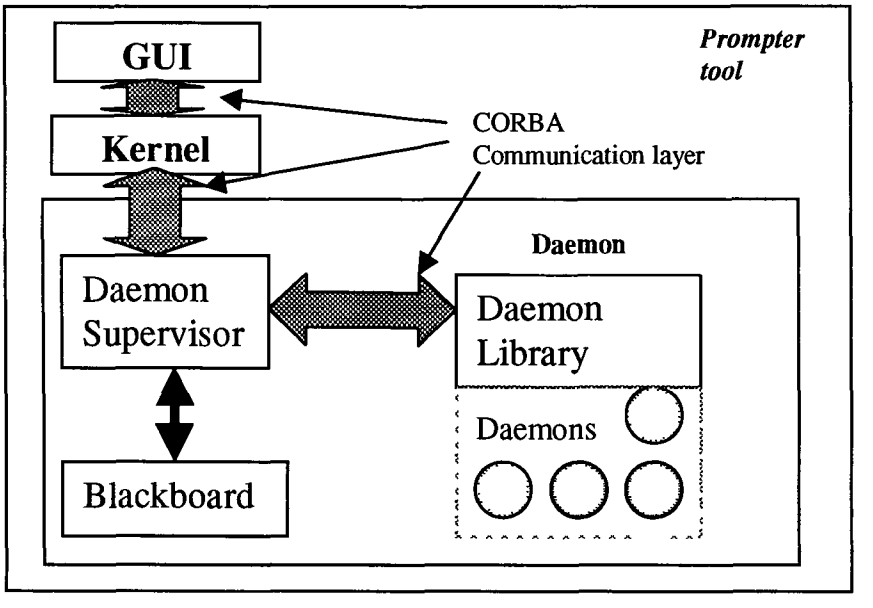

Figure 4.1 Prompter Overview

The daemon supervisor is the main controller of the daemon architecture and is responsible for the passing of information between the daemons and the kernel, and between the daemons and the blackboard. When the blackboard informs the daemon supervisor that a daemon can execute it informs the daemon library to retrieve it from its repository. The daemon supervisor then gives the daemon the information it needs to execute. Finally, the advice the daemon returns is loaded back into the blackboard for later retrieval by the kernel.

As illustrated in figure 4.1 the blackboard interface is not across the communication layer due to the level of communication between itself and the daemon supervisor being so large. Between the other three components there is a communication layer called the CORBA layer (section 5.2). This layer allows the components on both sides to communicate with each other and yet operate asynchronously. For example, the GUI can reside on many machines and still maintain its connections to the kernel and daemons. The CORBA layer automatically takes care of routing all method calls, security issues, passing information across a network etc. In fact the kernel can be completely rewritten in a different language to the other components who would remain unaware, due to each components independence.

4.5 Tokens

The token structure is the method of passing information from the user to the daemons. A token is the base type variable within ***Prompter.*** It stores a value that is set by the user or the system. All information that the user enters is converted into these token values. Each token has a unique name and number which distinguishes it from others, and a description or definition to illustrate its function.

There are two types of tokens within ***Prompter***, qualitative and quantitative. Qualitative tokens have scale values, such as high medium and low, while quantitative tokens have actual values such as 36 or 2.52.

4.6 Daemon Detailed Design

Following on from figure 4.1, the overall OMT class diagram for the daemon architecture is outlined in figure 4.2 highlighting only the class relationship firstly and then each component is explained separately.

As illustrated in figure 4.2 the three main components of the daemon architecture are:

- Daemon Supervisor
- Daemon Library
- Blackboard
- Daemons

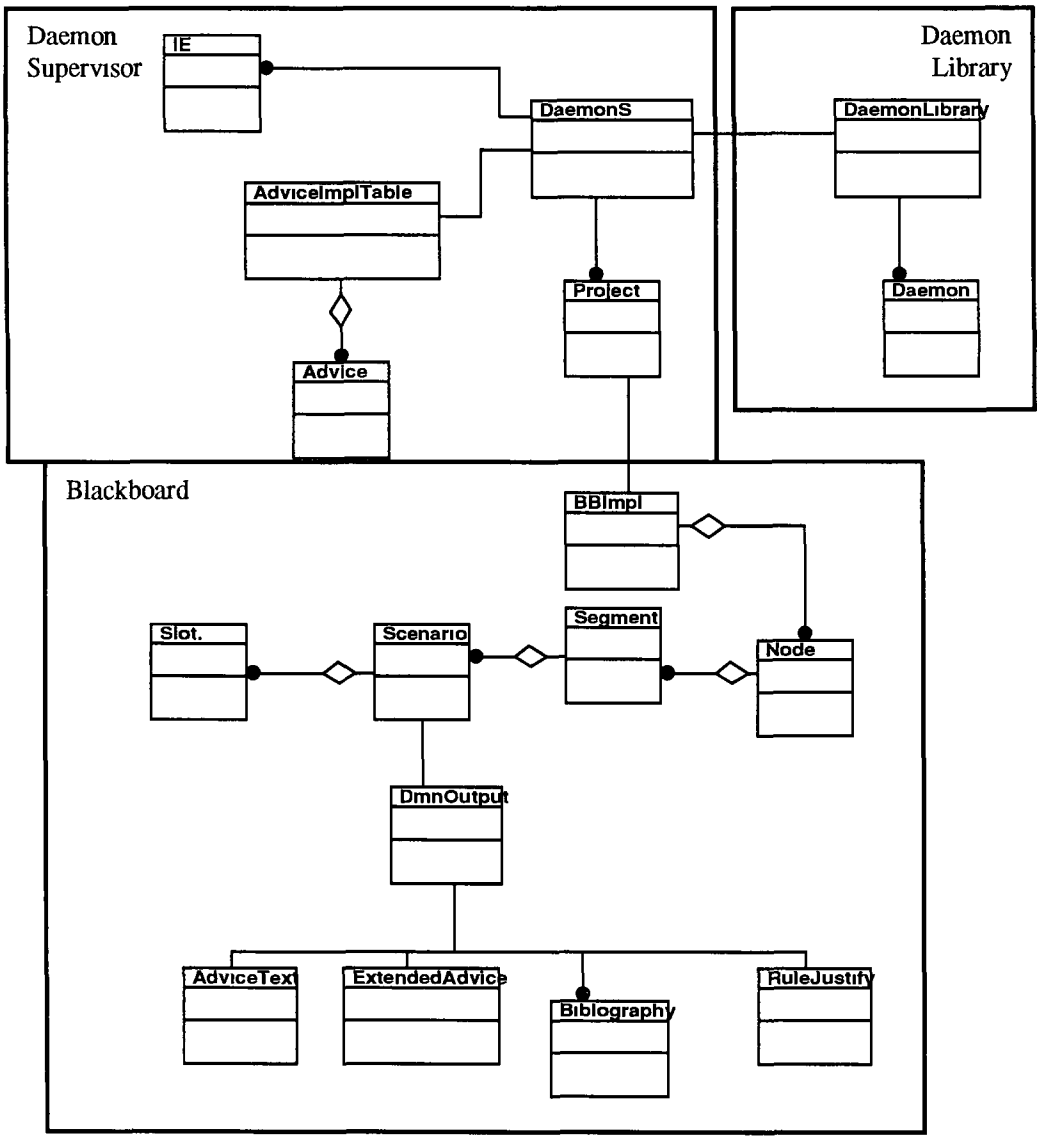

Figure 4.2 Daemon Architecture OMT Diagram

Each of these components aids the daemons in their critiquing process. They act as the interface between the daemons and the rest of the tool. They ensure that the correct advice is returned at the right time and the information passed back is always the most up to date available.

4.6.1 Daemon Supervisor

The daemon supervisor is responsible for all communication between the kernel and the daemon components. It is the main controller of the daemon architecture and is concerned mainly with

scheduling issues, communication issues and of course daemon execution and the handling of the advice returned from them. Most of the tasks of the daemon supervisor are outlined below:

- It is responsible for the creation of new threads of control for daemons available to execute.
- It maintains these daemon threads and times out them after a period.
- It is responsible for the acquisition of information from the kernel relating to information the user may have entered or changed.
- The daemon supervisor is responsible for the creation and maintenance of the advice table. This is a buffer to temporarily store the advice until the kernel is ready to receive it.
- It ensures the advice from the daemons is forwarded to the blackboard and the advice table. The advice is forwarded to the blackboard because it may be used by other daemons at a later date

The class diagram for the daemon supervisor is shown in figure 4.3. The daemon supervisor class contains a number of project classes. Within each of these are the details of a specific software project the user may be working on. Each project is kept completely separate and has a unique blackboard that stores the state of the daemons for that specific project.

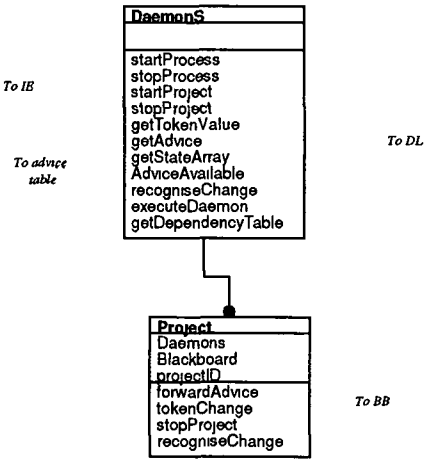

Figure 4.3 Daemon Supervisor OMT Diagram

The reason for abstracting the project information away from the daemon supervisor is to allow related information such as project details to be grouped together and processed separately and independently of other projects. It also allows the maintenance of multiple projects to be handled more efficiently, such as the deletion or creation of an entire project.

The methods contained within the daemon supervisor are concerned primarily with communicating information to the kernel or the advice table. They require little explanation. However the following have more specific roles:

- **void recogniseChange(..);** - This method is the signal for the daemons to start monitoring what the user has done. Up until this invocation the daemon remains inactive while the user is selecting default information. Note that it is not necessary for the daemons to be executing all the time and there may be times when the user wants to make a number of changes before the daemons supply advice. The user may decide to try an experiment and at the end let the daemons analyse the result.

- **void executeDaemon(..);** - This method is responsible for the execution of a daemon and is one of the key methods in this component. The Blackboard tells the daemon supervisor which daemons can execute and it is up to the daemon supervisor to inform the relevant inference engines. However the events that occur on either side of this call can be better understood with the help of the following state diagram.

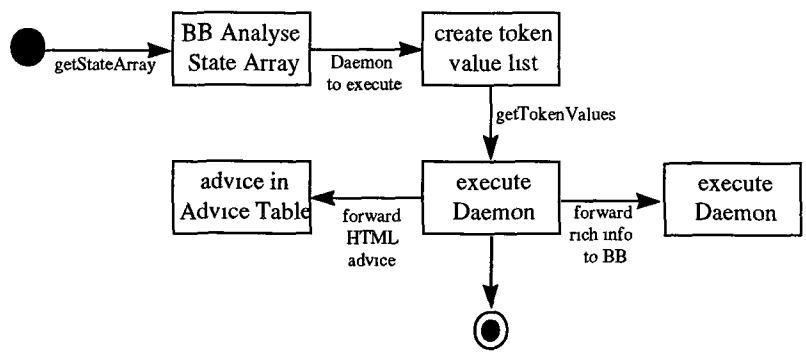

Figure 4.4 State diagram for generation advice

To execute the daemon, the token values for its dependent tokens must be acquired from the Kernel. These values determine which rule within the daemon fires. When the daemon has executed the advice returned is in a very rich format that is of little use to the user but may be of great use to other daemons. For this reason the rich advice is passed on to the blackboard for use by other daemons and a more user friendly version is sent to the advice table for the user.

- **void processAdvice(DmnOutput dOut);** - This method acts as the dialogue generator for the daemon architecture. It is responsible for the conversion of advice from one form to

another As mentioned above the advice must be presented to the user in a readable format so they can understand it better i e HTML The rich advice structure containing all the information returned from the daemon is passed as a parameter This method takes the user-relevant information from each section and places it in a HTML string HTML tags are inserted so the GUI can identify the various pieces of advice for display purposes This process is illustrated in the following state diagram

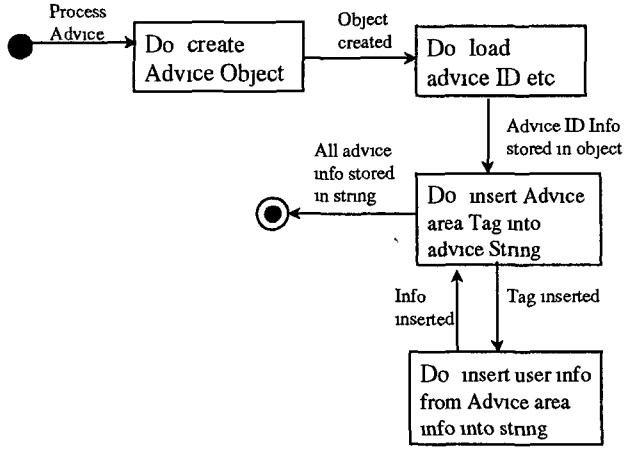

Figure 4.5 Process of advice state diagram

The advice object is created, and the various ID information added in An advice tag is inserted followed by the actual advice When the current section is finished the next tag is inserted and so on until all the advice has been converted into HTML This string is then added to the advice object, which is forwarded to the advice table where the Kernel can retrieve it

4.6.1.1 Daemon Supervisor Interfaces

A number of CORBA interface definition language (IDL) interfaces were created to allow the daemon supervisor to communicate with the Daemon Library and the Kernel across CORBA These interfaces can be viewed in the Appendix B and contain all the methods needed for the modules to communicate with each other

4.6.2 Blackboard

The blackboard is the main data structure in the daemon architecture that monitors the daemons Its purpose is to hold state information about daemons and inform the daemon supervisor when

one can execute The structure of the blackboard is outlined diagrammatically in figure 4 6 and its OMT diagram in figure 4 7

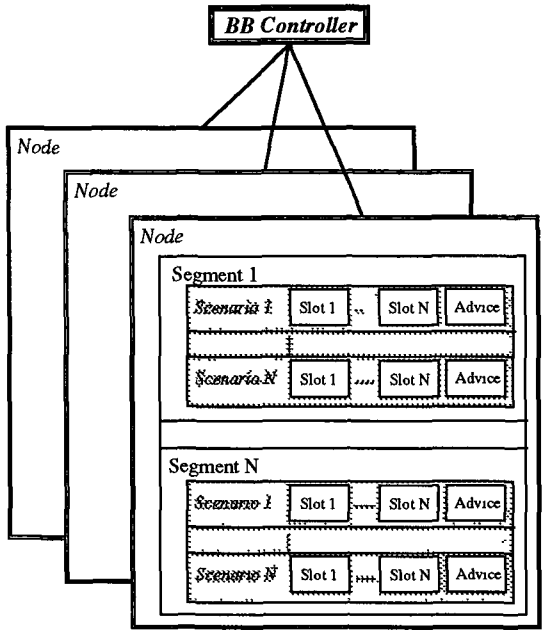

Figure 4.6 Blackboard structure

The blackboard is a tree structure with a number of branches called nodes, which are monitored by the blackboard controller Each node represents one area of the knowledge taxonomy specified in the previous chapter Contained within each node are a number of segments A segment represents a specific daemon of that node Thus if there are a number of daemons dependent on Estimation, this will result in a number of corresponding segments in that node

Within each segment is a structure called a scenario There can also be many scenarios within one segment A scenario can be described as follows, if a user wants to try a number of possible paths but is unsure of which one is correct they create a scenario which allows them to try a possible alternative solution to a problem thus producing an alternative state for a daemon This results in a daemon having a number of different states each one stored within a different scenario So since a segment represents a daemon in general, a scenario represents the daemon in one of the possible paths

A daemon is dependent on a number of tokens that are used to help give advice These tokens can exist in one of two states at any time, either **changed** or **unchanged** Thus within each scenario there is a number of structures called slots, one for each dependent token and a slot for the last

piece of advice the daemon provided The class diagram for the blackboard module is shown in figure 4 7 The advice structure will be discussed further in the section 4 6 4 1

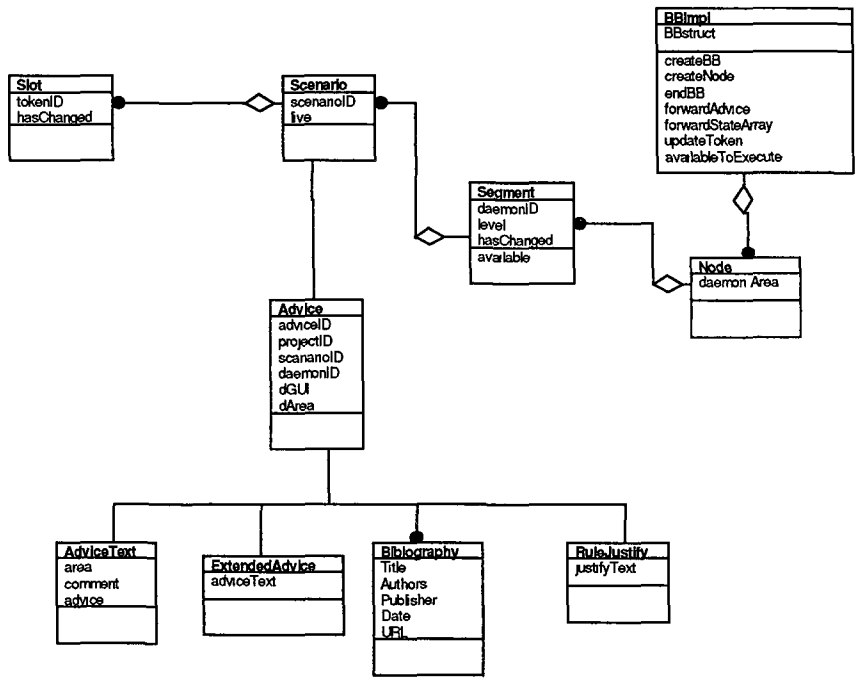

Figure 4.7 Blackboard OMT Diagram

As mentioned above a token can hold one of two states, at any one time

- **unchanged** the token value has not changed since the last execution of the daemon If this is true for all the tokens in the daemon then the daemon has no need to execute
- **changed** the token value has changed If this is the case the daemon's advice is no longer up to date and must execute again

Again a lot of the above methods are related to the updating and the searching for tokens but some of the more important methods are outlined below

- **void createBB(DependencyTable);** - This method is responsible for the creation of the blackboard structure and involves
    - The acquisition of the daemon information from the Dependency table
    - The extraction of the relevant data from this table
    - The construction of the node and segments from this information
    - The addition of these nodes along with an output area to the blackboard

The parameter "DependencyTable" or DT is an array of objects that is received from the Daemon Library and contains all the information about the daemons that the Blackboard needs to construct itself Each object contains the daemon's ID, the daemon's area of expertise and finally a list of all the tokens each daemon depends upon The state diagram for this method is illustrated in figure 4 8

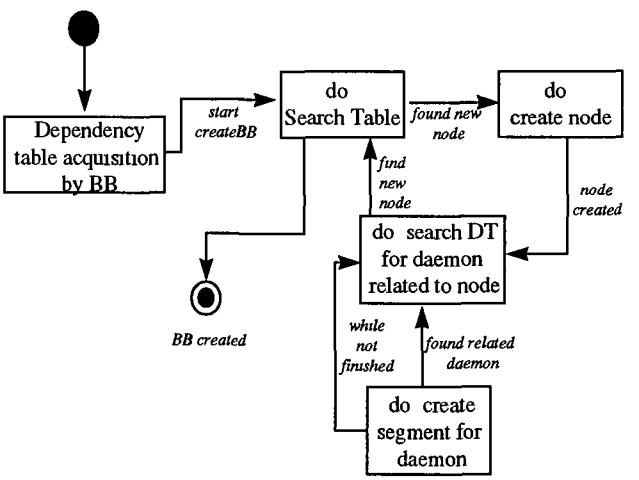

Figure 4.8 State diagram for the construction of Blackboard

When the blackboard has a copy of this "Dependency table", it retrieves the relevant data from it to construct the relevant nodes and segments It first finds an uncreated node and creates one It then searches the "Dependency table" for daemons that are related to it When it finds one, it takes the daemonID from the "Dependency table" and retrieves the array of "tokenIDs" (integers) which it depends on To create the segment the blackboard uses the call "segment createSegment (daemonID, tokenID[])," This in turn creates a default scenario with space for the advice the daemon may produce at a later stage All that remains now is to add this segment to the node When all the daemon's segments that relate to this node are added, the node is added to the blackboard

- **DaemonInfo availableToExecute()**, - This method is responsible for asking all the nodes if they have any daemons that are capable of executing The criteria for a daemon executing is if one or more of its tokens have changed since its last execution and all of its slots are marked available The event diagram for this method call is illustrated in figure 4 9

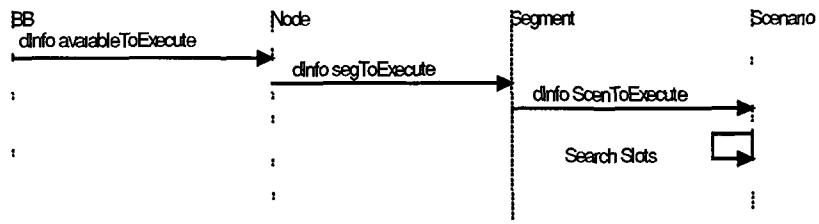

Figure 4.9 Event diagram to illustrate the availabletoexecute

- **void forwardAdvice(DmnAdvice adv);** - This method is called from the daemon supervisor and passes the advice from the daemon, to the blackboard after it executes This allows the blackboard to keep a constant record of the last advice generated by the daemon This can be used to provide advice to the user without having to re-execute the daemon, or to other daemons during their execution The method takes the advice and searches the blackboard for the segment it belongs to and stores it within the correct scenario

4.6.3 Daemon Library

The daemon library is responsible for the physical management of the daemons themselves It controls the physical storage of the daemons and their acquisition when available to execute Since this is not that critical to the daemons themselves it is not discussed in detail

The daemon library is a separate component conceptually from the rest of the architecture The reason for this is the distributed nature of the tool Since it may be required to store all the daemons on some machine separate to the tool itself, the CORBA layer was introduced Thus if a user is mobile they can still execute their specific daemons if they have a network link to the daemon library This also makes the integration of new daemons, much easier Another advantage of this link is even though there is only one set of daemons in the daemon library, many users can access them

Broadly speaking the functions of daemon library are

- When ***Prompter*** is started the daemon library supplies dependent token information (dependency table) to the daemon supervisor when requested (as part of the blackboard construction process) The dependency table as mentioned above contains all the relevant information required for the blackboard to correctly construct itself
- The daemon supervisor instructs the daemon library to extract a named daemon from the library for execution when its dependent tokens are available

4.6.4 Daemon Design

An OMT diagram for a daemon is illustrated in figure 4 10 and a conceptual view is illustrated in figure 4 11

Figure 4.10 Daemon OMT diagram

To better explain the structure in figure 4 10, a daemon can be thought of consisting of three main structures

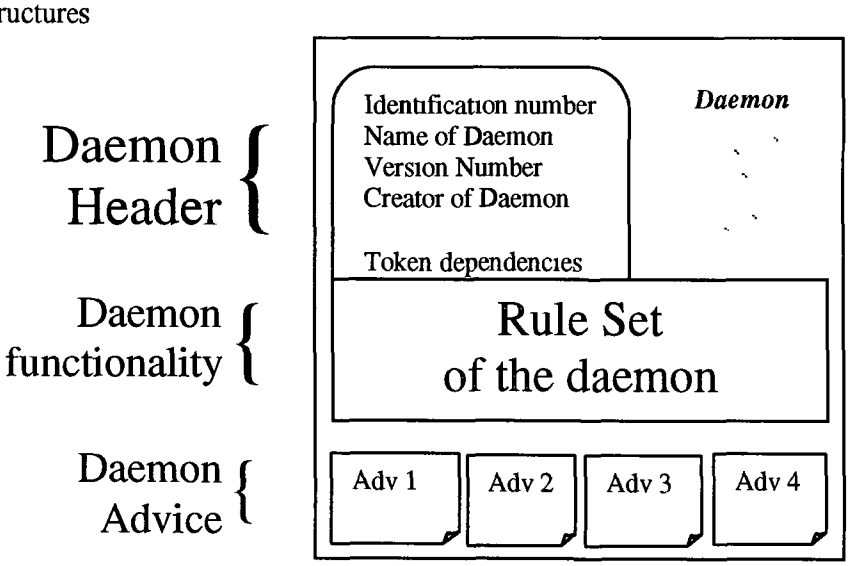

Figure 4.11 Daemon Internal structure

The **Daemon Header** contains information relating to the unique identifier of the daemon, its version number and who wrote it Also contained within this section is the list of dependent tokens (the tokens the daemon needs information on to supply advice), its area of expertise and what inference engine it requires to execute

The **Daemon Functionality** is the specific rules of the daemon It is from within this section the daemon knowledge is stored Obviously this section differs from daemon to daemon Some have a simple knowledge base which may be represented as IF THEN statements while others need more complicated concepts such as frames or fuzzy logic to achieve their goals It is this area of

the daemon that changes depending upon the implementational language chosen However as outlined in section 4 7, the daemon is controlled by its respective inference engine which handles its execution and so the overall daemon architecture can remain oblivious to the daemons functionality The design I employed allows for the future expansion of the daemon library with new daemons and daemon languages Agent languages are discussed further in section 4 7

The **Daemon Advice** is the third component Within the daemon the advice is divided up into a number of structures The rule set dictates which advice object is returned The advice when returned to the inference engine is broken up into various components and forwarded to the daemon supervisor This advice structure is now outlined in more detail

4 6.4.1 Daemon advice

There are two distinct views of advice within the daemon architecture because the advice itself servers two purposes The advice to the user only contains information that is relevant to the user, however the advice that is returned from the daemons contains much more Such as the token values that caused certain rules to fire etc This information is not relevant to the user but may be of some use to other daemons

Kernel/GUI view - Both the kernel and GUI, and therefore the user will view the advice as one large page of text/graphics which is presented in HTML format This advice is supplied to the kernel in the form of an advice object as m figure 4 12

Figure 4.12 Advice OMT diagram

This class (figure 4 12) contains all the identification material needed for the tool to identify what the advice is related to and where it came from The actual advice is stored in the advice field as a string Contained within this is the HTML tags that help the GUI differentiate between the components of advice

Daemon view - This alternate view of the advice for the daemons The data is in a more detailed format The advice is structured as a hierarchy of classes, which reflect the individual components that make up the advice This more detailed view of advice provides a more flexible means of presenting daemon outputs and allows for easy enhancement of the daemons over time The class diagram in figure 4 13 illustrates the structure followed by an explanation of the various categories

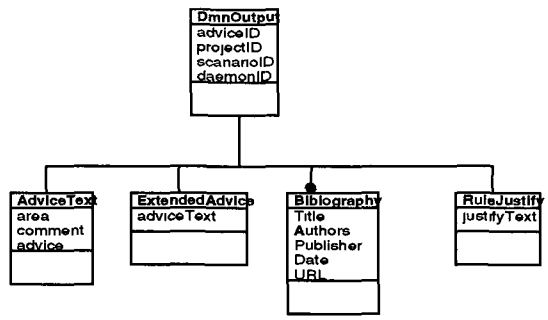

Figure 4.13 Daemon output OMT diagram

1 **Advice text** - this is the actual dynamic advice text and is made up of the following

- **Area of expertise** - What is the area of competency of the expert giving you advice, for example, 'cost risk management' or 'lifecycle selection'
- **Comment** - The experts comment on your project plan For example, 'A high degree of risk to the cost of the project has been identified'
- **Advice** - A short paragraph of advice on how to handle the situation identified above

2 **Extended advice text** - A comprehensive explanation of the area in which the expert is giving you advice and a fuller explanation of the mechanisms you can employ to address concerns The rationale behind having this section is to provide the user - in particular an inexperienced project manager - with more in-depth information

3 **Bibliography** - This is a suggested reading list for the user in respect to the area under consideration This points to such things as the related sections in the project handbook, published works and internet resources

4 **Justification** - How the daemon arrived at its conclusions (in rule terms), i e what rule executed for it to give its advice

4.6.4.2 Daemon Execution

The following diagram (figure 4 14) shows the various steps involved in a daemon's execution

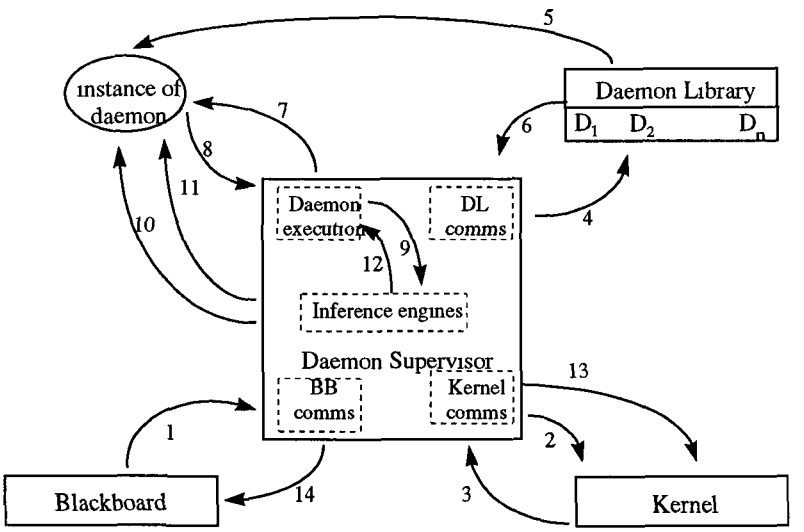

Figure 4.14 Overview of a daemon's execution

1 Blackboard tells the daemon supervisor blackboard_comms which daemon is ready to run by the method **daemonID[] availableToExecute()**

2 The daemon supervisor Kernel_comms asks the Kernel for the actual values for the tokens the executable daemon needs by passing it the tokenIDs using the function **TokenValue[] getTokenValue(tokenID[])**

3 The required values are returned by the kernel in the above function

4 The daemon supervisor DL_comms requests the extraction from the DL of the required daemon by the function call **daemonRef getDaemon(daemonID)**

5 The DL instantiates the requested daemon

6 The DL then returns a reference to the daemon as the return value in function 4

7 The daemon supervisor Daemon_execution interrogates the instantiated daemon to find out which inference engine (IE) it needs, by using the function **int getIEType()**

8 The type of IE is returned

9 The daemon supervisor Daemon_execution then passes the daemon reference, tokenIDs and their values to the IE for evaluation of the production rules using the function **advice processRule(daemonRef, tokenID[], tokenValues[])**

10 The IE processes the daemon's rules

11 The IE then closes the daemon

12 The IE passes the advice to the daemon supervisor Daemon_execution using the return value of function 9

13 The daemon supervisor then builds the advice object and passes it to the kernel

14 The daemon supervisor passes the rich advice to the blackboard

4.7 Agent Languages

There are many AI languages currently available to represent agents which have their good and bad points There are currently efforts being made to construct an agent standard know as FIPA97 [FIPA 97] The question however arose which languages and standards were suitable for agent implementation within ***Prompter*** and whether a tool for creating them should be used

There have been many attempts at producing tools specifically for designing agents However most of these have achieved only modest success in the market place as most of these tools were concerned with mobile agents on the Internet, IBM's Aglets was one of the main contenders It was focused on building network based applications that used mobile agents [LangeD 96] to perform tasks on some network However due to its complexity and the ability to create "Intelligent" agents it only received modest success Telescript [WhiteJ 96] produced by General Magic was another attempt, again for mobile agents and it was also unsuccessful due to it being too cumbersome OMG have now become involved and are, at present, trying to develop an agent standard in the form of a CORBA facility This is known as the FIPA97 standard Since this is work to try to standardise agents it is discussed further in the following section

As a result of surveying the above languages it became evident that there was no clear language that stood out I decided the daemon architecture should not be dependent on one language but instead be open to many Different languages can be easily inserted into the architecture e g Lisp or Prolog This concept is illustrated in figure 4 15

These agents are controlled by their respective inference engines which in turn are controlled by the Agent Controller, otherwise known as the master facilitator introduced in the previous chapter If agents wish to communicate with each other or with agents implemented in another language, they do so using the blackboard structure described previously

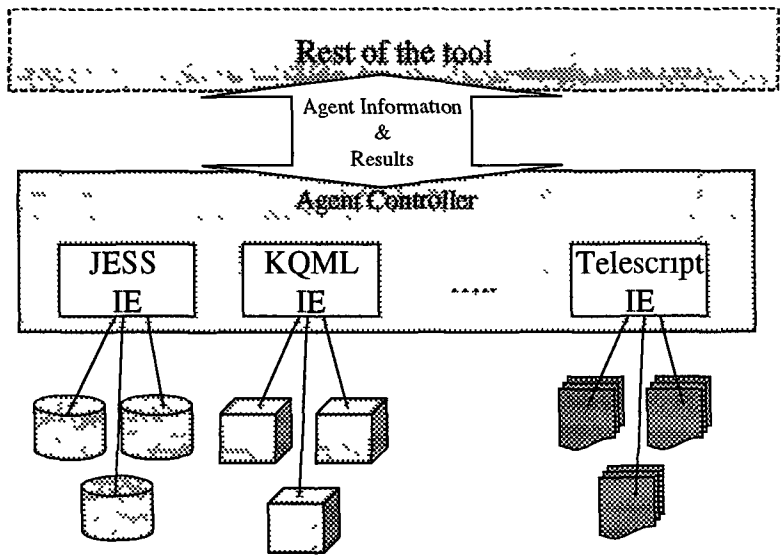

Figure 4.15 Agent integration into Prompter

During the implementation stage of this architecture an agent languages had to be chosen as the default agent language To make this decision a number of factors had to be taken into account

- the tool was written in Java Thus a language that was capable of interfacing with it was required
- the issues of commercial licences had to be considered
- Efficiency Agents needed to be executed efficiently and be capable of returning their advice back to the tool
- The agent language chosen had to be well known so users could implement their own daemons if the case arose

4 7.1 FIPA97

FIPA97 is a standard that is being developed by the FIPA (Foundation for Intelligent Physical Agents) organisation FIPA itself is an international association of companies, which agreed to share efforts to produce specifications of generic agent technologies They stated that specifications must be produced before industries make commitments to their own methodologies, thus avoiding incompatibility In other words this specification is designed to avoid conflicts between agents and to ensure the interoperability between heterogeneous agents

Some of the main goals of FIPA are to achieve standardisation in the following areas [FIPA 97]

- Agent Management
- Agent/Agent Interaction
- Agent/ Software Interaction

Their current Developers Guide [FIPA 97] states that its main goal is not to state information on specific implementation issues such as "How do we implement FIPA compliant agents in language xxx?" But instead to act as a guidance for people implementing FIPA compliant platforms

On the more implementational side of the specification, one of the main trends is the use of CORBA as its recommended transfer protocol, in particular IIOP However since this in general refers to agents travelling across intranets or internets it is not highly relevant to the daemon architecture Also, this specification was still tackling the problem of the internal structure of agents and what language they should be implemented in at the time of writing this thesis In general they suggest ACL (Agent Communication Language) as their communication language

For these reasons it was decided that the FIPA was not of high relevance to my work since the issue of communication was solved using a blackboard, thus requiring no need for ACL or CORBA to be used

4.7.2 CLIPS/JESS

JESS (Java Expert System Shell) [FriedmanE 97] is a Java derivative or clone of the popular expert system shell CLIPS (C Language Integrated Production System) [Giarratano 84] designed by researchers at NASA As it is written entirely in Java makes it compatible with the daemon architecture CLIPS is a well known expert system language used in many systems and as a result has a wide programmer base So although JESS itself was still in development at the time of writing this thesis, it is based on a language that has proved itself

The language itself is similar to LISP, another AI language It is a rule based expert system shell meaning that its purpose is to continuously apply a set of if-then statements or rules to a set of data or fact list These rules are contained in the daemon and the Java interpreter supplies the information for them to analyse

JESS production rules consist of conditional statements known as production rules and a working memory Contained in these production rules are one or more conditions, which lead to one or more actions The JESS runtime cycles through the working memory trying to match conditions on the LHS with data If it finds a rule that has enough information to fire then it places those production rules and the conditions into a conflict set and executes the relevant actions which may change this conflict set causing more rules to fire This cycle continues until there are no more conflicts left

It may seem less efficient to program if-then statements in CLIPS rather than some simple custom-built interpreter However if there were a large number of rules in a daemon it would result in unnecessary cycling by the IE to find rules that can fire JESS overcomes this problem by using the Rete algorithm [Giarratano 84] This algorithm cycles through the rules and remembers past test results across iterations of the rule loop Only new facts are tested and in addition these facts are tested against only the rule LHS's to which they are most likely to be relevant This results is a reduction of the computational complexity per iteration to O(sqrt(RP)), down from O(RPF) where R is the number of rules, P is the average number of patterns per rule and F is the number of facts on the fact list

The structure of a simple daemon can be viewed as follows

```
, The header of the daemon can be represented in JESS as a template
(deftemplate daemon0 "Expert in process Selection"
        (slot  )            , relevant daemon information
        (slot version)      , daemon version
        (slot _31)          , UserProficiency token
        (slot _32)          , CustomerAccesibility
        (slot _33)          , ApplicationType
        (slot _34)          , DevelopmentOrganisation
)                                                                    \

, rule to provide advice can be easily encoded into a simple if-then condition
(defrule process_selection "rule taken from FCPR Vs of daemon 0"
        (daemon0 (_31 ?tk1)(_32 ?tk2)(_33 ?tk3)(_34 ?tk4))      , using the daemon template
        =>                                                       , give each token a value
        (if (and (= ?tk1 1) (= ?tk2 1)(= ?tk3 1)(= ?tk4 1))
          then (printout t " use iterative model" crlf))
        (if (and (= ?tk1 1) (= ?tk2 2)(= ?tk3 2)(= ?tk4 2))
          then (prmtout t " Prototype for HCI, incremental development" crlf))
        (if (and (= ?tk1 2) (= ?tk2 3)(= ?tk3 3)(= ?tk4 3))
```

```
        then (printout t " V or M model" crlf))
      (if (and (= ?tk1 3) (= ?tk2 4)(= ?tk3 1)(= ?tk4 1))
        then (printout t " Prototype" crlf))
      (if (and (= ?tk1 4) (= ?tk2 5)(= ?tk3 4)(= ?tk4 4))
        then (printout t " Spiral model" crlf))
)
```

The daemon0 deftemplate is the template name for the daemon which tells the JESS runtime what tokens the daemon needs to execute Each of these tokens is stored in a slot and each slot is given a variable to store the token value The RHS of the rule condition is now checked The token values are checked and if one of the *if* conditions is satisfied the output is piped back to the inference engine from where it executed from This integration between the JESS environment and daemon architecture is now described below

4.7 2.1 JESS Integration

As described in the previous chapter, a set of agents implemented in a common language are controlled by an inference engine specific to that language This engine handles all communication between the daemon supervisor and the daemons Thus to integrate an inference engine a wrapper class must be constructed to forward and receive information As JESS is implemented in Java, the construction of this class was relatively simple

In general there were three mechanisms of integrating JESS into the architecture, High Medium and Low coupling With High coupling, a Java object can be created from within JESS and/or have their attributes monitored by it However this level of sophistication was not necessary since all that was required was the passing of the relative token values to the JESS daemon and allowing them to execute on their own For this reason the low coupling option was chosen At this level, only information is passed to the JESS daemon, which is allowed to execute until completion within the inference engine The output is then piped back to the awaiting method call where it is organised and stored within a DmnOutput object The class diagram for this wrapper is illustrated in figure 4 16

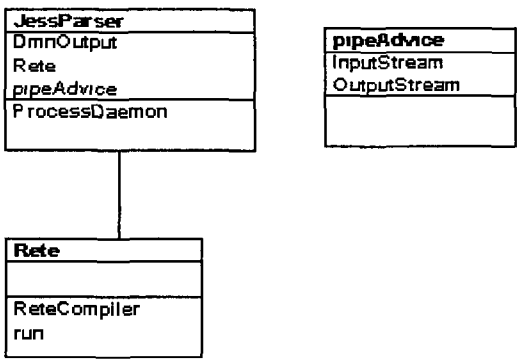

Figure 4.16 Agent Integration OMT diagrams

The Rete class is the main JESS class used to run and compile JESS code The JessParser class is the constructed JESS wrapper that is bound to the Rete engine When the JESS environment has executed the daemon its output is returned to the tool using the pipeAdvice class also shown above This class has an input and output attribute through which information is passed and returned from the JESS inference engine

The set-up and execution of the JESS daemons is explained further in this method below

- **DmnOutput ProcessRule(Daemon dnm, int [] tokenIDs, int [] tokenVals);** - This method is responsible for preparing an inference engine to execute a specific daemon It receives the relevant information related to it such as the tokenIDs and the token values There is a different processRule call for each inference engine and its implementation is situated within the wrapper For JESS, when the inference engine is ready, it constructs a JESS command to load the tokens and the respective values This command is them submitted to the IE which executes the daemon When this is completed the output i e the advice is piped back into a DmnOutput class and returned to the daemon supervisor

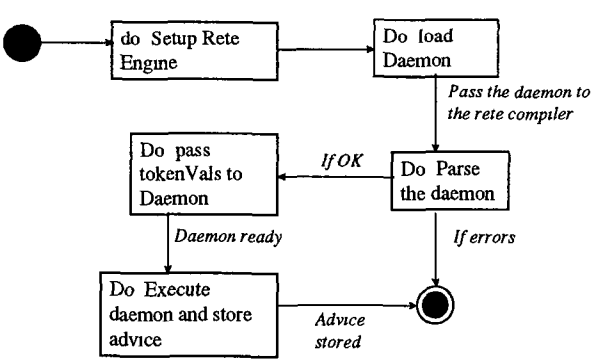

Figure 4.17 Preparing the daemon for execution

This above process (as also outlined in figure 4 17) of executing daemons is just as simple for other languages that are incorporated into the tool However if there is no functionality provided for this integration, a wrapper must be constructed There is currently work on Lisp [AIST 98] and Prolog wrappers for integration into Java software and so the languages available for the daemons are in no way limited The only requirement for new daemons is that they conform to the interface for the daemon supervisor's method call (processRule), name and parameter structure for launching the daemons

4.8 Knowledge

In this section I describe the daemon knowledge base and the engineering process I used to construct daemons The term 'knowledge base' is used to refer to the expertise within the ***Prompter*** daemons, which are represented in terms of JESS production rules

4.8.1 The Knowledge Engineering Process

The knowledge engineering process [SmithP 86] is a process by which knowledge is collected in such a way that it is implemented correctly within agents or daemons The process is generally divided up into a number of sections

- Knowledge Identification
- Knowledge Extraction
- Knowledge Validation
- Knowledge Representation
- Knowledge Verification

A brief description of the above process is now given

- **Knowledge identification** - This is the process of identifying actual advice/knowledge sources in the software engineering world For example, this process could include identifying studies on subjects like 'lifecycle selection methods' or 'risk analysis measures' which contain information such as a method of identifying a situation and the actions to taken as a result
- **Knowledge extraction** – When appropriate knowledge sources have been identified they must be documented in a suitable form which can be translated into actual daemons

- **Knowledge validation** - Validation of the knowledge base has been identified as an important step in the construction of the daemons. The purpose of this validation process is to ensure that the knowledge used and the advice given is technically accurate and appropriate for a given situation.
- **Knowledge representation** – This is the mechanism for translating the knowledge into actual rules or some other form of mathematical structure.
- **Knowledge verification** - This is the process of verifying whether the actual knowledge representation accurately reflects the documented knowledge source. For example, in a rule-based system, the concerns include rules that have correct condition parts and when executed give appropriate advice. Other issues include circular or redundant rules.

The area that is most relevant here is Knowledge Representation, or how to get from knowledge into rules.

Within the ***Prompter*** project a number of software experts within the project team were approached and asked to construct a number of documents [Prompter 98] related to their area of expertise within SPM. They collected their data from sources such as case studies, research, and their own experience working in the area. These documents contained all the knowledge the daemons need to perform their task and sources where more information can be acquired. An example of one of these can be found at [Prompter 98].

The documents were then inputted into the knowledge validation process and further developed to produce a number of comprehensive design and implementation documents for the daemons. These documents can also be used in the future maintenance of daemons. Within the knowledge validation stage, all the documents were circulated to experts external to the project, whose opinions were sought. Once these documents had been fully agreed upon they were ready for the next phase, Knowledge Representation.

4.8.2 Knowledge Representation

This is the process of translating the extracted knowledge in the Daemon Knowledge Base Design documents into daemon rules or into a more mathematical structure other than text alone. There are several mechanisms for performing this task that vary in sophistication and complexity. One mechanism is through the use of decision trees. I chose these because the people who had to construct the knowledge may not have had much experience in knowledge representation and so

the mechanism had to be simple to develop and understand It also had to eliminate any ambiguity in the knowledge For this reason, I decided that knowledge should be presented in a more structured and mathematical format

4.8.2.1 Decision Trees

Decision trees are derived from simple predicate logic and are in general easy to construct and understand They are structures which map out all possible routes that can be taken through the knowledge space They show graphically the relationships of the problem and can deal with more complex situations in a compact form

A decision tree is composed of nodes representing goals, and links representing decisions These links either join nodes to nodes or nodes to end nodes These end nodes store the advice for the path followed If a node has more than one link exiting it means there is more than one possible route that can be taken The deciding factor on which route is taken is dependent on which node can be satisfied If neither is satisfied then no advice is given If one is satisfied the other route is forgotten about

Thus for constructing these types of trees we only need to use two forms of logical operators In predicate logic all logical rules can be reduced to a number of AND and OR operators These can be easily represented in the decision tree structure in figure 4 18

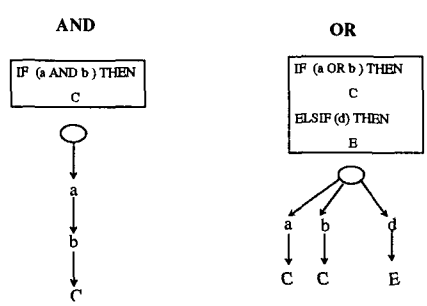

Figure 4.18 AND and OR trees

The AND operator consists of adding the test condition to the current path Thus to get to the node C the conditions a and b must be satisfied In the case of the OR operator to get to C, a or b must be satisfied, but to get advice E, d must be satisfied. These diagrams must be deterministic at all times i e there should be only one possible path that can be taken at any one time A sample daemon decision tree can be constructed from one of the test daemons used in "Cavan" (one of the early prototypes of the tool). The daemon file firstly is as follows

Daemon zero

Version 1 0a

Written by ROC as demo for Christmas prototype

Expert in process selection

31,32,33,34

IF 31 = 1 AND 32 = 1 AND 33 = 1 AND 34 = 1 THEN The suggested lifecycle for your project is to use Iterative prototypes

IF 31 = 1 AND 32 = 2 AND 33 = 2 AND 34 = 2 THEN It is suggested that you use Prototyping for systems development

IF 31 = 2 AND 32 = 3 AND 33 = 3 AND 34 = 3 THEN The suggested lifecycle for your project is to use the V model

IF 31 = 3 AND 32 = 4 AND 33 = 1 AND 34 = 1 THEN It is suggested that you use Prototyping for systems development

IF 31 = 4 AND 32 = 5 AND 33 = 4 AND 34 = 4 THEN It is suggested that you use the Spiral model

The decision tree that would have lead to the above daemon would appears in figure 4 19

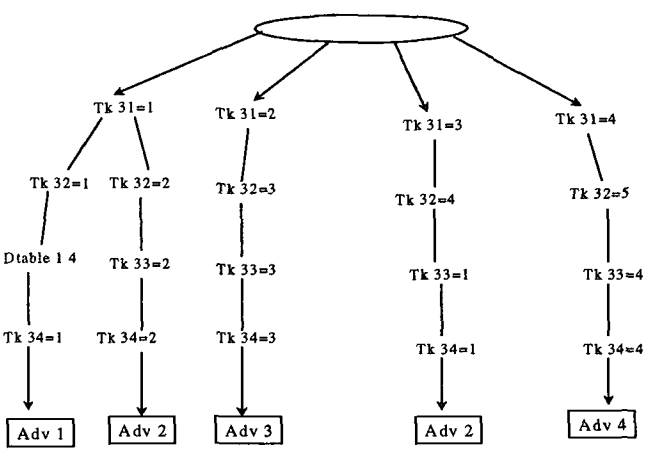

Figure 4.19 Example decision tree

where the test condition in each case is "tk X = value" As can be viewed from the diagram, patterns can be noted in the tree such as tk 31's value being similar in two IF statements Each path is terminated in an end node, which contains the advice ID The advice to be given is then specified in the following tables

Advice Index	Advice Text
Adv 1	The suggested Life cycle for your project is to use Iterative prototypes
Adv 2	It is suggested that you use prototyping for system development
Adv 3	The suggested life cycle for your project is to use the V model
Adv 4	It is suggested that you use the spiral model

Table 4.1 Table outlining the various advice objects

If the decision tree becomes overly complicated there are a number of mechanisms that can be utilised to simplify it

- A node can contain more than one piece of information If a number of tokens in an AND statement all have the same value they can be specified in the same node e g from the decision tree the path that leads to adv 1 - tokens 31 and 32 can be combined into the same node i e tk 31,32 = 1
- Branches of a tree do not always have to be divergent, they can converge as well if the knowledge allows Thus there can be only one advice box for Adv 2 in the decision tree with two possible paths to it
- If the tree becomes large, certain sections can be separated. They are drawn separately

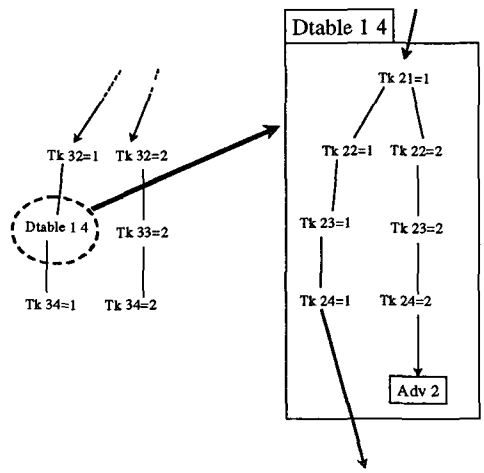

Figure 4.20 Decision tree simplification

- and given a name such as Dtable 1 4 This sub tree can then be included in the full tree by placing its ID as a node This is similar to the method call As can be viewed from the diagram, since the table has only one input and one output, this should be consistent with its location in the overall tree

4.8.3 New Daemons

The field of SPM is by no means stagnant and for this reason there must exist the ability to expand and in some way alter the knowledge of the daemons As a result the daemons are kept as independent of the tool as possible It is, as described previously, rather simple to incorporate a new daemon into the tool, especially if the inference engine is already available

To add a new daemon to the architecture the following must be done It must be ensured that the daemon conforms to the high level structure of a daemon specified above The file that contains the daemon must be added to the daemon directory and the tool restarted This will result in the

daemon reading the daemon headers as usual including the new addition, and the blackboard being constructed automatically with the new daemon information Once the daemon is inserted into the correct directory the rest is automatic However some care must be taken to ensure the knowledge it contains is consistent with that already available

4.9 Summary

In conclusion, the goal of this chapter was to outline the detailed design of the daemon architecture

It began with a brief discussion on some of the high level decisions that were made before the development of the tool such as standards which included coding standards, design standards, and documentation standards Following on from this a high level OMT diagram of the architecture was given, with each component broken down and explained in detail

The daemons and their advice were the last component described in this chapter A brief discussion concerning the knowledge of the daemons was outlined It described the various protocols that were followed to ensure the knowledge was correct and a suggested mechanism for bridging the gap between documented knowledge and rule based knowledge was given

The following chapter discusses some of the implementation issues that arose during the development of this architecture into a working prototype

5. Implementation of a prototype

The goal of this chapter is to explain some of the implementation issues that were encountered during the development of a prototype of the daemon architecture

5.1 Introduction

This chapter is primarily concerned with the implementation stage of the daemon architecture and some of the issues I encountered during this phase A brief discussion is given relating to some of the requirements of the architecture such as what exactly mobile code is, what platform independence is and why these are necessary at all This leads on to a discussion concerning some of the problems that were encountered in the implementation stage using the languages chosen and the solutions used to overcome them

A brief discussion of the implementation languages is given so the above mention problems can be better understood Iona's implementation of the CORBA standard was the distributed language chosen while the main implementation language was Java I also outline a number of reasons for their inclusion and the advantages and disadvantages they brought Finally an outline of the development strategy is given detailing how the architecture evolved from the design to a working prototype, and the problems that were encountered

5.2 CORBA

I chose Iona's implementation of CORBA, OrbixWeb3 [Orbix 97] as the main communication language for the ***Prompter*** tool The Common Object Request Broker Architecture (CORBA), is the Object Management Group's standardised specification [OMG 98] for interoperability among the rapidly increasing number of hardware and software products available today CORBA allows applications to communicate with each other independently of their location or designer

The CORBA bus allows transparent access to distributed objects over a heterogeneous network of machines and operating systems - distributed meaning the various objects can be hosted across many computers and heterogeneous meaning many languages and many operating systems can be used, yet all operating together transparently The client remains completely independent of the server It may be written in Java and the server written in C++ and neither would be aware of the difference. Also the client can be completely unaware of the server's location This makes

network programming much easier as it allows you to create distributed applications that interact as though they were implemented for one machine[Orbix 97] Figure 5 1 shows a high level diagram of the CORBA structure

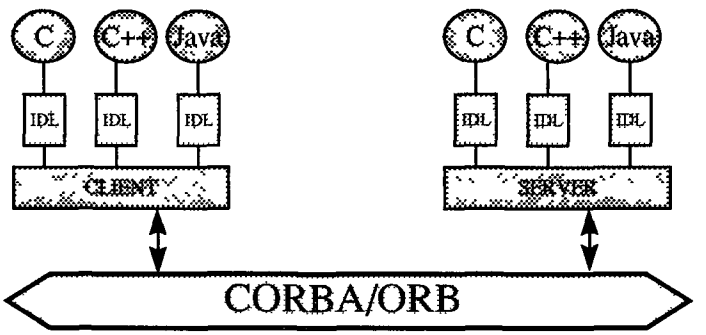

Figure 5.1 CORBA overview

As can be viewed from this diagram, CORBA is a client/server mechanism, whose communication is handled by an underlying communication layer known as an ORB CORBA distributes any messages between the client and the server via its Object Request Broker (ORB) The ORB receives requests from a 'client' to send a message to an object The broker locates the object referred to by the client and delivers the message to it The ORB smoothes over the system differences between the individual components of an application It can be described as an Object Bus, a software equivalent to the computer hardware bus [WeissM 96]

5.2.1 IDL Interface

In order to enable the client and server to operate with such transparency, CORBA must maintain some mechanism of linking them i e through interfaces (see figure 5 2)

Figure 5.2 Interfaces

These interfaces are defined using CORBA's Interface Definition Language or IDL This language allows classes, their respective attributes and methods and any other information to be defined in some common syntax

```
interface grid {
    readonly attribute short height, // height of the grid
    readonly attribute short width,  // width of the grid
    // IDL operations
```

```
        void set(in short n, in short m, in long value), // set the element [n,m] of the grid, to value
        long get(in short n, in short m),        // return element [n,m] of the grid
};
```

In the above example taken from the CORBA documentation [Orbix 97], the interface contains two attributes and the respective method calls that can be made on them The Server and Client have the same view of the interface but are unaware of what lies on the other side

This file is compiled into a stub for the client This contains the method's location, its implementation language and the parameters it requires etc If the client wants to execute a method it makes a call to this stub The client is unaware of the mechanisms used to communicate the call This stub processes the call and passes it to the underlying ORB runtime This runtime communicates with the server ORB, and passes it on to the server

This procedure differs from other distributed mechanisms such as RPC (Remote Procedure Calls) in that IDL is completely object-orientated and thus supports inheritance and polymorphism as well as encapsulation

5.2.2 CORBA programming

In the following two sections, an outline is given of the main CORBA mechanisms for the set-up of a Client and Server, accompanied with an explanation of the main method calls involved

5.2.2.1 Writing the Server

A class is constructed to contain the server e g *public class Server { }* The first task within a server class is to connect to the CORBA ORB itself This initialises the ORB and returns a reference to the server class This call appears as follows

org omg CORBA ORB orb = org omg CORBA ORB init(),

The next task is for the server to create a reference to its implementation code This is where the implementation of the methods defined within the IDL interface is stored Thus, if the implementation code for this server is in a class called ServerImplementation, the call appears as

TestInterface test = new ServerImplementation(int x,int y),

This creates an instance of the generated Java file from the interface, and this instance is then bound to an instance of the implementation class All that remains is for the server to indicate its readiness to the ORB This can be done using the call

CORBA Orbix impl_is_ready("ServerName"),

In this call the server name is also given to the ORB so it may be distinguished from other servers

5.2.2.2 Writing the Client

A class is constructed for the client e g *public class Client { },* As before, the client must initialise the ORB from its perspective again using the call

org omg CORBA ORB orb = org omg CORBA ORB init(),

The client can then attempt to bind to the server using the call

TestInterface test = TestInterfaceHelper bind(" ServerName",srvHost),

This tells the ORB to bind to a server called ServerName and inform it that a client with a machine address srvHost wishes to bind to it At this point the client can now make method calls on the server using simple method calls with the format *test methodCall( ),*

5.3 Java language

Java was chosen as the main implementation language for the tool It is an object-oriented, architecture neutral, portable, multi-threaded, dynamic language It is designed to support applications on networks, such as the Internet, which made it an ideal choice for the ***Prompter*** tool since ***Prompter*** is designed to be a distributed system capable of operating with its components situated on various computers across an Intranet or Internet It was also a requirement that it be platform independent All these requirements were met by Java In addition since it is fully object-orientated it provides a natural choice for integration with CORBA

Unlike other languages the Java compiler does not generate machine code but specific binary code (bytecode) which is run by a virtual machine This virtual machine is a layer of software on a computer, which takes Java bytecodes and executes them on that platform The Java structure is highlighted in the following diagram taken from [Horton1 97]

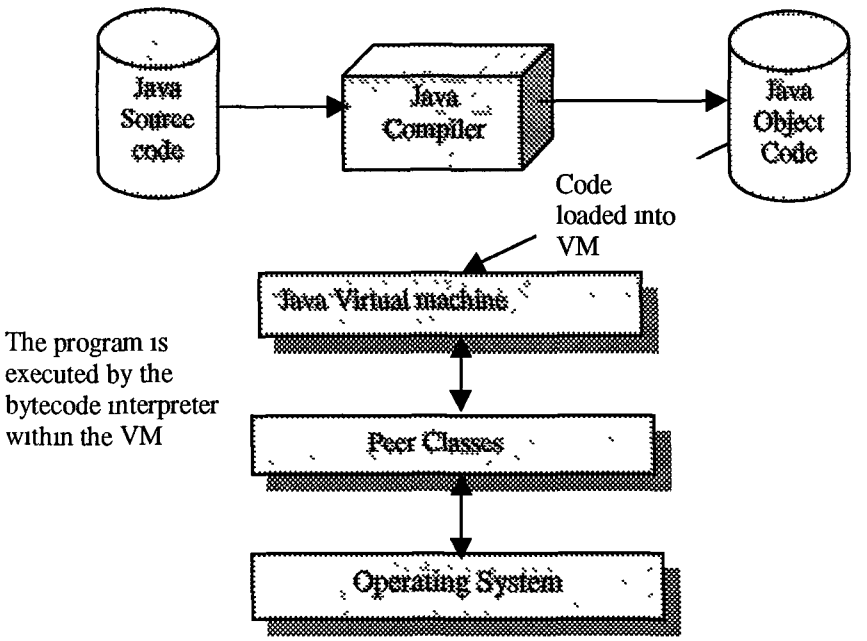

Figure 5.3 Java Overview

This structure results in code that executes the same way, no matter what its underlying architecture is Thus Java bytecodes can be shipped over the net and are guaranteed to function the same on all platforms A program written and compiled on a Pentium with a WindowsNT O/S can be simply transferred to a Unix machine and run without the need for recompilation

Java has achieved widespread acceptance in the programming world [SrinivasK 97] Due to Java's mobility it can also be considered as an option for implementing mobile agents

5.4 Where Java and CORBA fit in.

Although the basic Java support for bytecode migration implies Java code mobility this capability is not really viable With the standard implementation of Java, all Java objects reside on a single host Also Java lacks mechanisms for transmitting arguments from one host to another In contrast the fundamental premise of CORBA is that an object on one host can invoke a method of an object on another host CORBA passes references rather than objects and thus avoids plunging into the issues of object migration CORBA also provides a persistent object service that is not possible with Java

Thus for the purposes of the daemon architecture, using both Java and CORBA together can satisfy all the tasks of a distributed multi-platform system Java will allow CORBA objects to run

on any system Some suggest [OrfaliR 97] that Java is the ideal language for writing client/server objects Its built in multithreading and garbage collection makes it easy to implement robust objects Thus the two languages complement each other Java deals with implementation transparency and CORBA provides the network services not covered by Java It links the Java portable application environment and the rest of the world

5.5 Design and Implementation

In this section some of the implementation issues I encountered in the development of the prototype are discussed This includes explanations of some problems encountered and the mechanisms used to overcome them

5.5.1 Daemon Architecture

The daemon architecture was broken down into two CORBA servers The first server, the Daemon Supervisor controls all communication between the daemons and the kernel through the IDL interface The second CORBA server, the Daemon Library maintains the daemons themselves This second CORBA layer allows the daemon supervisor and the daemon library to reside on different machines and to the rest of the tool if necessary

One issue that is discussed later relating to the daemon architecture is the deletion of a CORBA layer between the Daemon supervisor and the Blackboard This layer was designed into the initial architecture but was deleted during the implementation stage due to the complications it caused

5.5.2 Implementation Strategy

The life cycle model chosen for the development of a prototype of ***Prompter*** was the Spiral model [McDermuidJ 91] This resulted in a number of prototypes that increased in sophistication and complexity as time passed It was thought that for simplicity it was best to implement the prototype in Java initially to prove the architectural concept and then introduce the CORBA communications layer at a later stage It was thought that this strategy would give an easy transition from design to implementation

SUN's JDK1 1 6 was chosen as the development environment over other versions of Java such as Microsoft's J++ which is not 100% pure Java compatible unlike the sun version This compatibility was deemed important as, at the time of coding, a question exists over the compatibility of certain vendors' Java environments. The version of CORBA used was Iona's

OrbixWeb v3 0 and was chosen because it was implemented fully in Java thus maintaining the tool's mobility across platforms and JDK compliance

On the completion of the initial coding of the prototype in Java, the IDL interfaces were introduced At this stage it became clear to me that many of the class structures and protocols would have to be altered as IDL did not support many of the rich structures specified during the design phase

5.5.3 Complex Coding

The introduction of CORBA increased the complexity of the code and the increased length of execution time Since CORBA uses references to objects and servers, the code becomes more complex to debug as it becomes impossible to trace through these references

Another problem was the overhead in running the software When the software was run on one machine, the server calls were almost instantaneous, however, as more distribution was introduced the overhead became more dependent on the state of the network This was expected but it did lead to a lot of waiting by some clients on results from the servers and a general slowing the down of the tool This was most evident in the initial starting up of the tool This required the creation of all the servers which resulted in the ORB having to search for the location of the servers This meant requesting information from the network DNS (The network supervisor), leading to unwanted delay

5.5.4 Improving the performance of the prototype

As stated above, there was a great overhead to initially start the tool due to CORBA One method of overcoming this problem was by specifying exactly what servers were where i e by giving their exact IP addresses on start-up Obviously this reduces the distribution factor of the tool

Another method was the reduction in the distribution factor of the daemon architecture Originally there were three CORBA servers within the architecture, those being the daemon supervisor, the daemon library and finally the blackboard Thus on start-up of the daemon component, three servers had to be created and initialised I decided that one of these had to be removed The blackboard server was chosen because there was a closer link (thus meaning more CORBA communication) between the daemon supervisor and the blackboard than with the Daemon Library

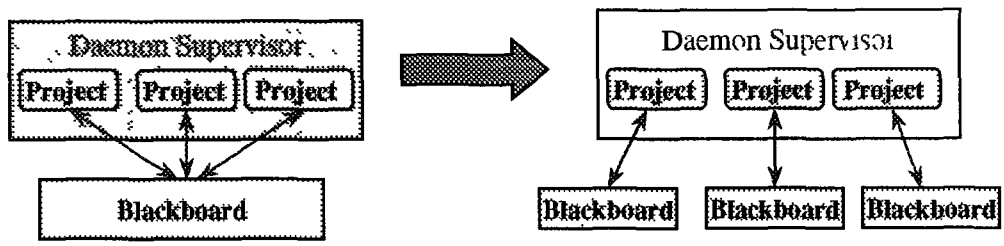

Figure 5.4 Removal of Blackboard server

The removal of this CORBA link meant the daemon server and specifically each project now had direct control over their own blackboard and resulted in a much quicker prototype (see figure 5 4) The prototype now operated with a 50% improvement in overall operational time It also reduced the complexity of the code

The elimination of the blackboard server as it stood also reduced a potential bottleneck at the blackboard interface As it stood, the blackboard server handled all information dealing with all projects from all users However with the elimination of the interface, it allowed each project to have its own unique blackboard which it controlled This reduced the complexity of maintaining data relating to projects It also made the design more OO as the data for one project became only available to that project The trade-off here however was the tool was now less distributed

The final mechanism I incorporated to improve the performance of the prototype was the reduction of method calls made across CORBA This meant that when a CORBA call was made the maximum amount of information had to be passed with it An example of this is the token change protocol Initially the kernel signalled a tokenchange after one token was changed Thus it was decided to create a structure called the state array that would buffer token changes and transmit a number of them at once Obviously this meant the daemon architecture was not as up to date as possible, but a trade-off had to be made for efficiency This solution reduced the CORBA calls by almost 60%

5.5.5 IDL aiding the Design

Since IDL was used as an interface between the main components within the architecture and the rest of the tool, it forced me to make decisions about some design issues before they would normally be considered Initially the interfaces were specified and stubs created This then provided a base for the design stage, which allowed each logical structure to be separated from the rest of the architecture As a result of this property, it forced the design stage of the

architecture to be linked more closely to some implementation It also allowed the easy addition of new code to the prototype which was necessary since the life cycle was a spiral model for the tool

There is another side to this however Since these interfaces had to be specified early in the design process it resulted in many problems and errors occurring that caused re-writes of the interface So although it helped to work with some interface, it also introduced the problem that if the interfaces were wrong the design itself would have to be changed resulting in many changes

5.5.6 Problems with the Spiral development

The Spiral model was used as the life cycle model for the tool This meant that a simple version of the prototype was developed initially and through a process of revisiting, it became more complex, with more functionality as time progressed

This seemed a good method of developing the architecture However it did have its problems In the first iteration of the prototype many lessons were learnt about CORBA and IDL etc However when they were revisited in the second iteration, the documentation had to be revisited, code had to be studied again and in some cases re-implemented or improved upon, resulting in wasted time on each iteration Thus this life cycle model resulted in a lot of time taken consulting documentation, redesigning some of it, scrapping some of it, etc instead of having a constant turnout of new code

However some good points also resulted from this life cycle model It allowed certain paths to be tried and if unsuccessful, reworked Also it allowed the ability to improve on old code as new code was introduced As the java language progressed new functionality was discovered that allowed operations to be performed in different ways, e g the reading in of files The life cycle model allowed this method of improvement This was also useful was with the advice structure I decided that the advice would be more useful to the tool as actual HTML instead of its OO form This allowed the advice to be restructured and also made the daemon architecture much more adaptable to other systems as the advice from the daemons was in a standard format that could be easily incorporated

5.5.7 Daemon Supervisor availability

The daemon supervisor acts as a server to the kernel and must be available to execute a method invocation at any time If a tokenChange method was called it would result in the daemon supervisor having to perform a number of operations including a callback before it was free to perform the next task from the kernel This was unacceptable since if the kernel was kept waiting, the rest of the tool and the user would also have to wait

It was not possible to make all the calls to the daemon supervisor one-way as most of them would be missed Thus one mechanism of overcoming this was to introduce multi-threading When a call is made from the kernel, a thread is created in the DS to handle it, and is allowed to execute, thus ensuring the daemon supervisor spends as little time as possible being unavailable to the kernel

5.5.8 Bottlenecks

The above is a simple mechanism of threading the daemon architecture as each thread is given its execution time by the operating system It was considered that problems would arise when more than one of these threads wanted to write to the blackboard and the advicetable (the structure for buffering advice to the kernel) However within the Java language threading classes there is functionality to cover this With the addition of the word "*synchronised*" to the definition of a method, it prevents more than one thread accessing that method at any time This does however cause bottlenecks in the system

To ensure consistency, methods concerned with the blackboard, the state-array and the advice table had to be synchronised in other words allowing one thread in at a time This meant that at the interface to these methods bottlenecks occur To reduce this the size of these methods were reduced to perform a smaller set of tasks, thus making the time to execute this method smaller and the bottleneck smaller

5.5.9 Callbacks

Callbacks are a mechanism used by CORBA and OrbixWeb to allow a server to invoke methods on a client In other words the implementation of the method is in the client and is invoked from the server These appear many times in the interface between the DS and the Kernel They also produced most of the problems that were encountered during this interface's development during the implementation phase

To understand the problems that followed, the implementation mechanism must be explained further Within the server there is a waiting mechanism that serves calls made by the client This waiting mechanism is the Impl_is_ready call Thus for a server to invoke a method on the client, there must also be a serving mechanism to serve these invocations This mechanism is the processEvents call Thus when the server invokes a method on the client the processEvent serves it in the same way as the Impl_is_ready

The first problem these callbacks produced was the fact that the client did not know when the server would make one of these calls and so did not know when to call processEvents It was thought that placing the processEvent in a separate thread would solve this problem However this would greatly slow the client down as the scheduling of the thread would be left up to the underlying operation system to schedule whether the processEvent call or the client thread should be run

I noted that the callbacks in general would not be that common and that normal client/server operations would be in the majority Thus I decided to schedule a processEvent when it was thought that one was most likely to occur It was known that most callbacks would occur when the daemon supervisor wanted to inform the kernel that it had advice available Thus the kernel would know when it told the daemon supervisor which tokens had changed and that it may result in a daemon executing and thus advice being generated Thus after a tokenChange method was called, it would be soon followed by a processEvent call

5.5.10 Deadlocking

Another problem that was introduced due to callbacks was deadlocking When a method was invoked from a server to the client it may cause what is called deadlocking This occurred when a call tokenChange was made to the server (Daemon Supervisor) It caused the server to request the state array from the kernel However what was not considered was that the kernel was still waiting for the method tokenChange to finish as illustrated in figure 5 5

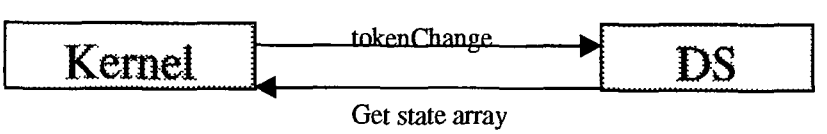

Figure 5.5 Problem with deadlocking

But the return call cannot be processed until the tokenChange method is finished which is when the state array is returned As can be seen this will never be resolved

One solution was to introduce a timing mechanism for the calls thus insuring if the method did not finish after a certain time the call was revoked However if the client was blocked, how could it check the system time? To overcome this problem of blocking, the method call itself was encapsulated in an individual thread so if it becomes blocked only that thread would block while the client as a whole remained active

However this was still unacceptable as the thread would remain blocked indefinitely Thus the decision was made to make the method tokenChange one-way Now this allows the method invocation to be equivalent to a fire and forget mechanism, meaning that when the kernel invokes the method it does not wait to ensure the daemon supervisor receives it, but instead presumes it has and that the daemon supervisor server will process it Of course, there is the potential for the daemon supervisor to miss the call but it is a trade-off to ensure that deadlocking is not encountered

5.5.11 How open is the daemon architecture

The prototype has no hard coded references to the domain of software project management Thus to change the domain of the daemon architecture would not be incredibly difficult All that is required is to change the daemons contained within the directory "dlib" in the "DmnPack" directory of the prototype When these daemons are changed the prototype must then be restarted This causes the daemon library to re-read the daemon headers and the blackboard is then reconstructed with the new daemon information Thus the daemon architecture has incorporated the new daemons

Obviously this is not possible in the case of all domains The new domain must be structurally similar to that used in the prototype since the blackboard must be capable of representing the state of the daemons at any stage and it has to do this with its structure of nodes, segments and scenarios Also the new domain must utilise the concept of tokens as this is the basic data type used within the prototype Thus as long as the new domain maintains a similar structure to software project planning, the prototype will have little problems representmg it

Obviously the daemons must also retain the same structure as the ***Prompter*** daemons as well They must have a daemon header that identifies itself to the tool and the advice structure should remain similar This should not be highly difficult because in general when advice is given it is usually followed by a justification for it, and the ability to give background information to the advice and some sort of bibliography of where more information can be acquired

5.6 Summary

In this chapter some of the main implementation languages used in the development of a prototype and the problems that were encountered are discussed Contained within the CORBA description an outline was given about how simple servers and clients can be created However this is by no means the full potential of CORBA as there is a great deal more functionality that has not been mentioned here

The rationale was then given for why these languages were chosen over others This rationale which was stipulated in the requirements documentation for the tool, forced the rejection of languages which may have had a quicker execution time than Java, but which were not as platform independent or as easy at multithreading etc

However the question must be asked whether the languages achieved their objective The answer to this would be yes in that the prototype is fully JDK compliant and distributed The prototype that is machine independent and platform independent, and with the use of CORBA the tool is capable of operation across a distributed network This architecture is also easily adaptable to other domains in that there is no software project management domain-specific information hardcoded into the prototype, thus allowing it to be used with many other domains The architecture also allows the easy incorporate of new daemons into it As stated in the previous chapter this incorporation only requires a new inference engine wrapper if its implementation language is new or if the inference engine for its language is already in the tool, it just has to be added to the daemon directory

In the later sections of this chapter some of the problems encountered during the implementational stage and the solutions to them were described Most of the problems discovered were eliminated or at least reduced The questions about the complexity of the code and the runtime are issues that cannot be ignored as they are still evident in the performance of

the prototype On the other hand in respect to the entire tool, the runtime of the daemon architecture is not as critical to the overall tool as with other components

In conclusion, the languages chosen to implement the prototype did achieve their tasks It must be expected that with the incorporation of CORBA, an overhead is in some way incurred The detailed design that was described in the previous chapter was implemented and the architecture did perform decision support for the user in some fashion while still retaining its conceptual and implementational independence from the rest of the tool

In the following chapter the conclusions that were drawn from this thesis are given with suggestions as to how this architecture can be developed further in the future

6 Conclusions

6.1 Introduction

In this chapter I outline what has been achieved in this thesis including a discussion on the current state of the prototype and its architecture, the architecture's strengths and weaknesses and a discussion on the future work and development that could be performed on the architecture It finishes with some of my personnel remarks relating to the work I have performed and what I have learned

6.2 Open architecture

There were a number of requirements the daemon architecture had to satisfy, most notably to provide decision support within the ***Prompter*** tool However it also had to be as open as possible, maintain a level of abstraction between the architecture and its knowledge, highly dynamic and mobile, and like the rest of the ***Prompter*** tool and it had to represent its knowledge in a convement and easily expandable manner

Thus, the requirements above called for an architecture that was

- Dynamic
- Generic
- Mobile
- Distributed
- Efficient

In the following sections it is shown how these criteria were achieved

6.2.1 Mobility

The daemon architecture is not specific to any operating system or computer architecture since it was implemented in Java (chapter 5), a platform independent language, the distribution language was OrbixWeb3 which is a Java implementation of CORBA, and JESS, a Java implementation of

CLIPS, was chosen as the agent language These languages achieved their tasks while also maintaining the 100% JDK compliance of the tool

6.2.2 Degree of Distribution

Iona's implementation of the CORBA standard, OrbixWeb3 (chapter 5) was the distribution language chosen Although this added a major level of complexity into the development stage of the tool, it did allow, the daemon architecture to operate asynchronous from the rest of the tool across a network

I also introduced a level of distribution into the daemon architecture between the daemon supervisor and the daemon library and also between the daemon supervisor and the blackboard, which was later deleted due to efficiency problems

6.2.3 Generic

In order to achieve an architecture that was generic, a level of abstraction between the architecture and the daemons had to be maintained Consequently I decided to have the architecture control the daemons using inference engines which are tied to it using wrappers These inference engines are responsible for the execution of the daemons and the retrieval of their advice This does not prevent daemons implemented in different languages from communicating Daemons/agents controlled by different inference engines communicate with each other usmg the blackboard As a result, the architecture has no contact with the daemons themselves Thus to replace the domain, only the daemons have to be changed

6 2.4 Expandability

The daemon architecture had to have some mechanism of altering its knowledge base easily As described in chapter 4 the knowledge base is mapped to the daemons in a 1 to 1 relation such that for each section of the taxonomy there is a daemon to represent it

Thus to expand the knowledge of the taxonomy the process is simple A new daemon is written to contain the new knowledge to be incorporated There is no requirement on how the daemon's internal rule structure should appear as long as it conforms to the high level structure laid out in

chapter 4 If the daemon is written in a language familiar to the architecture i e there is a wrapper for it already in the architecture, then the daemon file can be inserted into the "dlib" directory where it is incorporated upon start-up of the tool However, if the daemon is written in a new language a wrapper must be constructed to incorporate the inference engine for it into the architecture This is not as complicated as it may seem as there are a number of Java representations for AI language currently available (at the time of writing there were Java implementations of Prolog and Lisp as well as the already described CLIPS available) Again the new inference engine and daemons are incorporated upon start-up of the tool

An advantage of this integration process is, the tool does not have to be recompiled each time a new daemon is introduced This property of expandability ensures that the knowledge can be easily maintained in the future

6.2.5 Efficiency

One problem that was encountered during the implementation stage and discussed in more detail in the previous chapter was the delay introduced by the CORBA layers If the daemon architecture was to perform satisfactorily it had to be efficient at its tasks

Since there are two CORBA layers above the daemon architecture there is already a major delay to the user However, as was also noticed, this delay was greatly increased by the CORBA layer between the daemon supervisor and the blackboard Smce its removal, the distribution factor of the architecture was reduced but the efficiency of the prototype was greatly improved

Also because the interface between the daemon supervisor and the rest of the tool contains a number of methods it was possible to easily introduce multithreading into the daemon supervisor So on method calls from the kernel, the daemon supervisor creates a thread to handle the task, thereby utilising its CPU time better

6.3 Weaknesses of the Architecture

In this section some of the weaknesses in the design and in the implementation of the daemon architecture are described

Although the architecture is an open one, there are still some domain-specific traces within it The main one is the structure of the blackboard The blackboard currently provides one node structure for each individual area of software project planning Within each node is a segment for each daemon and so on Thus the blackboard in some way mirrors the domain it represents If a new domain is introduced it may be difficult to incorporate this into the blackboard A new set-up program for the blackboard may be needed if it could not be represented the same way This leads me to conclude that the daemon architecture is best suited to domains where knowledge is easily broken up into categories similar to those of software project planning

As stated above if a new daemon is added to the system with an existing inference engine, there is no problem However if a daemon written in a new language is created, a wrapper for its inference engine must be created For certain languages this may not be possible if there is no functionality provided

At present the prototype runs slowly However since the daemon architecture is at the backend of ***Prompter*** this may not be an issue If the architecture is incorporated into a tool that requires faster results, problems may occur This lack of speed is as a result of the use of CORBA within the architecture

Since the architecture communicates with the rest of ***Prompter*** through a CORBA interface, it is necessary to have an ORB situated on the machine that runs it as well as a Java virtual machine As a result a lot of resources are required to run the prototype

6.4 Future development of the tool

It is not thought that the knowledge base of ***Prompter*** will remain static, A tool to allow the user to insert new daemons into the system would be very useful i e a Daemon Developers Kit This tool would allow the user to create a daemon and hide all the implementation issues from them It should ensure that the daemon is correct and insert it into the system leaving the user to concentrate on the knowledge it must contain

It is known that there are some unresolved issues with how agents co-operate with each other and how they resolve problems In this architecture these issues were overcome with the use of a blackboard However this is not the only way this can be performed The introduction of an agent communication language may provide a better solution to this issue

One major piece of functionality that has not been tackled is agent learning Since the daemons at present have no idea of the profile of the user, they will supply information when they execute However project managers in general all have different mechanisms of managing projects Some managers hold their budgets as one of their critical issues while others think of this as being of less importance To cater for this the daemons or indeed the blackboard could build up a profile of what information the user accepts and what they leave for a later stage If a daemon's advice kept getting rejected, it could inform the system administrator that its advice could be wrong Also if the manager likes to be kept aware of their Risk at all stages, the Risk daemon would supply advice more often To allow the architecture to perform this task, some Artificial Intelligent techniques could be introduced One suggestion would be the introduction of neural networks into the blackboard that learn which daemon the user accepts advice from and which they reject

6.5 Conclusions

The goal of this thesis was to design and develop an agent based architecture to provide decision support for the ***Prompter*** tool This was achieved through the design and the development of a daemon architecture as described in the previous chapters of this thesis Below some of the final conclusions from this thesis are given along with the resolution of some general questions that may have arisen

The first issue that must be addressed is whether the architecture achieved what it set out to do The answer is yes The architecture is capable of providing advice or critiquing the user's project from a number of different perspectives It is distributed, and mobile, and most importantly is an open architecture that allows the knowledge base to be easily expanded or even completely changed

The architectural structure is also sound, since it achieves its tasks successfully Even though there are possible mechanisms for improving it, there are no components within it that are flawed However if the design was performed again the mistake of over distribution would not have been made Also, more investigation would be performed concerning other distribution languages instead of OrbixWeb The question still remains, would the prototype have run better using Java's Remote Method Invocation mechanism? Another point that could be re-examined would be making the daemon architecture more independent At present the link between the tool and the daemon architecture is similar to a client-server protocol It was initially thought that this architecture would be completely independent

Another question that remains is whether it would have been better to use an agent communication language such as KQML instead of using the blackboard structure since a language of this type would allow better agent coupling However there are advantages with the blackboard, the first being that it has been tested and it allows the architecture to remain in control of the daemons However, if a language that allowed agents to communicate among themselves was introduced, it would greatly increase the complexity of the relationship between the daemons and make it much easier for daemons to work together

I can also conclude that Java was a good choice for the implementation language It allowed the tool to be machine independent and highly mobile OrbixWeb allowed the individual components of the tool to be distributed It introduced a great deal more complexity into the system, but it did allow the tool to be divided up well Also the choice of another distribution mechanism would most likely have introduced the same complexity

One of the main questions that has arisen over the course of this thesis is whether the daemons produced were actually agents Some sacrifices were made to the idea of an agent such as surrendering some of their autonomy However they still have a lot of the properties associated

with agents as outlined in chapter 3, such as independence, being goal driven, the blackboard allows them to be communicative, and reacting to their environment

Another level of complexity that was introduced was JESS Was it a good language for implementing daemons? Since it is an mdependent component, JESS takes care of problems that can arise with agents such as memory allocation, conflict resolution, rule resolution etc Also it demonstrates the adaptability of the architecture to new languages It was capable of representing the knowledge that was provided and therefore I must conclude that it was a good choice

6.6 Concluding Remarks

From the design and development of the daemon architecture I have learned a substantial amount about software development within a team In this section I will make some general comments that strike me as being of some importance

The design of an architecture for mtegration within another tool is a complicated process which is usually performed in stages In developing the daemon architecture I learned this process, and how important it is to the quality and maintainability of the resulting software Initially within this process, there must be a requirement specification phase to define what it is that the component is expected to perform From this an architectural design must be developed and then a detailed design At every stage there must be adequate documentation to explain the decisions made or protocols introduced I realised the importance of this process when the time came for me to design the daemon architecture This documentation acted as a mechanism of reminding me why one decision was made over another while also helping me produce higher quality code Also each component was constructed using a traceable mechanism of documentation from the requirements to the coding stages This prevented the coding stage from veering from the desired path

Since the daemon architecture was developed as part of the ***Prompter*** tool, I had to design and develop it so it was capable of interacting with software developed by other organisations This introduced me to the idea of team development I learned the importance of correctly versioning my documentation and code, of keeping up to date with the tool's functionality, and most importantly, working with other members of the team to create solutions to problems such as efficiency, saving protocols, CORBA problems etc For the three components to work coherently together, it was crucial for the members of the team to do as well

Prompter was developed according to a strict schedule of builds and deliverables and as a result so was the daemon architecture This illustrated the importance of scheduling and developing software within a time limit It meant that solutions to problems had, in some cases, to be abandoned due to lack of time etc It required me to organise my time more thoroughly, making me prioritise functionality and tasks in a better manner

In developing the daemon architecture and the knowledge it represents, I was introduced to a different level of the project I was dealing with people who were responsible for supplying the knowledge for the agents/daemons It helped illustrate that not all work within a project is implementation oriented and that other issues must also be considered This was also true when working with users who provided feedback about prototypes developed, documentation, and knowledge issues Since they do not care what happens behind the scenes, they have a better view of the tool as a whole This consultation showed that their most important concern was not how, for example, the blackboard was structured, but instead with the appearance of advice for the user and its content I learned that the tool should be viewed at several levels of abstraction and not only at the design level

Since ***Prompter*** was developed for commercial purposes a number of commercial issues had to be considered during its development One that was highlighted to me during the development of the agents/daemons was the problem of commercial licences for agent languages When I was investigating agent languages to implement daemons in, I was not considering how much they would cost to integrate into a commercial tool When this issue was raised a number of languages had to be abandoned Commercial software that uses other software to operate, generally must acquire a licence to do so and this usually incurs cost

Since ***Prompter*** will be used in real project development, it had to be ensured that within the tool, and specifically for myself, within the daemon architecture, components were well documented and the structures and interfaces to components were as simple as possible This was to ensure that the cost of maintaining the code would be low It had to be easy for components to be replaced or rewritten if necessary This was also one of the reasons for maintaining the abstraction of the tool from the knowledge within the daemons

7. Bibliography

[AIST 98] Agency of Industrial Science and Technology, "http //www aist go jp/ETL /~matsui/javalisp/index html", Electrotechnical Laboratory, JavaLisp, accessed 14-9-98

[AmblerS 97] Ambler, Scott W , "Java Coding Standards 17 01a", AmbySoft Inc 1997

[Broadcom 97] Trinity College Dublin, Broadcom Eireann Research, "Software Agents A Review" May 1997

[CDMCS 92] University of Alabama, University of Tulsa, PED-MICOM-Army "Compostite Design and Manufacturing Critiquing System", USA, 1992

[ChunH 97] Chun, W Hon , Lai, M Edmund, "Intelligent Critic System for Architectural Design" IEEE Transactions on Knowledge and data engineering, Vol 9, No 4, July/Aug 1997

[CroftD 97] Croft, David Wallace, "Intelligent Software Agents Definitions and Applications", Special Projects Division, Information Technology, Analytic Services, Inc, USA 1997

[EngelmoreR 88] Engelmore,R, Engelmore, Morgan, Engelmore Tony, "Blackboard Systems" , Addison-Wesley, Great Britain, 1998

[EckertC 95] Eckert, Claudia, "Intervention Strategies for Critiquing Professional Designers", Design Discipline, The Open University, Milton Keynes, UK, 1995

[FarleyS 97] Farley, Steven R , "Mobile Agent System Architecture", Java Report May 97

[FininT 92] Finin, Tim, Finin, Rich, McKay, Fritzson, McKay, Don, "A Language and Protocol to Support Intelligent Agent Interoperability", Proceedings of the CE \& CALS Washington `92 Conference , June 1992

[FischerG 93] Fischer, G , Nakakoji, K , Ostwald, J , Stahl, G , Sumner, T , "Embedding critics in design environments", Knowledge Engineering Review, 1993

[FIPA 97] "FIPA97 Developer's Guide" – an output from FIPA98 Technical Committee 10, Version 2, 1997

[FriedmanE 97] Friedman-Hill, Ernest, "JESS, The Java Expert System Shell", Distributed Computing Systems, Sandia National Laboratories, USA, 1997

[GeneserethM 94] Genesereth, Michael R , Ketchpel Steven P , "Software Agents", Papers from the Spring Symposium, Stanford University 1994

[GernerA 93] Gerner, Abigail S, "Critiquing Effective Decision Support In Time Critical Domains", Dissertation Proposal, Department of Computer and Information Science, University of Pennsylvania, 1993

[GernerA 94] Gerner, Abigail S, "Critiquing Trauma Management Plans On-line", CliFF Abstract, cliff-group, 1994

[Giarratano 84] Giarranto R , Riley, "Expert Systems – Principles and programming", Second Edition, PWS publishing, 1984

[HenryW 94] Henry, William, "Software Project Risk Management A support Tool" Dublin City University, M Sc in Computer Application, 1994

[HortonI 97] Horton, Ivor " Beginning Java", Wrox, UK, 1997

[LangeD 96] Lange, Danny, B , Chang, Danial T , "IBM Aglets Workbench, Programming Mobile Agents in Java A White Paper", IBM Corporation September 1996

[LangeD 98] Lange, Danny, B , "Mobile Agents Environments, Technologies and Applications", PAAM98, 3rd international Conference and Exhibition proceedings

[MaesP 97] Maes, P , "Software Agents", Unicom Conference on Agents and Intelligent User Interfaces proceedings, Unicom, 1997

[MartinL 96] Lockheed Martin advanced concepts, "Succeeding with the Booch and OMT Methods, a practical approach", Lockheed Martin advanced concepts centre, Rational Software corporation, Addison Wesley, USA, 1996

[McDermuidJ 91] McDermuid, John, A , "Software Engineer's Reference Book", Butterworth-Heinemann, Oxford, 1991

[MoynihanT 94] Moynihan, T , Power, J , Henry,W , "A critiquing system architecture for Software Risk Management", 5th European software control and metrics conference proceedings (ESCOM) Italy, May 1994

[OMG 98] "The Common Object Request Broker Architecture and Specification", Object Management Group, MA, USA 1998

[OrfaliR 97] Orfali, R , Harkey, D , Edwards, J , "CORBA, Java and the Object Web", Byte magazine, October 1997

[Orbix 97] "OrbixWeb programmers guide", Iona Technologies plc Dublin, 1997

[PetrieC 97] Petrie, Charles J , "What's an agent and what's so intelligent about it?", IEEE Internet Computing, July/August 1997

[PowerJ 94] Power, Jane A.; "A critiquing system architecture in the risk management domain: Riskman2" DCU M.Sc. in Computer Applications thesis, 1994.

[PressmanR 94] Pressman, Rodger S.; "Software Engineering, A Practitioners approach", Third Edition, European Adaptation, McGraw Hill 1994.

[Prompter 97] O. Connor, R. O; Floch, C; Moynihan, T.; Renault, T.; Combelles, A.; "Prompter- A decision Support Tool using Distributed Intelligent Agents", Dublin City University 1997.

[Prompter 98] Prompter knowledge team, "Software Cost Risk", Daemon Knowledge Base Design, Project and Process Prompter, ESPRIT 22241, Dublin City University, 1998.

[RichE 91] Rich, Elaine; Knight, Kevin; "Artificial Intelligence, Second Edition ", McGraw-Hill, 1991, Ch 20

[Rumbaughj 91] Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick; Lorensen, William; "Object-Oriented Modelling and Design", Prentice Hall, Englewood Cliffs, NJ, USA, 1991.

[SilvermanB 92] Silverman, Barry; "Communications of the ACM", April 1992, Vol.35, No.4

[SmithP 86] Smith, Peter; "An Introduction to Knowledge Engineering", International Thomson, 1986.

[SrinivasK 97] Srinivas, K.; Jugannathan V.; Karinthi, R.; "Java and Beyond Executable Content", IEEE Computer, June 1997.

[ThayerR 88] Thayer, Richard H.; "Tutorial: Software Engineering Project Management", IEEE Computer Society Press, USA, 1988.

[TurbanE 95] Turban, Efraim; "Decision Support Systems and Expert Systems, Management Support Systems", 4th Edition, Prentice Hall, USA 1995

[Verbruggen 87] Verbruggen, R; Jenkins,J; Bosco, M; "Integrated Management Process Workbench: Intelligent Assistance for the Software Project Management", CASE '87, First International Workshop on Computer Aided Software Engineering, Cambridge, Massachusetts 1987.

[WebbeB 94] Webbe, Bonnie Lynn; Clarke, John; Chi, Diane, Gerner, Abigail; Kaye J.; Neumann, S.; Ogunyemi, L.; Singh M. "The TraumAID Project", CliFF Abstract, cliff-group, 1994.

[WeissM 96] Weiss, M.; Johnson, A.; Kinsey, j.; "Distributed Computing: Java, CORBA, and DCE", Technical Report Open Systems Foundation, 1996.

[WhiteJ 96] White, Jim; "Mobile Agents white paper", General Magic, USA, 1996.

Appendix A The main classes in the architecture

Blackboard Class

```
/**
 * @author Eamon Gaffney
 * @author DCU
 *
 * This is the Blackboard class which is responsible for the BB structure and the states of the tokens within
 * each of the daemons (segments) It also stores the outputs from these daemons in case it may be required
 * at a later date
 */

import java io *,        // The Imported Java classes
import java util *,


/*
 * The BB structure is implemented as a vector and within each element there exists a segment The BB
 * creation can occur when the BB has a copy of the daemon libraries Dependency table Within this class
 * there exists a number of methods which answer queries from the DS however there are also a few
 * callbacks resident here as well
 */

public class BB {

  // The BB structure will be a number of nodes all collected together using the vector data structure

  private Vector BBstructure,    // vector to store all the nodes
  public int projectID,              // store the projectID the BB is related to
  public DaemonInfo [] DT,     // This is the Dependency Table received from the
                             // library It is used by the method newCreateBB to
                                  // create the BB


  public BB() {
        // constructors to create the BB vector
        BBstruct = new Vector(),          // the vector is initialised and ready to store information
  }


  // Methods for the BB Class

  public void newCreateBB(DaemonInfo [] dTable, int pID) {
     /*
      * This method is the new BB constuction It receives the DT from the DL This contains an array of
      * daemonInfo objects which is then placed into structures called node It first checks if the node is
      * already created If not it should be created After this stage the segment which contains the
      * daemon information is added to the relevant node
      * @param  dTable The dependency table
      * @param . pID . projectID that the BB is related to
      */
```

```
        DT = dTable, // Create a local copy of the DT
        projectID = pID,

        for (int index=0, index< dTable length, index++) {  // for all elements in the dependency table
array do
            String dmnArea = dTable[index] area(),  // read the area of the current daemon index
            int j=0,
            Node newNode,  // Search current BB to ensure node is not created already

            while ((j<BBstruct size()) && (dmnArea != ((Node)(BBstruct elementAt(j))) area)) {
                j++,  // if not found move to the next index in the BB
            }
            if (j< BBstruct size()) {  // if node is found in the BB already
              newNode = (Node)(BBstruct elementAt(j)),  // The set the var equal to its position
              // The reason for this is so a daemon may be added to it
            }else {
              newNode = new Node(dmnArea),  // create a node instance
              BBstruct addElement(newNode),  // add new node to the BB
            }

          Segment seg = new Segment(dTable[index] daemonID() , dTable[index] tokenIDs()),
          // create a segment giving the relevant information
            newNode segs addElement(seg),          // add segment to the BB structure
          }  // This will be done for every entry in the DT
}


public void endBB() {
    // method to end the BB structure or reset it
    BBstruct = new Vector(),

}


public void startScenario(int sID) {
    /*
     * Method to create a new scenario within the project  This method must
     * then tell each node to create a new scenario for each daemon
     */
        for (int i=0,i<BBstruct size(),i++) {
            ((Node)(BBstruct elementAt(i))) startScenario(sID),
         }
}


public void stopScenario(int sID) {
    /*
     * Method to delete a scenario within the project  This then tells each node to create a new scenario for
     * each daemon
     */
     for (int i=0,i<BBstruct size(),i++) {
            ((Node)(BBstruct elementAt(i))) stopScenario(sID),
    }
}
```

```
public int [] findDaemons(int tokenID) {
    /*
     * Used to find all the daemons which contain a certain token  It does this by searching the Dependency
     * table and constructing an array of integers which contains the daemonIDs
     * @param   tokenID   the tokenID to be searched for
     * @retur   int []   array to store the daemonIDs the token appears in
     */
        Vector did = new Vector(),

        for (int k=0,k<DT length,k++) {  // for all elements in the DT do
          // for all the tokens in the segment do

          for (int j=0,j<DT[k] tokenIDs() length,j++) {

            if (tokenID==DT[k] tokenIDs()[j]) {
              // if the tokenID required for this daemon do store the daemonID
              did addElement( new Integer(DT[k] daemonID())),
             }
          }
        }
        int [] daemons = new int [did size()], // store these IDs in an array
         for(int i=0,i<did size(),i++) { //copy the buffer into an array
             daemons[i] = ((Integer)( did elementAt(i) )) intValue(),
        }

        return daemons, // return this array
}



public int[] daemonSearch(int dID) {
    /*
     * Method used my many methods for the purpose of searching for a specific daemon within the BB
     * @param   dID   daemonID
     * @retur   int[]   index 0 contains index of the daemon within the node
     *                  index 1 contains the index of the node within the BB
     */
        int i =0,
        int [] ans = new int[2] ,

        do{ // do a search for the daemon in each node  daemon search returns -1
          // if it is not in a node and the position in the node if it is there
          ans[1] = ((Node)(BBstruct elementAt(i))) daemonSearch(dID),
          i++,
        } while((i<BBstruct size()) && (ans[1] <0)),

        i--,  // required to negate the last i++
        ans[0]=i,
        return ans,
}
```

```
public void updateToken(int daemonID , tokenInfo tInfo) {
    /*
     * update the token in the BB in all the segments it appears in Firstly it searches for the DaemonID
     * within the BB then searches for the token within that daemon When it finds it, the state
     * is changed and the daemon flag hasChanged will indicate a token has changed
     * @param   daemonID   the ID of a daemon
     * @param   tInfo   contains where the token is found i e
     *              its projectID and its scenarioID
     */
    int [] ans = daemonSearch(daemonID), // search for daemon  ans[0] = position of daemon within node

        if (ans[0]< BBstruct size()){  // if i < BBstruct then the daemon was found at i
                        // since the while ends before when pos is set
          ((Node)(BBstruct elementAt(ans[0]))) updateToken(ans[1],tInfo), // update the tokens
        }                                                                  // at position and[0]
}



public void forwardStateArray(tokenInfo tInfo) {
    /*
     * method which receives an object of tokenInfo This contains all the information about the token that
     * has been changed from where the token is related to etc Now all that
     * remains is for the all the slots related to the token to be updated
     */
    int [] daemonIDs = findDaemons(tInfo tokenID()), //find all the daemons, tokenID exists in
          for (int j=0,j<daemonIDs length,j++) { // for each of these daemons
            updateToken(daemonIDs[j],tInfo),// update the token within them
          }
}



public Vector availableToExecute() {
    /*
     * Searches the BB for any daemons (segments) that are available for execution  A daemon is available
     * if all its tokens are available  and there has been a change in one of its tokens
     * @retur   BBExeInfo[]   the array of all the daemons that can execute
     */
        Vector did = new Vector(),          // vector to store all the daemons and their relavent tokens
        for (int i=0,i<BBstruct size(),i++) { // for all segments do
          if (((Node)(BBstruct elementAt(i))) hasChanged()) { //if a token in the node had changed do
            ((Node)(BBstruct elementAt(i))) hasChanged = false,
            Vector temp = new Vector(), // create a temp vector to store the daemonIDs
            temp = ((Node)(BBstruct elementAt(i))) availableToExecute(),  // ask the node to return a list
                                                                          // of  daemons that can execute
           for (int j=0, j<temp size(), j++) { // copy them into the proper vector
             did addElement(temp elementAt(j)),
           }
         }
       }
       for(int i=0,i<did size(),i++) { //copy the buffer into an array
          ((BBExeInfo)(did elementAt(i))) projectID = this projectID,
       }                                  // an array as it is easier to transport
       return did,  // send list to DS project,
}
```

```
public void forwardAdvice(DmnOutput adv) {
    /*
     * Forwards the Daemons advice to the output area of the related segment
     * @param   adv   the advice object reference to be added to the BB
     * @param   dInfo   the projectID, scenarioID of where the advice is related to
     */

        int daemonID = adv daemonID,  // retrieve info from object
        int pos, i =0,
        int [] ans = daemonSearch(daemonID), // search for daemon  array returned

        if (ans[0]< BBstruct size()){  // if ans[0] < BBstruct then the daemon was found at i
                        // since the while ends before when ans[1] is set
          ((Node)(BBstruct elementAt(ans[0]))) updateAdvice(ans[1],adv),      // update the advice
        }                                                                     // at position ans[1]
}


public String saveBB() {
    /*
     * Save the information stored within the BB  This method stores this information within a string which
     * is then returned to the DS and on to the Kernel which stores it in the project file
     * @param None
     * @return String the the BB info is saved in
     */
        String info = "BB",
        for(int i=0,i<BBstruct size(),i++) { // for all elements do
              // access the element at index i which is a secment and add node tag to string
              info = info                  // store the returned string from that node
              +((Node)(BBstruct elementAt(i))) getNodeInfo(),
        }
        info = info+ ",END",
        return info,
}


public void loadBB(String info) {
    /*
     * Method to load the information in the string back into a BB structure
     * @param String that stores the information to be loaded
     * @return None
     */
         BBstruct = new Vector(),          // tokemze String usmg Node as the delimiter
        StringTokenizer t = new StrmgTokenizer(info,",",false),  // tokemze the TOKEN line
        String temp = t nextToken(),
        while (!temp equals("END")) {
          if (temp equals("Node")) {
                    Node newNode = new Node(t nextToken(),t nextToken()), // pass node name and state
                    temp = newNode createSegments(t),
                    BBstruct addElement(newNode);
          } else { temp = t nextToken(),}
        }
 }


} // end of blackboard class
```

Node Class

```
/**
 *  @author Eamon Gaffney
 *  @author DCU
 *
 *  This class is responsible for the daemon object node that will be store the segments from a specific area
 *  of the knowledge base  The purpose of this class is to group the a number of daemons that are in the
 *  same area the KB together
 *
 */

package p3 DmnPack,

import java util *,

public class Node {

    public String area, // the area the daemons contained within it are related to
    public boolean hasChanged,    // a Boolean flag  If a token is changed in a daemon
            // in this node this flag is set  This prevents unnecessary searching of nodes that nothing has
                    // changed  This flag is  normally checked when the BB is checking for daemons to
   execute   public Vector segs = new Vector(), // store all the segments i e  daemons relating to the node

    // constructors

    public Node(String value) {
        area = value,
       hasChanged = false,
    }

    public Node() {
        area = " ",
        hasChanged = false,
    }


    public Node(String value, String state) {
        area = value,
        hasChanged = state startsWith("true"),
    }



     // Some of the methods of the class

    public String createSegments(StringTokenizer t) {
        String temp = t nextToken(),

        while (temp equals("Segment")) {
            int seg = Integer parseInt(t nextToken()),
            temp = t nextToken(), // pass node name and state
            int [] tokenIDs = new int [Integer parseInt(t nextToken())],
            for (int i=0,i<tokenIDs length,i++) {
                tokenIDs[i] = Integer parseInt(t nextToken()),
```

```
            }

            Segment newSeg = new Segment(seg,temp,tokenIDs);
            temp = newSeg.createScenarios(t);
            segs.addElement(newSeg);
        }
      return temp;
}


public void startScenario(int sID) {
 /*
  * Method to create a new scenario within each segment contained within the node
  */
       for (int i=0;i<segs.size();i++) {
          ((Segment)(segs.elementAt(i))).startScenario(sID);
       }
}


 public void stopScenario(int sID) {
  /*
   * Method to delete a scenario within each segment contained within the node
   */
       for (int i=0;i<segs.size();i++) {
          ((Segment)(segs.elementAt(i))).stopScenario(sID);
       }
  }


  public int daemonSearch(int dID) {
  /*
  * search the segments for a specific daemonID. If it finds one it returns its index. However if it is not
  * found it returns -1 which will be interpreted as such from where it has been called.
  * @param : dID : the daemonID
  * @retur : int : the index of the daemonID in the node
  */

       int i = 0;
       while( (i<segs.size())&& (dID != ((Segment)(segs.elementAt(i))).daemonID())) {
          i++;  // while daemonID is not found continue
       }

       if (i<segs.size()) { // if i< segs.size this means the while didnt reach the end
         return i;        // of segs and so must have found daemonID thus return it
       } else { return -1; }// if not found return -1 which indicated this.
  }


 public void updateToken(int pos,tokenInfo tInfo) {
 /*
  * mothod to update a token within a node. This method forwards the call to the segment class
  * @param : pos : the position od the daemonID the token is in
```

```
 * @param   tInfo   the information about the token i e  ID and scenarioID etc
 */
    hasChanged = true,
    ((Segment)(segs elementAt(pos))) updateToken(tInfo),
 }


 public void updateAdvice(int pos, DmnOutput adv){
 /*
 * Method to update the advice within a specific segment within the node
 * @param   pos   the position of the segment within the node
 * @param   adv   the actual advice object to be stored
 */
     ((Segment)(segs elementAt(pos))) updateAdvice(adv),
 }


 public Vector availableToExecute() {
 /*
 * ask the node to construct a vector of alll the daemons that are available to execute  It does this by
 * asking each segment to return any scenario information if they can execute and copying them into
 * the vector total
 * @retur   Vector   the vactor of all the scenarios that can execute  This
 *              vector contains BBExeInfo object only
 */
     Vector total = new Vector(),
     for (int i=0, i<segs size(), i++) { // find all daemons capable of executing,
       if (((Segment)(segs elementAt(i))) hasChanged()) {  // if tokens in segment have changed
         ((Segment)(segs elementAt(i))) hasChanged(false), // reset hasChanged variable
         Vector temp = new Vector(),  //temporary vector to store info from each segment
         temp = ((Segment)(segs elementAt(i))) AvailableToExecute(),// copy details of scenarios that
                                                 //can execute
         for (int j=0, j<temp size(), j++) {    // copy temp vector into total
           total addElement(temp elementAt(j)),// thus all info is stored in one vector
         }
       }
     }
    return total,
 }


 String getNodeInfo() {

     String nodeInfo = "",
     nodeInfo =",Node,"+ area + "," + hasChanged,
     for(int i=0,i<segs size(),i++) { // for all elements do
           nodeInfo = nodeInfo + "," +
           // access the element at index i and store the mfo related to  the segments in a string
           ((Segment)(segs elementAt(i))) getSegInfo(),
     }
     return nodeInfo,
 }

}  // end of class Node
```

Segment Class

```
/*
*     @author Eamon Gaffney
*     @author DCU
*
* responsible for the segments within the BB structure. Each segment contains the daemonID, a list of
*     the tokenIDs and an output area for the Daemons to write their into. Thus each segment
*     has a number of scenarios under its control.
*/

package p3.DmnPack;
import java.util.*;

public class Segment {

    private int daemonID;       // unique daemonID
    public Vector scenarios = new Vector(); // vector to store the scenarios
    public int [] tokens;       // variable to stored the dependen tokens for the segment
    private boolean hasChanged; // flag to indicate if a token within the segment
    // has changed. This is used to stear the search  to the changes and allow it to ignore those that haven't

    // constructor
    public Segment(int dID, int tID[]) {
        daemonID = dID;
        tokens = tID;
        hasChanged = false;
    }


    public Segment(int sID, String state, int [] tID) {
        daemonID = sID;
        hasChanged = state.startsWith("true");
        tokens = tID;
    }


    public String createScenarios(StringTokenizer t) {
        String temp = t.nextToken();

        while (temp.equals("Scenario")) {
            int sID = Integer.parseInt(t.nextToken());
            boolean state = (t.nextToken()).startsWith("true"); // pass node name and state
            DScenario s = new DScenario(sID,state,tokens);
            temp = s.createSlots(t);
            scenarios.addElement(s);
        }
        return temp;
    }


    public void startScenario(int sID) {
    /*
     * Method to create a new scenario within the segment
     */
         DScenario temp = new DScenario(sID, tokens);//((DScenario)(scenarios.elementAt(0))).tokenID);
```

```
        scenarios.addElement(temp);
}


public void stopScenario(int sID) {
 /*
  * Method to delete a scenario within the segment
  */
     int i=0;
     while ((i< scenarios.size()) && (sID != ((DScenario)(scenarios.elementAt(i))).scenarioID())){
          i++;
     }
     if(i<scenarios.size()) {
       scenarios.removeElementAt(i);
     } else { System.out.println(" scenario not found");  }
}



public void updateToken(tokenInfo tInfo) {
 /*
  * update the token in a scenarios. This method forwards the info on to the appropriate scenario
  * @param : tInfo : the information that tells where specifically the token
  *                  is ment for
  */

     int sID = tInfo.scenarioID();
     hasChanged = true; // indicate that the segment contains changes
     int i=0;
     while ((i< scenarios.size()) && (sID != ((DScenario)(scenarios.elementAt(i))).scenarioID())){
          i++;
     }
     if(i<scenarios.size()) {
       ((DScenario)(scenarios.elementAt(i))).updateToken(tInfo);
     } else { System.out.println(" scenario not found");  }
}



public void updateAdvice(DmnOutput adv){
/*
 * Method to update the advice of the a specific scenario within this segment
 * @param : adv : the advice object to be stored.
 */
     int sID = adv.scenarioID;
     // find the scenario index the advice relates to
     int i=0;
     while ((i< scenarios.size()) && (sID != ((DScenario)(scenarios.elementAt(i))).scenarioID())){
          i++; // search for the scenario. i will indicate its position in the vector
     }         // if i < the size then the while didnt reach the end and so must have
     if(i<scenarios.size()) {  // been found
       ((DScenario)(scenarios.elementAt(i))).updateAdvice(adv); // method call to forward advice
     } else { System.out.println(" scenario not found");  }// forward on the advice
}
```

```
public Vector AvailableToExecute() {
 /*
 * method to construct a vector of all the scenarios that are capable of
 * executing under its control The information is stored in BBExeInfo object
 * which contain their daemonId, scenarioID etc
 */
     Vector temp = new Vector(), // vector to store all the BBExeInfo object to be returned
     for (int i=0,i<scenarios size(), i++) { // search all scenarios
        if (((DScenario)(scenarios elementAt(i))) hasChanged()) { // if it has changed do
          ((DScenario)(scenarios elementAt(i))) hasChanged(false), // ask the scenario to return info
          BBExeInfo bRef = ((DScenario)(scenarios elementAt(i))) availableToExecute(),
          if (bRef != null) { //if not all tokens are available null is returned
                    temp addElement(bRef), // if OK add object to vector
          }else{// System out println(" null scenario "), }
        }
     }
     for (int i=0,i<temp size(),i++) { //add the daemonID to all since they are related to
        ((BBExeInfo)(temp elementAt(i))) daemonID = this daemonID,// the same daemon
     }
     return temp,
}


 public String getSegInfo() {
     String segInfo = "",

     segInfo = (",Segment,"+ daemonID + ","+ hasChanged + "," +tokens length),
     for (int i=0,i<tokens length,i++) {
        segInfo = segInfo + ","+ tokens[i],
     }

     for (int i=0, i<scenarios size(), i++) {
        segInfo = segInfo + ((DScenario)(scenarios elementAt(i))) getSnroInfo(),
     }
     return segInfo,
}


} // end of segment class
```

Scenario Class

```
/*
*    @author Eamon Gaffney
*    @author DCU
*    @version 3 2 7 2  date          29/10/98
*
*    This class is responsible for the specific scenario of a daemon  On creation of a segment the token info
is stored in
*    the default scenario  Each scenario store a list of the tokens and their states
*/
package p3 DmnPack,
import java util *,


public class DScenario {

    public int scenarioID,  // specific scenario identifier
    public Slot [] slots,  // each scenario contains a number of slots contained in there is token details
    private DmnOutput advice,     // actual deaemon advice
    private boolean hasChanged, // flag to idicate if tokens in this scenario have changed
    public int [] tokenID,
    public int [] tokenVal, // This method will store the token values
    // The advantage for using these arrays is purely for efficiency of running
    // it is better to have then here than to go serching and constructing them


    // Constructors

    public DScenario(int sID, int [] tID) {
        scenarioID = sID,
        slots = new Slot[tID length],
        hasChanged = false,
        tokenID = tID,
        for (int i = 0, i<tID length, i++) {
           slots[i]= new Slot(tID[i]), // create a slot for each tokenID

        }
        advice = null,
    }


    public DScenario(int sID,boolean state, int [] tID) {
        scenarioID = sID,
        slots = new Slot[tID length],
        hasChanged = state,
        tokenID = tID,
        for (int i = 0, i<tID length, i++) {
          slots[i]= new Slot(tID[i]), // create a slot for each tokenID
        }
        advice = null,
    }
```

```
public void updateToken(tokenInfo tInfo){
/*
 * Method to update a specific token within the scenario  The token ID is stored in the tInfo
 */

     hasChanged = true,
     int tID = tInfo tokenID(),
     int tValue = tInfo tokenVal(),
     int i=0,
     while ((i<slots length) && (tID !=slots[i] tokenID)) {
         i++,
     }
     if(i<slots length) {
      slots[i] tokenUpdate(tValue),
     } else { System out println(" Token not found"), }
}


public void updateAdvice(DmnOutput adv){
/*
*  Method to update the advice in the specified scenario
* @param   adv   the advice object
*/
        advice = adv, // add advice to the scenario
}



public BBExeInfo availableToExecute() {
/*
 * check if scenario is available to execute  The scenario is available if all the slot states is set to true
 * Thus details are placed in the BBExeInfo class and all the slot states are reset to false
 * @retur   dInfo   the details of the scenario available to execute
 */
     BBExeInfo dInfo,
     int j=0,
     tokenVal = new int[slots length],
     while ((j<slots length) && (slots[j] tokenValue != -1)) {
         j++, // j == slots length only if all slots have changed
     }
     if (j<slots length){
       return null,
     } else { // if all tokens have changed and are available

          hasChanged = false,
          for ( int i=0, i<slots length,i++) { // reset all slots to false
             slots[i] hasChanged(false),
              tokenVal[i] = slots[i] tokenValue,
          }

          dInfo = new BBExeInfo(scenarioID,tokenID, tokenVal),
     }
     return dInfo,
}
```

```
public String getSnroInfo() {
/*
 * Method to get the information from all the slots  THis method is used during the saving of the BB
 * @param None
 * @returnString   the information retrieved is stored in a string
 */
     String snroInfo = "",

     snroInfo = ",Scenario,"+scenarioID+","+hasChanged,
     for (int i=0, i<tokenID length,i++) {
        snroInfo = snroInfo +","+ slots[i] hasChanged+","+slots[i] tokenValue,
     }
     if (advice !=null) {
          snroInfo = snroInfo + ",Advice," + advice getAdviceInfo(),
     }
     return snroInfo,
}



public String createSlots(StringTokenizer t) {
/*
 * Method to create the scenarios from the information passed in  This information
 * takes the form of a string tokenizer which tokenizes the information from
 * the string from the saved BB file
 */

     for (int i=0,i<slots length,i++) {
        slots[i] hasChanged = (t nextToken()) startsWith("true"),
        slots[i] tokenValue = Integer parseInt(t nextToken()),
      }
      String temp = t nextToken(),
      if (temp equals("Advice")){
        advice = new DmnOutput(Integer parseInt(t nextToken()),
        Integer parseInt(t nextToken()),Integer parseInt(t nextToken())),
        advice storeAdvice(t),
        temp = t nextToken(),
      }
      return temp,
 }

} // end of scenario class
```

Slot class

```
/**
 *        @author Eamon Gaffney
 *        @author DCU
 *
 *        the information relating to the token value of a specific daemon  Within this class the
 *        tokenID is stored  The purpose of this class is to group the a number of daemons that are in the
 *        same area the KB together   Token value attribute added  THis stores the value of the token EG
 *
 */

package p3 DmnPack,

public class Slot{
    public int tokenID, // the unique identifier for the token
    public boolean hasChanged, // the state of the token, true indicates the token has changed
    public int tokenValue,

    // constructor

    public Slot(int tID) {
        tokenID = tID,
        hasChanged = false,
        tokenValue = -1,
    }



    public void tokenUpdate(int value) {
     /*
     * method to update the state of the token to true indicating the token
     * has changed
     */
        hasChanged = true,
        tokenValue = value, // This stores the actual value of the token
    }


} // end of slot class.
```

JessParser class

```
/**
*     @author Eamon Gaffney
*     @author DCU
*
*          responsible for executing a jess daemons rules  It is passed a reference to an instance of a jess
*      daemon in the DL  it then opens this this daemons rules and processes them, returning advice
**/

package p3 DmnPack,
import java io *,
import p3 DmnPack jess *,


public class JessParse {

    public DmnOutput adviceToGive,          // advice to return
    public JessParse(){}

    public DmnOutput ProcessRule(BBExeInfo dInfo, Daemon liveDaemon, int [] tokenValues,
String P_DaemonFileLocation) {

        ByteArrayOutputStream bs = new ByteArrayOutputStream(), // output device from the jess files
        PrintStream out = new PrintStream(bs),
        PipeAdvice id = new PipeAdvice(out,System in,bs), //create the info passing mechanisms

        FileInputStream fis,
        String dmnfile = P_DaemonFileLocation + liveDaemon daemonID() + " dmn",
        Rete rete = new Rete(id), // create an instance of the Rete algorithm to setup the
                        // variables for the JESS files to be processed
        try {
          fis=new FileInputStream(dmnfile), // tell the jesp compiler which file to read
          Jesp j = new Jesp(fis,rete), // pass the file and the options to the compiler
          try {
            j parse(false), // parse the file
            String command = "",     // string to store the jess command that must be constructed
            command = "(assert(daemon0",
            for (int i=0,i<dInfo tokenIDs length,i++) {  // pass the info to the clp
               command = command +"("+"_" + dInfo tokenIDs[i]+" "+ tokenValues[i]+")",
            }
            command = command + "))",
            rete executeCommand(command),
            rete executeCommand("(run)"),  // run the CLIP
            String ans = bs toString(),    // convert advice into a string
            AdviceText adv = new AdviceText(ans),
            adviceToGive = new DmnOutput(adv), // package up the advice
           }catch (ReteException E){System out println(E),}
        }catch(Exception ex){System out println(ex),}
        return adviceToGive,          // return the advice
    }


} // end of jess parser class
```

Appendix B The CORBA interfaces

Daemon Supervisor/Kernel Interface

```
exception P3Exception {

//Internal Prompter Exception for use in the interfaces
                string      exceptionReason,        // reason for the Exception
                short       exceptionCode,          // unique exception code
                short       exceptionLevel,         //Level of importance of Exception
};


interface Kernelcall;

/**
 *              ============================
 *              Section 1 - Interface Objects
 *              ============================
**/



/**              STATE ARRAY
 *              ==========
 *
 * This state-array, is periodically downloaded by the DS, acts as a buffer
 * or temporary storage area where the Kernel places tokenIDs that have changed
 * in the project workspace  This list is downloaded by the DS and reset by the Kernel
 *
**/

interface tokenInfo {
// A token object holds all the info  needed to identify a token's context

        attribute long projectID,           // The unique projectID
        attribute long scenarioID,          // The unique scenarioID
        attribute long tokenID,             // The unique tokenID
        attribute long tokenVal,            // The value of the token
};


typedef sequence<tokenInfo> stArray,        // create an array or sequence of these tokenInfo objects


interface StateArray {
// The State Array is a sequence of token objects
        attribute stArray sArray,
};
```

```
/**              Advice Object
 *               ==============
 *
 * This is the Advice Object that will be used to communicate the daemons advice
 * to the rest of the tool  These objects are stored in as a sequence in
 * the advice table
 *
**/

interface AdviceObj {
// This is the AdviceObj interface

        readonly attribute long projectID,  // The unique projectID
        readonly attribute long scenarioID, // The unique scenarioID
        readonly attribute long daemonID,   // The unique daemonID
        readonly attribute long adviceID,   // The adviceID of the advice
        readonly attribute string dArea,    // The area the advice pertains to
        readonly attribute string advice,   // The advice from the Daemon

};


// Sequence of advice objects make up the advice table
typedef sequence<AdviceObj> aTable_vector,


interface AdviceTable {
// This is the actual advice table interface

        readonly attribute aTable_vector aTable,
};







//              ========================
//              Section 2 – Kernel to Daemon Interface
//              ========================
//
// startProcess
// ============
// This is the initial startup call to the DS  It results in the DS starting up all its
// housekeeping processes  No parameters are passed or returned for this method  It is
// used once when the project tool is started  This method indicates the start of the process
//
// stopProcess
// ===========
// This method is used when the project is ending  It results in the DS performing some clean
// up operation and saving any unsaved information  It is also used to tell the DS to finish
// up all its processes.
//
```

```
// newProject
// ============
// This method indicates the start of a new project  It differs from the startprocess method in
// that we don't want the DS to restart all its threads of control but instead to create a new
// BB etc  and all the other tasks associated with a new thread  This also tells the DS to ignore
// any changes to the tokens since the project is new and the user has returned to the APM stage
// of development  The field "maxNo" is passed which contains the number of tokens stored in the
// project repository  The unique project identifier is passed to the DS with the parameter
// projectID to distinguish it from other projects  This number is needed for setup purposes in the DS
//
// startProject
// ===============
// thid method is used when the ds is told to create a project but is given blackboard information
// to be stored in the BB structure  This method is used when the BB was saved when the project was
// previously closed
//
// stopProject
// ===========
// This method differs from the stopProcess method in that it signifies that the project's development
// has ended and all relative information relating to it must be saved  This does not mean that the
// overall process has ended however  The unique ID of the project must be passed to signify which
// project must be ended, if multiple projects are running
//
// createScenario
// ============
// This method is used when a new scenario is created  It results in the required structures being
// created to store the relevant information relating to it
//
// deleteScenario
// ============
// This call tells the DS to end a particular scenario  The unique ID is passed which tells the DS
// which Scenario the Kernel wants stopped
//
// recogniseChange
// ===============
// This method results in the DS starting to monitor token states of a specific project with the ID
// projectID  This would be used when there exists enough information for the daemons to give advice
// on  It is called when the IPM stage of the current project is reached
//
// tokenChange
// ===========
// This method is used to inform the DS when a token or a number of tokens have changed
//
// getAdvice
// =========
// This method is used by the Kernel to retrieve the advice from the DS (i e  the advice table)  The
// return value is an instance of the class AdviceTable where the advice is stored minus the advice
// previously returned  Once the Kernel
// has retrieved this table the DS resets the table to null
//
//
//
```

```
interface DS {

        void startProcess() raises (P3Exception),
        void stopProcess() raises (P3Exception),
        void newProject(in long projectID),
        void startProject(in long projectID, in string bbData),
        string stopProject(in long projectID),
        void createScenario(in long projectID, in long scenarioID),
        void deleteScenario(in long projectID, in long scenarioID),
        void recognise Change(in long projectID),
        void tokenChange(in long projectID, in long scenarioID, in Kernelcall ref),
        AdviceTable getAdvice(),

};




//
//              ========================
//              Section 3 – Daemon to Kernel Interface
//              ========================
//
// getStateArray
// ============
// This method is used to download the "state-array" from the Kernel, which consists of a
// list of all the tokens that have changed since the array was last downloaded The method
// returns an instance of the class stArray, which contains the tokenIDs This class can be
// found in the IDL file in the interface of the same name
//
//
// adviceAvailable
// ==============
// This method will be used after daemons have finished execution The DS will execute this
// method to inform the Kernel that advice is stored in the "Advice-table", which the Kernel
// will have direct access to No variable is returned as this method just acts as a flag to
// show that advice is available

interface Kernelcall {

        StateArray getArrayState() raises (P3Exception),
        void adviceAvailable(in long projectID, in long scenarioID),
};
```

Daemon Library / Daemon Supervisor Interface

```
typedef sequence<long> Array,          // array to store a collection on long numbers


interface DaemonInfo {
// method used by the dl to pass daemon header information to the Blackboard which it need to set-up

        attribute long daemonID,       // a unique identifier for the daemon
        attribute string area,         // the area the daemon is an expert in
        attribute Array tokenIDs,      // the daemons dependent tokens
};


typedef sequence<DaemonInfo> daemonInfo_vector,        // creating an array of DaemonInfo


interface DependTable {
// this daemon information is passed to the Blackboard in the form of an array

        readonly attribute daemonInfo_vector table,
};


typedef long TokenIDArray[35],   // an array for the tokens

interface Daemon {
// the information stored in the daemon header

        attribute long daemonID,
        attribute string dName,
        attribute string dVersion,
        attribute string dOrigin,
        attribute string dArea,
        attribute long dGUI,
        attribute long dIE,
        attribute TokenIDArray TokenID,
        readonly attribute string P_DaemonFileLocation,
        readonly attribute long P_DaemonCount,

};


interface DL {
// the method calls the Daemon S can make on the daemon supervisor

        DependTable getDependTable(),
        Daemon getDaemon(in long DaemonID),
        void DLSetup(in string P_DaemonFileLocation, in long P_DaemonCount),
        oneway void killDL(),
};
```

Appendix C The OMT class diagram of the Daemon Architecture

This class diagram contains the full OMT diagram of the daemon architecture including class attributes and methods