

## **CHAPTER 6 SYSTEM ARCHITECTURE**

### **6.1 Introduction**

This chapter describes a high level view of the architecture of the proposed intelligent assistant system, its component modules and the interfaces between these modules.

### **6.2 Architectural Issues**

The design of the system architecture has been motivated by a number of issues previously discussed. This section will summarise these choices, prior to description of the system architecture.

In chapter 3, a number of candidate solution technologies were discussed - Decision Support Systems, Expert Systems, Expert Critiquing Systems, Blackboards and Intelligent Agents - with the proposal of a hybrid approach being set forth [O'Connor, 98]. This hybrid approach consists of a library of intelligent agents (multi-agent system), where each expert advisory agent is a specialist in an individual area of software project planning within the framework, on an overall tool which represents the Decision Support System. Communication amongst agents is facilitated by a Blackboard structure in which agents may write their conclusions or interim findings and the rest of the multi-agent community may read, thus allowing for effective inter-agent communication.

The uses of Java and CORBA have been described in chapter 4 as solution technologies to the issues of building a platform independent, distributed, client-server system, with the use of JESS being put forward in chapter 5 as an agent implementation language. The choices allow for the creation of a Java-based DSS system containing a JESS-based multi-agent community, in which the various components communicate via CORBA [O'Connor and Moynihan, 98].

### 6.3 Architecture Components

Figure 6.1 illustrates a high-level view of the system architecture being proposed to implement the intelligent assistant system. This consists of a user interface, the decision support system itself and the underlying knowledge base which contains the expertise and knowledge (ie. agents).

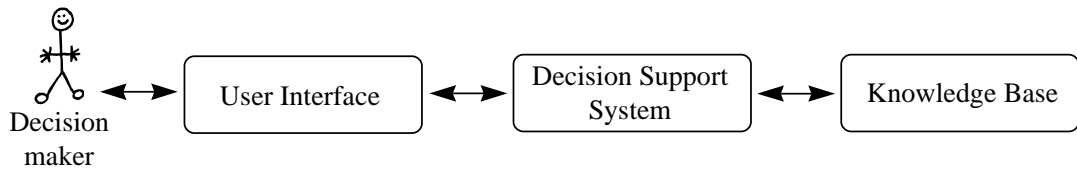


Figure 6.1 - High-level View of Architecture

These architectural components can be further subdivided into their constituent modules which represent CORBA (client and server) components. This view of the system architecture is illustrated in figure 6.2, followed by a description of the component modules.

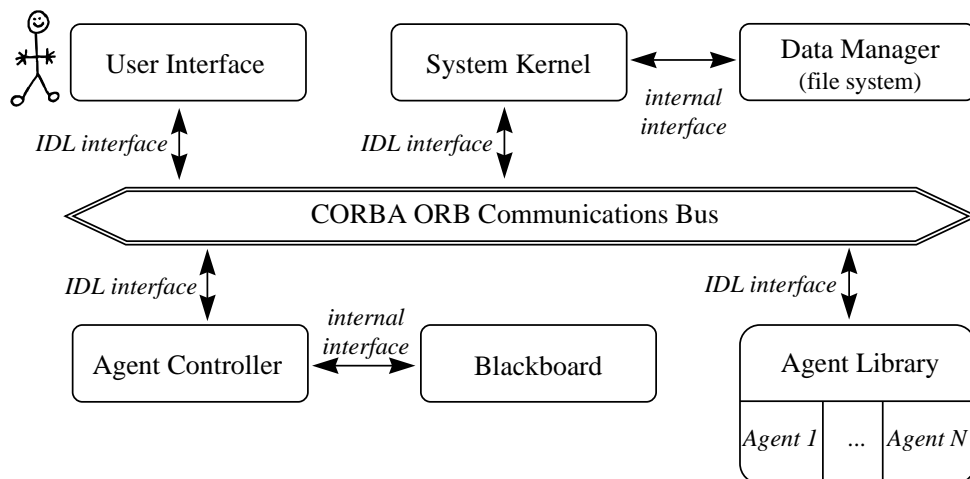


Figure 6.2 - Component Architecture

- **User Interface** - This component handles the management of all the screen elements (menus, dialog boxes, etc.), validates data entered by the user and passes on clear functional messages to the rest of the system.

- **System Kernel** - This is the core component of the system and handles all the processing and storage of user entered data. It manages all aspects of project plans and channels advice from the agents to the user.
- **Data Manager** - This manages all aspects of the mapping from the logical view of data to its physical storage. It is under the control of the System Kernel, through which all requests for data must be channelled.
- **Agent Controller** - This module acts as a controller (or supervisor unit) over the agent community and manages the scheduling and execution of agents, as well as governing write access to the Blackboard.
- **Blackboard** - This represents the global problem solving state of the system. Over time, agents produce changes to the Blackboard which lead incrementally to advice on the project under consideration. The Blackboard is under the control of the Agent Controller and all requests for data read / write must be channelled through the Agent Controller.
- **Agent Library** - All agents are contained in an Agent Library, but remain under the control of the Agent Controller. The purpose of the Agent Library is to manage the physical agents themselves and to service requests for agent interactions from the Agent Controller.

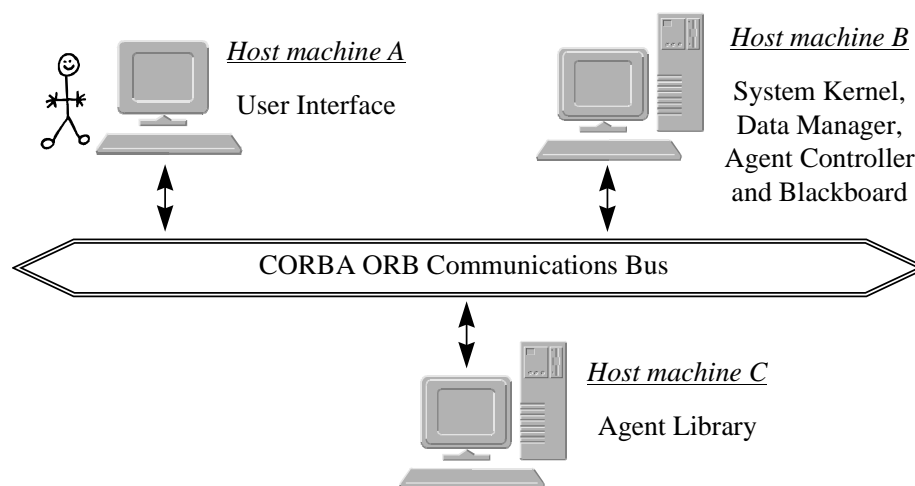


Figure 6.3 - Component Configuration

This CORBA based architecture allows for several possible implementation configurations in terms of what host machine a particular component may be installed on (in a network environment). Figure 6.3 illustrates one such configuration in which

the User Interface is installed on host machine A (say a typical desktop PC), the main DSS components (ie. System Kernel, Data Manager, Agent Controller and Blackboard) are on host machine B (say a standard network fileserver) and the Agent Library is installed on host machine C (say a backend server).

### **6.3.1 Component Interfaces**

The systems component modules communicate via interfaces (or API - Application Programmer Interface) and are of two types: public CORBA IDL interfaces and internal interfaces, which are not implemented in IDL.

IDL interfaces essentially define each components boundaries - that is, the services (or methods) the objects in that component can provide to other components. As IDL is completely language independent, objects specified using IDL are portable across languages, operating systems and networks. Thus the component modules with IDL interfaces may be implemented in different languages and reside on different machines (in a network or Internet environment). For example, in an extreme case, the four main modules (GUI, Kernel, Agent Controller and Agent Library) could reside on four separate host machines in a network, with all inter-module communication being automatically handled by the CORBA ORB.

The use of internal (or private) interfaces between modules provides a control mechanism over the global data structures of the Blackboard and file system and also provides for encapsulation of data with its appropriate control structure. This is achieved by allowing access to both the Blackboard and Data Manager only via calling an IDL method in the appropriate controlling module, which can then choose to pass on the request or modify it. In such a configuration, these two (non-IDL interface modules) must have their interfaces implemented using standard programming language constructs, thus both modules must reside on the same machine as their controlling module.

CORBA IDL is a declarative language used to define object APIs. When an IDL file (i.e. an object API specified in the IDL language) is compiled (by an IDL compiler), it produces two sets of code files: stub code to create proxy objects that a client can use for making invocations on object references of the interface types defined in the IDL file, and skeleton code for access to objects that support those interfaces. The role of the stub and skeleton are illustrated in figure 6.4. If the client and the target object are in the same address space, no extra code is required to communicate between them. If they are in different address spaces, on the same or different hosts, then extra code is required in the client to send the request to the server side, and extra code is required at the server side to accept the request and pass it to the target object.

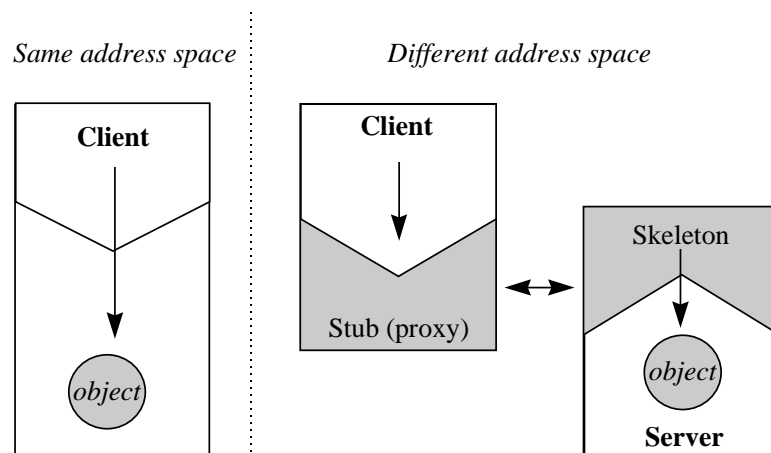


Figure 6.4 - Stub and Skeleton Code (Proxy Objects)

The following sections will describe each of the component modules in further detail, including a description of the modules functionality and its interface.

### 6.3.2 User Interface

The user interface (or GUI) has two tasks within the system; GUI management, which includes managing the screen elements, validating data inputs and passing on commands to the system kernel; and the management / control of the overall structure of screens and dialog boxes. This functionality is represented (at an architectural level) by three components, as illustrated in figure 6.5:

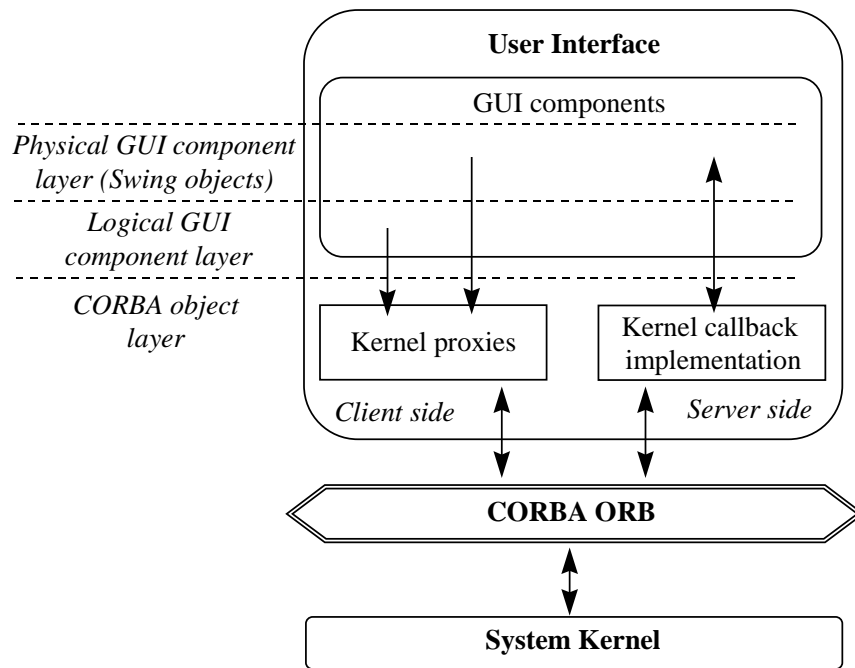


Figure 6.5 - User Interface Component Architecture

- **GUI components** - are divided into two layers: The first layer maps the proxy objects (CORBA object layer) to the logical component layer, which allows for the provision of a model of user interface components to be developed. The second layer (physical component layer) contains the actual GUI objects based on the Sun Swing toolkit. This layer (or separation) approach is based on the user interface management paradigms of *Model-View-Controller* and *Presentation-Abstraction-Control* model [Coutaz, 87].
- **Kernel proxies** - are Java objects which are created by compiling the IDL interface specification and are used to communicate with the Kernel.
- **Kernel callback implementation**<sup>2</sup> - handles service requests from the System Kernel, and represent objects which implement the interface as specified by IDL specification. This component is necessary to allow the Kernel get the attention of the GUI when, for example, an agent has produced some advice and it wishes it to be presented to the user.

<sup>2</sup> In CORBA, a callback is a method call from a server to a client, which reverses the client-server roles. They extend a clients architecture to include messages from their servers and are generally invoked when a server has something important to send to a client. The implementation code of the callback method exists on the client side, with a callback proxy method on the server side.

GUI management is an abstract layer of the user interface which exists between the actual GUI functions (as implemented using the Sun Swing JavaBeans components) and the rest of the modules. It's main areas of functionality are:

- **Data input checking** - Receive data inputted by the user and check it for validity. This may involve ensuring that an integer or real number is within certain bounds, checking that a date is valid and checking that text strings are correct when possible.
- **Data pre-processing** - Some types of data need to undergo an element of pre-processing prior to transmission via a CORBA ORB. For example, a date in the format DD/MM/YYYY would need to be converted to a long<sup>3</sup> number of the format NNNNNNNN.
- **Data request batching** - In many cases it is desirable (and more economical from a CORBA data transmission point of view) to batch groups of related data for subsequent processing. For example, many of the agents will require a series of data inputs simultaneously in order to execute - this user-entered data could be batched in small groups for communication to the rest of the system. Likewise, when advice is received from agents, it may be more desirable to batch the incoming advice data and present it in small groups to the user, instead of interrupting the user on a frequent basis with individual items of advice.
- **General screen house-keeping** - While most screen house-keeping functions (such as updating menus, windows management etc.) can be performed by the physical GUI components themselves, some functions require more complex processing. For example, whenever the GUI needs to process an item of data which is managed by the kernel, the GUI must first negotiate with the kernel to send/receive that data.

---

<sup>3</sup> In the CORBA IDL mapping scheme, the Java datatype int maps to the IDL datatype long.

### 6.3.3 System Kernel

The System Kernel provides the main interface between the User Interface component, the knowledge (advice) generator component provided by the Agents and the Data Manager. The Kernel represents the main data processing / management aspects of the system and acts as the overall controlling component of the system. It is represented (at an architectural level) by five main components and five further sub-components, as illustrated in figure 6.6:

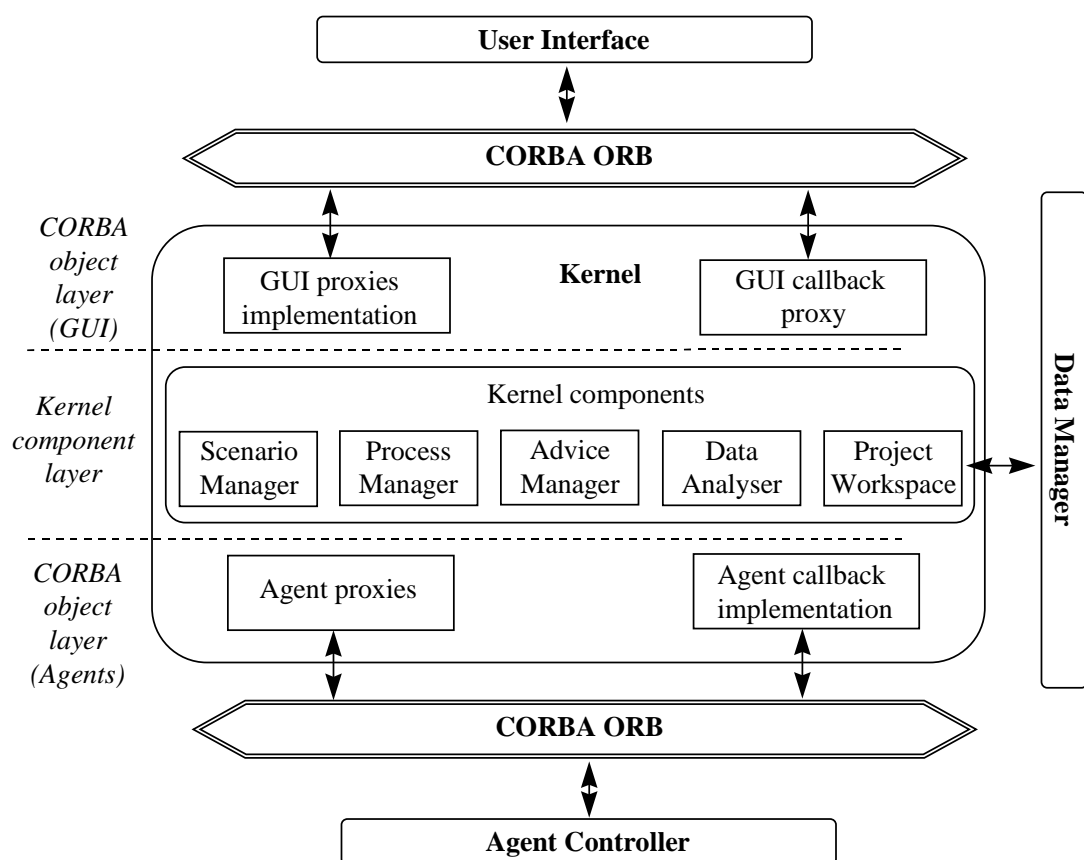


Figure 6.6 - System Kernel Component Architecture

- **GUI proxies implementation** - are the implementation of the Kernel Proxies as seen by the GUI. These objects implement the interface as specified by IDL specification and are used by GUI components to communicate with the Kernel to send or receive data.

- **GUI callback proxy** - are Java objects which are created by compiling the IDL interface specification. They handle service requests which are generated by the Kernel and destined for the User Interface and are represented on the User Interface side by the Kernel Callback Implementation.
- **Kernel components** - The main kernel components are as follows:
  - **Process Manager** - is concerned with assisting the user to select a suitable process template from the process repository (via data manager) to form the basis of a project definition.
  - **Scenario Manager** - Projects are categorized by scenarios, which are different views (work breakdown structures) of the same project. Two scenarios may be practically the same except for differences in two or three small pieces of information. For example, the duration of a sub-task of one scenario may be longer than another, thus enabling the user to 'experiment' with small variations in a project and possibly receive different advice from agents.
  - **Advice Manager** - packages advice received from the agents for subsequent delivery to the user interface. As advice is generated in an asynchronous manner by agents, it is the job of the Advice Manager to ensure advice is batched according to scenario under examination, etc.. The Advice Manager also ensures that advice is in the appropriate format (e.g. HTML) before being sent to the User Interface.
  - **Data Analyser** - undertakes any calculations required on project data, particularly in the creation and analysis of scenarios.
  - **Project Workspace** - acts as a temporary storage container for all project data (i.e. scenarios and related data) for any given user session. At the start of a session, the project under consideration will be retrieved by the Data Manager and placed in the Project Workspace until the end of the session, when it be stored by the Data Manager.
- **Agent proxies** - are Java objects which are created by compiling the IDL interface specification. They are used by Kernel components to communicate with the Agent Controller to send or receive data.

- **Agent callback implementation** - handles service requests generated by the Agent Controller and represent objects which implement the IDL interface. This component is necessary to allow the Agent Controller get the attention of the Kernel when, for example, an agent has produced some advice and it wishes it to be given to the kernel's Advice Manager.

### 6.3.4 Data Manager

The Data Manager handles all aspects of the mapping from the logical view of data to its physical storage and maintenance. It is under the direct control of the System Kernel, thus all requests for data must be channelled through the Kernel. Figure 6.7 illustrates the main components of the Data Manager:

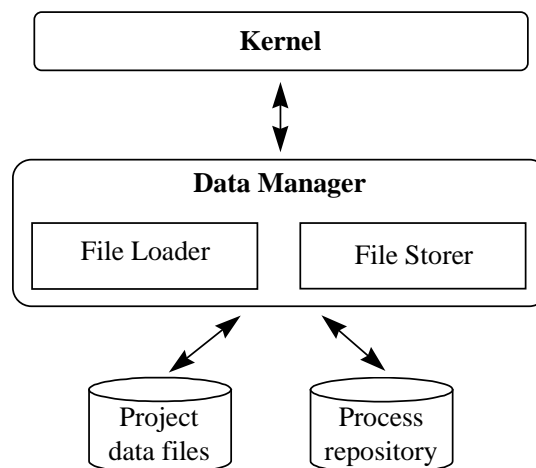


Figure 6.7 - Data Manager Component Architecture

- **File Loader** - reads in data from a physical file (project data file and/or process repository) and passes that data to the Kernel.
- **File Storer** - is used to write out to a physical data file (project data file) all data passed to it from the Kernel.
- **Project Data Files** - are the physical files containing project data.
- **Process Repository** - are the physical data files which contain all the process templates.

### 6.3.5 Agent Controller

The Agent Controller is a supervisor unit over the agent community, which manages the scheduling and execution of agents. In addition, it governs read / write access to the Blackboard. It is responsible for all communications between the Kernel and the Agents / Agent Library. At an architectural level it is represented by 3 main components and 4 further sub-components, as illustrated in figure 6.8:

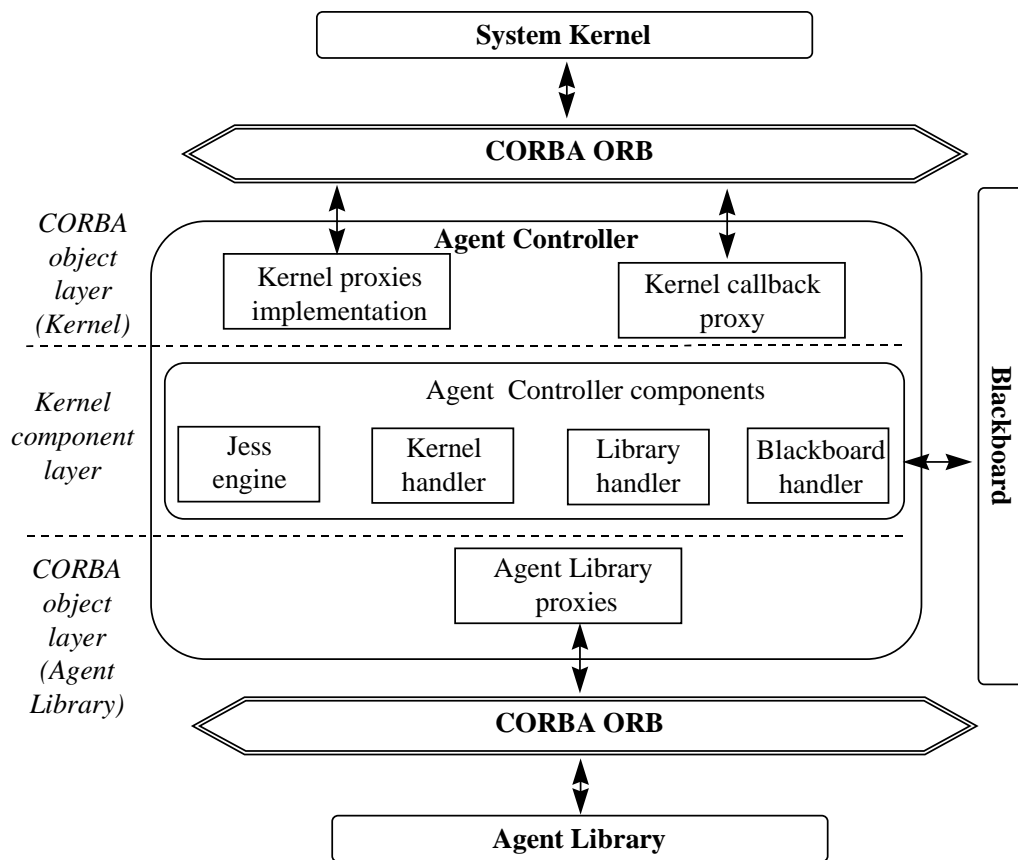


Figure 6.8 - Agent Controller Component Architecture

- **Kernel proxies implementation** - are the implementation of the Agent Proxies as seen by the Kernel. These objects implement the interface as specified by IDL specification and are used by the Kernel to communicate with the Agent Controller to send or receive data.
- **Kernel callback proxy** - are Java objects which are created by compiling the IDL interface specification. They handle service requests which are

generated by the Agent Controller and destined for the Kernel and are represented on the Kernel side by the Agent Callback Implementation.

- **Agent Controller components** - The main agent controller components are as follows:
  - **Kernel Handler** - is concerned with managing all communication between the Kernel and the other Agent Controller components. For example, when the JESS Engine requires actual (live) data values (from Data Manager) about a project, it is the Kernel Handler component that deals with processing the request. All such communication takes place via the CORBA ORB using the Kernel Proxies implementation
  - **Library Handler** - has a similar management role to the Kernel Handler. It manages all aspects of communication with the Agent Library, via the Agent Library Proxies.
  - **Blackboard Handler** - like the other handlers is responsible for the management of communications with the Blackboard. As the Blackboard interface is non-IDL (i.e. not using the CORBA ORB), communication is via a direct set of APIs. For example, when an individual agent wants to write information to the Blackboard, it is the Blackboard handler which manages the communication of this data to the Blackboard component.
  - **JESS Engine** - This component contains the inference engine and related modules for the JESS system. When an agent executes (under the control of the Agent Controller), its rules will be processed by the JESS engine and subsequent output processed by the system. As previously described, this architecture allows for a number of distinct inference mechanisms to be used, in which case they would reside as separate sub-components of the Agent Controller components.
- **Agent Library Proxies** - are Java objects which are created by compiling the IDL interface specification. They are used by Agent Controller components to communicate with the Agent Library to send or receive data.

### 6.3.6 Blackboard

The Blackboard represents the current problem solving state of the agents with respect to the project under consideration. For each area of expertise, the Blackboard holds state information on Tokens (data items in a process template / project model) which are manipulated by agents, state information on the agents and any advice agents have provided to date. The Blackboard is under the direct control of the Agent Controller. Any agents wishing to access the Blackboard must have read requests channelled through the Blackboard Handler component of the Agent Controller. Any write requests to the Blackboard are similarly processed, allowing co-ordination of read / write access, thus avoiding any possible access conflict.

To impose order on the information stored in the Blackboard it is necessary for the Blackboard to have a strictly defined structure. In keeping with traditional Blackboard framework design [Englemore and Morgan, 88], this Blackboard is organised as a tree structure where each node in the tree represents an area of project planning expertise. Figure 6.9 illustrates this tree structure with nodes from 1...N, where a node represents data stored about one specific area such as activity planning, risk analysis, etc.

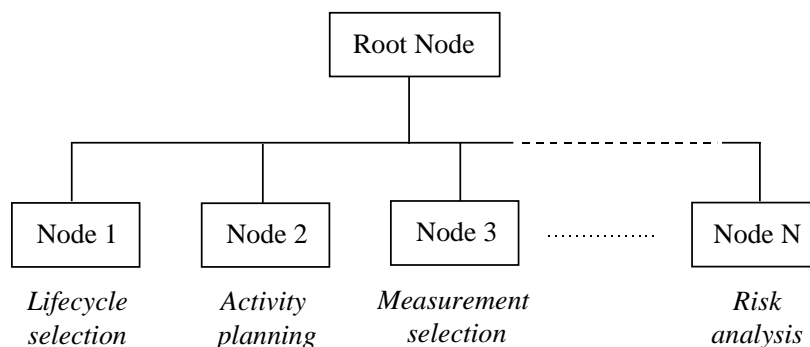


Figure 6.9 - Blackboard Node Hierarchy

As there may be many agents operating in any one area (ie. node in the tree), each agent is assigned a segment in a node within which there is a container for multiple scenarios. Figure 6.10 illustrates a detailed view of Node 1. Within this node there are

segments 1...N for each of the agents operating in this area. Likewise within each segment there are a number of containers 1...J for each of the possible project scenarios.

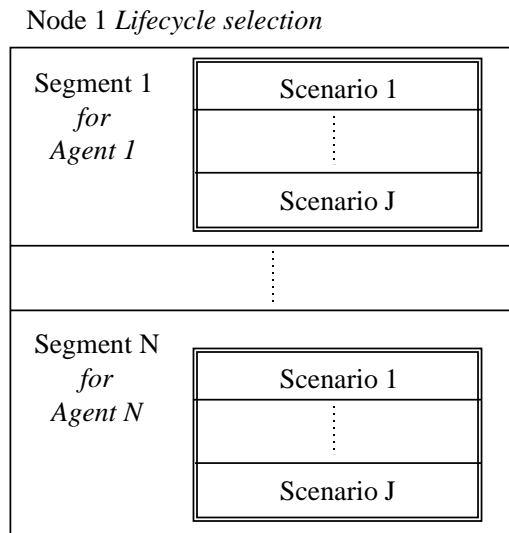


Figure 6.10 - Blackboard Segment Structure

In each scenario container there are a number of slots which hold state information on a specific agent for a specific scenario. These include a slot for each token an agent manipulates and a slot for any advice information an agent may have previously generated. Figure 6.11 illustrates a detailed view of segment 1 of node 1 and shows slots 1...K for each scenario.

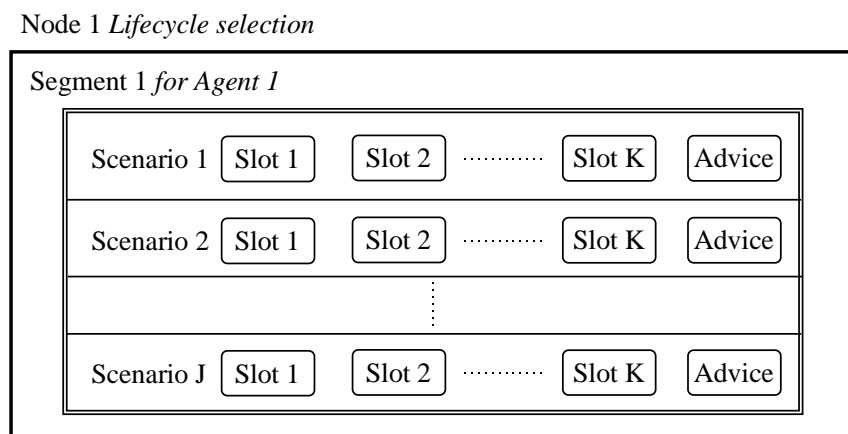


Figure 6.11 - Blackboard Scenario Structure

### 6.3.7 Agent Library

Agents are located in the Agent Library, but remain under the control of the Agent Controller. The purpose of the Agent Library is to manage the physical agents themselves and to service requests for agent interactions from the Agent Controller, where the physical agents themselves are represented as JESS rule scripts which are held on the file system. An example usage would be when the Agent Library interrogates the JESS scripts at the request of the Agent Controller and creates an instance of an agent for execution. The architecture of the Agent Library is illustrated in figure 6.12.

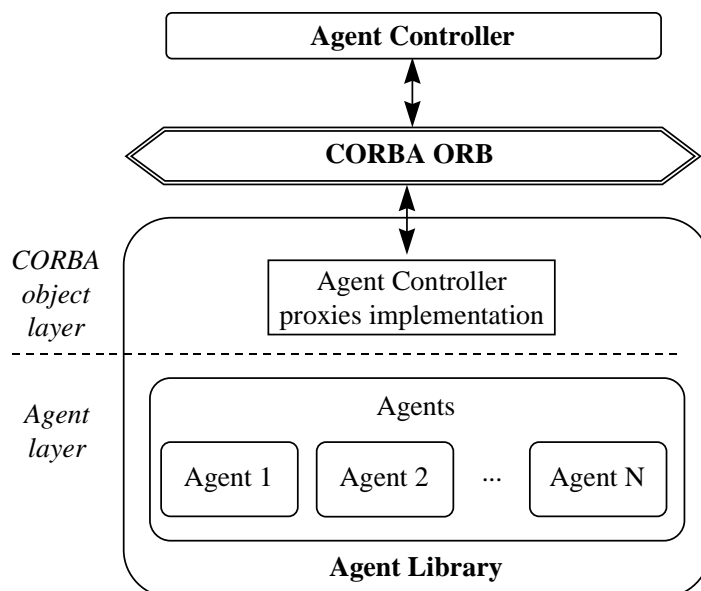


Figure 6.12 - Agent Library Component Architecture

- **Kernel proxies implementation** - are the implementation of the Agent Library Proxies as seen by the Agent Controller. These objects implement the interface as specified by IDL specification and are used by Agent Controller to communicate with the Agent Library to send or receive data.
- **Agents** - are the JESS rule scripts which implement the knowledge base. In addition to the actual rules, the script also contains identification information about the agent.

Individual Agents are represented as separate files within the system, which contain JESS rule scripts and identification / configuration information. The structure of an agent is shown in figure 6.13.

- **Agent Header** - contains all the basic information which an agent needs to identify itself to the system, including an identification and version number, and the area of proficiency of the agent.
- **Agent Tokens** - are a list of the data items in a process template / project model (that represent the actual data items held about a project) which the agent uses in its rule set.
- **JESS Rule** - is the actual JESS rule script.

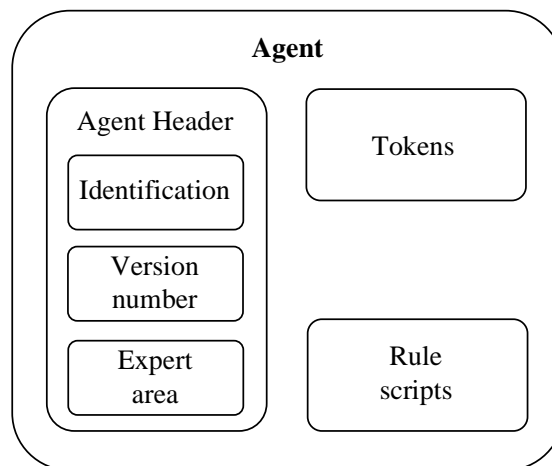


Figure 6.13 - Agent Structure

As can be seen from the above, agents are not directly embedded within the Agent Library or Agent Controller. Instead there is a link from the Agent Library to the set of available agents, i.e. agents installed on the file system. This architectural property allows for a dynamic agent population as the Agent Library will interrogate the file system at the initiation of a session and dynamically build a view of the set of available agents for the duration of that session. Furthermore, this data is passed to the Blackboard which creates the appropriate number of segments in its nodes, therefore any individual agent may be added or removed from the Agent Library by the addition or deletion of their corresponding agent file on the file system.

The architectural property of a resizable agent population is a key factor in the agent-orientated framework as it allows for dynamic updating of the knowledge base, where knowledge (agents) may be added, revised or removed without impact on the rest of the system. In particular, any such alteration in the knowledge base is completely independent of the other architectural components and does not require any explicit changes in system configuration. Thus the knowledge base may grow over time to take account of new expertise or techniques in software project planning or to add organisational specific knowledge. For example, if an organisation was planning a project to CMM level 2, they may wish to add a series of agents dedicated to that standard, or a company / project specific standard.

## 6.4 Review of Architectural Characteristics

Chapter 4 proposed a number of desirable architectural characteristics which should be taken into consideration when developing an architecture for an intelligent assistant system for less well understood domains, and in particular for the domain of software project management. In the light of this, and having discussed the proposed architecture above, it is necessary to pause and consider the how well this architecture satisfies these desirable characteristics.

1. **Specialism** - In this architecture, each agent is a specialist in a particular area of software project management, thus providing for a community of cooperating agents and the associated flexibility of the system.
2. **Information exchange** - This multi-agent approach provides for enhanced information sharing, as agents within the community can exchange information. Although each agent is a specialist in its own area, they have the ability to communicate with agents of other specialisms via the Blackboard.
3. **Collaboration** - The concern here is the overhead of both complexity and system performance due to collaboration. This has been streamlined by the use of a Blackboard attached to the Agent Controller, thus reducing both

system overhead and complexity of implementation, whilst providing for collaboration.

4. **Blackboard** - The blackboard approach was put forward in chapter 3 as being an aid to reducing both the level of complexity and volume of communications traffic. As stated in point 3 above, this has been incorporated into the proposed architecture.
5. **Supervision** - In an agent community, there does not have to be a single controlling agent. However, the indications are that the existence of such a supervisory agent led to more organised communications in the community. The provision of the Agent Controller will lead to more organised activity within the agent community.
6. **Hierarchy** - The concern here was that a hierarchy could diminish the flexibility of a system, thus agents should be allowed to operate regardless of rank. In the proposed system architecture, all agents are equal, but the Agent Controller may choose to impose a level of importance at run-time in accordance with local system performance, conditions, etc..
7. **Knowledge evolution** - The multi-agent approach to knowledge base implementation allows for dynamic updating of the knowledge base by adding, updating and deleting agents. In addition, CORBA provides for remote discovery of objects and services, and thus provides an ideal transportation bus for new agents.
8. **Data separation** - Is concerned with the separation of data and agents leading to a more open and flexible system, which is provided by the data communication between the Data Manager and the Agent Controller. This allows alterations in the data storage mechanisms and the agents themselves to be carried out independently of each other.
9. **Platform independence** - As the proposed architecture takes full advantage of both CORBA and Java, true platform independence is achieved.
10. **Multiple inference strategies** - As previously described, the proposed architecture allows for a number of distinct inference mechanisms to be employed. Each separate inference engine would reside as a separate sub-component of the Agent Controller.

11. **Component separation** - The separation of the agents components (which represent the 'intelligence' of the system) from the Kernel and GUI components (which represents the user-orientated framework or 'application') results in a flexible architecture, where either application or intelligent components may be altered independently.
12. **Multiple paradigms** - An agent approach allows the development of a number of different supporting frameworks. A traditional expert system approach may be employed by some agents, while others can adopt an expert critiquing system approach of critiquing user entered plans or data.

## 6.5 Summary

This chapter presented the architecture of the proposed intelligent assistant system. Each of the main architectural components have been further described, with the interfaces between these components outlined and the module interactions outlined.

Chapter 7 presents the issues surrounding the design and implementation of a prototype of the proposed system. It will outline both the design approach and the actual system design. In addition, the development of the prototype implementation of the system is presented.