BORG: A SYSTEM FOR THE ASSIMILATION OF LEGACY CODE INTO DISTRIBUTED OBJECTS

by

Nicholas Rahn

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

The University of Utah

March 1997

ABSTRACT

Heterogeneous computing creates a need for transparent communication between distributed software components. Transparent communication is often handled by preexisting low-level communication software packages. A high-level abstract communication architecture leverages off of a pre-existing low-level communication software package to facilitate the creation of distributed software components and to provide simple methods for their integration into distributed applications.

BORG is a high-level abstract communication architecture that fosters the evolution of legacy systems into an object model by providing tools that simplify the creation, acquisition and use of distributed software components within that legacy system. A distributed bulletin board assists applications in obtaining and integrating distributed software components which have been created from the legacy system. Distributed software components can be crafted from the legacy system via tools that abstract and encapsulate the functionality provided by the low-level communication software package.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

There is a need in today's distributed and heterogeneous computing world for transparent software communication. It may be necessary for one piece of software to communicate with others, whether they are currently executing or not, and whether or not they execute on a local or remote machine. The user should see a seamless interaction between the separate pieces of software, while invisible layers of communication code handle the technical aspects of their interaction.

It is no longer necessary or adaquate for an entire application to run in one process and on one processor. Applications should be dynamic and flexible in their execution, allowing multiple processes to handle the job formerly reserved for the old monolithic application. Moreover, these processes no longer need to be confined to a single processor. With a host of machines at one's disposal, it is advantageous to "farm out" the work load of an application to other available machines, processing in separate parallel processes, parts of the application that can be executed independently[6].

The ability of an application to run in multiple processes benefits the user through scalability and composeability[1]. The individual processes that constitute a multi-process application can be thought of as objects interacting with one another to form a single application. These objects are a division of the application's work load into smaller, more manageable parts. Individually, the objects provide one particular service. Yet, composing them can produce arbitrarily large and complex applications. For example, an object that provides an input/output service and an object that provides a geometric model viewing service may be composed to form an application that can display a geometric model that has been read from disk. The individual objects can also be replicated to enhance their power for a given application. For example, an object that performs a matrix multiplication could be used multiple times by an application that is computing large transformations.

If the objects that make up an application are actually separate processes, they may be executed on multiple processors to benefit the user in terms of performance optimization. Objects that can be executed in separate processes are referred to as *distributed objects*. An application that is comprised of one or more distributed objects is termed a *distributed application*. Distributed applications that must perform large computations can "farm out" individual parts of the computation to multiple distributed objects executing on separate processors. A distributed application that has several unique distributed objects can execute the objects individually on separate processors in order to tune performance to an acceptable level.

The user requires that the individual objects of a distributed application communicate seamlessly. Users do not want to be, nor should they be subjected to anything more than a simple transparent command or click. All of their actions should flow smoothly from one object to another, never concerning the users with the fact that the objects may be executing in separate processes on machines all over the world rather than contained in one large local process[6].

Whereas the user focuses only on the top level seamlessness of a distributed application, the programmer must be concerned with a broad spectrum of issues, ranging from low-level communication aspects such as sockets and marshaling, to the difficulties of high-level transparent interaction. The programmer must deal with writing low-level communication software, integrating it into high-level application code and finally making the user visible interaction appear seamless.

Fortunately, due to reusable object-oriented software, a programmer may not have to rewrite every aspect of a distributed application from scratch. There are many different systems and products that have abstracted and encapsulated the low-level communications necessary to help create distributed software. Implementations of the Object Management Group's (OMG) Common Object Request

Broker Architecture (CORBA)[10] and Douglas Schmidt's ADAPTIVE Communication Environment (ACE)[18] are examples of these *distributed system software packages* which help programmers focus on the design and implementation of a high-level abstract communication architecture.

A *high-level abstract communication architecture* facilitates the writing of distributed applications by providing a simple method for the creation of distributed objects and for their integration into a distributed application. An application programmer is presented with tools for creating new distributed objects and for combining those objects with others to form an arbitrarily complex distributed application. The tools use preexisting distributed system software packages and provide the programmer with a large degree of flexibility while requiring only limited knowledge of the tools' underlying structure.

This thesis will examine the aspects of one such high-level abstract communication architecture; **BORG**. Written for use by the Alpha_1 Geometric Modeling and Manufacturing Software System, the BORG System contains tools for creating distributed applications from legacy code. The BORG System employs a distributed bulletin board which helps applications obtain and integrate distributed objects that have been created from the legacy code. The BORG System tools take advantage of the abstracted and encapsulated functionality provided by a distributed system software package, namely an implementation of the Common Object Request Broker Architecture (CORBA). An application programmer can use the tools to create specialized distributed objects that are crafted from the legacy code, and ready for use in an application. The application programmer can also use preexisting distributed objects as part of the implementation of a distributed application in the BORG System.

The legacy code that is assimilated by the BORG System is software that performs useful tasks or actions for the user, but could benefit from being updated with more state-of-the-art software engineering methods. The tasks or actions performed by the legacy code are called *services*. BORG System tools convert such legacy code services into BORG System service providers. These service providers place an object-oriented interface on legacy code services that are not necessarily object-oriented. The BORG System tools produce service providers that can be used as distributed objects and composed into a distributed application.

The BORG System bulletin board allows applications to gain access to service providers. Applications that request access to a service provider through the bulletin board may use that access to employ the service or send a message to it. Applications will use descriptive tokens as qualifiers in their requests. Tokens describe the service providers whose access is sought or the set of service providers to which an application may wish to send a message.

Chapter 2 describes work related to the BORG System. This includes discussions of distributed system software packages as well as current methods of service composition with in the Alpha_1 System. Chapter 3 presents a general overview of the BORG System while also exposing its components. Chapters 4 through 7 explain the details of individual BORG System components and Chapter 8 presents the results of this work.

CHAPTER 2

RELATED WORK

The BORG System is based on concepts from artificial intelligence and distributed application programming environments. The "bulletin board" concept is similar to the AI HEARSAY-II blackboard system[2], whereas the high-level communication architecture has its roots in distributed programming environments such as ACE[18], OLE[5], CORBA[9] and OpenDoc[11].

2.1 Blackboard Systems

Blackboard systems were first developed for the HEARSAY-II project between 1971 and 1976 at Carnegie-Mellon University[2]. HEARSAY-II was a speech recognition system that used the blackboard approach to construct the meaning of a spoken sentence and perform the requested action. It was designed to be task independent, depending on the vocabulary and semantic specifications, but for experimentation purposes was instructed to respond to queries of a database containing artificial intelligence abstracts.

Blackboard systems consist of three parts: knowledge sources, a blackboard, and control structures. For each application, the domain of knowledge is split into small parts called knowledge sources, each of which contains a portion of the total information from the domain. These knowledge sources are independent entities, not aware of any of the other knowledge sources in the system. As such, they need only communicate with the blackboard and not with each other.

The blackboard is a globally accessible repository containing data produced by knowledge sources. It consists of multiple levels that express the different stages of hypothesis formation for the given domain. Thus, for the HEARSAY-II project which was designed for speech recognition, the blackboard levels were the wave form of the spoken command, sound segments, syllable classes, words, word sequences and phrases[15].

The control structure or scheduler determines when an individual knowledge source should become active, applying its knowledge to the current state of the blackboard. When the blackboard has been altered, it is the job of the control structure to determine which knowledge source or sources should next be applied to the blackboard and in what order. The control structure must have some basic information concerning the individual knowledge sources, as well as the general outline of a solution path in order to make decisions as to the order of knowledge source activation.

The general flow of a blackboard system begins with the input of data from an outside source into the lowest level of the blackboard. The control structure then determines which knowledge source should be applied in order to make the best gains toward a solution. The chosen knowledge source will then examine the current data on the blackboard and alter that data based on its specific knowledge of the domain. A final solution is obtained when the control source determines that there are no appropriate knowledge sources to apply to the current blackboard data.

The base design of the BORG System relates closely to the design of HEARSAY-II. Much like HEARSAY-II, the BORG System uses individual entities (BORG: service providers, HEARSAY-II: knowledge sources) which communicate with a common repository (BORG: bulletin board, HEARSAY-II: blackboard). The BORG System combines HEARSAY-II's blackboard and control structure into a central location, called the bulletin board, which ultimately controls the use of the individual service providers in the system.

2.2 ACE

The ADAPTIVE Communication Environment (ACE)[18], developed at Washington University by Dr. Douglas Schmidt, is an object-oriented framework designed to ease the development of distributed applications.

> By reusing the C++ wrappers and frameworks provided by ACE, developers are freed from spending their time reinventing solutions to commonly recurring tasks. In turn, this enables them to concentrate on the key higher-level functional requirements and design concerns that constitute particular applications.[18, p. 13]

The ACE wrappers abstract and encapsulate into an object-oriented framework the low-level UNIX system functionality that is used to create distributed applications. General categories of C++ classes abstract the following low-level services[18]:

- Event demultiplexing and service dispatching;
- Local and remote IPC mechanisms;
- Memory-mapped files;
- System V IPC mechanisms;
- Multi-threading and synchronization mechanisms;
- Explicit dynamic linking mechanisms.

Object classes from these categories may be combined in applications to obtain the required functionality. By using the object-oriented C++ interfaces, ACE is able to monitor the problematic type checking of ambiguous system variables at compile-time, rather than at run-time.

The ACE wrappers are used by the ADAPTIVE Service eXecutive (ASX) framework to simplify the creation of distributed applications. ASX provides common design patterns for development, while separating them into application independent and application specific concerns. The greatest benefit of ASX is that it "encourages the development of standard communication-related components by decoupling processing functionality from the surrounding framework infrastructure."[18, p. 14]

ACE is a mid-level communication system that has abstracted low-level communication functionality. The BORG System is a high-level communication architecture that uses a preexisting communication subsystem, such as ACE, to handle the more mundane aspects of the underlying communication code.

2.3 OLE

Object Linking and Embedding (OLE)[5], developed by the Microsoft Corporation for its Windows based applications, is a standard set of object services. OLE allows a document to consist of objects produced from many different applications. These objects can be anything from text to graphics to video. The programmer can incorporate these objects into an application using the OLE application programming interface.

Objects contain two types of data: presentation data and native data. Presentation data is that data necessary to display the object in a window. Native data are that data necessary for editing the object. As the Object Linking and Embedding name implies, objects can be either linked or embedded into a document. When an object is linked, only its presentation data are physically in the document. The object's native data are accessed via a link to its place of "permanent" residence (e.g., a file or disk). Alternatively, both the presentation and native data of an embedded object reside wholly in the document. Linked objects have the disadvantage that the document in which they are used cannot be transported outside the local file system, whereas embedded objects enlarge the document size since the whole object must be stored therein.

OLE supports visual "in-place" editing. This capability allows the user to activate and edit an object, whether inside the document or inside another object, without having to start a completely new application. Objects may be transfered between documents or between objects via "drag and drop" capabilities provided by OLE. Objects may even be converted from one application's format to another. Additionally, OLE allows programmers to define operations for an application that are accessible to other applications. For instance, a document containing a spreadsheet table can send a macro to a spreadsheet program, requesting that it sort and recalculate the table[5].

The BORG System gleans from OLE the concept of separate objects that are combined to construct one document. The BORG System's individual service providers can be combined to form a large application. Thus, just as OLE objects can be used in multiple documents to create a standard appearance, BORG service providers can be used in multiple applications to take advantage of pre-existing functionality.

2.4 CORBA

The Object Management Group (OMG) is a nonprofit consortium of industry leading software companies, vendors and individuals, who have come together in a joint effort to propose standards for an object-oriented application development environment based on object technology and distributed computing. The Object Management Architecture (OMA)[3] is the central outline of the OMG's work, dividing all object systems into four constituent parts for which the OMG will propose standards. These areas are [7]:

- Object Request Broker (ORB)
- Object Services (OS)
- Common Facilities (CF)
- Application Objects (AO)

The ORB is the central component of any object system, handling the communication between all objects in the system, regardless of their location, platform or implementation. The OMG's proposed standard for an ORB is the Common Object Request Broker Architecture (CORBA).

ORBs, and specifically CORBA compliant ORB implementations (that is to say, an ORB that has been implemented to comply with all of the specifications of the CORBA standard), handle the messages passed between objects. All interobject messages in the system will move from the sending (client/stub) object, through the ORB internals and on to the receiving (server/skeleton) object.

A CORBA object is defined as an object-oriented service plus a CORBA proxy object. Object-oriented services are contained within a CORBA server process. Typically, a client application will want to invoke an operation of some object-oriented service. CORBA handles this invocation with the use of a CORBA proxy object. A *proxy* provides access to other objects for the purposes of encapsulation or distribution. A CORBA proxy object is made up of a client side, the ORB, and a server side. The client side will receive the invocation request from the client program. Using the ORB, the client side will send the invocation message to the server side of the CORBA proxy object. The server side is connected to the actual object-oriented service and will invoke the appropriate operation of that object-oriented service (see Figure 2.1).

Since the client and server sides of the CORBA proxy object are often not located within the same process space, the ORB must handle any low-level communication requirements that may be necessary for their interaction such as delivery services, activation and deactivation of remote objects, method invocation, parameter en-

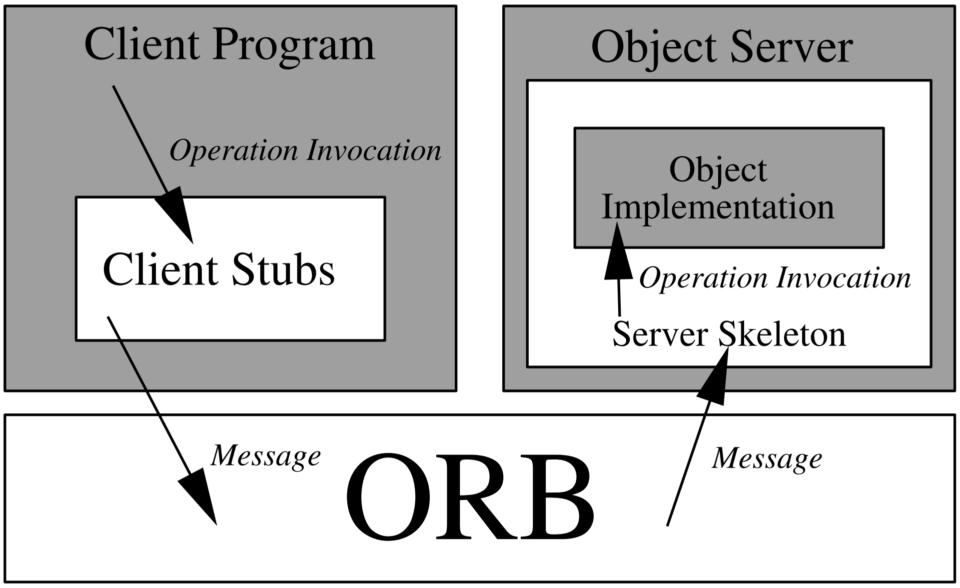

Figure 2.1. CORBA Invocation Roadmap

coding, synchronization, and exception handling. The client program is also not concerned with any of the system issues related to the internals of the CORBA proxy object, such as the location of the client and server sides, the programming language used to implement the object-oriented service, or what type of machine the object-oriented service is running on. All are handled transparently by the operation of the CORBA proxy object.

CORBA objects are defined using the OMG's Interface Definition Language (IDL). IDL is a language-neutral way of describing the public interface to an object-oriented service. An IDL description of an object-oriented service is similar to the declaration of a C++ class in that it contains the elements of the interface that will be publicly visible to clients. Using an IDL interface description, an IDL compiler will generate a CORBA proxy object (stubs and skeleton functions) in one of the languages specified in the OMG's IDL language binding. The client side of the CORBA proxy object is used in clients as a representation of the actual instantiation of the object-oriented service. Clients may interact with the client side of the CORBA proxy object as if it were a local instantiation of the object-oriented service. The interaction passes from the client stub, through the ORB and up to the server skeleton and the actual instantiation of the object-oriented service. Thus, through the use of IDL and CORBA proxy objects, a client is shielded from any of the ORB controlled system aspects of the client-object communication.

CORBA implementations abstract the low-level communication code necessary for transparent distributed object computing through the use of IDL and CORBA proxy objects. The BORG System uses a CORBA implementation as a tool for the creation, acquisition, and use of its distributed objects. By using CORBA as its distributed system software package, distributed objects in the BORG System communicate seamlessly and transparently.

2.5 OpenDoc

Component Integration Laboratories (CI Labs) is a vendor-neutral industry association sponsored by companies such as Adobe Systems, Apple, IBM, and

Novell. CI Labs is the owner and distributor of OpenDoc, a vendor-neutral standard for compound documents[11]. Compound documents are those fashioned with distributed, cross-platform component software. The component software are elements of a document such as text, graphics, spreadsheet cells, or buttons. The compound document scheme allows each of the components of the document to be edited "in place," simply by changing the focus. It does not require a new application to be started for each different component, but instead allows each type of component to do its own editing.

Each OpenDoc document consists of a shell and any number of individual components. The shell holds the components and dispatches any events to the currently active component. When any component type becomes active, usually via a single activation click, the editing tools (such as menus and tool bars) switch to ones useful for editing this type of component. All events will be handled by the editing methods of the active component type. For example: a text component will have editing tools to allow changing characteristics such as font size or line spacing. Events generated while a text component is active will be sent to the text component type methods that change the font size or line spacing. The text component will then redraw itself and wait for more events.

OpenDoc supports an efficient compound document storage mechanism. The default storage subsystem is Bento, the standard container for multimedia interchange adopted by the Interactive Multimedia Association, but can be replaced with any other applicable storage system that implements the OpenDoc meta-format[13]. Additionally, OpenDoc supports scripting based on the Open Scripting Architecture (OSA) standard that allows application-independent scripting. The scripting feature permits component types to transparently use OSA compliant scripting languages in their user interface design.

The Object Management Services of OpenDoc are based on the System Object Model (SOM)[8]. SOM handles dynamic object linking, giving OpenDoc multiple language and distributed object services support. SOM is also CORBA compliant, allowing OpenDoc to communicate with other distributed object systems that are also CORBA compliant. This means that an OpenDoc document may contain distributed components, running on machines other than the one holding the main shell. The BORG System uses this same idea for the creation of distributed applications. A BORG application may contain many service providers, each of which may be executing on a processor other than the one executing the main application.

2.6 Alpha_1

The BORG System attempts to facilitate the assimilation of the Alpha_1 Geometric Modeling and Manufacturing System into a distributed object model. Alpha_1, developed at the University of Utah, is a B-spline modeling software system, whose models can be converted into numerically controlled (NC) machining code and realized as manufactured physical parts. Alpha_1 models are generated through the Shape_edit Command Language (SCL) and an associated interpreter, `c_shape_edit`.

Alpha_1 currently uses an Inter Process Communication (IPC) scheme based on UNIX sockets to connect its `c_shape_edit` SCL interpreter and programs that display its models such as `tk3d` and `motif3d`. A user creates models with the `c_shape_edit` interpreter usually running within the GNU `emacs` environment. A model viewer can be connected to the `c_shape_edit` interpreter. When the SCL command `show` is called on a model, the `c_shape_edit` interpreter uses a UNIX socket connection to send the model to the viewer for display.

In this approach, the instance of the model in the `c_shape_edit` interpreter is serialized into a byte stream and written to the UNIX socket. The model viewer is continually checking that socket for any incoming data. When data arrive, they will be read in and converted back into the model instance. The viewer can then display the model.

This IPC scheme is limited in that it is only used within the `c_shape_edit/tk3d` environment. Most programs interact by saving their output to a file where other programs can access it. For example, the `c_shape_edit` interpreter allows a model to be saved to a file. A viewer program can read that file later and display the model. As another example, the `c_shape_edit` interpreter can save a model in a file. A user can read and display that model in a viewer which can create another file containing a viewing matrix. The model file and the viewing matrix file can be piped into the `render` program in order to generate a rendered image.

The BORG System allows the services provided by Alpha_1 programs such as `c_shape_edit`, `tk3d`, and `render` to be used by applications in an object-oriented manner. These services can be running as separate CORBA servers which an application can treat as normal objects. Instead of manually writing socket code to make a connection, applications are free to make member function invocations on these objects, transparently passing data between the individual processes as parameters and return values.

CHAPTER 3

SYSTEM ARCHITECTURE

This chapter discusses the overall architecture of the **BORG System**. We will first discuss the general concept of BORG, followed by a high level examination of the system architecture and its individual components.

3.1 What is the BORG System?

The BORG System is a software architecture that promotes the conversion of legacy systems into distributed systems. *Legacy systems* are old software that is still in use, but could benefit from reengineering using more modern methods. *Distributed systems* are collections of objects whose interaction (i.e., whose decomposition into objects, processes and processors) is transparent to the user. The objects of a distributed system are themselves distributed (*distributed objects*) and their interaction is also transparent to the user. The BORG System attempts to foster the evolution of legacy code into an object model by providing tools that simplify the creation, acquisition and use of distributed objects in the legacy system.

3.2 System Overview

The BORG System consists of a set of services, a set of clients, and an Agency. Clients use the Agency to request access to services. The Agency responds by contacting providers of a service and establishing a connection between the client and the service provider (see Figure 3.1). The provider of a service is referred to in the BORG System as a *functionality engine*.

A functionality engine defines an object-oriented interface to a service, and is often implemented as a C++[19] class. Functionality engines range from simple to complex, yet always provide only one specific service. For example, a simple functionality engine may only provide a basic service, such as multiplying two matrices. A complex functionality engine may provide a more sophisticated service such as geometric model viewing. This larger, more complex type of functionality engine may use smaller functionality engines as helpers in providing the service. In the case of a geometric model viewing service, it could use the base matrix multiplication functionality engine as a helper in its model viewing transformations.

The services provided by BORG System functionality engines are encapsulations of those found in the legacy system. BORG System functionality engines foster the conversion of legacy services into an object model by providing programmers with access to these services through object-oriented interfaces (see Figure 3.2). The object-oriented interface may be just an encapsulation of an underlying function based program, or it may be the interface to a class in an object-oriented system. Once encapsulated within the object-oriented interface of a functionality engine, the legacy service may be used as an object, even if the underlying code is not object-oriented.

The BORG System provides tools that allow functionality engines to be created and used as distributed objects. By using these tools, a functionality engine may be executed as a separate process while being composed with others to form a distributed application. The tools require only limited knowledge of the distributed system software package upon which they are based. The BORG System tools were created using CORBA as the distributed system software package. An instantiation of the C++ class that implements a functionality engine is embodied as a CORBA server and can be used in applications via a handle, also known as a CORBA proxy object. An instantiation of the C++ class that implements a functionality engine will be referred to as a *functionality engine instance*, or *FEI*. CORBA proxy objects provide applications with transparent access to FEIs through the use of an ORB.

Clients obtain handles to FEIs in order to employ their service. *Clients* are applications or functionality engines that use, as part of their implementations, the services provided by functionality engines. A client may be completely composed of functionality engines, or it may use them sparingly, possibly in only one

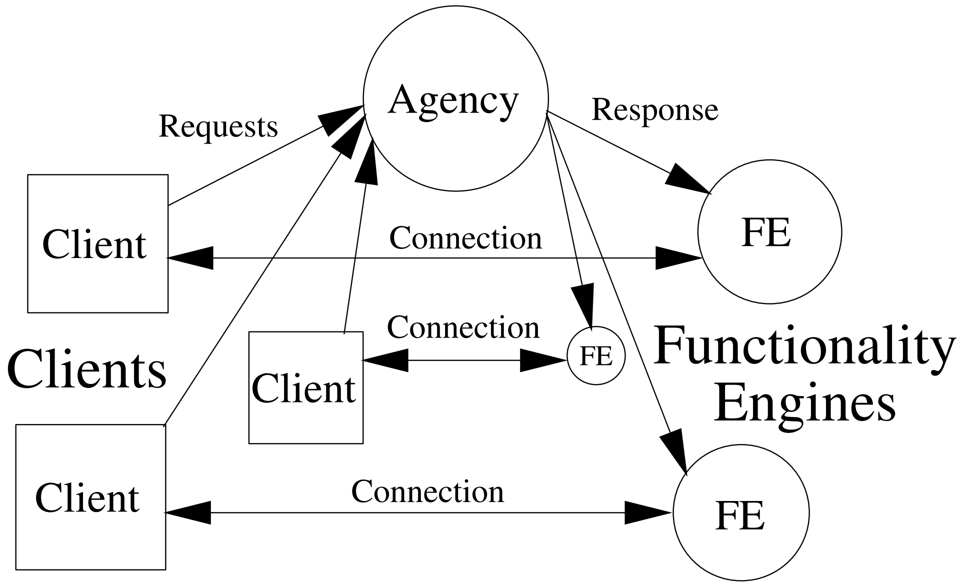

Figure 3.1. BORG System Architecture

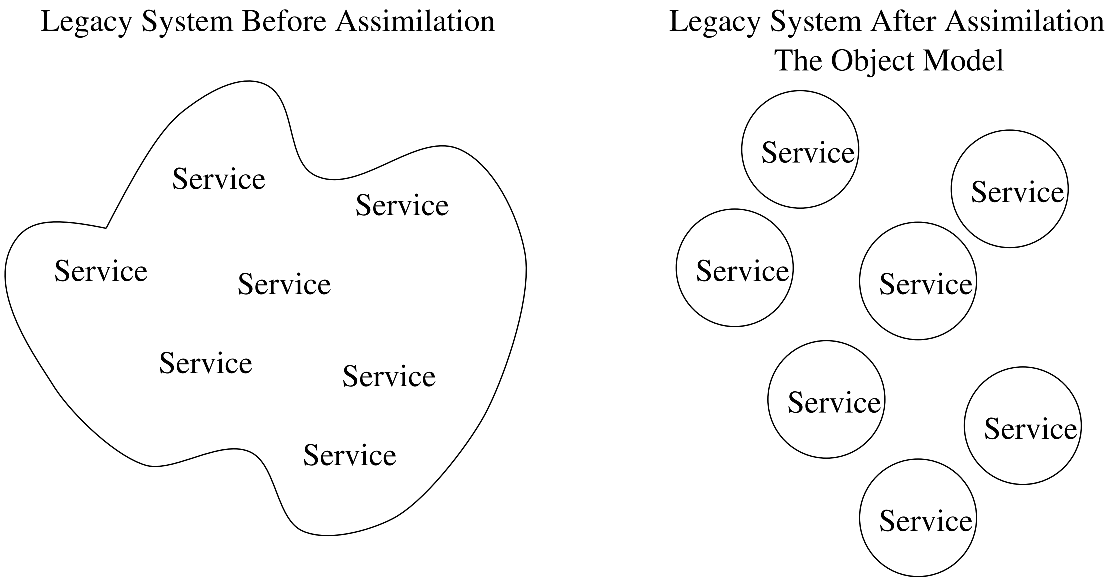

Figure 3.2. Legacy Services to Object Model

circumstance.

The *Agency* is a functionality engine whose service is to provide clients with access to other functionality engines. Its service includes providing clients with handles to FEIs as well as passing messages from clients to groups of FEIs. All clients in the BORG System have access to the Agency as it is the only way for a client to initiate contact with an FEI.

When a BORG System client needs the service provided by a functionality engine, it will contact the Agency and request that it be provided with an FEI. The request is made by passing to the Agency a token that names and describes the functionality engine. The Agency will use the ORB to find an FEI specified by the token and respond to the request by returning a handle (CORBA Proxy) to that FEI (see Figure 3.3). A BORG System client may also need to broadcast a message to a group of FEIs. In this case it will contact the Agency, requesting that it send a notification message to a group of FEIs. This request is made by passing to the Agency a token describing the group of FEIs and a descriptor of the message to be passed to them. The Agency will respond to the request by contacting all of the FEIs in the group and delivering the message to them (see Figure 3.4).

3.3 System Components

The components of the BORG System are the functionality engines, the Agency that provides access to the functionality engines, the tokens that describe functionality engines, and the clients that use the functionality engines. All of these components are essential to the creation of a distributed application in the BORG System.

First and foremost among the BORG System components are functionality engines. A functionality engine is a provider of a service and places an object-oriented interface on that service. The underlying service is part of the legacy system that is being assimilated by the BORG System and is not necessarily object-oriented.

The object-oriented interface placed on a legacy service is implemented as a C++ class. Using tools provided by the BORG System, the functionality engine can be created and used as a distributed object. The BORG System tools use CORBA as

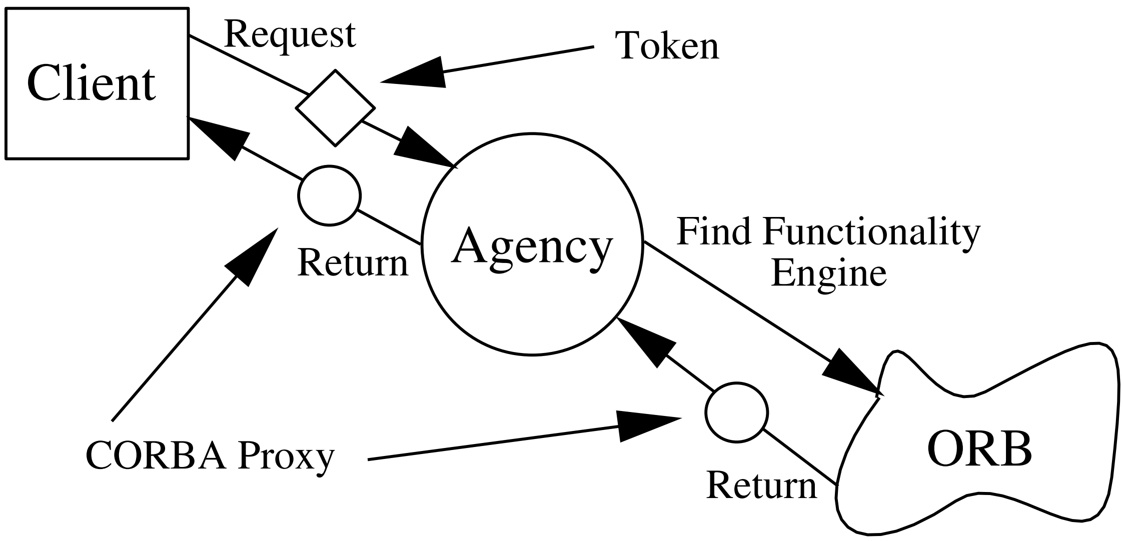

Figure 3.3. Request for Service

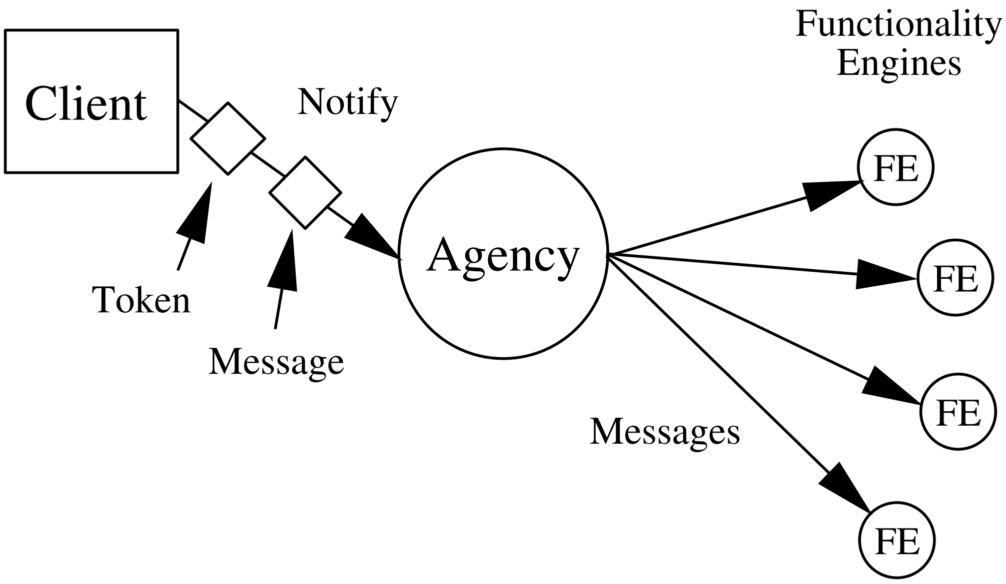

Figure 3.4. Request to Notify

the underlying distributed system software package, creating CORBA servers and CORBA proxy objects as part of the implementation of each functionality engine. The CORBA servers are embodiments of an instantiation of the C++ class that implements the object-oriented interface of a functionality engine. A CORBA proxy object is used as a handle to a CORBA server and using an ORB, encapsulates all of the code necessary for transparent interaction with the FEI embodied in the CORBA server.

The Agency is a functionality engine whose service is to provide access to other functionality engines. It was created using BORG tools and will almost always be used as a distributed object. Its CORBA server is executed as a daemon process on every machine in the BORG System. The Agency functionality engine's CORBA proxy object provides transparent access to its CORBA server. Clients wishing to use the Agency will access it via an instantiation of the `local_agency`, a C++ class that is a proxy to the CORBA proxy object of the Agency.

The Agency's service provides access to other functionality engines through requests made by clients. The requests have two forms: the request for the services provided by an FEI and the request to send a message to a group of FEIs. Requests for the services provided by an FEI require the use of a descriptive token detailing the requested instance. The Agency responds to this form of request by returning a CORBA proxy object that provides access to the requested FEI. Requests to send a message to a group of FEIs require the use of a descriptive token that details the group of FEIs to whom the message should be sent, as well as a descriptor of the message. The Agency responds to this form of request by contacting all of the FEIs in the group and delivering the message to each of them.

BORG tokens are used within a client for describing functionality engines. Often, tokens are used to describe a set of FEIs, as when they are used in conjunction with the Agency. A token instantiation can describe a unique FEI, or a set of FEIs.

Tokens are implemented as C++ classes. A hierarchy of tokens exists that is rooted at the base `token` class. The `token` class constructors initialize the member data of a token instance so that it uniquely represents a set of FEIs.

Each functionality engine of the BORG System has an associated token class that is ultimately derived from the base `token` class. The derived class constructors will call parent constructors that initialize the instance to represent the desired set of FEIs.

BORG clients are any piece of code that uses an FEI. Clients are normally either applications or functionality engines and use FEIs as part of the implementation of some part of their design. An application may be a client if, for instance, it uses a FEI as part of the implementation of an internal function. A functionality engine may also be a client if it uses a FEI as part of the implementation of its service.

A client must use the Agency to initiate any interaction with other functionality engines. Interaction with the Agency is initiated by the client instantiating the `local_agency` class, since it is a proxy for the Agency functionality engine. The client may instantiate tokens and use them as descriptors in requests made of the Agency through the `local_agency`. Apart from the `local_agency`, clients will access functionality engines through CORBA proxy objects. The client may use the CORBA proxy as if it were the actual FIE the proxy provides access to.

The chapters following will look in depth at these components of the BORG System. Chapter 4 will examine functionality engines. Chapter 5 will look at tokens, and Chapter 6 will peer into the workings of the Agency. Finally, Chapter 7 will discuss the "how, what and why" of the clients that use the BORG System.

CHAPTER 4

FUNCTIONALITY ENGINES

This chapter examines the BORG System functionality engines. We will first discuss what a functionality engine is, followed by detailing the relationship between functionality engines and the OMG's IDL, and concluding with a look at specific examples of functionality engines in the BORG System.

4.1 What is a Functionality Engine?

Functionality engines encapsulate services provided by the legacy code being assimilated by the BORG System. Functionality engines attempt to coerce the services offered by the legacy system into an object model by placing an object-oriented interface on each service (see Figure 3.2). Functionality engines can be simple or complex, but provide only one service. They are built using tools of the BORG System that allow their use as distributed objects and provide for their transparent interaction within a distributed system.

For example, a large legacy system, such as the Alpha_1 Geometric Modeling and Manufacturing System, contains complex services such as interactive modeling, rendering, and visual display applications. Each of these services is contained within an individual program that is executed from the command line with its output and execution customized through command line arguments. Outside of the BORG System, the services provided by these programs may be composed only via UNIX pipes. But, within the BORG System, these services have been encapsulated into functionality engines. Each service may be accessed via the object-oriented interface provided by the functionality engine. Thus, inside an application, the services that once could only be used as stand-alone programs may now be accessed programmatically via the functionality engine.

Functionality engines are implemented as C++ classes. By using BORG tools that leverage off of a distributed system software package, an FEI may be used as a distributed object. FEIs may then be composed to become part of a distributed application. The distributed system software package upon which the BORG tools leverage is CORBA. FEIs become CORBA servers that may be accessed transparently within a distributed application via a CORBA proxy object.

4.2 Functionality Engines and IDL

The BORG tools leverage off of CORBA to create, acquire, and use FEIs as distributed objects. The fulcrum used for this leveraging is the OMG's Interface Definition Language (IDL). IDL provides a standard, language-neutral means of describing the public interface of an object. The object-oriented interface placed upon a legacy system service by a functionality engine is described in IDL. An IDL complier will take a functionality engine's IDL description and generate a CORBA proxy object that is used for remote communication with the CORBA server containing the FEI.

4.2.1 Describing Functionality Engines in IDL

IDL is a purely descriptive language which supports a subset of ANSI C++ with additional keywords introduced to support distribution concepts [10]. This section will discuss only selected parts of IDL that are generally relevant to describing functionality engines. Interested readers are referred to [10] for the complete grammar specification of IDL.

Object-oriented interfaces are described in IDL using the `interface` construct. An IDL `interface` contains the publicly accessible parts of an object-oriented interface. All elements of the interface are described in a language neutral way so that CORBA proxy objects may be generated in various programming languages. The IDL compiler used in the implementation of this thesis generates CORBA proxy objects in C++.

The most important element of an IDL interface description is the *operation*. An operation declaration "specifies the operations that the interface exports and the format of each including operation name, the type of data returned, the types of all parameters of an operation, ...and contextual information which may affect method dispatch." [10, p.62] Operation declarations of an interface can be thought of as being equivalent to member function declarations of a C++ class with the addition of a few IDL specific keywords such as `oneway`, which specifies the operation's method of dispatch to be asynchronous (i.e., the operation will not wait for a return value after being dispatched.)

All data types used in an IDL interface definition (these are most notably a operation's return types and parameter types) must be either the IDL basic types (`short`, `long`, `float`, `boolean`....), IDL template types (`sequences`, `strings`, `arrays`), or IDL constructed types (`structs`, `unions`, `enums`). This is important to note when creating IDL interfaces for functionality engines that are encapsulations of legacy code. One cannot expect to use an arbitrary C++ class or C struct found in the legacy code as the parameter or return type of an IDL operation. The class or struct must have some sort of corresponding IDL template or constructed type previously declared which is used in place of the class or struct in the definition of an IDL interface. The IDL type will be converted by an IDL compiler into a class or struct defined in the OMG's IDL language binding. When used in a functionality engine that implements a legacy service, the struct or class generated by the IDL compiler must be converted into the original legacy class or struct. This has direct impact on the implementation of BORG System tokens which are used as parameters to the request operations of the Agency functionality engine and on the use of Alpha_1 objects as parameters and return types to operations. See Chapter 5, for a complete discussion of tokens and Section 4.3.2 for a discussion of Alpha_1 objects in IDL.

Every functionality engine of the BORG System is described by a corresponding IDL `interface`. For example, Figure 4.1 shows an IDL interface for a generic mathematical functionality engine. The `compute_sqr` operation takes as its parameter a pass-by-value **float** representing the value to square and returns its square as a **float**. The `multiply` operation takes two pass-by-value **floats** representing the values to multiply and returns their product as a **float**. The `increment` operation takes a pass-by-reference **long** and increments it by one, but does not return a value as indicated by the **void** return type.

An IDL **interface** description does not contain executable code. It is only a descriptive declaration of the interface. Thus, it is up to the implementor of the interface to decide how to implement its operations. In the case of Figure 4.1, implementations of the `compute_sqr`, `multiply` and `increment` operations of the `generic_math` functionality engine must be provided. BORG System functionality engines are implemented as C++ classes with member functions corresponding to the operations declared in the IDL interface. A first step towards creating the C++ class implementing the functionality engine is for an IDL compiler to generate a CORBA proxy object from the functionality engine's IDL interface.

4.2.2 The CORBA Proxy Object

BORG System tools allow an FEI to be used as a distributed object. A distributed object is an instantiation of a class (object) and its proxies. The object often executes within its own separate process. The proxy provides access to the object, encapsulating the low-level technical aspects of the interprocess communication between the process using the distributed object and the process containing the object. By using a CORBA IDL compiler, a CORBA proxy object is generated from the IDL `interface` description of a functionality engine (see Figure 4.2). An FEI executes within a CORBA server process and access to it is provided by a CORBA proxy object.

Not all functionality engines need to run as distributed objects. A client, which is any code that uses a functionality engine, may choose to instantiate the class that implements a functionality engine within its own process space where it can be interacted with locally, instead of through a CORBA proxy object. This option is useful for clients who do not wish to incur any of the performance penalties inherent to interprocess communication.

Normally though, functionality engines will be used as distributed objects, in which case the functionality engine will consist of a CORBA server and a CORBA

```
interface generic_math
{
    float compute_sqr( in float val );
    float multiply( in float val1, in float val2 );
    void  increment( inout long val );
};
```

Figure 4.1. Generic Mathematical Functionality Engine IDL Declaration

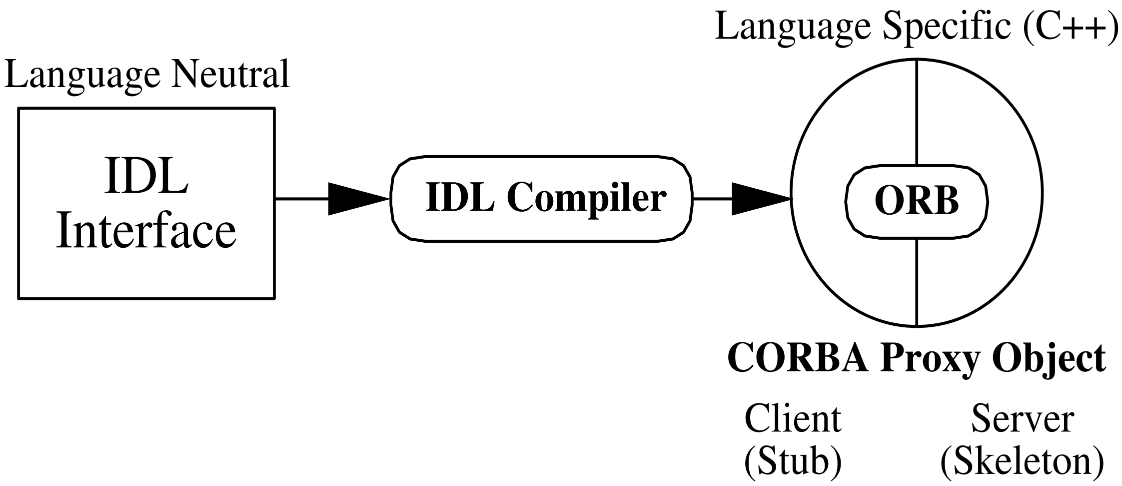

Figure 4.2. IDL Interface Definition to CORBA Proxy Object

proxy object. A CORBA server is a process in which the implementation class of a functionality engine has been instantiated. The CORBA proxy object consists of a client side (stub), an ORB, and a server side (skeleton). When a client uses a CORBA proxy object to access an FEI, the client's messages travel from the client side of the CORBA proxy, to the server side via the ORB, and up to the FEI (see Figure 2.1). The ORB component of the CORBA proxy object transparently handles all of the minor subtleties of this communication including, dynamic delivery services, method invocation, parameter encoding, synchronization and exception handling. The CORBA proxy object acts as the mediator between the client initiating the communication and the FEI that the client is communicating with. The implementation of the internal structure of a CORBA proxy is dependant upon the CORBA implementation being used, but always provides transparent and seamless interaction between the client and the FEI. The BORG System uses ORBeline[14], a CORBA implementation from PostModern Computing.

4.2.3 The Implementation of a Functionality Engine

After the IDL `interface` has been defined for a functionality engine, the implementor of the functionality engine must provide a C++ class that implements the that interface. Generally, the elements of the IDL `interface` become the public section of the resulting implementation class. The implementor is required to provide an implementation of the class and its member functions.

The implementation class of the functionality engine need only include the operations that were described in the IDL `interface`. The implementor must provide the actual code that implements these operations before the functionality engine can be used by a client. A client will have access to the functionality engine only through the operations described in the IDL `interface`. The implementation class of the functionality engine may have protected and private sections as well, but these are not described in IDL and thus are useful only for the purposes of the implementation. Figure 4.3 shows a C++ class that could be used as the implementation of the **generic_math** functionality engine whose IDL interface

```
/*
 * C++ implementation class of the
 * generic_math functionality engine.
 */
class Generic_math
{
public:
    generic_math();
    ~generic_math();

    /* Methods that implement the generic_math interface. */
    float compute_sqr( float val );
    float multiply( float val1, float val2 );
    void  increment( int & val );
};
```

Figure 4.3. C++ Implementation Class of the generic_math Functionality Engine

appears in Figure 4.1.

The only public member functions, other than the object constructors and destructors, are the three that implement the operations of the IDL `interface`: **compute_sqr**, **multiply**, and **increment**. The class may have other protected and private sections, but since they only pertain to how the class was implemented, they are not shown here. The OMG provides language bindings for IDL that specify how IDL types such as `long` map to language specific types such as C++ `int`s. Currently, the OMG has defined language bindings for C, C++, and Smalltalk.

A single IDL `interface` may have many implementations. Each implementation can be different from the others, but must contain the operations described by the IDL `interface` in the public section of the implementation class. Conversely, an implementation may be suitable for use as the implementation of many interfaces as long as all of the operations declared in the interfaces are contained in the public section of the implementation class. Implementors are free to switch interfaces and implementations around as suits their needs, as long as the set of operations declared in the IDL `interface` is completely contained within the public section of the implementation class.

An instantiation of a class that implements the IDL interface of a functionality engine is called a functionality engine instance (FEI). FEIs reside within CORBA servers which execute on specific machines in the BORG system and may be identified by names given them when they are created. For instance, a `generic_math` FEI, executing on a machine named Centauri may be named "Centauri_generic_math", where as a `generic_math` FEI executing on a machine named Gemini may be named "Gemini_generic_math".

4.2.4 Connecting the Proxy and Implementation

ORBeline provides an IDL compiler that generates CORBA proxy objects from IDL `interface` definitions. The CORBA proxy objects are generated in C++, using the OMG's IDL to C++ language binding. There are three elements to the ORBeline CORBA proxy object: the client side, an ORB, and a server side. Working together they allow for a transparent interaction between clients and FEIs.

The client side of an ORBeline CORBA proxy object is implemented as a C++ class. It contains member function declarations that are the C++ language binding equivalent of the operations described in the IDL `interface`. The client side class is derived from an ORBeline defined, `CORBA_Object`, so that it has access to basic member functions inherent to all CORBA proxy client side objects. Figure 4.4 shows the client side stub class generated for the generic_math functionality engine.

When a client uses a CORBA proxy as a stand in for a FEI, it is actually

```
class generic_math: public virtual CORBA_Object
{
    // ...
    virtual CORBA::Float compute_sqr( CORBA::Float val );
    virtual CORBA::Float multiply( CORBA::Float val1,
                                   CORBA::Float val2 );
    virtual void increment( CORBA::Long& val );
    // ...
};
```

Figure 4.4. Client Side Stub C++ Class

interacting with an instantiation of the client side class of the CORBA proxy. The client will interact with the client side class instantiation as if it were the FEI. The member functions declared for the client side class are the same as those declared for the functionality engine implementation class. The member functions of the client side class are referred to as *stubs*. Stubs are invoked by the client program in lieu of the operations of the FEI. They marshal the invocation request and its arguments, send it to the server side with the help of the ORB component of the CORBA proxy and then wait for a response. Once a response has been received, the stub unmarshals the return parameters and returns to the client program. The interprocess communication necessary for an operation invocation is transparent to the client. Figure 4.5 shows the code generated by the ORBeline IDL compiler to implement the **multiply** stub member function.

The server side of a CORBA proxy object is connected to an FEI. Functions on the server side are often referred to as *skeletons*, with a skeleton function corresponding to each operation declared in the IDL `interface`. The skeleton receives invocation requests sent from the client stubs. The skeleton will unmarshal a request and invoke the corresponding member function of the FEI. When the member function returns, the skeleton will marshal the return parameters and, with the help of the ORB component of the CORBA proxy, send them back to the client side stub. ORBeline uses a server side class as a place holder for the skeleton functions which are then declared as static member functions of the server side class. Figure 4.6 shows the skeleton functions declarations that are generated by the ORBeline IDL compiler for the generic_math functionality engine. The parent class of the `_sk_generic_math` class is the client side stub class of the CORBA proxy object. Figure 4.7 shows the code generated as the implementation of the skeleton **_multiply** function.

ORBeline has two programming strategies for the connection between the server side skeletons and the FEI: the inheritance strategy and the delegation or TIE strategy. In the inheritance strategy, the implementation class of the functionality engine inherits from a server side class generated by the IDL compiler as part of

```
CORBA::Float generic_math::multiply( CORBA::Float val1,
                                     CORBA::Float val2 )
{
    CORBA::Float _ret = (CORBA::Float)0;
    CORBA::MarshalStream *_strm = _create_request( "multiply",
                                                   1, 9001);
    *_strm << val1;
    *_strm << val2;

    try {
            _invoke();
    }
    catch (const CORBA::TRANSIENT& ) {
        return multiply( val1, val2 );
    }
    *_strm >> _ret;
    _strm->flushRead();
    return _ret;
}
```

Figure 4.5. IDL Compiler Generated Stub Member Function Implementation

```
class _sk_generic_math : public generic_math
{
    // Skeleton Operations implemented automatically
    // by IDL compiler.
    static void _compute_sqr( void *obj,
                              CORBA::MarshalStream &strm,
                              CORBA::Principal_ptr principal,
                              const char *oper,
                              void *priv_data );

    static void _multiply( void *obj,
                           CORBA::MarshalStream &strm,
                           CORBA::Principal_ptr principal,
                           const char *oper,
                           void *priv_data );

    static void _increment( void *obj,
                            CORBA::MarshalStream &strm,
                            CORBA::Principal_ptr principal,
                            const char *oper,
                            void *priv_data );
};
```

Figure 4.6. Server Side Skeleton Functions

```
void _sk_generic_math::_multiply( void *_obj,
                                  CORBA::MarshalStream &_strm,
                                  CORBA::Principal_ptr ,
                                  const char *,
                                  void *_priv_data )
{
    generic_math *_impl = (generic_math *)_obj;
    CORBA::Float val1;
    CORBA::Float val2;
    _strm >> val1;
    _strm >> val2;

    _strm.flushRead();
    CORBA::Float _ret = _impl->multiply( val1, val2 );
    _impl->_prepare_reply( _priv_data );
    _strm << _ret;
    _strm.sendMessage();
}
```

Figure 4.7. IDL Compiler Generated Server Side Skeleton Function Implementation

the CORBA proxy object. The server side class contains pure virtual member functions that are the C++ language binding equivalent of the operations declared in the functionality engine's IDL interface. Since the implementation class of the functionality engine inherits from this server side class, it must provide an implementation of the pure virtual member functions. The skeleton functions receive invocation requests from the client stubs. They unmarshal a request and invoke the corresponding member function of the server side class. Since this invocation is of a virtual function, the invocation is passed up to the corresponding method of the functionality engine's implementation class. Figure 4.8 shows the server side class generated by the ORBeline IDL compiler for the generic_math functionality engine. Figure 4.9 shows the implementation class of the generic_math functionality engine that uses the inheritance approach.

In the delegation or TIE strategy, the implementation class of the functionality engine does not inherit from any server class in the CORBA proxy object. The implementation class of the functionality engine may have its own inheritance structure based upon legacy system classes, or none at all. The only requirement is that it have member functions corresponding to the operations declared in its IDL `interface`. A wrapper class delegates the invocation requests received from the server side skeletons up to the FEI. The ORBeline IDL compiler generates a C++ template class as the wrapper. The constructors of the template class take a reference to an object as a parameter. The template class contains member

```
class _sk_generic_math : public generic_math
{
    // The following operations need to be implemented
    // by the server.
    virtual CORBA::Float compute_sqr(CORBA::Float val) = 0;
    virtual CORBA::Float multiply(CORBA::Float val1,
                     CORBA::Float val2) = 0;
    virtual void increment(CORBA::Long& val) = 0;
};
```

Figure 4.8. IDL Compiler Generated Inheritance Approach Server Side Class.

```
class Generic_math : public _sk_generic_math
{
public:
    Generic_math();
    ~Generic_math();

    /* Methods that implement the generic_math interface. */
    virtual CORBA::Float compute_sqr( CORBA::Float val );
    virtual CORBA::Float multiply( CORBA::Float val1,
                                   CORBA::Float val2 );
    virtual void  increment( CORBA::Long & val );
};
```

Figure 4.9. generic_math Functionality Engine Implementation Class Using Inheritance Approach

functions that correspond to the operations declared in the IDL `interface` of the functionality engine. The implementation of these member functions invokes the corresponding member functions of the object reference. Figure 4.10 shows the TIE template server side class generated by the ORBeline IDL compiler for the generic_math functionality engine. Figure 4.11 shows the implementation class of the generic_math functionality engine that uses the TIE approach.

The delegation strategy is useful for converting legacy class hierarchies into distributed objects. Since there is no need to inherit from a class generated by the IDL compiler, any class that implements the operations declared in the IDL interface may be used as the implementation class. The inheritance strategy is more intuitive, but is also more difficult to use when assimilating legacy classes. Ultimately the inheritance strategy wrappers legacy classes with an implementation class that inherits from the server class of the CORBA proxy. Figure 4.12 shows the inheritance hierarchy for an ORBeline CORBA proxy object and implementation classes.

```
template <class T>
class _tie_generic_math : public generic_math
{
    public:
        _tie_generic_math(T& t, const char *obj_name=(char*)NULL) :
                        generic_math(obj_name),
                        _ref(t) {
                _object_name(obj_name);
        }
        _tie_generic_math(T& t, const char *service_name,
                        const CORBA::ReferenceData& id)
                        :_ref(t) {
                _service(service_name, id);
        }
        ~_tie_generic_math() {}
        CORBA::Float compute_sqr(CORBA::Float val) {
                return _ref.compute_sqr(
                val);
        }
        CORBA::Float multiply(CORBA::Float val1,
                CORBA::Float val2) {
                return _ref.multiply(
                val1,
                val2);
        }
        void increment(CORBA::Long& val) {
                _ref.increment(
                val);
        }

    private:
        T& _ref;
};
```

Figure 4.10. IDL Compiler Generated TIE Approach Server Side Class

```
class Generic_math
{
public:
    Generic_math();
    ~Generic_math();

    /* Methods that implement the generic_math interface. */
    virtual CORBA::Float compute_sqr( CORBA::Float val );
    virtual CORBA::Float multiply( CORBA::Float val1,
                                   CORBA::Float val2 );
    virtual void  increment( CORBA::Long & val );
};
```

Figure 4.11. generic_math Functionality Engine Implementation Class Using TIE Approach

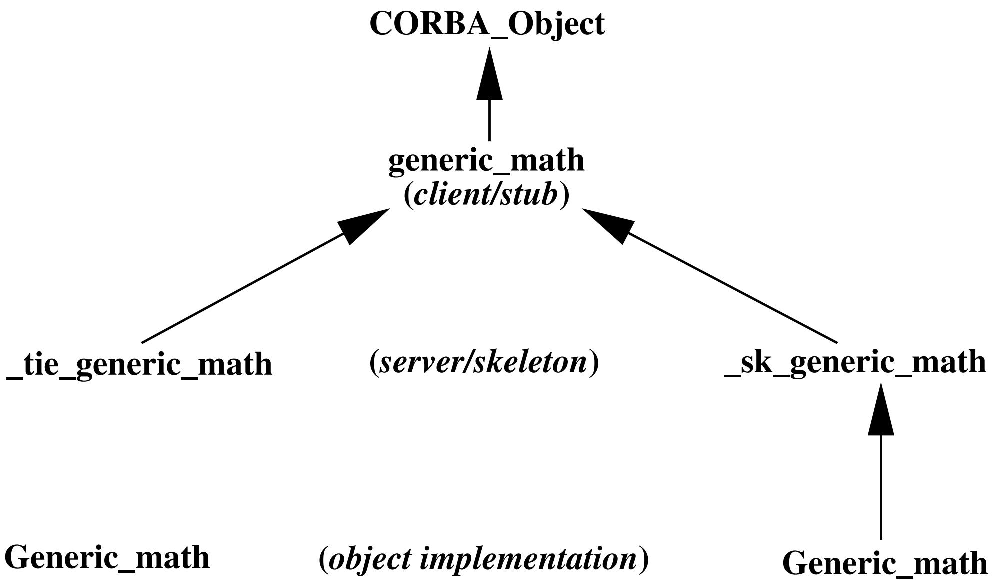

Figure 4.12. Inheritance Hierarchy for ORBeline CORBA Proxy Objects and Implementation Classes

4.3 BORG Functionality Engines

IDL allows for inheritance among its interfaces. The BORG System takes advantage of this by inheriting all of its functionality engine IDL `interface`s from a single base `interface`. This section discusses several of the functionality engines currently implemented in the BORG System.

4.3.1 The Base Functionality Engine Interface

The IDL `interface` construct allows for interface inheritance. An `interface` can be derived from a parent `interface`, inheriting all of the operations defined in the parent. Multiple inheritance among interfaces is also supported in IDL. The BORG System takes advantage of inheritance among interfaces by deriving all functionality engine IDL `interface`s from a base functionality engine IDL `interface`, called the `employee`. One exception to this rule is the Agency functionality engine which is not derived from any IDL `interface`. Figure 4.13 shows the inheritance hierarchy of the BORG System functionality engines.

The `employee` interface contains standard operations that are available for use with all functionality engines in the BORG System. The implementation class of each functionality engine IDL interface that derives from the `employee` interface must have its own implementation for each of the operations defined in the `employee` IDL interface. Figure 4.14 shows the IDL interface definition for the Employee functionality engine. The **this_token** operation returns a base representation of the token associated with this functionality engine. The values contained in the base representation define the specific characteristics of a particular set of FEIs. Tokens and their base representations will be discussed at length in Chapter 5.

The **message** operation takes a base message structure as a parameter and interprets the message it describes. The implementation of the operation will handle any types of messages it knows about. All Employees know how to handle a deactivate message, but more specific functionality engines will also know how to handle more specific messages. Section 5.4 will discuss BORG messages at length. The IDL `oneway` keyword is used to declare the **message** operation to be

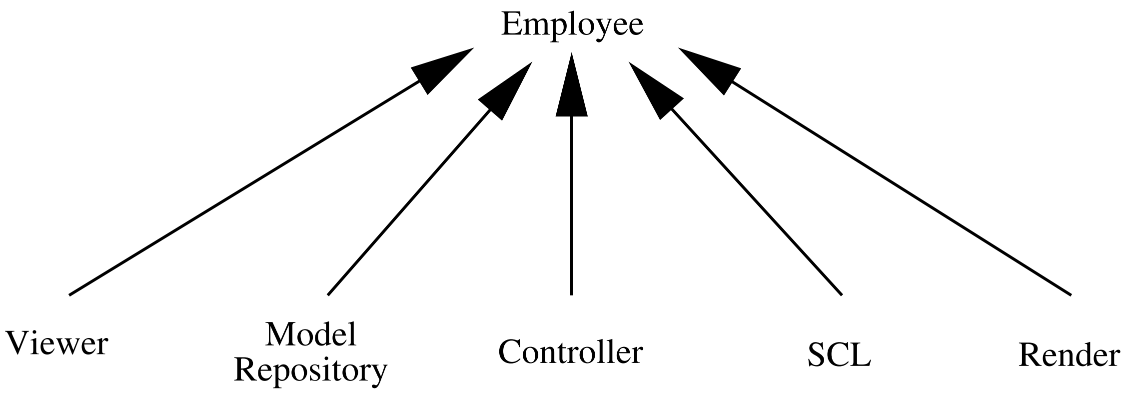

Figure 4.13. BORG System Functionality Engine Inheritance Hierarchy

```
// employee IDL interface definition.
interface employee
{
    base_token  this_token();
    oneway void message( in base_msg msg );
};
```

Figure 4.14. Employee Functionality Engine IDL Interface

asynchronous. Thus, a client will invoke this operation via a CORBA proxy object, but will not wait for the operation to return before continuing to execute.

4.3.2 The Model Repository Functionality Engine

The Model Repository is a functionality engine whose service is to provide clients with Alpha_1 models. Figure 4.15 shows the IDL interface of the Model Repository functionality engine. The Model Repository uses a hash table containing information about known Alpha_1 models. Clients may query the model repository using a specific model name. If the Model Repository hash table contains an entry for the model, it will be returned to the client. The `model_rep` interface inherits from the `employee` interface and so requires that its implementation class provide member functions for the operations declared in its IDL `interface` as well as for those operations declared in the `employee` interface. Figure 4.16 shows the C++ class that implements the `model_rep` IDL interface. Note that `Model_rep` is derived from the server class of the CORBA proxy object and that it provides member functions that implement the `employee` IDL operations as well as the `model_rep` IDL operations. Figure 4.17 shows the implementation of the `model_rep`'s **query** member function.

Figure 4.15 shows that the return type of the **query** and **read_file** operations as well as a parameter of the **add_model** operation are of type `a1_object`. The

```
interface model_rep : employee
{
    a1_object query( in string model_name );
    void      add_location( in string model_name,
                            in string location );
    void      add_model( in string model_name,
                         in a1_object a1obj );
    void      remove( in string model_name );
    a1_object read_file( in string file_name );
};
```

Figure 4.15. Model Repository IDL Interface

```
class Model_rep : public _sk_model_rep
{
public:
    Model_rep( local_agency* age );
    ~Model_rep();

    /*
     * Public CORBA interface member functions.
     */

    /* Member functions implementing the employee IDL interface. */
    virtual base_token* this_token() {
        return new base_token( t_.base() );
    }
    virtual void        message( const base_msg & msg );

    /* Member functions implementing the model_rep IDL interface.*/
    virtual a1_object * query( const char * model_name );
    virtual void        add_location( const char * model_name,
                                      const char * location );
    virtual void        add_model( const char * model_name,
                                   const a1_object & a1obj );
    virtual void        remove( const char * model_name );
    virtual a1_object * read_file( const char * file_name );

    // . . .
};
```

Figure 4.16. C++ Implementation Class of Model Repository IDL Interface

```
a1_object *
Model_rep::query( const char * model_name )
{
    /* Look up the model name in the hash table. */
    model_rep_key mrk( ( const char * )model_name );
    model_rep_elm *mre = models_.lookup( model_rep_key( mrk ) );

    if ( mre == NULL )
    {
        /* Generate an error message if model name is not found
         * in table.
         */
        fprintf( stderr, "Model '%s' not found in repository.\n",
                 model_name );

        return NULL;
    }
    else
        /* Read the object from the file. */
        return read_model_file( mre->location() );
}
```

Figure 4.17. Implementation of Model Repository query Member Function

`a1_object` type is the IDL type corresponding to any Alpha_1 C++ class that inherits from the Alpha_1 base object class, `object_type`. Figure 4.18 shows the IDL definition of the `a1_object`. The Alpha_1 Software System contains hundreds of C++ classes, all of which we would like to be able to use in declarations of IDL interfaces. As mentioned before, types used in IDL cannot be C++ classes. Nevertheless, there must be an IDL type corresponding to each C++ class or struct in order to use it within an IDL interface. Thus, the most obvious solution is to create an IDL type (`struct` or `interface`) for each Alpha_1 C++ class that we wish to use in an IDL interface. Apart from the large amounts of time, effort and disk space that would be required to implement this solution, other non-trivial problems arise. The IDL `struct` construct cannot be used since `struct`s do not support inheritance and thus the Alpha_1 object hierarchy would not be represented properly. IDL `interface`s do support inheritance, but instances of the `interface`s translate to CORBA servers. Thus, every time an application wished to pass an Alpha_1 object, either as a parameter or return value, that object would have to be executing as a CORBA server. The performance of this solution is problematic in terms of memory usage and execution speed, expecially for applications that create, use and pass large numbers of Alpha_1 objects.

The solution implemented by the BORG System maintains the Alpha_1 class hierarchy while also accounting for performance issues. Most Alpha_1 C++ classes derive from the base Alpha_1 class, `object_type`. `object_type` provides member functions that allow an instance of any class derived from `object_type` to serialize itself into a byte stream, either for transmission to another process via a UNIX socket or for recording on a persistent storage device. This serialization process is assisted by the Alpha_1 `a1_stream_type` class. The IDL template type `sequence` is mapped by an IDL compiler to a C++ class that is essentially a container class for a large array of `char`s. The BORG System implements a new Alpha_1 class, `a1_corba_stream`, which allows an `object_type` instance to write itself to an IDL `sequence`. The sequence object may then be used as a parameter or return value of a member function corresponding to an operation of an IDL interface. A sequence may also be converted back to an instance of the original Alpha_1 C++ class.

4.3.3 The Alpha_1 Render Functionality Engine

The Render functionality engine provides a rendering service to BORG clients. Figure 4.19 shows the IDL interface description of the Render functionality engine. The implementation class encapsulates the capabilities of Alpha_1's stand-alone render program, placing an object-oriented interface on a non-object-oriented legacy program. Render's IDL interface declares only one operation, **render_to_file**. The operation takes two parameters: an `a1_object` and a file name. The `a1_object` parameter is a scene of objects including viewing matrices and camera objects. The file name specifies where the Render functionality engine should place the rendering output. The output is a Run Length Encoded (RLE) file containing the rendered view of the scene. The **render_to_file** operation was declared with the `oneway` keyword. This specifies an asynchronous invocation. In an asynchronous invocation, the client stub will not block and wait for a response from the FEI. The client is free to continue executing other code while the Render functionality engine executes the **render_to_file** operation. When the **render_to_file** operation is invoked, the render functionality engine will pop up an information window telling the user that the rendering is taking place. When the rendering is completed, the window will inform the user and change the window so that it contains an **OK** button. When the user clicks on the **OK** button, the window will close and the Render functionality engine will be deactivated (see Figure 4.20). The Render functionality engine could easily be extended with operations to return the rendered image or to render without the graphical information dialog box.

A special implementation feature of the Render functionality engine as well as several other functionality engines is that it is multithreaded. Multithreading is necessary in functionality engines that require a graphical interface. Functionality engines are implemented as CORBA servers and are communicated with via CORBA proxy objects. The CORBA server is a separate process that contains an instantiation of the implementation class of a functionality engine. The main

```
typedef sequence< char > a1_object;
```

Figure 4.18. Alpha_1 Object IDL Representation.

```
interface render : employee
{
    // Asynchronous invocations.  (Specified by keyword "oneway")
    oneway void render_to_file( in a1_object a1obj,
                                in string file_name );
};
```

Figure 4.19. Render Functionality Engine IDL Interface

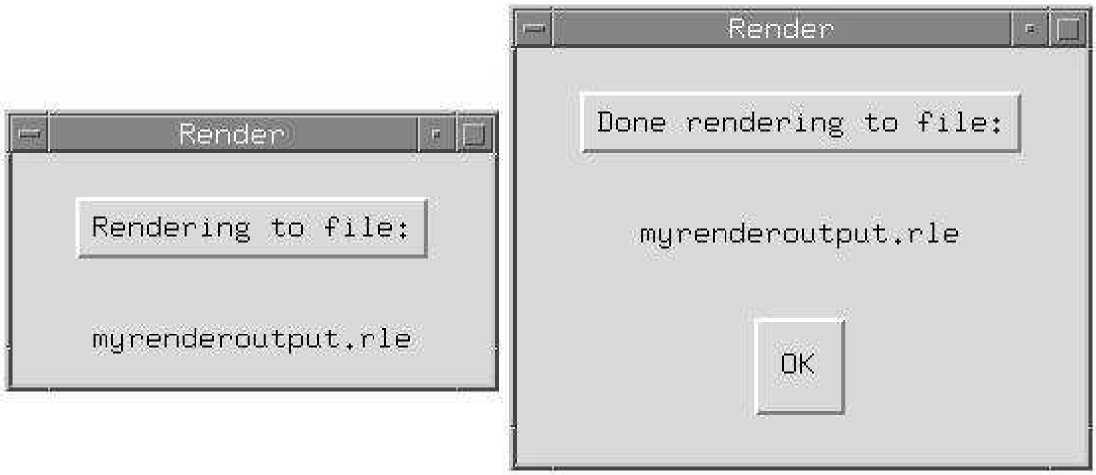

Figure 4.20. Render Functionality Engine Graphical Interface

thread of execution in a CORBA server runs a CORBA event loop which processes invocation requests made by clients on the FEI.

The BORG System uses Tcl/Tk[12] to produce graphical interfaces. Tcl/Tk (like any graphical interface system) requires a way to process events generated by the user. Normally this is done in an event loop running in the process's main thread of execution. This is not possible in a CORBA server since the main thread of execution is already running the CORBA event loop.

The BORG System's solution is to create a new thread of execution that runs an event loop to processes user generated Tcl/Tk events. Thus, the render functionality engine has two threads of execution, one running the CORBA event loop, and the other running a Tcl/Tk event loop. This means that FEIs may have a graphical user interface just like other applications, while also processing invocation requests made by clients. This is an essential requirement for functionality engines such as the Render functionality engine where a Tcl/Tk event loop is required to manage the information window while the CORBA event loop renders the scene.

4.3.4 Creating a BORG Functionality Engine

Figure 4.21 is a flowchart diagram outlining the steps required to create a new BORG functionality engine. A programmer will start the design of a new

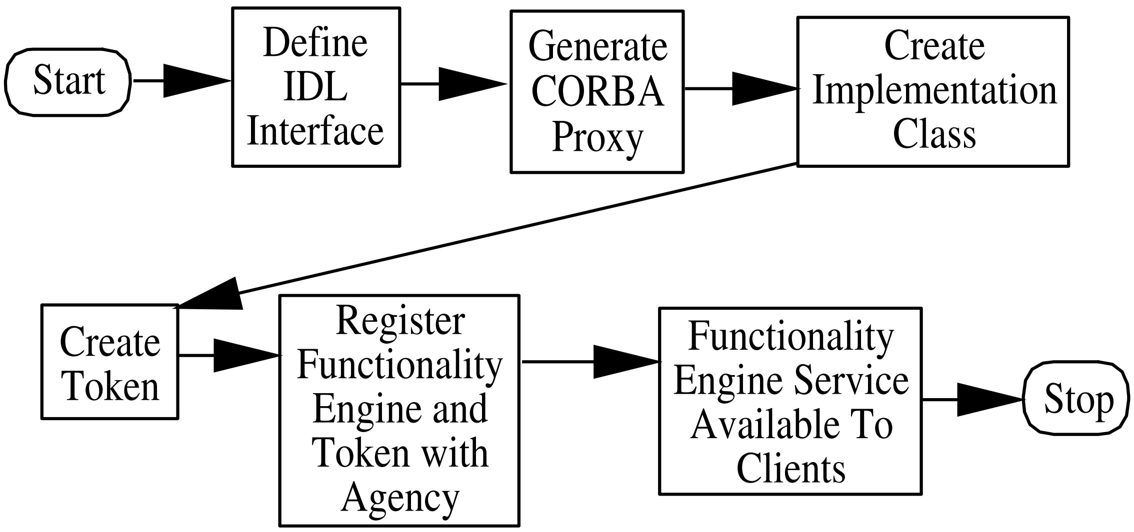

Figure 4.21. Functionality Engine Contruction Flow Chart

functionality engine by first determining what interface it will support (i.e., what are its publicly available operations). The fully defined interface is then described in IDL using the IDL `interface` construct. The IDL `interface` is used by an IDL compiler to generate a CORBA proxy object. The implementation of the functionality engine is unimportant and irrelevant to the client application. The client application will only interact with an FEI via the CORBA proxy object, and so does not need to know anything about the functionality engine except its IDL interface.

The programmer must provide an implementation class for the functionality engine as well as an implementation of that class and its methods. Additionally, the programmer must create a token that will be associated with the functionality engine (see Chapter 5 for a discussion of BORG tokens). This token will be used by clients when sending requests to the Agency (see Chapter 6). The functionality engine and its associated token must then be registered with the Agency before the Agency can provide a client with access to that functionality engine. Section 6.3.3 discusses registering tokens and functionality engines with the Agency. Once the CORBA proxy object has been created, the implementation of the functionality engine and its associated token have been completed, and the functionality engine and its token have been registered with the Agency, clients are free to use the service provided by the functionality engine.

CHAPTER 5

TOKENS

This chapter discusses the BORG System's tokens and messages. First, we will explain the purpose of tokens, followed by how and where tokens are used as well as examining the structural implementation of a token. The concluding section discusses BORG System messages, their use and implementation.

5.1 Purpose of Tokens

Tokens are used to represent functionality engine instances. They can uniquely represent any set of FEIs, whether those particular FEIs exist or not. This makes tokens useful as a means of describing a specific FEI to a client or other functionality engine.

A token is implemented as C++ class which represents a functionality engine. There is a token class associated with every functionality engine. For example, the `render_token` class represents the Render functionality engine, and the `model_rep_token` class represents the Model Repository functionality engine. An instance of a token class will be referred to simply as a token.

A token specifies a set of FEIs. Tokens can be created with varying degrees of information concerning the set of FEIs they represent. At the broadest, a token can represent the entire set of FEIs associated with that token's class. At the narrowest, a token can represent one unique FEI associated with the token's class. If two tokens are equal, they represent equivalent sets of FEIs.

5.2 Using Tokens

The special nature of tokens that allows them to represent a set of FEIs provides for their use as qualifiers in messages passed between clients and functionality engines in the BORG System. A client may send a token to a functionality engine as part of any message that requires the specification of a set of FEIs. The set of FEIs that the token represents can be used for anything from notification operations to client request operations.

During execution, a client may instantiate the token class representing a functionality engine. The constructors of a token class provide for the specification of all of the possible varying degrees of information concerning the set of FEIs that this token will represent. The client will decide, based on its current needs, how specific a set of FEIs this token will represent.

Different operations require different degrees of FEI set specification. Broad set specification can often be used for operations such as notification. For example, a client may wish to send a notification message to all of the FEIs executing on a specific host. A token can be instantiated with the proper parameters so that the set of FEIs it represents contains all FEIs running on the specific host. Once instantiated, the token may be passed to the Agency functionality engine as part of its notification operation.

A narrow FEI set specification may often be used for operations such as the request operation. For example, if a client wishes to use a functionality engine as a specific element of its implementation, it may instantiate a token representing that functionality engine. As part of the instantiation process, the client may specify very explicit parameters so that the set of FEIs that this token represents is very small and specific. Having instantiated the token, the client may pass the instance to the Agency as part of the operation requesting access to an FEI. The Agency will return a proxy to an FEI contained in the set of FEIs represented by the token.

5.3 Token Implementation

Tokens have two representations, each serving a distinct purpose. A token's external representation is that seen and used by a client program and is implemented as a C++ class. Internally, a token is represented as an IDL `struct`.

5.3.1 External Token Representation

Externally, tokens are represented as a C++ class which is only a container class for the internal representation of a token. All token classes are derived from a single base `token` class, which does not describe any functionality engine. The `token` class is declared to contain a data member that is an instance of the internal representation of a token. Classes derived from the `token` class are declared without any data members, but use and initialize the data contained in the `token` class. The data contained in the `token` can be set only through the `token`'s constructors. Figure 5.1 shows the C++ class declaration of the `token`.

Classes derived from the `token` class represent functionality engines. Instantiations of these classes represent FEIs. Using the proper constructors and providing pertinent parameters to them, the client has the freedom to instantiate a token class so that the token represents a specific set of FEIs. Clients will deal only with the derived token classes and not with the token's internal representation, for which the derived token classes are only a container.

Figure 5.2 is an example of a derived token class. Note that it is derived from the `token` class and that its interface consists only of constructors that invoke the constructors of the parent class. This derived token class represents the Employee functionality engine described in Section 4.3.1. Instantiations of this class represent Employee FEIs. Since all BORG functionality engines derive their IDL interfaces from the IDL interface of the Employee functionality engine, all FEIs may be considered Employee FEIs. Thus, `employee_token`s may be used to specify a broad set of FEIs, and in the limit, specifying all FEIs in the BORG System.

Figure 5.3 shows the C++ class declaration of the `model_rep_token` which represents the Model Repository functionality engine. Note that `model_rep_token` is derived from `employee_token` and that it contains only constructors that invoke the constructors of the parent `employee_token`. The other currently derived tokens of the BORG System, the `controller_token`, `render_token`, `scl_token` and `viewer_token`, are declared in a parallel manner. Figure 5.4 shows a diagram of the token inheritance hierarchy.

```
/* Base class of the BORG token hierarchy */
class token
{
public:
    token( const token & t ) : base_( t.base() ) {
        *this = t;
    }
    token( const base_token & bt ) : base_( bt ) {
        *this = bt;
    }
    token( const char *obj  = NULL,
           const char *mach = NULL );
    token( const char        *obj,
           host_machine_enum  e );
    ~token() {}

    // . . .

    /* Overloaded operators. */
    inline int     operator==( const token & t ) const;
    inline int     operator!=( const token & t ) const;
    inline token & operator=( const token & t );
    inline token & operator=( const base_token & b );

    /* Accessor member functions for private data. */
    const base_token & base() const { return base_; }

protected:
    /* Constructor used by derived classes to pass on the type
     * name of their derived token. */
    token( const char        *type,
           const char        *obj,
           host_machine_enum  mach );

private:
    /* Internal representation of a token.  CORBA struct. */
    base_token base_;
};
```

Figure 5.1. Token C++ Class Declaration

```
/* Token class representing the Employee Functionality Engine */
class employee_token : public token
{
public:
    employee_token( const employee_token & t ) :
        token( t.base() ) {}
    employee_token( const base_token & b )     :
        token( b ) {}
    employee_token( const char *obj  = NULL,
                    const char *mach = NULL ) :
        token( "employee", obj,
               token::parse_host( mach ) ) {}

    employee_token( const char        *obj,
                    host_machine_enum  e ) :
        token( "employee", obj, e ) {}

    ~employee_token() {}

protected:
    /* Constructor used by derived classes to pass on the type
     * name of their derived token. */
    employee_token( const char        *type,
                    const char        *obj,
                    host_machine_enum  mach ) :
        token( type, obj, mach ) {}
};
```

Figure 5.2. Example of a Derived Token Class

```
/* Token class representing the Model Repository
 * functionality engine.
 */
class model_rep_token : public employee_token
{
public:
    model_rep_token( const model_rep_token & t ) :
        employee_token(t.base()) {}
    model_rep_token( const base_token & b )      :
        employee_token( b ) {}
    model_rep_token( const char *obj  = NULL,
                     const char *mach = NULL ) :
        employee_token( "model_rep", obj,
                        token::parse_host( mach ) ) {}

    model_rep_token( const char        *obj,
                     host_machine_enum  e ) :
        employee_token( "model_rep", obj, e ) {}

    ~model_rep_token() {}

protected:
    /* Constructor used by derived classes to pass on the type
     * name of their derived token. */
    model_rep_token( const char        *type,
                     const char        *obj,
                     host_machine_enum  mach ) :
        employee_token( type, obj, mach ) {}
};
```

Figure 5.3. Model Repository Functionality Engine Token Class

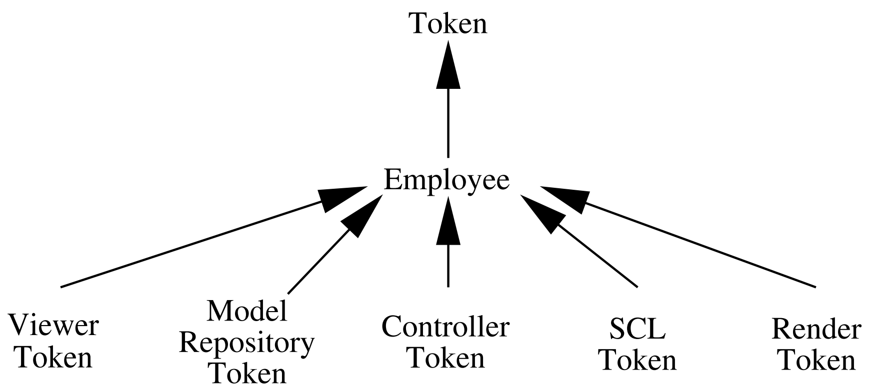

Figure 5.4. Token Inheritance Hierarchy

5.3.2 Internal Token Representation

Internally, every token instance contains a compact structure that stores the specific data that allows for the specification of a set of FEIs. This structure is called a `base_token` and is generated as the implementation of the IDL `struct` construct (see Figure 5.5). As a generic structure, the `base_token` does not represent any functionality engine, but, once initialized, can be used to represent any FEI in the BORG System.

The `base_token` has slots for all of the data elements needed to specify a unique set of FEIs.

- **type** : The type name of this token (i.e., "model_rep_token", "render_token"). This places a textual representation of the token's class name into the generic

```
// Base token IDL definition.
struct base_token
{
    string type;        // Type name of this token.
    string obj;         // Object name of requested server.
    short  machine;     // Machine on which associated
                        // functionality engine runs.
    short  host_machine;// Machine on which this token was created.
};
```

Figure 5.5. IDL Declaration of the base_token

base_token structure. Since the base_token structure does not represent a functionality engine, this data slot initializes and sets the boundaries of the set of FEIs represented by the base_token.

- **obj** : The object name of all the FEIs contained in the set of FEIs represented by this token. The object name can be very specific, such as "Alpha1_1 Model Repository" or "Centauri_generic_math", limiting the set to a small number of elements. On the other hand, the object name does not have to be specified, whereupon the set of FEIs could be quite large.

- **machine** : The host machine for the all of the FEIs contained in the set of FEIs represented by this token. This is an enumerated type with possible values:

  - ANY
  - HOST
  - CENTAURI_CS_UTAH_EDU
  - GEMINI_CS_UTAH_EDU
  - OBTUSE_CS_UTAH_EDU
  - SANCTUM_CS_UTAH_EDU
  - NONE

  The ANY value specifies all of the FEIs executing on all the machines in the BORG System, potentially creating a large set. The HOST value specifies all of the FEIs executing on the current host machine, whereas the other values specify specific machines in the BORG System. The enumerated type can be expanded to include any additional machine added later to the BORG System. An enumerated type was chosen for this data slot due to its ease of implementation. A more complex implementation, such as a dynamic namespace, may provide the base_token with more flexibility.

- **host_machine** : The host machine on which this token was instantiated. This is also an enumerated type whose domain of values is the same as that of the **machine** slot. This data slot does not limit the set of FEIs associated with this token.

The `base_token` structure encapsulates the unique data of a token in a structure that is the implementation of the IDL `struct` construct. Using an IDL `struct` to define the `base_token` structure allows the BORG System to use tokens as parameters and return values of the operations of a functionality engine's interface.

The interface to a functionality engine is defined using the IDL `interface` construct. All types used in an IDL `interface` definition must be either IDL basic types, IDL template types or IDL constructed types. It is not possible to use C++ classes or structures in IDL. This precludes the use of the C++ class representation of a token within an IDL definition.

This does not mean that tokens cannot be used in the operations of functionality engines. The data of a token are encapsulated within a C struct that implements the IDL `base_token` structure. The IDL struct can be used within the IDL `interface` definition of a functionality engine, and its corresponding C struct can be used in the implementation of the IDL interface. Thus, since `base_token`s are the IDL embodiment of a token, the BORG System has a way of sending messages containing tokens between clients and functionality engines.

Whereas one would like to use a token in the description of an IDL `interface`, the IDL `base_token` structure must be used instead, without the loss of any of the specific data of an individual token. One of the functionality engines that benefits from this design is the Agency. Chapter 6 peers into the workings of the Agency and how this design of tokens is used in practice.

5.4 Messages

BORG System messages represent signals passed from a client to a set of FEIs. These signals can be anything from a signal to deactivate an FEI, to a signal that a file has changed on disk. Like tokens, messages are implemented as a class hierarchy, with each class representing a unique message.

Messages are used in the Agency's notification operation. Clients will specify a set of FEIs and a message to send to them. The set of FEIs is determined by a token, and the message to be sent is determined by a message instance.

A C++ class hierarchy is used to represent messages. The class `message` is the base class of the hierarchy and does not represent any specific message. Classes derived from the `message` class represent specific types of BORG System messages. For example, the `deactivate_msg` represents the deactivate signal available for use with all FEIs.

Like tokens, the `message` class contains member data that determine the specific message represented by this class. A derived class constructor will set the member data to specify the message this object represents. The member data of a message is an instance of the `base_msg` struct. The `base_msg` struct is the C++ structure generated from an IDL `struct`. It encapsulates the data of a message in an IDL structure and allows messages to be used within IDL operations. Figure 5.6 shows the IDL declaration of the `base_msg`.

Currently, the `base_msg` struct contains only the variable, **msg**, of type **short**. **msg** is an enumerated type representing the specific type of message. Currently, only a deactivate message is defined in the BORG System. An FEI that receives the deactivate message is no longer available for use by clients. The FEI is deleted and its CORBA server process is killed. Future expansions of the message capabilities of the BORG System may enrich the data contained within the `base_msg` struct, and enlarge the domain of defined messages.

```
struct base_msg
{
    short  msg;  // Enumerated type of the message.

    // More data can be encapsulated here if needed.
};
```

Figure 5.6. IDL Declaration of the base_msg

CHAPTER 6

THE AGENCY

This chapter discusses the BORG System's Agency functionality engine. We will first explain the purpose of the Agency, followed by how and where the Agency might be used, and concluding with an examination of the Agency's implementation.

6.1 Purpose of the Agency

The Agency is a functionality engine that handles information exchanges among clients in the BORG System. By acting like a bulletin board, the Agency mediates requests for access to FEIs that are made by clients. This mediation encapsulates and abstracts the underlying distributed system software package that is necessary for the exchange of information between clients and functionality engines in the BORG System.

The service provided by the Agency is the mediation of requests made by clients. Clients in the BORG System must use the Agency as an intermediary when interacting with FEIs. A client is unable to communicate with an FEI unless it uses the Agency as a catalyst or as a disseminator (see Figure 3.3 and 3.4).

As a catalyst, the Agency is used by clients when they are in need of the service provided by a functionality engine. The client will request that the Agency provide it with an FEI. The Agency will either find an existing FEI or create a new one, returning a CORBA proxy to a FEI in response to the request made by the client.

As a disseminator, the Agency is used by clients to send notification messages to a specified group of FEIs. Clients will contact the Agency, requesting it to send a specific notification to a set of FEIs. The Agency will then find all the specified FEIs, and send the notification to them.

The Agency encapsulates and abstracts the underlying distributed system software package that is necessary to provide its service. The Agency was designed so that BORG System application programmers need only limited knowledge of the underlying distributed system software package.

6.2 Using the Agency

The Agency is used for requesting access to functionality engine instances and for requesting the sending of notification messages to sets of functionality engine instances. These bulletin board characteristics are available to all clients through the use of a special proxy class called the **local_agency**. The client can use the bulletin board features provided by the Agency anywhere they are required, via an instantiation of the `local_agency` class.

6.2.1 Agency as Catalyst

A client may use the service provided by a functionality engine. Clients obtain an FEI by sending a request to the Agency via the `local_agency` proxy. The Agency will find an appropriate FEI and return a CORBA proxy object for that instance.

Clients specify what FEI they require access to by using tokens. A client will instantiate a token that represents the FEI it wishes to use. The token will specify either a specific FEI or a general set of FEIs from which any element would be appropriate. The token is passed to the Agency, via the `local_agency` proxy, as a parameter of the operation.

6.2.2 Agency as Disseminator

A client may also find it useful to send a message to a set of FEIs. This is accomplished by sending a notification request to the Agency via the `local_agency` proxy. The Agency will find the appropriate FEIs and send the notification message to them.

Clients specify to whom the message should be sent by using tokens. The token may specify one specific FEI, or it may specify a broad category of FEIs. The message to be sent is determined by an instance of a message descriptor class. There is a message descriptor class for every predefined message in the BORG System. The message descriptor and the token are sent to the Agency via the `local_agency` proxy, as parameters of the Agency's notification operation.

6.2.3 Current Limitations of a Dissemination

Currently, there is only one message descriptor class defined in the BORG System. This message descriptor represents the **deactivate** message (see Section 5.4). When a client sends this message, all FEIs that receive it automatically shut down and are no longer available for use by clients. The **deactivate** message can be sent to any FEI.

Other message descriptors may be defined that can be sent only to specific FEIs. For instance, a model repository functionality engine may define a message descriptor that says the repository of models has been updated. A model repository FEI that receives this message descriptor may interpret it as a signal that its internal database of models is out of date and needs to be reloaded using the repository of models on disk.

Currently, the bounds of the Agency's implementation limit the size of the set of FEIs to which a message will be sent. A token may represent a large set, but currently, the message will only be sent to a single FEI within that set. This is due to CORBA's lack of message broadcasting support. CORBA is a point-to-point communication system, whereas notification is a broadcast communication operation. Thus, an intricate algorithm would be needed to ensure complete broadcast coverage. The Agency does not implement such an algorithm and as such, dissemination is most useful when a message must be sent to a specific FEI.

6.3 Implementation of the Agency

The Agency acts as a catalyst and a disseminator while encapsulating and abstracting the underlying distributed system software package. Clients interact with the Agency via a proxy object called the `local_agency`.

6.3.1 The local_agency

The local_agency is a proxy object of the Agency functionality engine. It provides clients with transparent access to the Agency. The local_agency is designed to encapsulate the interface of the Agency and provide a general interface based solely on BORG System constructs.

The local_agency is declared as a C++ class (see Figure 6.1). Clients will use an instance of the class where ever there is a need for access to the Agency. The constructors of the local_agency class will obtain a CORBA proxy object that is connected to the Agency FEI by using the CORBA communication infrastructure (see Section 6.3.2). The **request** and **notify** member functions of the local_agency class provide access to the Agency's CORBA proxy object and its operations. Internally, the **request** and **notify** member functions take the token that they receive as an argument, and pass on a reference to its base_token member data as a parameter to the Agency's operations. The **notify** member function also takes a message as an argument, and passes on a reference to its base_message member data as a parameter to the Agency's operation.

For example, if a client needs to use the Model Repository, it may instantiate a model_rep_token and pass it as a parameter to the invocation of the local_agency's **request** member function. Figure 6.2 shows code that describes how the local_agency can be used to request access to a functionality engine. Here we have instantiated the local_agency class and the model_rep_token class. The local_agency constructor obtains a CORBA proxy object for the Agency functionality engine. The model_rep_token constructor uses default arguments to initializes the set of Model Repository FEIs represented by this token. Then, passing the model_rep_token instance to the local_agency's request member function, the Agency contacts a Model Repository FEI and returns its proxy to the client. The **_narrow** function of the model_rep class that is used in Figure 6.2 is a static member function that type casts the employee CORBA proxy returned by the local_agency to a model_rep type. ORBeline implements this member function as part of its run-time type inheritance system.

```
class local_agency
{
public:
    local_agency(); /* Stand alone client. */
    local_agency( int & argc, char * argv[] ); /* Functionality
                                                 * engine client. */
    ~local_agency();

    /* Member functions that encapsulate operations of the Agency
     * interface.
     */
    employee *       request( const token & t,
                              find_enum find );
    void             notify( const token & t,
                             const message & m );

    /* General object member functions. */
    int              bind_to_agency();

    /* Accessor member functions or private data. */
    const char * name() const       { return name_; }
    const int    corba() const      { return corba_; }
    const int    persistent() const { return persistent_; }
    const int    bound() const      { return bound_; }

private:
    agency * a_;  /* CORBA proxy of the Agency FEI. */
    // . . .
};
```

Figure 6.1. local_agency Class Declaration

```
{
    local_agency     agency;
    model_rep_token  mt;
    model_rep *      repository;

    repository = model_rep::_narrow( agency.request( mt ) );
}
```

Figure 6.2. Using the local_agency

6.3.2 Binding To FEIs

In previous sections we have mentioned that clients use the Agency to contact FEIs and that the `local_agency` is used to contact the Agency FEI. This contacting is done with the help of the ORB. The Agency and its proxy, the `local_agency`, encapsulate and abstract the ORB and the FEI connection process.

The Agency and the `local_agency` use the ORB to obtain handles or CORBA proxies to FEIs. These handles are instantiations of the client side stub class of the CORBA proxy object and provide transparent access to the FEI via the process described in Section 4.2.2 and shown in Figure 2.1.

Handles are obtained through a binding process that uses the ORB to locate and establish a connection to an FEI. A **_bind** static member function is generated by the ORBeline IDL compiler as part of each client side stub class. The **_bind** member functions uses the ORB to locate and connect to an FEI. Figure 6.3 shows the declaration of the **_bind** static member function of the `employee` client side stub class. The parameters of the **_bind** member function can be used to specify a specific FEI to which to bind. The parameter, **object_name**, can specify the name of the desired FEI, and the **host_name** parameter can specify the name of the host machine where this FEI is executing. The **CORBA::BindOptions** parameter is discussed in [14]. If the parameters to the **_bind** member function are ambiguous, such that there is more than one FEI that fits their description, internal algorithms of the specific ORB implementation in use will choose the FEI with which to connect.

Figure 6.1 shows the `local_agency` declaration. The `local_agency` normally binds to the Agency FEI during its instantiation by invoking the **bind_to_agency** member function. Accessor member functions of the `local_agency` can be used to determine if a connection has been made to an Agency FEI. A private data member, **a_**, is a pointer to the instantiation of the client side stub class of the Agency obtained via the bind process. Figure 6.4 shows the code that implements the **bind_to_agency** member function. The parameter to the **_bind** function is the value of the environment variable, HOST. This specifies that the Agency FEI

```
class employee : public virtual CORBA_Object
{
    // . . .
    static employee * _bind(const char *object_name = NULL,
                            const char *host_name = NULL,
                            const CORBA::BindOptions* opt = NULL);
    // . . .
};
```

Figure 6.3. _bind Function Declaration Generated by the ORBeline IDL Compiler

```
int
local_agency::bind_to_agency()
{
    if ( ! bound_ )
    {
        /* Connect to the Agency FEI via the bind function. */
        a_ = agency::_bind( getenv( "HOST" ) );

        if ( a_ )
            bound_ = 1;
    }

    return bound_;
}
```

Figure 6.4. Binding to the Agency Within the local_agency

that we wish to connect to is located on the current host machine.

6.3.3 The Agency

The Agency is a functionality engine of the BORG System that is generally used as a distributed object. Normally, the Agency will run as a daemon process on every machine in the BORG System. Clients will use the `local_agency` to connect to and interact with the Agency daemon on the current host. Clients do not need to be aware that the Agency functionality engine is running as a daemon process, nor will they invoke its methods directly.

Figure 6.5 shows the IDL `interface` representing the Agency functionality engine. The Agency's IDL `interface` describes the **request** and **notify** operations. Both operations take a `base_token` as a parameter, and the **notify** operation takes an additional `base_message`. The `base_token`s are used by the Agency's implementation to bind to FEIs. Once bound, in the case of the **request** operation, a CORBA proxy representing the FEI is returned. In the case of the **notify** operation, the FEIs are notified with the message represented by the message descriptor parameter.

Figure 6.6 shows the implementation class of the Agency functionality engine. It declares member functions that implement the operations of its IDL interface, as well as a member function, **register_fe**, that registers tokens and functionality engines within an internal data structure. This internal data structure is the **employees_** private data member. **employees_** is a template hash table whose buckets contain an instance of the `token_emp_bind` class, shown in Figure 6.7. The hash table contains a bucket for each functionality engine that has been registered. The buckets contain a token representing the functionality engine and a pointer to a function which is used to bind to an FEI. For example, a bucket for the employee functionality engine would contain an `employee_token` and a pointer to a function that binds to an Employee FEI. Figure 6.8 shows the function that binds to an Employee FEI and is registered in the Agency's hash table. The function forwards the bind request to the `_bind` member function. This extra layer is provided to abstract any code that may be specific to the ORB implementation being used.

```
// BORG System's Agency IDL definition.
interface agency
{
    employee request( in base_token bt, in short find );
    void     notify( in base_token bt, in base_message bm );
};
```

Figure 6.5. The Agency Functionality Engine IDL Interface

```
class Agency : public _sk_agency
{
public:
    Agency();
    ~Agency() {}

    /* Public member functions. */
    virtual void register_fe( const token & t,
                              employee * (*bfp)(const char*) );

    /*
     * Public CORBA interface member functions.
     */

    /* Member functions implementing the agency IDL interface. */
    virtual employee * request( const base_token & bt,
                                CORBA::Short       find );
    virtual void       notify( const base_token & bt,
                               const base_msg & msg );

    // . . .

private:
    /* Private member data. */
    hash_table<token_emp_bind> employees_;
};
```

Figure 6.6. The Agency Functionality Engine Implementation Class

```
class token_emp_bind
{
public:
    token_emp_bind( const base_token &bt,
                    employee* (*bfp)(const char*) );
    ~token_emp_bind() {}

    /* Public member functions. */
    employee*       bind( const base_token &bt );

    /* Accessor member functions for private data. */
    const token          &tok() const { return t_; }

private:
    /* Private Data. */
    token                     t_;

    /* Private member functions. */
    /* This is a pointer to a function used to bind to an FEI. */
    employee * (*bind_fn_ptr)(const char*);
};
```

Figure 6.7. Agency's Hash Table Bucket Class Declaration

```
employee *
Employee_bind_to( const char *obj )
{
    return employee::_bind( obj );
}
```

Figure 6.8. Function Registered with the Agency for Binding an Employee

Figure 6.9 shows the Agency's registration process that takes place in the `main` function of the CORBA server containing the Agency's FEI.

When it is necessary for the Agency to contact an FEI, either because of a request or notify operation invocation, the agency will look up the specified functionality engine in the hash table. If a bucket is retrieved by the lookup, the Agency will use the bucket to obtain a handle to a specific FEI. Figure 6.10 shows the code used by the Agency to lookup and contact an FEI. The `base_token_key` used in the lookup takes a `base_token` and converts it into a key that is used in the lookup. Once a bucket has been retrieved, the **bind** member function of the `token_emp_bind` is called to obtain the handle to the FEI specified in the `base_token`. The Agency's **request** operation will return the handle to the client. The **notify** operation will invoke the `employee` **message** operation on this handle (see Section 4.3.1 for a description of the `employee` **message** operation).

By incorporating tokens, messages and the `local_agency` into their design, BORG clients are able to access any FEI in the system. In Chapter 7 we will look in depth at these clients, examining several current examples from the BORG System.

```
/* agencysrv.C */

#include <agency.H>         /* Agency; Agency functionality
                             * engine implementation class.
                             */
/* Token includes. */
#include <employee_token.H> /* Declares employee_token and
                             * Employee_bind_to
                             */
// . . .

int main( int argc, char *argv[] )
{
    // . . .

    /* Instantiate the Agency functionality engine's
     * implementation class (Agency FEI).
     */
    Agency a;

    /* Instantiate the derived token classes. */
    employee_token  et;
    // . . .

    /* Register tokens and their bind functions with
     * the Agency.
     */
    a.register_fe( et, &Employee_bind_to );

    // . . .
}
```

Figure 6.9. The Registration Process of the Agency

```
{
    /* bt is a const base_token &, received as a parameter to
     * the Agency's request or notify operation.
     */

    // . . .

    token_emp_bind *teb = employees_.lookup( base_token_key( bt ) );

    /* Bind to an employee. */
    employee * e = teb->bind( bt );

    // . . .
}
```

Figure 6.10. Lookup and Binding in the Agency

CHAPTER 7

CLIENTS

This chapter discusses clients of the BORG System. We first examine what constitutes a client, followed by what parts of the BORG System are available to clients, finishing with some examples of current clients of the BORG System.

7.1 What Is a Client?

BORG clients are programs or processes that use one or more FEIs to accomplish specific elements of their design. Theses clients may be software applications or functionality engines.

Applications are clients if they use an FEI as part of their implementation. For example, an application that computes the visibility of objects in a scene may be required to render the scene after completing the visibility computation. A functionality engine that defines a rendering service may be used by the visibility application in order to complete the application's required action of rendering the scene.

Functionality engines are also considered to be BORG System clients if part of the service they define is implemented using a service provided by an FEI. For example, a functionality engine that defines an interactive modeling service may use a functionality engine that defines a model viewing service in order to display the models it creates. Since the Agency is a functionality engine, a client that uses the Agency is a BORG client whether or not it uses the service of any other functionality engine.

7.2 The Structure of a Client

The first functionality engine that any client will interact with is the Agency. A client will use an instance of the `local_agency` class as a proxy to the Agency functionality engine. Figure 7.1 shows code that a client might use when requesting the services of the Render functionality engine. Figure 7.2 shows code a client might use when sending a message to a set of Viewer functionality engines.

A BORG System application client will often use the services of the Agency as part of the implementation of a local function. For instance, Figure 7.3 shows an example of an application that requests the services of the model repository in a function called `get_model`. In `get_model`, the `local_agency` is instantiated along with a `model_rep_token`. The token is then passed to the `local_agency` as a parameter to the **request** member function. The **request** member function returns a handle to a Model Repository FEI, which is then queried for a specific model.

Functionality engines are always BORG clients. Each FEI has, as part of the private member data section of its C++ implementation class, an instantiation of the `local_agency`. The `local_agency` can be used to help implement the service defined for the functionality engine by requesting or notifying FEIs.

7.3 Examples of Functionality Engines

This section describes three BORG System functionality engines. They are the *Controller*, the *Scl*, and the *Viewer*. The Controller funnels the actions from a user interface to appropriate functionality engines. The Scl is an Alpha_1 c_shape_edit modeling interpreter, and the Viewer displays Alpha_1 geometric models.

7.3.1 The Controller Functionality Engine

The *Controller* is a functionality engine with a graphical user interface that passes events to functionality engines such as the Viewer, the Model Repository and the Scl. Figure 7.4 shows the default interface of the Controller. This interface is written with Tcl/Tk and can easily be configured to allow for additional features.

Figure 7.5 shows the IDL `interface` declaration for the Controller functionality

```
{
    . . .
    local_agency agency;  // Instantiate local_agency.
    render_token rt;      // Instantiate render_token.
    . . .
    /* Use agency to request a Render FEI. */
    render *rend = agency.request( rt, EXISTING_OR_NEW );
    . . .
}
```

Figure 7.1. Client Code to Request the Services of a Functionality Engine

```
{
    . . .
    local_agency agency;  // Instantiate local_agency.
    viewer_token vt;      // Instantiate viewer_token.
    deactivate_msg dm;    // Instantiate deactivate_msg.
    . . .
    /* Use agency to notify Viewer FEIs to deactivate. */
    agency.notify( vt, dm );
    . . .
}
```

Figure 7.2. Client Code to Notify a Set of FEIs

```
a1_object * get_model( char * model_name )
{
    local_agency agency; // Instantiate the local_agency.
    model_rep_token mt;  // Instantiate a model_rep_token.

    /* Request a handle to a Model Repository FEI using the
     * local_agency.
     */
    model_rep * repository = agency.request( mt, EXISTING_OR_NEW );

    /* Query the Model Repository FEI for a specific model. */
    return repository->query( model_name );
};
```

Figure 7.3. Function get_model Using the Service of the Model Repository Functionality Engine

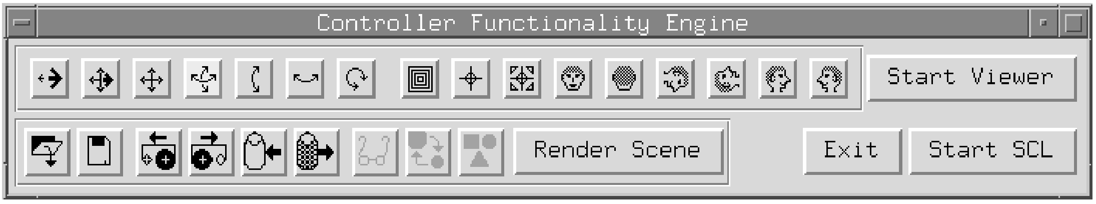

Figure 7.4. Graphical Interface of the Controller Functionality Engine

```
interface controller : employee
{
    // Declared asynchronous to avoid blocking.
    oneway void eval_tcl_code( in string tcl_code );
};
```

Figure 7.5. Controller Functionality Engine IDL Interface

engine. The sole operation declared for the Controller is **eval_tcl_code**. This operation allows a Controller FEI to be used as an enhanced Tcl/Tk interpreter. The Controller will evaluate Tcl/Tk code, as well as special "controller" events. The "controller" events signal actions unique to the Controller functionality engine. For example, Figure 7.6 shows the Tcl/Tk code that creates the "Start SCL" button. The button's command is "controller" and has an argument, "start_scl". When this command is interpreted, the Controller will request an Scl FEI through the Agency. All subsequent events that require the processing of SCL code will be sent to that Scl FEI.

Figure 7.7 shows the implementation class of the controller functionality engine. The public section of the class declares member functions that implement the `employee` IDL interface as well as member functions that implement the Controller IDL interface. The private member data contain handles to FEIs used by the Controller as well as a `local_agency` and an `a1_corba_stream`.

7.3.2 The Scl Functionality Engine

The *Scl* is a functionality engine that interprets the Alpha_1 Shape_edit Command Language (SCL) via the Alpha_1 c_shape_edit interpreter. It uses the Viewer functionality engine to visually display the generated models. Figure 7.8 shows the IDL interface of the Scl functionality engine. Clients can use the Scl functionality engine to interpret SCL code (**eval_code**), obtain the SCL code that generated a specific model (**model_code**), obtain the SCL model type name of a model (**type_name**), or get all of the models created by this Scl functionality engine (**all_models**). The Scl functionality engine can be used in any BORG client, but has a convenient graphical user interface through the Controller functionality engine.

```
button .startscl -text "Start SCL" -command {controller start_scl}
pack .startscl -in .newabframe -side right
```

Figure 7.6. Tcl/Tk Code for the Controller "Start SCL" Button

```
class Controller : public _sk_controller
{
public:
    Controller( local_agency * age );
    ~Controller();

    /* General object member function.
     * This member function is required in order to use
     * the Tcl/Tk interface.
     */
    static int controllercmd( ClientData clientData,
                              Tcl_Interp * i,
                              int argc, char *argv[] );

    /*
     * Public CORBA interface operations.
     */

    /* Member functions implementing the employee IDL interface. */
    virtual base_token* this_token() {
        return new base_token( t_.base() );
    }
    virtual void        message( const base_msg & msg );

    /* Member functions implementing the controller IDL
     * interface.
     */
    virtual void eval_tcl_code( const char * tcl_code );

private:
    /* Private Member Data. */
    local_agency *      a_;   // local_agency.
    a1_corba_stream *   acs_; // For serializing Alpha_1 objects.

    viewer *            v_;   // Viewer FEI.
    model_rep *         m_;   // Model Repository FEI.
    scl *               s_;   // Scl FEI.
};
```

Figure 7.7. Controller Functionality Engine Implementation Class

```
interface scl : employee
{
    void        set_viewer( in viewer v );
    string      eval_code( in string scl_code );
    string      model_code( in string model_name );
    string      type_name( in string model_name );
    a1_object   all_models();
};
```

Figure 7.8. Scl Functionality Engine IDL Interface

Figure 7.9 shows the declaration of the `corba_scl_obj` class, the implementation class of the Scl functionality engine. It should be noted that the `corba_scl_obj` class does not inherit from the server side skeleton class of the CORBA proxy object. It has its own inheritance hierarchy and as such, uses the TIE approach to skeleton-implementation class connection. See Section 4.2.4 for a discussion of the TIE approach.

Figure 7.10 shows the implementation of the **eval_code** operation. **eval_code** is basically a forwarding function that passes its invocation up to the parent `scl_obj` and returns that result. Figure 7.11 shows the implementation of the **all_models** operation. Here, the parent `scl_obj` member function, **get_all_models** is called. The return value Alpha_1 `model_list_obj` is serialized into a byte stream by the `a1_corba_stream`, which then returns an `a1object` to the client. Section 4.3.2 discusses `a1_corba_stream` and the `a1object`.

7.3.3 The Viewer Functionality Engine

The *Viewer* is a functionality engine that visually displays Alpha_1 geometric models. Figure 7.12 shows the IDL interface of the Viewer functionality engine. Clients may send Tcl/Tk commands (such as those that change the current view) to the Viewer via the **eval_script** operation. Alpha_1 objects, such as those obtained from the Model Repository, can be viewed in a Viewer FEI with the **show_obj** operation. A client may even query the Viewer for a list of the models that have been selected in the Viewer's display window (**get_selected_models**).

```
class corba_scl_obj : public scl_obj
{
public:
    corba_scl_obj( local_agency * age );
    ~corba_scl_obj();

    // . . .

    /*
     * Public CORBA interface operations.
     */

    /* Member functions implementing the Employee IDL interface. */
    virtual base_token * this_token() {
        return new base_token( t_.base() );
    }
    virtual void         message( const base_msg & msg );

    /* Member functions implementing the Scl IDL interface. */
    virtual void        set_viewer( viewer * v );
    virtual char *      eval_code( const char * scl_code );
    virtual char *      model_code( const char * model_name );
    virtual char *      type_name( const char * model_name );
    virtual a1_object * all_models();

private:
    /* Private member data. */
    local_agency *      a_;   // local_agency
    a1_corba_stream *   acs_; // For serializing Alpha_1 objects.
    viewer *            v_;   // Viewer FEI.

    /* Static member data for initialization of the SCL
     * interpreter.
     */
    static model_pkg_type * pkgs_[];
    static int              n_pkgs_;
};
```

Figure 7.9. The SCL Functionality Engine Implementation Class

```
corba_scl_obj::eval_code( const char * scl_code )
{
    /* Forward invocation to parent scl_obj. */
    char* res = scl_obj::eval_scl_code( scl_code );

    /* Return result. */
    return res;
}
```

Figure 7.10. Implementation of SCL Functionality Engine eval_code Operation

```
a1_object*
corba_scl_obj::all_models()
{
    /* Forward invocation to parent scl_obj. */
    model_list_obj* mlo = scl_obj::get_all_models();

    /* Instantiate the a1_corba_stream used to serialize Alpha_1
     * objects.
     */
    if ( acs_ != NULL )
        delete acs_;
    acs_ = new a1_corba_stream();

    /* Serialize model_list_obj into byte stream contained in
     * a1_corba_stream.  The serialization takes place below the
     * surface using acs_.
     */
    mlo->dp_obj();

    /* Since we are now done serializing the object, we can
     * delete it.
     */
    delete mlo;

    /* Return an a1object. */
    return acs_->a1obj_to_ORB();
}
```

Figure 7.11. Implementation of SCL Functionality Engine all_models Operation

```
interface viewer : employee
{
    void      show_obj( in a1_object a1obj, in short hint );
    void      eval_script( in string script );
    a1_object get_selected_models();
    void      connect_to_controller( in controller c );
};
```

Figure 7.12. Viewer Functionality Engine IDL Interface

Figure 7.13 shows the implementation of the **show_obj** operation. The **a1obj** parameter of the **show_obj** operation is a list of Alpha_1 objects that has been serialized into a byte stream in the client application. The objects to be shown are read from the **a1obj** via the `a1_corba_stream` and displayed in the Viewer's canvas. The **hint** parameter tells how the objects should be displayed.

The Viewer is also a multithreaded functionality engine like those described in Section 4.3.3. This permits direct interaction with the functionality engine via the mouse or keyboard. Like the Scl functionality engine, the Viewer functionality engine can be used in any BORG client, but the Controller functionality engine provides a convenient graphical user interface.

```
void
Viewer::show_obj( const a1_object & a1obj, CORBA::Short hint )
{
    /* Instantiate the a1_corba_stream used to serialize Alpha_1
     * objects.  Initialize it with the a1obj so that we can
     * read the objects contained in a1obj. */
    if ( acs_ != NULL )
        delete acs_;
    acs_ = new a1_corba_stream( a1obj );
    object_type * obj = NULL;

    /* Use the a1_corba_stream to get the objects from a1obj. */
    while ( obj = acs_->read_obj() ) {
        /* Show, highlight, unshow or unhighlight to object */
        switch ( hint ) {
        case H_SHOW:
            canvas_->win()->show( obj->dl_obj() );
            break;
        case H_HIGHLIGHT:
            canvas_->win()->highlight( obj->dl_obj() );
            break;
        case H_UNSHOW:
            canvas_->win()->unshow_from_all( obj->get_seg() );
            break;
        case H_UNHIGHLIGHT:
            canvas_->win()->unhighlight( obj->get_seg() );
            break;
        case H_NONE:
        default:
            break;
        }
        obj = NULL;
    }

    /* Process all display events and redraw the scene in the
     * window if necessary. */
    dlm.flush_dev();
}
```

Figure 7.13. Implementation of the Viewer Functionality Engine show_obj Operation

CHAPTER 8

RESULTS, CONCLUSIONS AND FUTURE WORK

Modern software technology allows distributed software components to transparently communicate. Technologies such as ACE, CORBA and OpenDoc abstract the low-level network communication infrastructure to provide the application programmer with an encapsulated distributed object communication system. Unfortunately, many legacy systems are not component based and cannot readily benefit from the advantages afforded by distributed object computing. Under these conditions, a certain amount of restructuring and even reengineering is often necessary.

The BORG System facilitates the conversion of a legacy system into a distributed object model. Services of the Alpha_1 Software System are identified and molded into BORG functionality engines. BORG provides tools that permit functionality engines to be used as distributed objects. Through a bulletin board, or Agency, applications are able to obtain handles to functionality engines which can then be exploited for the service they provide.

BORG was written using CORBA as the basis for transparent communication between the distributed functionality engines and a client application. BORG functionality engines are implemented as CORBA servers. Client applications interact with a functionality engine via a CORBA proxy object. The service provided by a functionality engine is used by clients via an interface defined in IDL. The IDL interface wraps the legacy service in an object-oriented interface.

As an abstract concept, the conversion from legacy system to distributed object model is fairly simple. Each legacy service is thought of as an object that provides a specific interface and can be transparently used in an application. In practice however, it is slightly more difficult. The interface of a functionality engine may be easy to design, yet the implementation proves to be more difficult.

For example, Alpha_1 services often translate into a natural object-oriented interface. Yet, these services tend to pull functionality from throughout the software system making it a complex operation to wrap legacy code into the object-oriented interface. In the case of the Render functionality engine, the old Alpha_1 render service was divided among several directories and libraries. The Render functionality engine takes the `main` render program and splits its code into different sections of the object-oriented interface. It is essentially a shell that provides object-oriented access to the legacy service below.

Functionality engines like the Model Repository were easier to create since there was no corresponding legacy program. Functionality segments were grafted from the legacy system into the object-oriented interface. All of the Model Repository's functionality is completely contained within its C++ implementation class and relies only on the legacy system for library support.

Other difficulties arise in the conversion from legacy system to distributed object model due to the semantic mismatch between IDL and C++. C++ combines data and operations into one structure, the `class`. IDL separates the concepts of data and operations into `structs` and `interfaces` respectively. Interfaces support inheritance and are instantiated as CORBA servers, but structs do not support inheritance and are translated to C structures. Thus, one cannot directly map from a C++ class to a combination of IDL interfaces and structs since IDL has no way of representing the polymorphic nature of the data contained in a C++ class hierarchy. Legacy systems that wish to directly translate C++ classes into IDL must provide workarounds on a case by case basis to circumvent this shortcoming.

CORBA is still an emerging standard. There are several implementations on the market, including ORBeline, that advertise compliance with the recent CORBA 2.0 standard. Part of the 2.0 standard was the Internet Inter-ORB Protocol (IIOP) which allows ORBs produced by different vendors to interoperate. IIOP may have the effect of moving CORBA object technology down into the the operating systems of workstations[4] which would provide worry free CORBA object interaction across all platforms, regardless of the ORB. Such industry support for the CORBA standard ensures the future of distributed object technology.

With industry support will come improved CORBA implementations. Current CORBA implementations are slower on the `ttcp`[20] protocol benchmarking tool for end-to-end data transfer throughput than are traditional C sockets[16]. This is mostly due to the overhead incurred by most CORBA implementations in areas such as fragmentation/reassembly, marshaling, and demarshaling[16]. For large data transfer systems such as those used in multimedia applications, increased overhead can result in diminished performance. Future advancements and enhancements to the CORBA standard and its implementations will certainly make it a viable option for all types of applications.

As future work, functionality engines designed and implemented for this thesis can be expanded and enhanced while other services of the Alpha_1 System can be outlined and converted into BORG functionality engines. As an example, the Viewer functionality engine could be enhanced to provide more than one viewing window on more than one machine. Multiple users could then interact with one Scl functionality engine that was connected to this "Multi-Viewer" functionality engine, producing a design environment where multiple users collaborate in the design of a model.

Several companies, including ORBeline, have or are now working on connecting CORBA objects with the Java language. This facilitates the use of CORBA objects in World Wide Web browsers such as Netscape while presenting new possibilities for interaction and collaboration. For example, Alpha_1 functionality engines could be made available to modelers and designers in the Science and Technology Center (STC) for Computer Graphics and Scientific Visualization. Collaborative projects might occur where users at the different STC sites can combine and use the services provided by the other sites via the World Wide Web.

The implementation of the BORG System in this thesis consists of 7 functionality engines, each described by an IDL `interface`. Three other IDL files were created to describe the `a1object`, the `base_token` and the `base_msg` structures. In total, about 150 lines of IDL description were written. The BORG System also consists of about 4000 lines of C++ code that was specifically written for this thesis. Some of this C++ code converts legacy system services into distributed object models and as such, replicates certain segments of the legacy code.

Distributed object computing has many benefits including composeability and scalability[1], collaboration, performance, and reliability[17]. The BORG system attempted to bring these advantages into the Alpha_1 Geometric Modeling and Manufacturing Software System. By defining legacy services and converting them into BORG functionality engines, Alpha_1 application programmers are able to utilize these services in a "plug and play" fashion. Complex applications, like the Controller functionality engine, are easily created, while new services can be cleanly integrated into any existing application.

On a broader scale, this thesis has shown that CORBA is a viable solution for any software system wishing to take advantage of the power of distributed computing. In addition, functionality engines have been shown to be an appropriate tool for legacy systems that could benefit from conversion to an object model. In the future, this kind of distributed component software design, use, and reuse will be the cornerstone of all software development.

REFERENCES

[1] ANSA. *ANSA Reference Manual Release 0.03 (Draft)*. Alvey Advanced Network Systems Architecture Project, 24 Hills Road, Cambridge CB2 IJP, UK, 1987.

[2] Barr, A., Cohen, P., and Fiegenbaum, E. *The Handbook of Artificial Intelligence*, vol. 4. Addison-Wesley, 1989.

[3] Belisle, D. J. Omg standards for object-oriented programming. *AIXpert* (August 1993).

[4] Brando, T. Comparing dce and corba. Tech. Rep. MP95B-93, The MITRE Corporation, March 1995.

[5] Brockschmidt, K. *Inside OLE2*. Microsoft Press, 1994.

[6] Coulouris, G., and Dollimore, J. *Distributed Systems: Concepts and Design*. Addison-Wesley, 1988.

[7] Guttman, M., and Matthews, J. R. *The Object Technology Revolution*. Schowalter, 1995.

[8] IBM. Overview of som. *AIXpert* (August 1995).

[9] Mowbray, T., and Zahavi, R. *The Essential CORBA*. John Wiley and Object Management Group, 1995.

[10] Object Management Group. The common object request broker: Architecture and specification, revision 1.2. Tech. Rep. OMG TC Document 93-12-43, The Object Management Group, Framingham, MA, 1993.

[11] Orfali, R., Harkey, D., and Edwards, J. *The Essential Distributed Objects Survival Guide*. John Wiley and Sons, 1995.

[12] Ousterhout, J. K. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[13] Piersol, K. A close-up of opendoc. *BYTE Magazine 19*, 3 (March 1994), 183–4+.

[14] PostModern Computing. *The ORBeline2.0 User's Guide*. Mountain View, CA, 1995.

[15] Rich, E. *Artificial Intelligence*. McCraw-Hill, 1983.

[16] Schmidt, D., Harrison, T., and Al-Shaer, E. Object-oriented components for high-speed network programming. In *Proceedings of the USENIX Converence on Object-Oriented Technologies (COOTS)* (June 1995).

[17] Schmidt, D., and Vinoski, S. Object interconnections (column 1). *C++ Report* (January 1995).

[18] Schmidt, D. C. The adaptive communication environment. In *11th and 12th Sun User Group Conference* (December and June 1993).

[19] Stroustrup, B. *The C++ Programming Language*, 2nd ed. Addison-Wesley, 1991.

[20] USNA. *TTCP: A Test of TCP and UDP Performance*, December 1995.