BeanBag

An Extensible Framework for Describing, Storing and Querying Components

Caroline O'Reilly B.A. (Mod)

A dissertation submitted to the University of Dublin,

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science

September 1999

Declaration

Permission to lend and/or copy

Abstract

When it became obvious that Object Oriented software applications were falling short of their promise of significant software reuse, the software industry began to look at another solution. Hence the interest in Component Technology. If developers have the means to use off the shelf components, application development time can be reduced and quality will improve. Components are by nature modularised and are more maintainable than other software applications, with easy integration of component versions. When companies want to minimise the expense of developing in-house solutions and the inflexibility of bought-in software, components are introduced, allowing assembly of components into customised solutions.

Presently, there are three main component driving forces, the OMG with the CORBA Component Model (CCM), Microsoft with COM Components (COM) and Sun with Enterprise Java Beans (EJB).

Enterprise Java Beans is the Java component architecture for developing server side components, as opposed to Java Beans, which is used to develop client side components. Threading, persistence, and security are handled by an EJB Container.

All types of components will have to reside in a repository for application developers to find them. In time it will become crucial to be able to query this repository and to view component descriptions. As the situation stands, the EJB Deployment Descriptor does not provide enough semantic information for a developer to understand exactly what a component does.

This thesis examines ways of storing components and extracting information transparently from them. It also examines ways of describing component semantics in a way that is extensible. Developers in the future, who realise that certain semantic information is crucial in a component description can create descriptions in a controlled way, and other developers can search this data. The descriptions developed are XML documents, and can be applied to all types of components, even though this thesis focuses on EJBs primarily.

Acknowledgements

Many thanks to my supervisor Dr. Paddy Nixon and Sotirios Terzis for their valuable ideas and encouragement. Thanks also to Malcolm and Peter from IONA for their XML and EJB expertise and for always having the answers. The M.Sc. class was invaluable in the final weeks for their humorous bantering in the early hours. Lastly, thanks to Niall for those gourmet sandwiches throughout the year and for many words of advice.

1. Introduction

1.1 Motivation

Anything less than complete commitment will doom component-based development to be yet another failed software engineering panacea [Kie98].

With the growing interest in software components in the software industry, ways of storing and retrieving relevant components is becoming increasingly important. Retrieval methods are crucial for reuse. As libraries of components grow in size, traditional information retrieval methods will not provide adequate precision and recall. Procedures are needed for the effective description of components and their roles if they are to achieve the promise of reuse. Describing syntax alone, as IDL does is not sufficient for many applications. What is required is a way of describing both the syntax and the semantics of components in a language independent way. These descriptions could be stored and queried by interested parties.

1.2 Proposed Solution

The objective of this thesis was to examine ways of storing, describing and querying components. The storage framework was designed with efficient retrieval in mind. The component description had to be component independent and extensible, and the querying facility had to provide flexibility.

To achieve these requirements the implementation of BeanBag went through a series of phases. The first was to choose a component framework to work with and to examine what information about a component was available already. Next, methods were investigated for transparently and automatically extracting syntactic information that was already available in a component. The next phase was to design a component independent description which was extensible, as it is impossible to predict what semantic information a developer may need in a description. When the description was complete the storage framework was designed to provide efficient retrieval of descriptions and a module for matching components was implemented. Finally the system was implemented as a CORBA Server.

1.3 Achievements

The main aim of this thesis was to investigate an extensible way of describing components. This requirement was achieved and in addition the descriptions are also component independent. The storage framework was designed in such away as to achieve efficient retrieval of components. The requirement of transparent insertion of components was achieved by the BeanBag system automatically extracting any data that was already available, generating extra data from the component and storing it in the database.

1.4 Roadmap

The chapter that follows describes the motivations and advantages behind component software. Also described are the component frameworks available and the current methods of describing what components provide. Chapter 3 outlines the design steps behind the BeanBag system and Chapter 4 outlines the implementation route and technologies used. Chapter 5 is an evaluation of the thesis and contains concluding remarks about the work and the directions that it could take in the future. The final chapters contain the Appendix.

1.5 Summary

In this chapter, the problem statement was outlined, as were the requirements of the BeanBag system. The steps taken to arrive at a system that fulfilled these requirements was outlined and whether the requirements were met.

2. Literature Survey

2.1 Introduction

In this section the motivation behind the current interest in component technology is discussed. The technologies and methodologies currently available for describing and constructing components are investigated. Areas that are examined are Component Frameworks, Component Description Languages, XML and XML query languages. It is necessary to look at what technologies are available and what they offer, before designing the BeanBag system.

2.2 Software Component Motivation

Components composed of other components allow thinking about problems at the appropriate level of abstraction. Without this, we wind up thinking about individual transistors instead of chips and boards [Chap97].

Component based development changes the software development process to be more industrial like. Developers buy pre-built components and assemble them to their own requirements. Speed of delivery improves, as does the quality and quantity of systems developed. If a large part of a software project can be fulfilled using pre-existing components, then skill sets will reduce. Component technology is often likened to that of computer hardware design, where a set of pre-built components exist such as motherboards and hard drives, and these are assembled to produce PCs. Redesigning and rebuilding these hardware components would be expensive and time-consuming. Instead one can configure existing components to fix the desired requirements.

Object-Oriented technology has fallen short of its promise of reuse, hence the emergence of components. If developers can buy pre-tested software components off the shelf to build their own applications, the software development cycle time will be significantly reduced. With the diffusion of components, complex systems will yield to modular components.

Most software projects result in software being bought in or built in-house. Each direction brings advantages and disadvantages. Software that is built in house is generally expensive, with customer requirements constantly changing and the development time being long. The resulting system may not interoperate with future systems but only with those already in-house. However, the system can be extended and re-adapted. When buying in software, the solution generally does not fit perfectly, however it is probably a cheaper solution in the long run, if the manufacturer stays around long enough to support the system. Component software attempts to find a compromise between these two dilemmas. A system can be assembled from pre-created components, but the application developer has scope for customization. Because the system is modularized, upgrades can happen seamlessly when individual component upgrades are released.

For system developers, short development times and transparent upgrades are tremendous gains. If developers are going to buy a software component over developing their own, the component must have significant advantages. Technical superiority, cost, and support contract may all influence the customer. Components will have to appeal to the market if they are to succeed where class libraries have failed.

2.2.1 The Benefits of Components

The technical component market is exploding, during the next five years leading application development organizations will move from building applications from scratch to assembling them from components [Chap97].

In the previous section the motivation behind choosing components was discussed. Component-based development also brings these advantages when designing applications. Component-based development:

- **can produce applications quickly**

- **can result in higher quality and more reliable software.** When third-party components are used, these have already been tested. Even though the application in its entirety must be tested, using components results in higher quality applications.

- **lets developers focus more on business problems.** Programmers do not have to worry about low-level programming details, such as database access and security.

- **can be cheaper than traditional development.** It can take less time and can save money

- **allows easy mixing and matching of languages and development environments.** Components written in one language can be used by another component of a different language, even on another machine. Because components provide a standard packaging model this transparency is possible.

- **offers the best of both alternatives in the build V's buy decision.** Components can be purchased and combined into a customized solution.

2.3 What are Software Components?

Software components are binary units of independent production, acquisition and deployment that interact to form a functioning system [Szy98].

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [Muh96].

Many applications already plug-and-play components. Netscape Communicator allows the installation of new components such as Apple's QuickTime or Macromedia's Shockwave. These components come from different vendors and can be deployed in another application. In this case, the duplication of effort for Netscape would not have been worthwhile and so components bring considerable advantages. Objects are not geared for a plug-and-play architecture. If a developer is to use a new set of class libraries, certain programming experience is required. In general it will only be application programmers that will know how to lever the use of the class libraries. They are not user-friendly or easy to integrate into applications. Components exist as at a different level of abstraction. Component plug-ins are easy to install and provide configuration at a higher level. The customer should not be interested in how the internals of a component actually work or how it is written, but they should be able to assemble components to provide a system tailored to their needs.

Many definitions of components exist, two are included above. The important points to take from these definitions are that components exist independently but can be assembled into a functioning system. They expose interfaces that are implemented internally, and it is through these interfaces that other components interact with them.

2.3.1 Components versus Other Kinds of Reuse

The oldest types of components that we use are procedural libraries. They are independent and in binary form and can be reused by other independent applications. Functions, classes or modules can form components if they are independent and in binary form. Components are generally easier to use than procedural libraries, because often they allow the developer to assemble them visually. Components have better version control than traditional libraries. Using components where libraries could be used allows the programmer to access functionality in a consistent way and also on remote machines.

Another popular form of reuse are class libraries, libraries of objects. These have become the foundation for reuse in C++ and Smalltalk. However, to use a class library effectively, a certain understanding of how the pieces of the library fit together is required. Sometimes the libraries can be complex. Using components makes this process easier, as the developer can often use a visual tool to place the component where it is needed.

Components allow cross-language reuse. Components written in C++ can be used with components written in Visual Basic or PowerBuilder. If the developer uses class libraries, the application must be written in the same language as the class library itself.

C++ makes it easy to reuse source code, but it is not easy to create reusable binary components. Most C++ libraries are shipped in source form, and not in compiled form. The source code is often required to discover how to inherit from an object. Sometimes the libraries are even modified and compiled into a private build of the library. Components are distributed in executable format, the source is not required. In this way, vendors do not have to divulge their source code. If the component is deployed internally in an organisation, the absence of source code can have the advantage of the component not being changed and its business rules are enforced.

As both components and objects are described in common terms the differences between them can seem hazy. [Szy98] outlines what the differences and similarities are.

A component exists as its own entity and is separate from its environment. Components make their services available to other components through *interfaces.* The implementation of the component is hidden and only available through well-defined *interfaces.* A component can provide many different interfaces each one providing a different service to the client. It cannot be partially deployed. A component may contain a set of objects and instanciate objects, or it could be implemented internally in assembly language or could contain procedures. Mutable state is assigned to component through resources. Because the resources are separate from the component itself, changing them does not require the recompilation of the component itself.

2.4 Component Frameworks

Choosing a component model is a critical decision. This choice determines which pool of components you'll be able to select from, what tools you can use to assemble applications using components, and how you'll create your own components [Chap97].

In the previous section, common features of software components were outlined. The design of BeanBag takes into consideration the component frameworks that are currently available. In order to construct a generic design for all components, it is necessary to understand component architectures and to be aware of the differences between existing component frameworks. This section outlines the architectures of the most common component frameworks.

A component framework is a set of interfaces and rules of interaction that govern how components 'plugged into' a framework may interact [Szy98]. Today there are three major forces in the component framework arena:

- The OMG's CORBA Components (CCM)
- Sun's Enterprise Java Beans (EJBs)
- Microsoft's COM.

For effective component assembly, components must be able to interoperate. Obviously without interoperability the component market will be fragmented and less successful. Hence the emergence of component standards such as EJB and CORBA Components.

Before Sun’s EJB Specification is discussed, it is worthwhile mentioning the component standard that preceded it, namely JavaBeans.

2.4.1 Java Beans

JavaBeans is a portable platform-independent component model written in the Java programming language. Developers take advantage of the platform independence of Java and write reusable components once and run them anywhere.

Mike Day defines beans as objects or components created with a set of characteristics to do their own specific job [URL7]. They have the ability to take on other characteristics from the container on the server in which they currently reside. This enables a bean to behave differently, depending on the specific job and environment where you place it.

A JavaBean is a component that has interfaces or properties associated with it. It can be interrogated by and integrated with other beans that were developed by different parties at different times. A JavaBean is different from other objects in that it has a *properties interface.* This interface can be read by certain tools as it describes what the component does. With this information the JavaBean can be hooked up with other beans and plugged into other environments.

JavaBeans, in contrast to Enterprise Java Beans are generally visible at runtime. The visual component can be a button, a list box or a graphic, but the component does not have to be visible. JavaBeans are intended to be local to a single process.

Individual beans will behave differently, but typical unifying features that distinguish a bean are:

- ***Introspection* -** this enables a builder tool to analyze how a Bean works
- ***Customization*** enables a developer, using an application builder tool to customize the appearance and behaviour of a Bean
- ***Events*** enable Beans to communicate and connect together
- ***Properties*** enable developers to customize Beans and program with them
- ***Persistence*** enables developers to customize Beans in an application builder, and then retrieve those Beans with customized features intact for future use.

More information on JavaBeans can be found at the JavaSoft JavaBeans Website [URL5].

2.4.2 Enterprise Java Beans (EJB)

Enterprise JavaBeans take JavaBeans to the next level, that being, to server-based components. The Enterprise Java Beans API allows developers to build large-scale business applications as reusable server components.

2.4.2.1 EJB1.0

The first Enterprise JavaBean (EJB) Specification, Version 1.0, from JavaSoft, was released in March 1998. It defined an API for the development, deployment and management of server side components. The original JavaBeans specification described the standard behavior of Java components that primarily ran on the client side. Over 50 products have already been developed to support EJB1.0. Companies such as Oracle, Borland, Symantec, IBM and IONA have announced and/or delivered products that adhere to this EJB specification.

2.4.2.2 EJB1.1

The EJB1.1 specification is now available to the public. The main changes from EJB1.0 are:

- Mandatory support for entity beans
- Enhanced deployment descriptors which are formatted in XML, rather than being serialized
- Java2 security replaces Java1.1 security

EJB1.1 is available from Sun, and can be downloaded from [EJB1.1].

EJB 2.0 has a due date of late 2000, and future specifications will look at issues of connecting legacy data, integrating messaging systems and representing bean relationships and inheritance.

Previously, developing multi-tier servers was a complex task, where developers had to deal with issues of concurrency, transactions, security and scalability, and had to manage threads, memory and network connections. EJB aims to make the development of server side components easier, by decoupling application semantics from infrastructure issues. However EJB is simply a model and vendors provide EJB solutions.

Enterprise JavaBeans, unlike JavaBeans are generally non-visual and designed to run on a server, and to be invoked by clients. An EJB could be built out of non-visual JavaBeans.

Figure 2.1 illustrates the architecture of EJB with the EJB home and EJB object implementing the home interface and remote interface respectively. The bean class is wrapped by the EJB Object [RMH99].

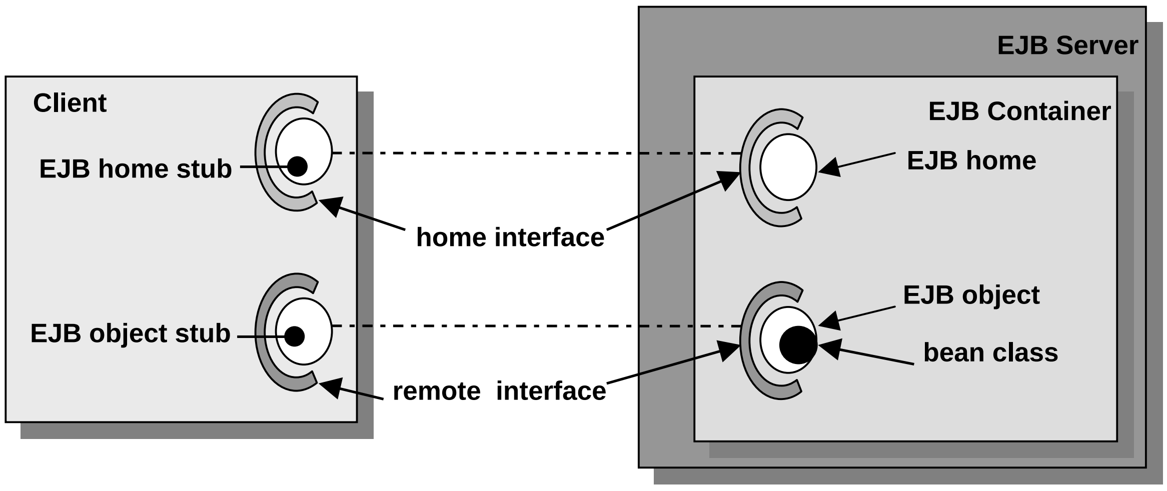


Figure 2-1 EJB Architecture

2.4.2.3 The EJB Client

The writer of an EJB cannot assume what the client will be or on which platform it will be running or on which machine. The EJB client could be a servlet, a client on someone's desktop or could be another EJB. The client could be written in any language, perhaps Java or C++. If the client is written in Java then the client will communicate with the EJB by using Remote Method Invocation (RMI) API. If the client is a non-Java client then it will communicate with the EJB using CORBA communication protocols such as IIOP.

Once the client has the name of the Enterprise JavaBean that it wishes to communicate with, it can create a communication channel with the EJB through a network directory service. Firstly the client receives a reference, like a proxy to the EJBHome object for the bean. By utilising this reference, the client can retrieve a reference to the EJBObject for that bean, and consequently can access the remote EJB's services.

2.4.2.4 The EJB Component

An EJB Component is an *Enterprise JavaBean.* It is a component like a JavaBean, written in Java by a developer that implements some business logic. Components live inside a container, and many component instances can exist in a container.

2.4.2.5 The EJB Container

The EJB component model provides an environment in which server side components, Enterprise Java Beans can be deployed. This environment (the *container*) provides transactional capabilities, security and management to the component. Developers create components that inherit the enterprise attributes from the container. The component interacts with the container and takes advantage of its transactional or security features without being aware of how they are implemented by the container. The EJB container manages the state of the object. An object can have persistent state or transient state. Components can be deployed in another vendor's EJB container and still work, without recompilation, providing it complies with the EJB specification.

2.4.2.6 The EJB Object

On the server side, the EJB object implements the remote interface of the bean. This distributed object is generated automatically by the EJB vendor and by the information provided by the deployment descriptor. The EJB object works with the EJB container to apply transactions, security and other operations to the bean at runtime.
When a client calls a method on an EJB object, the EJB object communicates with the EJB container and requests that the same method be called with the same parameters on the EJB component.

2.4.2.7 The EJB Home Object

The EJB home object is also generated by the EJB vendor when an EJB is installed in a container. This object is involved with the bean's life-cycle, and is responsible for the location, creation and removal of the Enterprise JavaBean.

2.4.2.8 The EJB Server

EJB components reside in an EJB container when they are deployed. The EJB Server must implement the EJB container. Any server that can host an EJB container and provide it with the services that it requires, can be an EJB Server. An EJB Server provides lower-level services such as network connectivity to the component.

2.4.2.9 EJB Design Patterns

There are two types of Enterprise JavaBeans that can be created, the Session Bean and the Entity Bean. This is how they differ:

Session Beans

- are non-persistent objects that implement some business logic
- are associated with a single client
- do not survive server crashes
- manage information relating to the conversation between the client and server
- are divided into two types
    - **Stateless Beans** do not store any information relating to the client between calls
    - **Stateful Beans** store information between calls from a client. Values and state are retained from previous communication.

Entity Beans

An entity bean maps a Java class to a data source. This source could be a row in a database or an entire table. Each entity bean has a primary key associated with it that identifies the data within [URL21].

Entity Beans:

- are persistent objects, which represent an object view of entities in persistent storage
- are associated with database transactions
- may provide data to multiple clients, concurrently
- survive server crashes, because the data that the bean represents is persistent.
- Represent and manipulate persistent application domain data
- Entity Beans are also divided into two classes
  - **Container-managed** is the simplest form of entity bean. Using container managed beans means that developers do not need to worry about how the persistent data is stored or retrieved. The container manages the database calls.
  - **Bean-managed** is used by developers who need more control on how the data is stored or retrieved from the database. The database calls are made from the bean itself and are hard coded in. The disadvantage is that the bean is more tightly coupled with the underlying architecture.

EJB Components are deployed using the Java Beans packaging format (JAR). Server components can be deployed in any vendor’s EJB implementation because the business logic of server components is not tied to the implementation details of a particular system [BW].

2.4.2.10 Interfaces

An EJB's provided methods are specified by the *remote interface*. The EJB object implements the remote interface of the EJB component. In order for the client to be able to create objects on the server, the *home interface* is used. The home interface is a contract between an EJB component class and its container, which defines construction, destruction and lookup of EJB instances [URL6]. Each EJB component has a home interface for this purpose.

2.4.3 The CORBA Components Model (CCM)

The CORBA Component Model (CCM) was approved through a vote by the ORB Task Force at the end of August 1999. CCM is the final piece needed to complete the CORBA 3.0 specification, which is a framework for building, assembling and deploying *plug-and-play* CORBA objects.

The RFP submitted to the OMG was constructed by an influential group of distributed middleware vendors: IONA, BEA Systems, DSTC, Expersoft, IBM, Inprise, Oracle, Rogue Wave and Unisys. As Sun are responsible for the Enterprise JavaBean specification, they cooperated in the submission.

CCM is essentially a language-independent extension of EJB

[URL11]

The problem that many organisations have with EJB, is that it only supports Java. Many organisations have software written in other languages and so EJB is not suited to their environment. CCM supports EJB components as well as components developed in other languages such as Ada, C++, Smalltalk, COM/DCOM and COBOL.

[Hub99] describes the differences between CORBA Components and Enterprise Java Beans, and examines why the OMG are developing their own component standard instead of endorsing the EJB directly. Some of these points are listed here:

1. The CCM designers assert that ‘the JavaBeans model is inappropriate for server-side development’. This may be the case in some ‘demanding situations’ but this does not mean that EJBs are generally inadequate for server-side development.

2. The EJB specification deviates from the OMG charter in the following areas:

   - OMG technologies must strive to integrate multiple language specifications (e.g. Java and C++)

   - OMG standards are approved according to the OMG’s open technology specification process

   - Technologies must be developed in compliance with the OMG’s Object Management Architecture (OMA) and should respect styles such as IDL, IIOP and the CORBA Services.

   - OMG technologies must provide an explicit metamodel based on the standard Metaobject Facility (MOF)

The CORBA Components model like EJB, is related to server-side CORBA Components. CORBA Components extend the core CORBA object model, and provide a higher level of abstraction, thus greatly simplifying CORBA programming. CCM Components use services such as transactions, security, events and persistence.

The CORBA Component model provides:

- Extensions to IDL that will support components and the relations between them
- CIDL, a mechanism for automatic code generation for defining servant implementations
- Extensions to the CORBA core object model
- A means to support navigation across multiple interfaces of a CORBA Component
- It includes a deployment model using XML, to describe the runtime properties of a component.
- It defines a container model
- It introduces the container programming model
- Defines policies that provide a simplified version of CORBA Transactions.
- Defines policies for servant lifetimes, security and persistence
- Defines a mapping to Enterprise JavaBeans so that they can be supported as a CORBA Component

[CCM99]

2.4.3.1 The Component Model

Component is a new meta-type in CORBA, it is an extension of the Object meta-type. A Component can be specified in IDL and stored in the Interface Repository. A component has a component reference, which is analogous to an object reference.

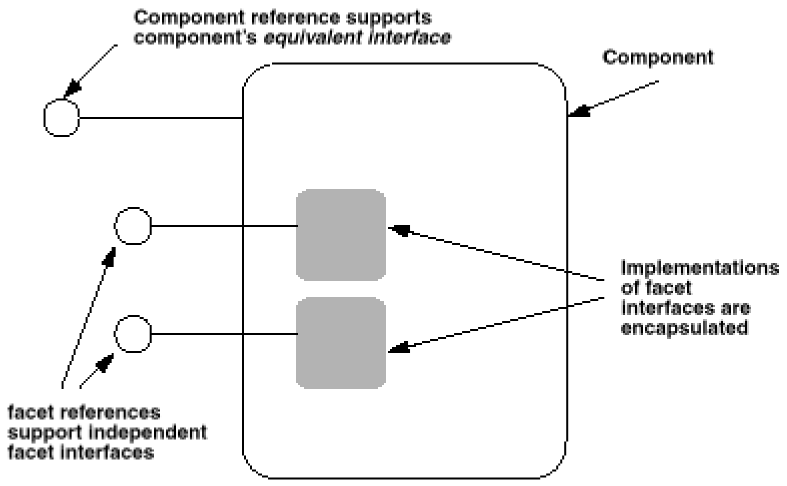

Component reference supports component's *equivalent interface*
Component
Implementations of facet interfaces are encapsulated
facet references support independent facet interfaces

Figure 2-2 The CORBA Component Model

Clients interact with components through surface features called *ports.* There are five types of ports:

- Facets - distinct named interfaces provided by the component for client interaction
- Receptacles - named connection points
- Event sources - named connection points that emit events
- Event sinks - named connection points into which events may be pushed
- Attributes - named values exposed through accessor or mutator methods

2.4.3.2 Facets

A component may provide multiple object references, called facets, which are capable of supporting CORBA interfaces.

2.4.3.3 Equivalent Interface

This interface makes the component's surface features visible to the clients. Other interfaces provided are called facets. Clients can navigate from any facet to the Equivalent Interface, and can obtain any facet from the component's Equivalent Interface.

2.4.3.4 IDL

A component definition in IDL, defines an interface of a component, that supports the features defined in the component definition body. Component definitions differ from interface definitions in that they can only support single inheritance.

A component body can contain the following type of port declarations

- `Provides` - provided interface declaration
- `Uses` - receptacle declarations
- `emits/publishes` - event source declarations
- `Consumes` - event sink declarations
- `Attribute` - attribute declarations

All these IDL declarations map onto operations in the components equivalent interface.

2.4.3.5 CORBA Components Vs Enterprise JavaBeans

The CORBA Component Model and the Enterprise JavaBean Model are actively evolving in parallel. Each new release of the specifications retests the leader/follower relationship. Some of the differences between the two models are detailed in [Hub99].

- CCM uses IIOP as its wiring protocol. EJB is less specific, some EJB based Java application servers require RMI, and others support IIOP, or both. This lack of specification on which to use in EJB adds design complexity.

- CCM containers can be implemented in C++ or Java. This is hardly surprising, as integrating heterogeneous technologies is one if the OMG's primary goals.

- CCM interfaces are specified in Component-IDL (CIDL). CIDL is an extension of CORBA IDL. EJB is based on pure Java interface specifications.

- CCM descriptors are specified in XML using DTDs. They are more extensive than EJB descriptors. EJB uses Deployment Descriptors to configure the Container properties. The Deployment Descriptor in EJB1.1 is XML based. Most vendors however have implemented EJB1.0 in which the Deployment Descriptor is itself a JavaBean and is delivered together with the EJB as a serialized Bean in the JAR file.

- EJB supports two component abstractions, Session and Entity Beans. CCM adds a Process component abstraction, which is like a stateful Session Bean in EJB. CCM Processes will allow more precise control of component lifetime, state management and identity than is possible with standard EJBs.

- CCM specifies properties and behaviours for managing the installation of components including managing software dependencies and component assemblies. EJB is much less complete in this area.

- CCM defines structural and behavioural features for extensive runtime component configuration. These aspects are less distinct in EJB.

- CCM places metadata (information about the component interfaces) in the CORBA Interface Repository, therefore maintaining language independence. EJB uses Java language features and EJB Container features to provide similar information.

- CCM defines a metamodel and expresses it in the XML Metadata Interchange format (XMI)

- CCM uses the CORBA Services (COSS) for Events, Naming, Life Cycle, Persistence, Security and Transactions. In some cases, CCM defines a subset of the COSS with some design changes to tighten integration. EJB has adapted subsets of COSS features in Transactions and Naming. JavaSoft has adapted or redesigned the other COSS themes, which means they cannot be seen as COSS-compatible subsets.

- CCM defines a bi-directional mapping to EJB. EJB specifies a CORBA mapping for client communication and intercontainer interoperability. EJB could not address component model compatibility because CCM did not exist at that time.

2.4.4 COM Components

Microsoft’s component technology has evolved from the non-object oriented Visual Basic components, to object linking and embedding with OLE and now to COM and DCOM. COM is a *binary standard.* This means that it is not concerned with the languages in which the components are written. Components do not have to be object-oriented behind their interface.

2.4.4.1 Visual Basic, OLE and ActiveX

Microsoft's first attempt at component technology was Visual Basic controls (VBXs). VBXs could be embedded in forms and could interact. VBXs are used to implement word processors, database connectivity and charting. However, VBXs are tightly coupled to Visual Basic forms, and so OLE controls were introduced (OCXs). OCXs are COM objects whereas VBXs are not. To qualify as an OLE component, the COM object has to implement many interfaces. The downside is that small controls have to carry extra baggage, and so implementing them is less attractive than VBXs.

After OLE controls came ActiveX controls. ActiveX is a new specification. An ActiveX control need only implement one interface, the *IUnknown*. The control must also be implemented in a self-registering server, so that when the server is started, it registers the classes with the system registry. ActiveX controls have regular COM interfaces, but also have outgoing interfaces, which are used for a notification mechanism.

2.4.4.2 COM

The Component Object Model (COM) is a way for software components to communicate with each other. It is a binary and network standard that allows any two components to communicate regardless of what machines they are running on (as long as the machines are connected), what operating systems the machines are running (as long as it supports COM) and what language the components are written in. COM also provides location transparency, in that it does not matter if components are in-process DLLs, local EXEs or if they are located on another machine.
COM objects are well encapsulated. There is no way to find out how they are implemented internally. To communicate with a COM object, interfaces are used.

The COM *interface* is fundamental to COM. The only way to communicate with a COM object is through an interface. An interface is a *contract* between the component and its clients, that defines what functions are available and what the object does when the functions are called [URL13].

An interface is represented as a pointer to an interface node. The interface node contains a pointer to a table of function pointers, called a *vtable*. The client sees a pointer to a pointer to a vtable, a double indirection. A COM component can implement multiple interfaces. The client never gets a pointer to the class itself, which means that the back end implementation can be replaced unknown to the client.

COM does not support any form of implementation inheritance. This does not mean that you cannot reuse COM components. To reuse them, *containment* and *aggregation* are used. When a class is inherited generally only a couple of methods are needed. Containment means the object required is instantiated, and requests are passed to it. One object contains another object, conceptually, i.e. one object holds an exclusive reference to another object. Passing on the requests to the contained object simply means calling its methods. Containment is completely transparent to the client, and it will not know that the contained object was invoked to carry out the method. If deep containment hierarchies occur, they can become a performance overhead, so COM defines *aggregation*.

Aggregation means that instead of forwarding requests to an inner object, the inner object's interface reference should be given to the client. Calls on this interface are not intercepted by the outer object, and are passed directly to the inner object, which eliminates the forwarding overhead that exists in containment. Transparency is important here also, and the client should be unaware that the interface it uses has been aggregated from an inner object.

In practice containment is used more often then aggregation. Aggregation is generally used where there are deeply nested constructions. Aggregation also brings an extra level of complexity [Szy98].

Every COM object implements the `IUnknown` interface. The identity of the `IUnknown` interface can serve to identify the entire COM object. All interfaces must derive from `IUnknown`. `IUnknown` is the only interface guaranteed to be present. The `IUnknown` interface supports the three mandatory methods of any COM interface, namely, `QueryInterface()`, `AddRef()` and `Release()`.

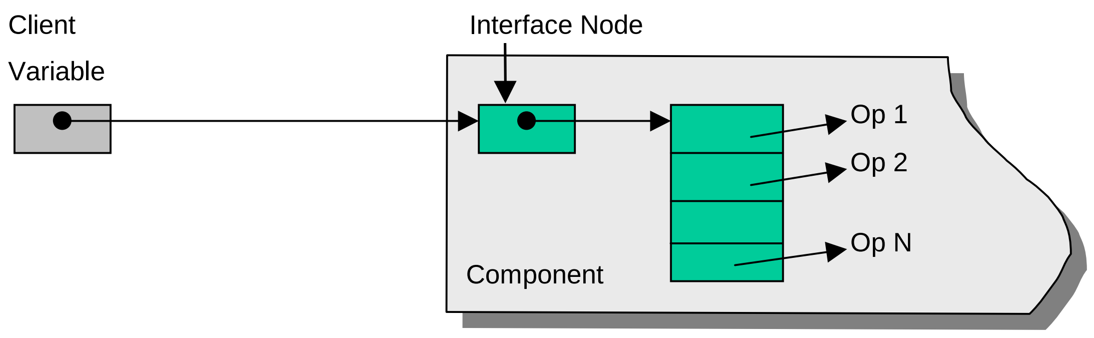


Figure 2-3 Binary representation of a COM interface [Szy98]

Every COM component has a common method called `QueryInterface()`. Given one interface, `QueryInterface()` can be used to obtain a pointer to a different interface. The method checks whether the given interface is supported by the COM component and if it is, it returns the corresponding interface reference. Using `QueryInterface()` the client can navigate from any provided interface to another. Interfaces are represented using *interface identifiers*

Polymorphism in COM is achieved by COM objects supporting a set of interfaces. The type of a COM object is the set of interface identifiers of the interface it supports. If a client requires that four interfaces are supported by a COM object, and the COM object supports these four and four others, then the client's requirements are satisfied. `QueryInterface()` is used to test whether a COM object supports the required interfaces.

The COM interface and its specifications cannot be changed after they have been published. The contract is immutable, you cannot add to it, you cannot delete, and you cannot modify it. You can improve the internal implementation once you still honour the contract. If the requirements change, you can always write a new contract, as COM supports multiple interfaces. COM also allows for the inheritance of interfaces, so an entire interface need not be rewritten.

2.4.4.3 DCOM

COM supports inter-process communication, but not communication across machines. DCOM builds on COM to provide communication across process boundaries and machine boundaries. When communication exists on a single machine, there is no need to know how data types are represented because the sending process is using the same representation as the receiving process. When communication is across machine boundaries, the data representations can be different. Therefore COM creates proxy objects on the client side and stub objects on the server side. To deal with differences in data representations across machines, DCOM marshals data into network data representation (NDR) which is a platform independent format.

2.4.4.4 COM+

COM+ was released by Microsoft in October 1997 and is essentially an extension of the Microsoft Transaction Server (MTS). MTS provides components with services, such as transaction processing monitoring, database connection pooling, and multi-user access. COM+ allows a single threaded object designed for single users to be used by multiple simultaneous clients. New services have been included for COM objects in COM+ such as events, asynchronous messaging, dynamic load balancing, and life cycle management [COM+99].

2.4.5 Commercial Application Servers

Since the CORBA Component RFP became available, vendors such as Fujitsu, IBM, Inprise, IONA, Oracle and Sun, have stated their commitment to either support and/or implement the CORBA Component Model. Since [EJB1.1] many vendors have already implemented Enterprise JavaBeans Application Servers. Examples of commercial Application Servers are:

- HomeBase from IONA. A beta release of OrbixHome was not available at the time of writing. OrbixHome will be a full implementation of EJB1.1.

- BEA WebLogic Enterprise Server, version 4.2 will add support for CORBA Components written in Java or C++. WebLogic Enterprise 5.0, scheduled for end of 1999 will support the Java2 Enterprise Edition platform, which includes EJB1.1.
- Netscape Application Server 4.0 includes EJB1.0 support and support for entity beans.
- Ejipt 1.2 is an all Java, low cost application server that implements both the required and optional features of EJB1.0, including entity beans. With a small footprint of 300K, it can be deployed on a laptop, and supports Java1.1 and 1.2 clients.

2.5 Component Description Languages

A lot has to happen to make software componentization and reuse a reality. One of the biggest hurdles is the lack of standards that let an application know what a particular component can be used for [Kin98].

If components are going to live up to the promise of reuse, then standards are needed for describing what a component does, so that relevant components can be retrieved. Component Description Languages do exist, and in this section a number of them are examined to investigate if they can sufficiently describe the semantics of the components listed in Section 2.4.

2.5.1 IDL

IDL is used to describe the interface of a component. The description is purely syntactic and does not give us any information about the behavior of the component or how components relate to each other. Hence, this language does not fulfil BeanBag's requirements.

2.5.2 OCL

In 1996, the OA&D Domain task force at the OMG issued a request for proposals on Object Analysis and Design. IBM and ObjecTime Limited jointly submitted a proposal in January 1997. An important aspect of this proposal was the inclusion of the Object Constraint Language, or OCL [URL4].

The Object Constraint Language (OCL) is part of the Unified Modeling Language from Version 1.1 onwards. UML includes common constructs for object oriented modeling such as class models, state machines, use cases and collaboration diagrams. UML is now the global standard modeling language of the OMG. Therefore OCL is likely to receive more attention than other normal specification languages such as VDM or Z.

The language is designed to augment class diagrams with additional information that cannot be represented in UML diagrams. A class diagram does not contain enough information to make it an unambiguous representation. Additional constraints about objects in a model are needed, to provide an unambiguous description. These are typically annotated in natural language. Formal languages have been developed to describe these special constraints, but they are not user friendly, and typically require a mathematical background to decipher.

OCL is a formal language that is easy to read. OCL is a modeling language and not a programming language; therefore it is not possible to write program logic in OCL. Because it is a modeling language, the implementation details are beyond the scope of OCL.

OCL can be used for a number of different purposes:

- to specify invariants on classes and types in the class model
- to specify type invariant for stereotypes
- to describe pre- and post- conditions on operations and methods
- to describe guards
- as a navigational language
- to specify constraints on operations:

[URL3]

OCL allows the expression of invariants and pre- and post- conditions that specify the behavior of a model, without getting involved with implementation details. The next section describes how BeanBag could use OCL to provide more semantic descriptions of software components.

2.5.2.1 OCL Invariants

```
Presenter

self.qualifiedFor->includesAll(self.offering.seminar)
```

This invariant for a scheduling system says that a presenter must be qualified for all seminars that he/she is assigned to present. *self* refers to an instance of a presenter.

2.5.2.2 OCL Preconditions and Postconditions

```
SeminarSchedulingSystem

MarkAsAbsent(p : Presenter, from, to : Date)

Pre:  true

Post: p.offering@pre->forAll(o  |  o.date  >=  from  and  o.date  <=  to
implies o.presenter = Set{})
```

In this example OCL is used to specify pre- and post- conditions for the method `MarkAsAbsent`. In OCL, the value of a property at the start of an operation is denoted by, `propertyName + @ + pre`. The post condition in this case marks a presenter as absent by cancelling his/her presentations within specific dates.

OCL fulfils the requirements for BeanBag as it can be used to describe the semantics of methods using invariants and pre- and post-conditions.

2.5.3 JBCDL

JBCDL is a component description language based on the Jade Bird Component Model (JBCOM). JCBOM describes component interfaces. The JBCDL is intended to aid component composition, component verification and component retrieval [QJHF98]. JCBOM describes a component as comprising of seven parts: template parameters, provided functions, requirements, members, connections, imported specifications and implementation. The component definition language was designed with reuse as a primary aim. The language is easy to understand so developers can quickly judge the suitability of the component. The syntax of JBCDL relates to the JBCOM and has 6 specification parts, *template parameters, provides, requires, contains, connection* and *imports.* The implementations are described in programming languages. JDBCL is most suited to describing object-oriented components; hence inheritance is included in the specification, as most OO languages support this.

2.5.4 KDL

KDL is a Component Description Language developed by Joseph R. Kiniry. The language is used to describe both the interface and behavior of software components. The language is an extension of the OMG's Object Constraint Language (OCL). It is used to specify the interface and the externally observable side effects of methods. Expression languages such as Eiffel or OCL use pre- and post- conditions on each method. KDL allows the specification of interfaces such as IDL interfaces plus it can also specify semantic relations of components. KDL includes relationship operators, like those used in UML to denote relations between components (*cdlHasA, cdlContainsA*). Also included are ways to specify if components are related under *cdlIsTypeOf* and *cdlIsKindOf.*

From examining the features of these component description languages, OCL is most suitable as it is has been adopted as a standard by the OMG and it provides the functionality that we require, i.e. representation of pre- and post- conditions and invariants.

There are other ways of describing the semantics of components and providing insights into the functionality of a component. In the next section modelling languages are discussed that could be utilised by BeanBag to extract more information from a component.

2.6 Modelling Languages

The widespread utilization of a specification language, much like a normal computer language, seems to be inversely proportional to the language's complexity - i.e., the simpler the language, the more system builders will use it [Kin98].

Designers need to know more about a component than just its interface. Components and objects can be described in a formal way, so that there is no ambiguity as to their function. Modelling languages have been developed for this purpose.

Current system specification methodologies can be divided into two communities, the informal and formal. Most designers are familiar with UML, which like OOCL and Cataysis belong to informal methodologies. These methods are not concerned with system correctness. OOCL can be used to model larger systems such as organisations, as well as software systems. It adds several diagrams to UML, but fails in the area of component representation and interaction.

2.6.1 UML

UML is used to informally document the interactions between the user and the system, and has become popular because its learning curve is manageable by most designers and because the OMG has adopted it as a modelling language. Plus UML diagrams can be interpreted easily by customers.

Component specification languages such as OCL, VDM and Z are formal specification methodologies, while ignoring object-orientation or component software. These methods are rooted in mathematical notation, and even though they are used to prove system correctness against a specification, they require a steep learning curve for the developer.

UML provides mechanisms for extending system specifications [Kin98]:

- Packages: a collection of model elements
- Stereotypes: indicates a usage or semantic extension
- Constraints: a semantic relationship between model elements that specifies conditions and propositions that are invariant. The conditions are generally described in a formal language such as Z or OCL.

  {*message.oclIsTypeOf(SummonRequest)* }

  $\{\forall n : \mathbb{N} \bullet n + n \in \text{ even.}\}$
- Properties and tagged values
- Notes: graphical symbols that contain information, usually textual, or comments about the model.

UML does allow component diagrams, but it does not have any utilities for describing the relationships that components can have with each other. It does not document a component's inbound and outbound interfaces sufficiently. DESML addressed these issues, and aims to provide a framework for describing components where UML stops.

2.6.2 DESML

[Kin98] describes DESML as a new set of modelling constructs, which can be used on top of other modelling languages. DESML aims to describe a component in a formal way to aid designers, but it also aims to be easy to use and easy to learn. [Kin98] examines the difficulties in designing a system that would not have a steep learning curve. DESML is a variant of UML, not an extension, as the UML core meta model was modified in the process.

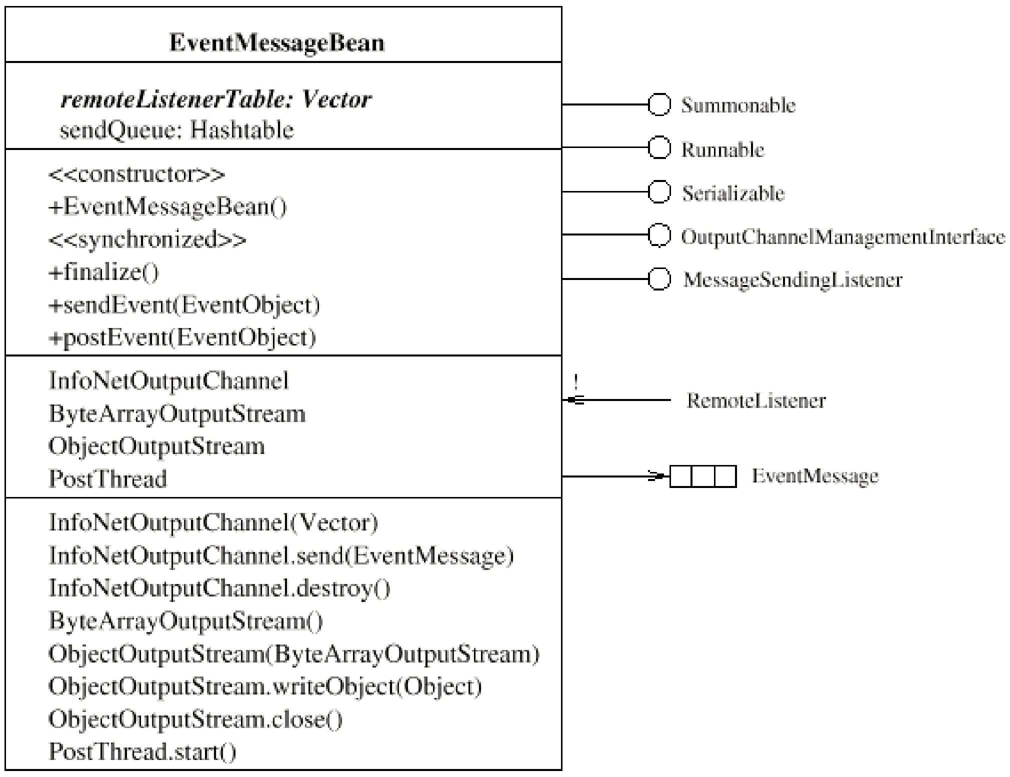


Figure 2-4 Using behavioural elements to denote the interface of a component [Kin98]

In [Kin98] some of the problems of component representation are discussed:

- **Core Component Representation**

[Kin98] describes the features that are missing in UML, when it comes to describing components.

1. The Outbound interface: aka the *needs* interface. There are other elements that the component needs to operate properly.
2. Properties and Attributes: the properties of components usually have special semantic values, i.e. the identifier might be a unique number.
3. Events and Methods: components are connected using events, which require their semantic information to be noted.

4. Dependencies and Associations

- **Partial Component Interface Specification**

A component depends on a set of other components, but not on all the interfaces of all these components. UML does not provide a way of describing this.

- **Component Associations**

Components have associations with other components. The most common associations are containment and aggregation. These associations are defined using stereotypes and constraints. However, these definitions are sufficient for the OO world, but components require more complex associations to be modelled. In [Kin98] a list of possible associations are defined:

- Standard Local Reference
- Garbage Collector Reference Type
  - Guarded Reference
  - Weak Reference
  - Phantom Reference
  - Soft Reference
- Indirect Association
- Renewable Association
- Mobile Association
- Constant Association
- Channel Association
- Event Association
- Method Association
- Tuple Association

    - Reflective Association
    - Meta Association
    - Semantic Association
    - Persistent Association

- **Dynamic and Emergent Structures of Components**

In [Kin98] a new way of describing networks of components is introduced, that of *the Object Network Diagram*. New constructs have been added, such as *Agents,* which represent an autonomous thread of control, *Types* are classifiers of objects, and *Kinds* which add semantics to types.

- **Tying Knowledge/Semantics to Components**

Because there is still room for mis-interpretation in UML, extra semantic information about a component will become crucial to helping developers find the components that they require for their system. The new metaclass *Kind* helps describe if two components are semantically similar.

If UML or DESML descriptions were available to BeanBag, they could be processed and the semantic information they provide could be stored persistently. Details about what other components are required by a component and what its semantic kind is, are of interest to a developer that is searching for a component

2.7 Component Repositories

BeanBag is intended to be a component repository for various components. There are many commercial component repositories available that combine a storage mechanism with version control and visualization. For a repository to be used successfully by developers it should be able to store many types of components, and should contain details about component interoperability, design models and interface specifications. The next section describes research in the area of component repositories and examines the features and shortcomings of some commercial repositories.

2.7.1 A Distributed Repository for Object-Oriented Software Components

[OYM] describes the design of a distributed OO repository. The main point to note is that this repository is distributed among machines, with no central datastore. The repository is used to store OO software components and the relationships between them are represented using hypertext. Useful features include multiviews of the components depending on whether the implementation is public or private.

Difficulties with using this system is that the target language is C++, and the repository is not designed for storing the components described in Section 2.4. Data about the C++ classes is extracted from the source code, such as class declarations, type information, class name, interface definition and dependencies. However this requires that the source code of the software is available, which in real life components, will probably not be the case. This system was in prototype version at the time of publication of [OYM].

2.7.2 DELOS

[Geor99] describes the development of a semantic repository for the DELOS environment, which focuses on integrating legacy systems. DELOS is an environment that supports the development of applications using distributed components that employs a central repository containing component meta-data about IS systems. The repository captures knowledge about the company's business processes, operational knowledge of legacy systems and IS knowledge of system components. The repository is based on the *Semantic Index System* (SIS) which has primitives that represent entity classes, attributes and relationships. Component behaviour is represented by using a Finite State Machine. Queries are constructed as a sequence of nested SQL.

[Geor99] describes how legacy data can be represented in the repository by using reverse-engineering rules to obtain semantics about data. Components are divided into two groups: generic or customised. Generic components may be suitable for reuse in many applications while customized components are specialised to perform a certain task.

A user constructs queries about the legacy systems. Internally the queries are converted from JDBC calls into the native query service that the legacy system supports. The result is returned to the user. Legacy components are wrapped with DELOS wrappers that are Java classes that can receive events from DELOS.

This system focuses on IS components. It is worthwhile to see how this system deals with legacy components. BeanBag must be designed to handle all component types. The DELOS system divides components into two groups depending on how reusable they are. BeanBag should also attempt to categorise components in this way.

2.7.3 Softlab's ENABLER Open Repository

This component repository was designed while considering the heterogeneity of components. The repository provides a component management system with versioning control, information sharing across workgroups and release management facilities. It is an integration framework that spans the desktop, Internet and the network. The components that are stored are a mix of program code, spreadsheets and forms. The WhitePaper for Enabler can be viewed at [URL1]. The repository is not specifically designed for the components that were discussed in Section 2.4.

2.7.4 The Microsoft Repository

Microsoft Repository is used to share software components and store information about them such as Web pages and design documents. Version 2.0 ships with Visual Basic 6.0. The Microsoft Visual Modeler (part of Visual Studio) allows developers to express a model in the form of Repository classes, interfaces, properties and relationships by using UML [URL14].

The Microsoft Repository stores the repository data in a relational database, and XML can be used to exchange data between the tools and the repository.

Investigating other commercial component repositories helps formulate the requirements for BeanBag. Features such as component classification need to be included in a component repository. It is also possible to store extra data with the component, such as its UML design documents and perhaps its source code if it is available. BeanBag must be designed to enable the storage of the components described in Section 2.4.

2.8 XML

XML is, essentially, a platform-independent way to structure information

[MR99]

As one of the main requirements of BeanBag is to describe components, XML should be investigated. This section describes what functionality XML could bring in terms of describing components.

At the moment, web content is tied to how it is displayed. What XML does, is to separate the content from the presentation. XML is not a language itself but rather a way of defining languages, which has many potential applications, such as EDI. Businesses may choose to interchange data, and XML can put structure on the data exchanged.

XML (Extensible Markup Language) is a formal specification for expressing the structure of data. It originates from the World Wide Web Consortium (W3C) and is used to separate structure and content from the presentation of data. By separating structure and presentation in this way, an XML document may have the same content, but can be presented differently depending on what device is being used to access the page, e.g. the document will look different on a mobile phone than on a computer screen.

Just as important is the fact that XML is inherently extensible. This means that developers can specify their own tag sets. If a tag is created to describe a set of recipes, and if tags describe the content then it is easier to search for the relevant data. It would be possible to retrieve the authors names and search in the bodies of recipes, because the tags describe the data. The XML for part of the recipe could look like this:

```
<author> Polly Jean Harvey </author>
<recipe_name> Dorset homemade strawberry jam </recipe_name>
```

XML and HTML are document formats derived from SGML (Standard Generalized Markup Language). HTML is an application of SGML, whereas XML is a subset of SGML. XML was designed with the Web in mind, while keeping the benefits of SGML and removing the complicated parts. HTML is suited best to fast data publishing on the Web. If the data requires more structure then XML may be used. For high-end highly structured publishing applications, SGML will continue to be used [URL15].

2.8.1 DTDs

Document Type Definitions (DTDs) are sets of syntax rules for XML tags. They define what tags are used in a document, what order they should appear in, which tags can appear nested inside others and what attributes tags have. A DTD is like a schema used in a database, but DTDs are less restrictive and allow for more variations of data. A DTD can specify that some fields are optional and that others can occur multiple times. HTML has its own universal DTD, however some people will need to define their own to suit their needs.

If an XML document is not well-formed it's toast [URL15]

An XML file must be well-formed and valid. There are certain XML syntax rules that need to be applied if a document is said to be a *well-formed* XML document. For instance, data must be ended with an end tag or an empty element tag. XML tags are case-sensitive. HTML contains error-handling code that deals with HTML with missing end tags and non-matching tags. However, this is not sufficient for XML, which demands a higher quality.

For an XML document to be well-formed it must follow three basic rules

1. The document starts with an XML declaration `<?xml version=”1.0”?>`
2. There is a root element in which all others are contained.
3. All elements must be properly nested. No overlapping is allowed.

[URL16]

If a document conforms to a specific DTD, then it is said to be a *valid* XML document. XML browsers need only concern themselves with whether an XML document is well-formed, in order to read it.

XML parsers examine XML code and report forming errors if the XML is not well formed. A DTD might specify that certain data should be present between two tags. The tags might be there and matching, and so it passes through the parser, but the data might not be there. DTDs are used to ensure the correct data is there also. A validating parser can be used to perform this task.

XML links have more functionality than HTML links. In HTML, links can be made to the middle of a page only if there is an anchor there already. The linking specification XLL (XML Linking Language) is being split into two separate specifications: XPointer and XLink. XPointer allows linking to any part of another page, even with no anchor there. XLink adds behavior to links, so that a new page might pop up, instead of the page arriving in the browser.

HTML is an adequate markup for humans to read, however for automatic data processing, XML is a more effective option. Using XML to describe components means that descriptions can be extended, and XML parsers can be used to check if the data is valid.

2.9 XML Schemas

The previous section described DTDs and ways to validate the content of an XML document. However, DTDs do not contain sufficient type information for some applications. Hence the interest in XML schemas.

Sun currently provides XML parsers for Java. Using these parsers to extract data from XML documents is not an easy task, as it requires an understanding of the API. Sun are looking at ways to enable the developer to access XML data in a Java-centric way by mapping XML schemas to in-memory Java objects. This work takes the form of the XML Data-Binding Facility.

Already we have schemas in XML in the form of DTDs, but these are a particularly weak type of schema with little support for complex structures. This has motivated the W3C Working Group to define a new schema language.

The XML binding facility is intended to be part of the Java2 platform. Including such a facility means an end to parsing XML manually, and developers could access XML content through Java classes. A schema compiler and a marshalling framework would be included in the facility. The schema compiler would translate an XML schema into a set of Java classes with matching accessors, i.e. get/set methods, thus hiding the parsing complexity from the application developer.

2.10 XML Query Languages

If component descriptions are represented in XML, a way of querying this data is required. This section describes research in the area of XML query languages.

2.10.1 XML-QL

In August 1998, a submission was made to the W3C for a Query Language for XML, called XML-QL. The submission takes a database view of XML as opposed to a document view, in that the XML document is the database and the DTD is the database schema.

XML-QL can express *queries*. Queries are used to extract pieces of data from XML documents. XML-QL also expresses transformations, which can map XML data between DTDs and can integrate XML data from difference sources [DFFLC98].

This is an example of XML-QL taken from [DFFLC98]:

Given this DTD, which describes the book, article, publisher and author elements,

```
<!ELEMENT book (author+, title, publisher)>
<!ATTLIST book year CDATA>
<!ELEMENT article (author+, title, year?, (shortversion|longversion))>
<!ATTLIST article type CDATA>
<!ELEMENT publisher (name, address)>
<!ELEMENT author (firstname?, lastname)>
```

a corresponding XML-QL query is formed:

```
WHERE <book>
      <publisher><name>Addison-Wesley</></>
      <title> $t</>
      <author> $a</>
      </> IN www.a.b.c/bib.xml
CONSTRUCT $a
```

The URL tells us where the XML document is, and it is assumed that it contains a bibliography, which conforms to the DTD above. This query matches every book element in the XML document that has at least one title element, one author element and one publisher element, whose name element is equal to *Addison-Wesley. t* and *a* are variable names unlike the string literal *Addison-Wesley*. The resulting list is a list of authors bound to *a*.

2.10.2 XQL

XML-QL has influenced the design of XQL while keeping in mind database standards such as SQL.
XQL has a select-from-where construct, which is similar to SQL.

```
Select $book.author
From bib:URL www.a.b.c/bib.xml, book:$bib.book
Where $book.publisher.name =”Addison-Wesley”
```

[IKK98]

In XQL element variables can be defined in a from-clause, such as `bib` and `book` in this example. The element variable `bib` is bound to an XML document. To make a reference to an element variable, it is prefixed with *$*.
The `groupby`-clause allows the resulting elements to be grouped together. The `orderby`-clauses, sorts the resulting elements.

2.10.3 XSL

XSL is a transformational language being defined by the XSL Working Group. XSL has similarities with the XML-QL approach proposed. The group believes that a convergence of XSL and XML-QL would result in a powerful and flexible query and transformation language for XML [SLR98].
XSL is already a W3C work in progress and can accommodate regular/irregular data or recursive data structures. There is no particular format for results, and results can be in

terms of XML text, primitives like strings or integers, or as structures representing parts of the documents.

The main difference between XSL and XML-QL is its syntax. XSL uses a URL like syntax for specifying queries, and XML delimiters. XML-QL uses keywords as delimiters and patterns for selection.

An example of XML-QL:

```
WHERE <book>
      <publisher><name>Addison-Wesley</name></publisher>
      <author>$a</author>
</book>
CONSTRUCT $a
```

The same query in XSL

```
<xsl:for-each select = “book[publisher/name =
‘AddisonWesley’]/author”>
      <xsl:value-of />
<xsl:for-each>
```

If results are required from more than one publisher, the query can be easily extended

```
<xsl:for-each select = “book[publisher[name = ‘AddisonWesley’]
                                 $or$ name=’Microsoft
Press’]]/author”>
      <xsl:value-of />
<xsl:for-each>
```

[SLR98]

XSL does not currently address some of the issues that are addressed in XML-QL, namely:

- Variables and Joins
- Object Identifiers

- Regular expressions in patterns
- Integrating Data from Multiple XML Sources

2.10.4 Lore

Lore is a Database Management System for XML that has been under development at Stanford University. Lore includes a functional prototype with a query language, indexing techniques, a cost-based query optimiser and support for logging and recovery. Lore also includes techniques for carrying out proximity searches.

The Lore project focuses on defining a declarative query language for XML, developing new technology for interactive searches over XML data and building an efficient XML query processor. More information can be found at [URL18].

2.10.5 W3C

In December 1998 the W3C held a workshop on query languages called QL'98. Companies discussed the problems and issues in creating a query language capable of handling XML. The aim of the workshop was to establish whether the W3C should start a new working group to define an XML-based query language. 92 participants from 31 different companies and 7 different research facilities discussed their business and commercial needs [URL17].

A query language is required that will take advantage of XML's data model while at the same time allowing the kinds of applications that SQL provides for databases. If there were a query language for XML it could improve current document searching techniques as queries would use the XML document structure to search relevant parts of the document.

The major issues that emerged relating to an XML query language were:

1. That a query language should take XML in and return XML
2. That a schema should not be required, but the language should have the ability to take advantage of one when present;

3. Companies are anxious to come to an agreement so they can move forward with shipping product.

[URL17]

It was proposed that the pattern matching utility in XSL, came closest in providing the basic functionality of an XML-query language. However, the XSL pattern language needs to be extended. XSL can only work on one document at a time.
XQL received a lot of attention at the conference because it extends the pattern language in XSL. As XQL includes indexing strategies, this makes it more suitable for a larger document set, and scalable.

2.11 Summary

This chapter described the technologies and methodologies required for designing a component repository. The motivation behind the current interest in component software was discussed, as were the benefits that components provide. Before designing the repository, the main component frameworks were investigated. This chapter included descriptions of the architectures of the main component frameworks, i.e. CORBA Components (CCM), Enterprise JavaBeans (EJB) and COM components. A comparison of CCM and EJB was included, as was a list of available commercial application servers.

A substantial requirement of BeanBag was to design an extensible component description. Therefore descriptions of component description languages such as IDL, OCL, JBCDL and KDL were given. Components can be described using modelling languages such as UML and DESML, and so these were described. XML was examined as a way of describing components in an extensible way and a note on XML schemas were included. Lastly, the XML query languages, XML-QL, XQL, XSL and the Lore project were investigated to discover their features and shortcomings.

3. Design

3.1 Introduction

Chapter 2 described research in the area of component repositories and what component technologies are available. This chapter uses that information to construct a set of requirements for each of the modules in BeanBag. Then, each module is designed with these requirements in mind. Lastly, a high-level view of the BeanBag process for insert and retrieval is outlined.

3.2 Requirements

The aim of this thesis was to design a framework for describing, storing and querying components that was extensible and that could be applied to all types of components. Each module of the BeanBag system has its own requirements. These are outlined in the following sections.

3.2.1 Storing Components

There is no requirement to store the component itself, it is sufficient to store the URL of the component. The main pre-requisite for storing components, is that the system must use the storage mechanism that is available, i.e. an Oracle relational database. The following functionality must be provided:

- If a description contains a new property, a dynamic way to store it must be supported
- Given the component name, BeanBag should return a reference to it
- Given the component name, BeanBag should return its remote interface description and its developer defined description

- Component descriptions will vary depending on the type of component to store. For instance Enterprise Java Beans has an XML Deployment Descriptor, and a remote interface, that can be retrieved using Reflection. The description of a COM object will be in a different format. How BeanBag extracts the description will depend on the type of component being inserted.

3.2.2 Describing Components

The primary requirement for the component description is that the component description must be component independent and easy to query. The component developer will be required to describe the semantics of a component before its deployment. This will be in the form of an extra file that the developer will populate, as in some cases it may not be possible to extend the description of the component that already exists. For instance, in the case of EJBs, the Deployment Descriptor cannot be extended to include user-defined descriptions.

Before methods used to describe semantic information about components are examined, the information that is already stored in a component must be discussed. In Enterprise JavaBeans, information about what is contained in the JAR at deployment time is contained in the Deployment Descriptor.

3.2.2.1 The Enterprise JavaBean Deployment Descriptor (DD)

EJBs are deployed in JAR archives. An EJB-JAR file can contain one or more Enterprise Java Beans as EJB Java class files that include the remote and home interfaces. The EJB-JAR contains an XML Deployment Descriptor.

In [EJB1.1] the Deployment Descriptor has been divided into two sections:

- Structural Information (mandatory)

  Enterprise Bean Name, Enterprise Bean Class, Enterprise Bean Type etc.

- Assembly Information (optional)

Typically only EJB-JAR files with assembled applications contain assembly information, therefore this is optional. The assembly information describes how the bean in the EJB-JAR is composed into a larger application.

The format of the Deployment Descriptor (DD) has changed from [EJB1.0] to [EJB1.1]. The new DD format is based on XML. The aim was to achieve a vendor-independent DD. The DD no longer contains information, which is specific to each EJB Server.

The role of the Deployment Descriptor is to provide information about the Enterprise Java Bean that is not contained in the code. The DD must be well formed XML and conform to the DTD in [EJB1.1].

The Deployment Descriptor must contain the following information about the Enterprise Java Bean:

- **`ejb-name`:** Enterprise bean's name - there is no relationship between this name and the JNDI name that the Deployer will assign the bean
- **`ejb-class`:** Enterprise bean's class - the fully qualified name of the class that implements the business rules of the bean
- **`home`:** Enterprise bean's home interface
- **`remote`:** Enterprise bean's remote interface
- **`session/entity`:** The bean's structural information
    - **`session-type`:** Stateful/Stateless (session)
    - **`transaction-type`:** Is transaction demarcation performed by the Container or the Enterprise Bean (session)
    - **`persistence-type`:** Is persistence management performed by the Container or the Enterprise Bean (entity)
    - **`primkey-class`:** A primary key class must be provided for an entity with bean-managed persistence (entity)
    - **`cmp-fields`:** The Container Managed fields (entity)

- The Environment Entries
- The Beans resource factory references
- The EJB references to homes of other EJBs
- Security Roles

A typical .ejbml file looks like the following:

```
<enterprise-beans>
      <session>
      ...
            <ejb-name>EmployeeService</ejb-name>
            <ejb-class>com.wombat.empl.EmployeeServiceBean
            </ejb-class>
            ...
            <ejb-ref>
                  <description>
                  This is a reference to the entity bean that
                  encapsulates access to employee records.
                  </description>
                  <ejb-ref-name>ejb/EmplRecord</ejb-ref-name>
                  <ejb-ref-type>Entity</ejb-ref-type>
                  <home>com.wombat.empl.EmployeeRecordHome</home>
                  <remote>com.wombat.empl.EmployeeRecord</remote>
            </ejb-ref>
            <ejb-ref>
                  <ejb-ref-name>ejb/Payroll</ejb-ref-name>
                  <ejb-ref-type>Entity</ejb-ref-type>
                  <home>com.aardvark.payroll.PayrollHome</home>
                  <remote>com.aardvark.payroll.Payroll</remote>
            </ejb-ref>
            <ejb-ref>
                  <ejb-ref-name>ejb/PensionPlan</ejb-ref-name>
                  <ejb-ref-type>Session</ejb-ref-type>
                  <home>com.wombat.empl.PensionPlanHome</home>
                  <remote>com.wombat.empl.PensionPlan</remote>
            </ejb-ref>
            ...
      </session>
...
</enterprise-beans>
```

Figure 3-1 An Example of EJB References in a Deployment Descriptor

3.2.3 Querying Components

Section 2.10 described XML Query Languages. If an XML component description and other component data are stored in a relational database, the minimum requirements for the query module are:

- To return a component from the data store that matches the interface provided by the user. The match may be exact or partial.
- The system should attempt to rank the returned components in terms of cost, version or preferred vendor. User-profiles could be used to tailor the search to the user
- The queries must be efficiently executed

3.3 High Level Architecture

From examining the requirements, it is obvious that the system will be a three-tier system, using a database, interfaced by a server with business logic and accessed by may clients.

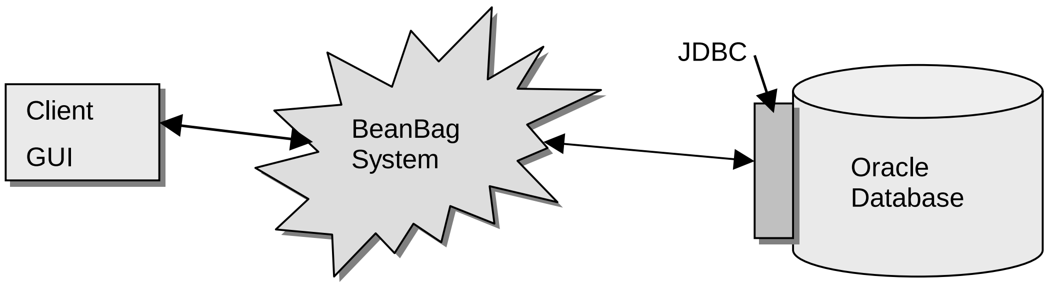


Figure 3-2 The BeanBag High Level Architecture

3.3.1 GUI

The user will interface with BeanBag via a GUI. It is anticipated that queries will be input by selecting combo boxes and Boolean statements or by entering SQL statements. GUI queries will have to be converted into a form that can be used to query the database.

Examples of queries that a user may request are:

- Retrieve a component that can add two numbers
- Retrieve a component like this component, but cheaper
- Retrieve a component that is a more recent version than this
- Retrieve two components that do the same work as this one
- Retrieve a component written by this author
- Retrieve all components that are from this company

3.3.2 The BeanBag System

The main function of the BeanBag system is to insert and retrieve components. Other functions that it must perform are:

- Match the user query with components that are already in the database
- Transparently insert new properties for components as they occur
- Keep track of statistics about the components. A user may like to know how many people have downloaded one component and what their reviews were.
- Retrieve/Insert the following data about a component:
    - The component URI, name and statistics
    - The component’s remote interface

- The component's semantic developer defined description
- The .ejbml file (in the case of EJBs)

The BeanBag system will be divided into a front and back end. The front end will be responsible for translating user queries into database queries, manipulating the component to extract information from it and parsing user input. The back end will manage the connection to the database and will create and delete database structures and execute JDBC calls.

3.3.3 Storage Framework

When designing the tables that are to contain the data the following functionality is required

1. An efficient means of retrieval
2. An extensible framework allowing new properties to be added as the user defines them
3. The framework should be component independent

3.3.3.1 Efficient Retrieval

Searching through large data structures in a database table is not efficient. If the user requests that BeanBag find a component matching a component interface, a direct comparison search would take a while. Also, two interfaces can be the same yet one may have the base classes in a different order, may include line feeds and comments or have the methods in a different order.

One way to implement this type of searching is to extract details from an interface at insert time that may be searched in the future. Elements that may be queried are method names, base classes extended, interfaces implemented and the number of parameters. When these details are extracted they can by inserted into an Indexed Table.

The Indexed Table for methods maps method names to beans that implement those methods. Then when a user requests all components that implement certain methods, BeanBag retrieves this data quickly and can AND or OR the results in much the same way as some Web search engines do.

This is an example of how the Indexed Table for methods looks, presuming that component names are mapped to a unique key, which is stored in another table.

methodName	beanNames
findProductBySKU	100063,100345,459990
findProductsByDescription	100063,987665

Figure 3-3 methods2beans table

3.3.3.2 XML Extensible Properties Table

Section 3.4 describes the design of the user defined component description. It is sufficient to know that this file is an XML file. From this, it can be deduced that the data to store is contained within XML elements and attributes. Because the XML elements and attributes describe the data contained, they also need to be recorded with the data.

A separate table for each element would be difficult to manage. When a new element type is inserted by the developer, a new table would be created, which might never be populated by another component. If elements and attributes were mapped to columns of a table, this would be inefficient and cumbersome to manage. Many of the columns would remain empty and new columns would be added regularly.

The design chosen was to have one Properties table. This would have a column for the bean unique identifier, element name, attribute name, value and the element index. These columns form a composite key. Using this format, new properties can be added easily and efficiently.

Pkey	ElementName	AttrbuteName	Value	Index
100063	name	id	Caroline	1
100063	name	email	oreillcr@cs.tcd.ie	1
104566	name	id	Jack	2
104566	Name	email	Jack@cs.tcd.ie	2

Figure 3-4 Extensible Properties Table

3.3.3.3 Bean Interfaces Table

It is likely that the most common request to the BeanBag system will be to retrieve the component description of a component in the database. For this reason the user defined descriptions and those extracted from the component will be stored in their entirety and indexed by the beans unique primary key. An example with regard to EJBs is to store the .ejbml file.

pkey	.ejbml description
100063	`<enterprise-beans>` `<session>` `<ejb-name>EmployeeService</ejb-name>` `<ejb-class>com.wombat.empl.EmployeeServiceBean` `</ejb-class> ...`
104566	`<enterprise-beans>` `<entity>` `<ejb-name>EmployeeService</ejb-name>` `<ejb-class>com.radio.empl.RadioerviceBean` `</ejb-class> ...`

Figure 3-5 Descriptions Table

3.4 The Component Description

To describe the semantics of a component that are not contained in the .ejbml file, a new file is needed, that will be populated by the component developer. This file, `xml/component.xml` will be included in the EJB-JAR at deployment time.

A component description may include the following data:

- Component Type { DCOM | EJB | CORBA | JavaBean }
- Author {name, email }
- Company
- Version
- Date Written
- Genre/Kind {Game, Mathematical, E-Commerce, CORBA, GUI }
- Comments
- Method Comments
    - Pre-condition
    - Post-condition
    - Parameters {unique, max number…}
- Outbound interface/Needs interface {what other components does this component need to perform}
- Transport {IIOP, RMI}

As the component description must be component independent and extensible, XML is a logical choice for representing the description. Using XML allows new elements to be added to the DTD and then the XML can be validated. As described in Section 2.10, an XML query language could be used to query the XML descriptions. There were several steps in designing the XML and the DTD. These are described in the next sections.

3.4.1 Abstracting the Information Content

Firstly, the information that was to be included in the component description was modelled as a hierarchy of elements.

```
Component
      Type
      Release
            Version
            DateOfRelease
            ExpiryDate
      Author
            Name
            Email
            WebSite
            Phone
      Institution
            Name
            Address
            WebSite
            Phone
      Genre
      TestEnvironment
            Package
            Version
      Needs
            ComponentName
      Method
            MethodName
            ReturnType
            Parameters
            Invariants
            PreCondition
            PostCondition
      Transport
      Comment
            Author
            Date
            Comment
```

3.4.2 Designing the DTD

The next step was to design a DTD that could represent the information content. The DTD is used in the validation stage to ensure that the description file contains correct data. The DTD ensures that the minimum data is present in the XML description. For more information on XML and DTDs see [NH99]. This is the DTD that was designed to support the component description:

```
<?xml version="1.0" encoding="UTF-8"?>
// This is the DTD for describing components. The Occurence Indicators
// are [?: 0/1, *: 0/more, +: 1/more]
// data & comment are used multiple times in the DTD and so they are
// defined as entities.

// there could be more than 1 component in an EJB-JAR archive
<!ELEMENT components (component+)>

<!ELEMENT component (type, date+, release, author+, institution*,
genre+, keyword*, test.environment*, needs*, method*, transport,
comment*)>

<!ENTITY % common.attrib
        "id     ID      #IMPLIED
        ">

<!ENTITY % date.attrib
        "day    CDATA   #REQUIRED
        month(jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)
#REQUIRED
        year    CDATA   #REQUIRED
        ">

<!ELEMENT date EMPTY>
<!ATTLIST date
        %common.attrib;
        %date.attrib; >

// there will be only 1 component type and it is required
<!ELEMENT type EMPTY>
<!ATTLIST type
```

```
        isa (EJB | CORBA | DCOM | JAVABEAN) "EJB">

// there will be only one release date and it is required
<!ELEMENT release EMPTY>
<!ATTLIST release version CDATA #REQUIRED
                  release-date IDREF #REQUIRED
                  expiry-date IDREF #IMPLIED >

// there could be more than 1 author
<!ELEMENT author EMPTY>
<!ATTLIST author %common.attrib;
                 email CDATA #REQUIRED
                 website CDATA #IMPLIED
                 phone CDATA #IMPLIED >

// A number of companies or universities could be involved
<!ELEMENT institution (company*,university*)>
<!ELEMENT company EMPTY>
<!ATTLIST company name CDATA #REQUIRED
                        address CDATA #REQUIRED
                        website CDATA #IMPLIED
                        phone CDATA #IMPLIED >
<!ELEMENT university EMPTY>
<!ATTLIST university name CDATA #REQUIRED
                        address CDATA #REQUIRED
                        website CDATA #IMPLIED
                        phone CDATA #IMPLIED >

// a component could belong to many genres. They have to be pre-
// defined for consistency
<!ELEMENT genre EMPTY>
<!ATTLIST genre belongs (e-commerce | games | journalism | maths |
cad) #REQUIRED>

// for efficient searching, components can be assigned keywords
<!ELEMENT keyword EMPTY>
<!ATTLIST keyword %common.attrib;>

// what envionments was the component tested in
<!ELEMENT test.environment (package?, version?)>
```

```
// what other components does this component need to function
<!ELEMENT needs ANY>
<!ATTLIST needs
        xlink:form      CDATA   #FIXED  "simple">

// what are the component methods in terms of pre- and post-
// conditions
<!ELEMENT   method   (parameter*,   pre-condition*,   post-condition*,
invariant*, comment*)>
<!ATTLIST method id ID #REQUIRED return.type CDATA #REQUIRED>
<!ELEMENT parameter EMPTY>
<!ATTLIST parameter name CDATA #REQUIRED type CDATA #REQUIRED>

// the transport used
<!ELEMENT transport EMPTY>
<!-- need to know how to say iiop or rmi or your own... -->
<!ATTLIST transport type CDATA #REQUIRED>

// comments
<!ELEMENT comment (#PCDATA)>
<!ATTLIST comment
        written.by IDREF #REQUIRED
        date IDREF #REQUIRED>

<!ELEMENT package (#PCDATA)>
<!ELEMENT version (#PCDATA)>
<!ELEMENT pre-condition (#PCDATA)>
<!ELEMENT post-condition (#PCDATA)>
<!ELEMENT invariant (#PCDATA)>
```

3.5 Component Insert and Retrieval

Given the location of a component, the first operation that BeanBag executes should be the module to check the type of the component. The component could be a DCOM component, a CORBA Component or an EJB. The insert module must discover the component type and follow that insert route.

Figure 3.6 outlines the high-level insert operation.

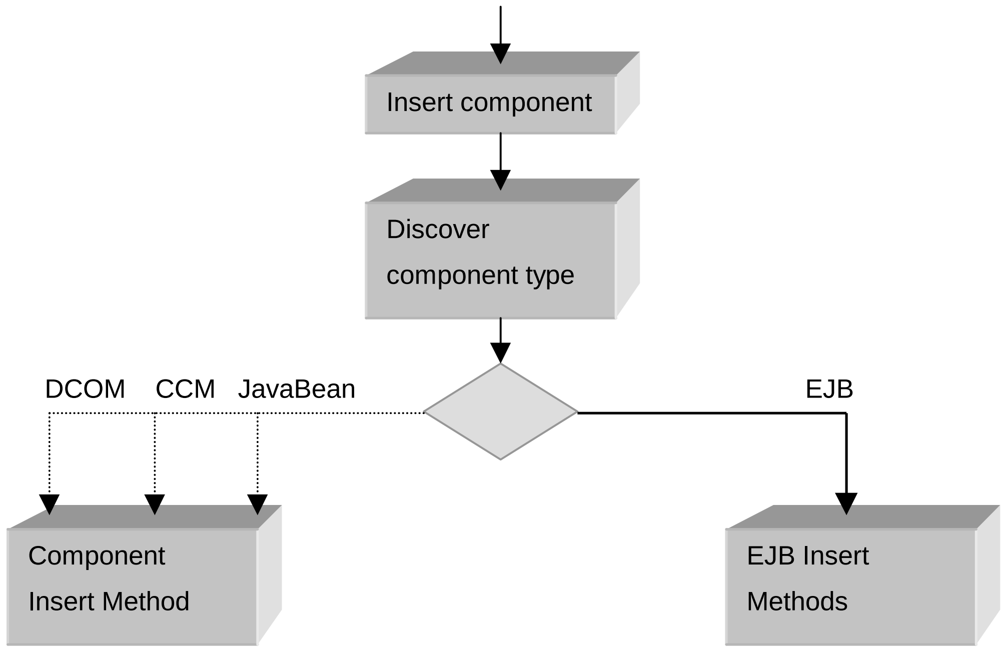


Figure 3-6 High Level Model of the Component Insert

In this thesis, only one route is examined to evaluate the feasibility of the BeanBag system. This is the EJB route.

Having designed the database tables and the developer XML description, the flow of data through the system can be outlined, from when the user inserts the location of a component to its insertion into the database. The following sections outline the main processes in BeanBag. Appendix C includes the sequence diagrams for these processes.

3.5.1 Inserting a component

The insert process takes the location of a component from the user, processes the component JAR file, and stores the semantic and syntactic data in the database. These are the modules that are used in the insert process.

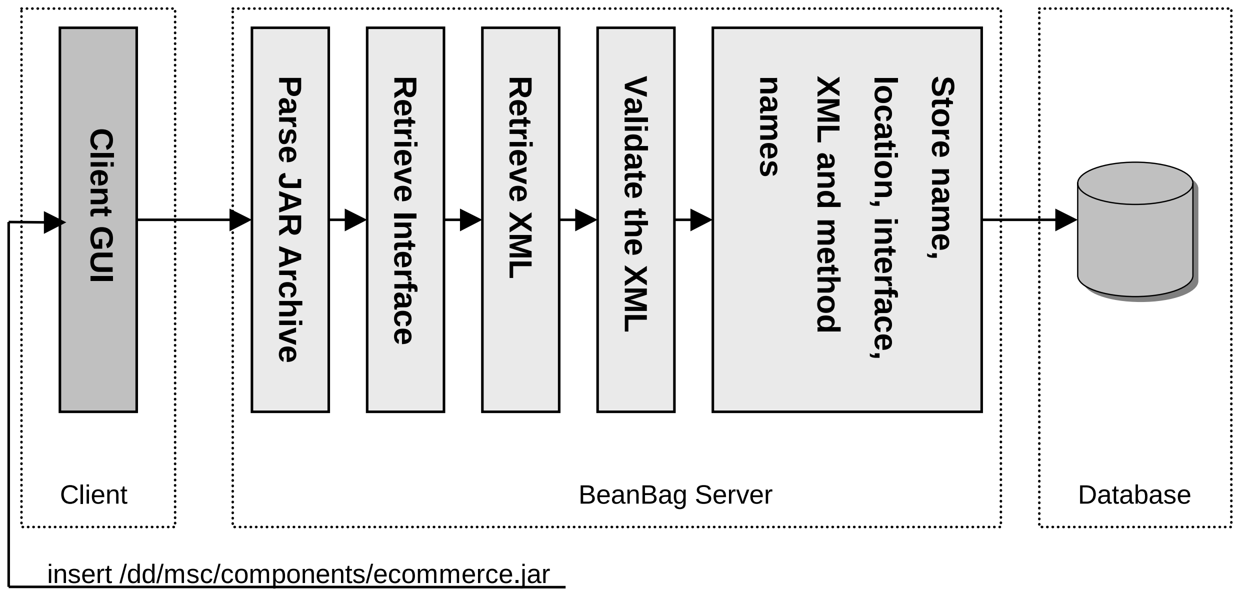


Figure 3-7 High Level View of the Insert Process

3.5.2 Retrieving Components

To retrieve a component the user selects criteria that a retrieved component must satisfy. The user input must be transformed into a database query. When the query has executed the user should be notified. What follows is a high level view of the modules that are used during the retrieval process.

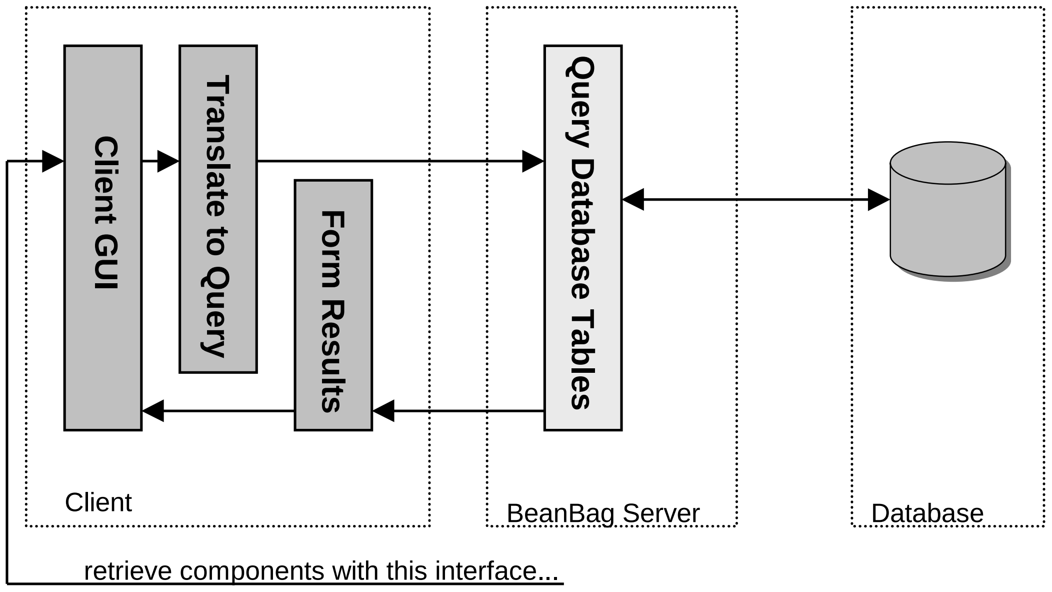


Figure 3-8 High Level View of the Retrieval Process

3.6 Summary

In this chapter the design of the BeanBag system was described. The initial sections outlined the requirements for storing, describing and querying components. The EJB Deployment Descriptor was examined, to determine what component information it contained before the component description was designed.

The next sections described the design of the system modules. The design for the storage framework discussed methods for achieving efficient retrieval and for accommodating extensible component descriptions, by describing the database tables. Next, the methodology behind designing the XML component description and DTD was outlined in detail. Lastly, a high level description of the operation of the system at component insert and retrieval was presented.

4. Implementation

4.1 Introduction

Chapter 3 discussed the design of BeanBag. In this chapter, the implementation of the system is described and any issues that were encountered during the implementation.

The implementation of BeanBag followed a path, from implementing components, extracting data from the JAR archive, to examining ways of querying the database. Each step describes what technologies were considered and which were chosen.

4.2 The Component Framework

The Enterprise JavaBeans Framework was chosen as the first component framework to be supported by the BeanBag system, as the EJB Specification had been released from Sun in March, and there were many implementations of the specification. At the time of implementing the system, there were no CORBA Component implementations.

4.2.1 HomeBase

HomeBase is the current Application Server from IONA. It was chosen because it was free and there were demos and documentation available on the IONA website.

4.3 The JAR archive

Section 2.4.2.9 described how EJB components are deployed as JAR archives. Given the location of the component, i.e. The JAR archive, BeanBag had to extract the .ejbml file and the developer described `xml/component.xml` file from the JAR archive. Every EJB JAR should contain at least one .ejbml file according to [EJB1.1]. If the

JAR archive does not include one, BeanBag throws an exception. A more detailed description of JAR archives can be found at [JAR].

The Java classes in `java.util.ZipFile` provide methods for finding and reading files in a JAR. The JAR corresponds to a `java.util.ZipFile` and the files it contains correspond to `java.util.ZipEntry` files.

To check if a particular file is in a JAR archive, the following code is used:

```
ZipFile jarFile = new ZipFile(jarfile);

  // check to see if a file ‘fileName’ exists in the archive
  Enumeration enums;
  for (enums=jarFile.entries(); enums.hasMoreElements();) {

      // get the next entry in the archive
      ZipEntry jarEntry = (ZipEntry)enums.nextElement();
      String filename = jarEntry.getName();

      if (filename.equals(fileName))
                 return true;
  }
```

A Java bug exists relating to reading files from JAR archives, which results in an ’Unexpected end of ZLIB input stream error’ occurring. The behaviour is described in more detail in [URL9]. This bug is fixed in jdk1.2 so the BeanBag application should be ported to this version in the future.

The workaround involves never attempting to read more bytes than the `jarEntry` contains, as shown in this code example:

```
...

  // workaround for Java Bug 4040920
  byte buffer[] = new byte[1024];
  int remaining, length;

  String line = new String("");
  try {
    DataInputStream dis = new DataInputStream
                             (archive.getInputStream(file));
    String nextline;

    // get the size of this file
    remaining = (int)file.getSize();

    // read in the file in 1024 byte blocks
    while (remaining>0 &&
           (length=dis.read(buffer, 0, Math.min(1024, remaining)))
                                                               != -1){

                line = line + new String(buffer);
                remaining -= length;

          }
       dis.close();
       }
  …
```

4.4 Extracting the Remote Interface from the JAR archive

Once the .ejbml file has been read from the JAR archive, the file is parsed to extract details about the components it contains. This file provides information such as the bean names, whether they are session or entity beans, and more importantly, the home interface name and the remote interface name. The entire file is stored in the database.

This is an example of an .ejbml file describing the cart EJB i.e. cart.ejbml

```
<ejbml>
     <session-bean
       name="cart"
       package="order"
       descriptor="order/CartDeployment"
       home="order.CartHome"
       remote="order.Cart"
       bean="order.CartBean"
       type="stateful"
       timeout="600"
       tx-attribute="TX_SUPPORTS"
     >
     </session-bean>
</ejbml>
```

Once BeanBag has parsed the remote interface class name from the .ejbml file, the Java Reflection API is used to extract the remote interface from that class. The Reflection API returns the signature of the remote interface class, except the names of the parameters to the methods. For more information about the Java Reflection API see [FLAN97_2].

The remote interface returned for the cart EJB is:

```
   public abstract interface order.Cart extends javax.ejb.EJBObject {
   // Constructors
```

```
// Fields
// Methods
public abstract void add(order.Product, int) throws
   java.rmi.RemoteException;

public abstract void remove(int) throws java.rmi.RemoteException;

public abstract javax.swing.table.TableModel getTable() throws
   java.rmi.RemoteException;

public abstract void purchase() throws javax.ejb.CreateException,
   java.rmi.RemoteException;
}
```

When the remote interface has been discovered, it is stored in the database. For efficient searching the remote interface is parsed to extract the method names, base classes and implemented interfaces. This information is stored in indexed tables, as described in Section 3.3.3.1.

4.5 Processing the XML

Section 4.3 described how to extract files from the EJB Jar archive. When BeanBag reads the `xml/component.xml` file, the file has to be validated, and then the elements and attributes must be parsed so that they can be stored in the database. An XML Parser is used to parse XML files and is also used for XML validation. The `xml/component.xml` file and its corresponding DTD were described in Section 3.4.

4.5.1 Choice of XML Parser

There are many XML Parsers available, but Suns ProjectX TR2 Parser was chosen as the error messages it generates during validation give details about line numbers in the XML file where the errors occur.

There are two methods of XML parsing, Event Driven (SAX) and Tree Based (DOM). Both are described in detail in [NH99]. The SAX Model was used to parse the XML file, as opposed to the DOM model because the SAX Parser is more memory efficient. Using the DOM parser involves constructing a tree corresponding to the XML file in memory, which can consume time and memory. Using DOM brings extra functionality such as look ahead, which means that an application can tell at what point in the element list it is. However, the SAX Parser was sufficient for BeanBag.

When the XML file is validated and the elements and attributes have been parsed by BeanBag they are stored in the Properties Table described in Section 3.3.3.2.

4.6 Choice of Database

The Oracle 8 Enterprise Edition Release 8.0.3.0.0 Relational database was used to store component data, as it was the only database available during the project. Had the Oracle 8i database been available, with XML support, the task of storing XML descriptions would have been made easier, as the XML would have been parsed automatically. The XML SQL Utility is described in more detail in Section 5.4.5.

4.6.1 Creating Sequences in Oracle

For effective searching, all the major tables had a numerical primary key. Each bean name corresponded to a unique number, which in turn was used to index into the tables. There are a couple of ways to go about creating unique sequence numbers in Oracle. One way would have been to get the MAX number in the column so far and add one. However, this solution does not scale when you have more than one client. It would have been possible for two clients to read the MAX of the column at the same time, add one and for both to write back the same number. Of course, one of the clients would get an integrity violation and would fail.

A better solution was to use Oracle Sequences. JDBC has no knowledge of sequences; they are created through SQL.

An Oracle sequence is created like this

```
// create a sequence of name my_seq
        CREATE SEQUENCE my_seq
// specify the interval between sequence numbers and give a start value
        INCREMENT BY 1
        START WITH 1
// specify the minimum and maximum value a sequence number can be
```

```
       MINVALUE 1
       MAXVALUE 9999999999999999999999
// the sequence cannot generate more numbers if it reaches the max value
       NOCYCLE
// guarantee that sequence numbers are generated in the order that the requests come. This is
// not important for primary key generation, but might be for timestamping
       ORDER
// specify how many values Oracle pre-allocates and keeps in memory for faster access
       CACHE 300
```

To use the sequence when inserting into a table, `nextval` and `currval` are used:

```
// insert a new primary key and bean name into the NAMES table
PreparedStatement pstmt = conn.prepareStatement
             ("INSERT INTO names VALUES(my_seq.nextval,?)");

// insert the bean name, as parameter number 1
pstmt.setString(1, componentName);
pstmt.executeUpdate();
pstmt.close();
```

Using sequences in Oracle means that one user can never acquire the same sequence number generated by another user. Two users can concurrently increment the same sequence but they will not get the same value back. Therefore when a user increments a sequence, the number numbers returned may be 1000, 1001, 1002, 1004, because another user is incrementing at the same time, and has received 1003. Sequence numbers are generated independently of tables and so the same sequence can be used for one or more tables.

4.6.2 Use of the Oracle `LONG RAW` datatype

The table structure for storing component descriptions needed columns large enough to store .ejbml files and developer XML descriptions. Each component's XML description and .ejbml file had to be stored in its entirety.

The Oracle `VARCHAR` and `VARCHAR2` datatypes specify variable length character strings. It is recommended that you use `VARCHAR2` as opposed to `VARCHAR` as in the future `VARCHAR` in Oracle may take on different semantics [OSQL]. A size for `VARCHAR` and `VARCHAR2` must be specified, the minimum begin 1, and the maximum being 4000.

4000 characters was not large enough for storing .ejbml files or XML files, so the `LONG` datatype was chosen. The `LONG` datatype allows character data of variable length to be stored up to 2 gigabytes.

The `LONG RAW` datatype was used to store .ejbml files and XML files as binary data. When storing binary data in Oracle the `RAW` datatype should be used. `LONG` and `LONG VARCHAR` get treated as character data and may get converted as data moves from one platform to another, whereas `LONG RAW` data is binary and never altered [URL19].

The use of the `LONG` datatype allows flexibility in size, but it brings restrictions that effected the design, such as:

- A table cannot contain more than one `LONG` column.
- `LONG` columns cannot be indexed.
- A stored function cannot return a `LONG` value.
- `LONG` columns cannot appear in certain parts of SQL statements:

  `WHERE, GROUP BY, ORDER BY`, or `CONNECT BY` clauses or with the `DISTINCT` operator in `SELECT` statements

For a complete listing see [OSQL].

4.6.2.1 Inserting the `LONG RAW` datatype

Storing the descriptions as binary data when using the Oracle Thin Drivers involved creating a prepared statement and then using `setBinaryStream()`, to insert the data as binary data.

```
...

  // insert the XML file into the table 'XML'
  PreparedStatement pstmt = conn.prepareStatement
                ("INSERT INTO xml VALUES(?,?)");

  // insert the componentName, as its numerical value
  pstmt.setInt(1, pkey);

  // insert the XML file, as retrieved from the ReadJar object
  InputStream fin = new StringBufferInputStream(xmldesc);
  len = xmldesc.length();
  pstmt.setBinaryStream(2, fin, len);

  pstmt.executeUpdate();
  pstmt.close();

...
```

4.6.3 Working around the Oracle Open Cursors Exceptions

During the development of BeanBag an Oracle exception occurred that the maximum number of open cursors was exceeded. Oracle opens cursors internally and so an application can not be sure how many are used. If the number of open cursors is not set, the default number is 50. [URL20] describes how to change this default value in the Oracle database.

In order to avoid open cursors errors, JDBC Statements and ResultSets need to be closed after each method is finished using them.

4.6.4 Achieving integrity in the database tables

Each client had a connection to a CORBA object on the Server. Each CORBA object had a connection to the Oracle database. This is described in more detail in Section 4.9. Therefore it could have been possible for two clients to do an update of a row of a table at the same time, and cause the data inserted to be incorrect. This problem would only occur with Indexed Tables when rows are updated to include new bean identifiers.

In Oracle there is no *lock row* command, by default Oracle does row level locking. Whenever you do an update or delete, Oracle locks the row. The syntax for doing an update on a row is

```
SELECT methodName, beans FROM methods2beans WHERE
```

```
          methodname='demoSelect' FOR UPDATE OF beans;
```

When this command is used. auto-commit must be turned off, which is set to true by default.

```
...

stmt = conn.createStatement();


// 1. turn off the auto-commit
  conn.setAutoCommit(false);

      // 2. select rows for update
      if (!newElement) {

          cmd = "SELECT " + cols[0] + ", " + cols[1] +
                  " FROM " + table + " WHERE " + cols[0] + "='"
                  + element + "' FOR UPDATE OF " + cols[1];

          rset = stmt.executeQuery(cmd);

          // 3. for each row, update it
          if (rset.next()) {

            cmd = "UPDATE " + table + " SET " + cols[1] +"='" +
            newValue +
                "' WHERE " + cols[0] + "='" + element +"'";

          stmt.executeUpdate(cmd);
          rset.close();

          }
      }
...

  // 4. tidy up and turn on auto-commit
  conn.commit();
  conn.setAutoCommit(true);
  stmt.close();
```

The `FOR UPDATE` clause locks the rows selected by the query. Once a row is selected for update, other users cannot lock or update it until the end of the transaction. The `FOR UPDATE` clause signals an intent to insert, update, or delete rows returned by the query, but does not require that one of these operations is performed. A `SELECT` statement with a `FOR UPDATE` clause is often followed by one or more `UPDATE` statements with `WHERE` clauses [OSQL].

4.7 Choice of Query Language

The next step in the implementation of BeanBag was to examine ways of querying the data that had been stored. Many XML query languages exist, as outlined in Section 2.10. However, XQL is designed to work against an object-oriented database (e.g. Versant or ObjectStore) but for the purposes of this project, only a relational database was available. Therefore SQL was used as a query language, even though XQL is specifically designed for this purpose.

4.7.1 Searching for an exact Interface match

As mentioned in Section 4.6.2, a `SELECT` statement cannot be used on a `LONG` column. This meant that the `SELECT` statement could not be used to find a component that implemented a particular interface. So the approach taken was to parse the interface of a component, extract details of its method names, base classes, interfaces implemented and to store this information in indexed tables. The time taken to insert a component increased because the component interface had to be parsed. But the component matching time improved.

In order for an interface to match another exactly the following members must match:

- The interface name and its signature
- The method signatures
  - The number of methods
  - The types of their parameters and number of parameters
  - The names of the methods
  - The return types of the methods
- The base classes
- The implemented interfaces
- The member variables, public, protected and private
- The constructors

BeanBag parses out method names and base classes from the user input, to find beans that match these requirements in the Indexed Tables. The only information that is not available in the database about the interface are the names of the parameters, as these are not returned by the Java Reflection API.

4.8 JDBC Drivers

As BeanBag used an Oracle database the Oracle Thin JDBC Driver was used, to interface with the database. There are two types of Oracle JDBC drivers that could have been used, each providing slightly different functionality. The Oracle Thin Driver is a 100% Java Implementation. It is quite small (300K, or 150K compressed). [SK] describes how the OCI driver is faster than the thin driver, because the OCI based driver does much of its data marshalling in C. OCI is a C library used to access Oracle databases and the Oracle OCI drivers are a JDBC layer written on top of this library. Using the OCI driver means that your application will not be 100% compliant. For a more detailed description of JDBC see [Flan97].

4.9 The CORBA Architecture

The next step after the database issues were resolved was to implement the CORBA architecture. OrbixWeb3.1 was used, as it was available and the ORB was sufficient for what was required. No advanced features such as object by value were needed.

4.9.1 Introduction to the CORBA Objects

The CORBA Architecture comprised of a three-tier model consisting of a client, server and a database, as shown below. The IDL interfaces are listed in Appendix B.

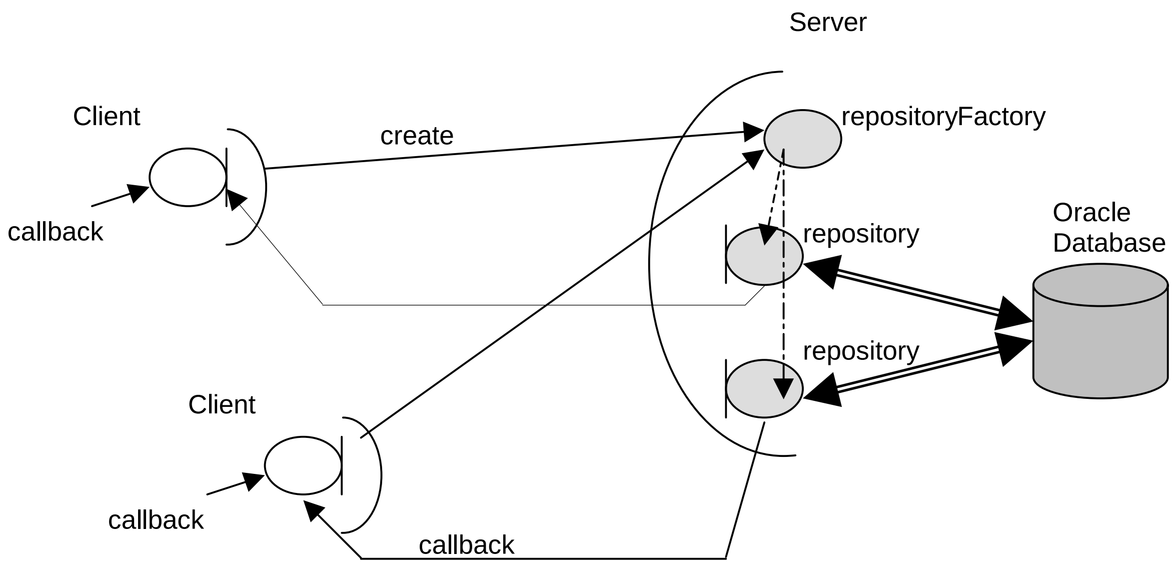


Figure 4-1 The CORBA Architecture

When a client enters the system, it connects to the `repositoryFactory` CORBA Object on the server, and calls its create method. The `repositoryFactory` object creates a `repository` object for the client and returns its reference. Each client maps to its own repository object on the server, which in turn has a connection to the database.

4.9.2 CORBA Callbacks

Each client contains a callback object. The client was implemented in this way, because when it asked the server to execute some SQL, it had to wait for a reply. Some SQL commands may take a long time to execute or perhaps may never return due to an Oracle error. By using callbacks, clients do not wait for replies. When the server gets the reply from the database, it packages the data and sends it in a callback to the client, which automatically updates the GUI JTable.

When implementing callbacks in OrbixWeb3.1, sometimes callbacks did not arrive on the client, specifically when they were implemented as oneways. The reason for this was that the oneway request was the first request on the proxy object. If this is the case, the proxy object has the daemons port number embedded in the object reference, and so it contacts the daemon, instead of the server. If it is a oneway, the daemon does not send a reply to the client, with a LOCATION_FORWARD reply and the oneway gets lost. The workaround is to set IT_IIOP_USE_LOCATOR to false in OrbixWeb.properties, so that the object reference contains the transient port number of the server, and not the well-known port number [URL12].

4.10 The GUI

With the server side completed the client was the next module to be implemented. The client was implemented as a GUI using Java Swing. More information on Java Swing can be found at [URL10].

If an error occurred at the database, it would not be visible at the client, unless the server returned the error. Therefore, each time the server catches a database exception, it is returned to the client, where it is caught and displayed in the Log Center.

Screen shots of the BeanBag application are in Appendix A.

4.11 Summary

This chapter discussed the implementation of BeanBag and the decisions made on the technologies used. Decisions were made on the component framework used, the XML Parser, the query language, JDBC driver and the distributed architecture. Each of these decisions was discussed in this chapter. Implementation details such as extracting files from JARs, implementing Oracle sequences and using the Oracle LONG RAW datatype were included. These details were discussed either because they refer to bugs in third-party products or because the documentation on how to implement certain features was not readily available.

5. Evaluation and Conclusion

5.1 Introduction

This chapter evaluates the current BeanBag system with regards to the requirements set out in Chapter 3. There were three components of the BeanBag system, namely the storage framework, the component description and the query system, each with individual requirements. In this chapter, each module is evaluated. Following the evaluation are the conclusions that can be drawn from the first BeanBag prototype, and how the system could evolve in future versions.

5.2 Evaluation of Modules

To evaluate the BeanBag system, each module of the application is evaluated against the requirements set out for it in Section 3.2.1.

5.2.1 The Storage Framework

The functionality required from the BeanBag storage module was that the end user should give the location of a component to the system and BeanBag should automatically extract information from it and store this data in a relational database. These requirements were achieved.

The design of the Properties table set out in Section 3.3.3.2 worked efficiently. When previously unseen properties are discovered in the `xml/component.xml` file, BeanBag parses them out like the others and inserts them in the Properties table. In this way, BeanBag uses one large table for storing user properties, but without wasting space.

5.2.2 The Component Description

The main objective of this project was to make it easy to extend the description of a component so that the developer of the component could define their own properties. This requirement was achieved in BeanBag, by using XML and allowing the properties to be defined in a DTD.

5.2.2.1 Defining new properties in the Developer Description

The steps involved in defining a new property for a component are

1. Add a description for the new property as an element in the components.dtd

A component can use security, e.g. SSL. The new element for security may look like this:

```
<!ELEMENT security EMPTY>
<!ATTLIST security id ID #REQUIRED>
```

2. Add this element to the overall component description in the components.dtd

For instance, a component can use zero or more levels of security

```
<!ELEMENT components (component+)>

<!ELEMENT component (type, date+, release, author+,
institution*, genre+, keyword*, test.environment*, needs*,
method*, transport, security*, comment*)>
```

3. Define the element in the xml/component.xml file, which in turn will be included in the EJB-JAR archive. The XML for the security element would look like this:

```
<components>
         <component>
               ...
               <security id="SSL"></security>
               ...

         </component>
</components>
```

Using the BeanBag system, insert the JAR. BeanBag will automatically parse the XML and validate it against its corresponding DTD. If there are no errors, the new property will be inserted transparently into the Properties table.

5.2.3 The Query System

The requirements for the Query Interface were described in Section 3.2.3. It was not possible to use an XML Query language such as XQL, as BeanBag had to use a relational database, whereas XQL requires an Object-Oriented database. Therefore queries on the database were achieved by composing SQL queries.

5.2.4 Queries Implemented

The GUI allows the user to input any SQL query and the client passes this to the server to be executed. The main queries that are implemented in this version are:

- Return all the properties for a component
- Return the interface for a component
- Return all components that implement particular methods
- Return a component that implements this interface

Section 4.7.1 describes how exact interface matching was implemented. The interface matching in this version of BeanBag, matches the method names, base classes and implemented interfaces exactly but has yet to match method parameters and types.

5.2.5 Evaluation of Exact Interface Matching

The Exact Interface Matching module could be improved. Presently the interface input by the user is parsed to extract the method names, base classes and the interfaces extended. Then components that correspond to these values are returned. Interface parsing is useful when a partial match is requested. However with an exact interface match, all parts of the interface must match.

Another way to achieve an exact interface match would be to strip out all white spaces and comments from an interface before it is inserted into the database. This interface would not have parameter names, as it is returned by Reflection. The input interface could also undergo processing to strip out comments and whitespace, and both interfaces could be compared. A hashing function could be applied to both interfaces, and if they are exact matches, the resulting hash codes should be the same.

5.3 Conclusions

5.3.1 Achievements

The BeanBag system that has been developed is a working prototype, for Enterprise Java Bean components. EJBs were used as a starting point for the component repository.

The requirements for the storage framework were outlined in Section 3.2.1. Component data was stored in the relational database and a dynamic way of storing new properties was implemented, using the Properties table [see Section 3.3.3.2]. The name, remote interface, user description and other data is stored transparently when a component location is given to the BeanBag system. An error system was implemented so that database errors are returned to the GUI client. The bottleneck of the storage mechanism is that the Properties table will become large and this may incur a performance overhead when searched on. The component type-checking model described in Section 3.5 was not implemented, but could be in the future.

The primary requirement for describing components was to design a component independent and extensible description. This was achieved by using XML. This meant that the component description was extensible, as XML is extensible by definition. The limitation of using XML was implementing the linking of components. Because the XLink and XPointer specifications are incomplete, the functionality in BeanBag for linking a component with all the sub-components it needs is not implemented.

The BeanBag retrieval mechanism implements different levels of interface matching and the retrieval of component semantics. Interfaces can be matched on their methods, base classes and interfaces implemented. The exact interface-matching module is not implemented in its entirety, and Section 5.4.2.1 describes how this module could be extended.

In Section 3.3 the requirement for keeping statistics about components was mentioned. This would be a useful feature for developers to provide feedback on components retrieved and how relevant they were, so that other developers could see these reviews when searching for components. This functionality was not implemented in the prototype due to time constraints. Similarly, a ranking system for components retrieved from the query module, was not implemented.

The BeanBag prototype is a working component repository that facilitates the efficient storage of EJB components. The query mechanism can retrieve components, their interfaces and their developer descriptions. The description mechanism is extensible and could be applied to other component types in the future. The following section describes how BeanBag could be evolved to include extra functionality.

5.4 Future Developments

BeanBag is implemented as a CORBA Server. However, it could develop in two directions. It could be extended as a CORBA Server, or it could be implemented as an EJB Component. Both directions are examined here.

5.4.1 BeanBag as an EJB Component

The BeanBag system operates as a typical three-tier model, comprising of clients that communicate with an application server. The server uses business logic and also communicates with a database. As outlined in [URL21] developing this middle tier is an expensive business. In a perfect world, the developer should only be concerned with the business logic required to implement the system. Details of how to implement transaction management, persistence and security should not be of primary interest to the developer. Abstracting these services is what the EJB Container does [see Section 2.4.2] and so BeanBag is a perfect candidate for evolving to an EJB Component.

During the implementation, an attempt was made to convert BeanBag into an EJB Component using HomeBase. The BeanBag storage framework used composite keys on some database tables, and therefore finder methods had to be written for the EJBHome object. This object is used to retrieve the components that are needed by the client. It uses finder methods to do this, the simplest of which matches a primary key. However, writing finder methods for composite keys was not implemented in HomeBase at the time (July 1999). This functionality will be supported by IONA in the future, but it was decided to implement BeanBag as a CORBA Server.

The design of BeanBag as an EJB Component is shown in Figure 5.1, with a set of EJB Entity Beans for persistent access and a number of Session Beans to execute the business logic for the system.

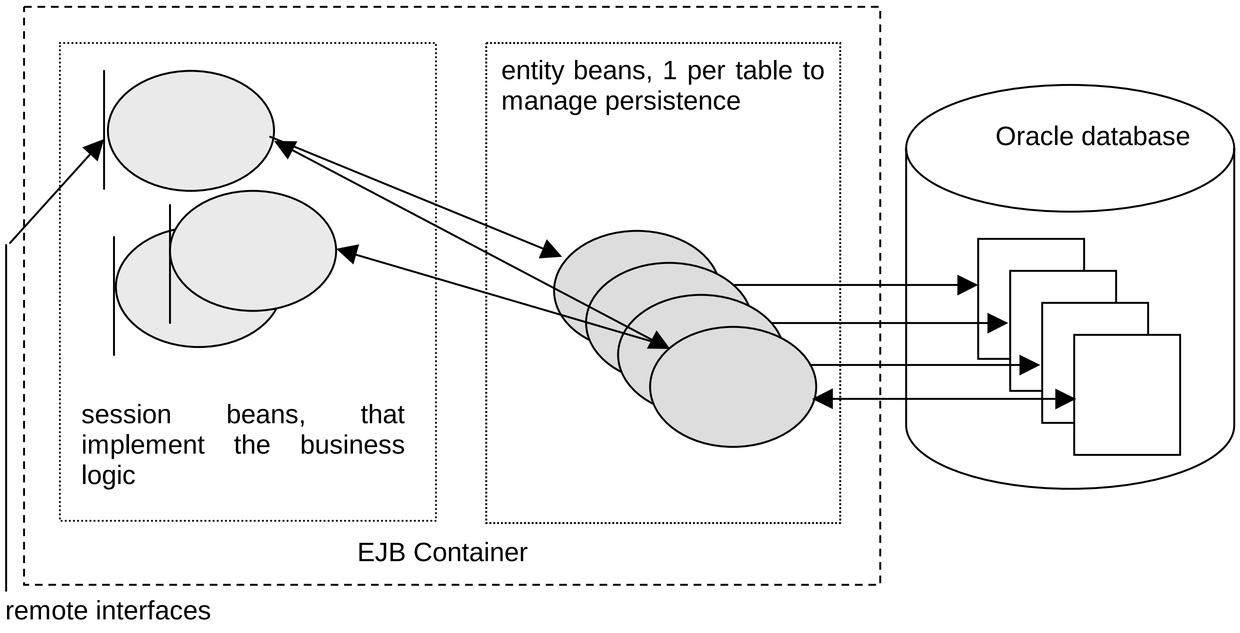

entity beans, 1 per table to manage persistence
Oracle database
session beans, that implement the business logic
EJB Container
remote interfaces

Figure 5-1 Design for BeanBag as an EJB Component

Session Beans contain the business logic that BeanBag already uses, for reading JAR archives, using the Reflection API and parsing XML files. However, the code for accessing the database will not be used, as the role of the EJB entity beans is to manage persistence. Business objects will operate on objects that represent the persistent data, without the knowledge of when and how the data is stored in the database.

For more information on constructing Enterprise JavaBeans, consult [RMH99].

5.4.2 Extending BeanBag as a CORBA Component Repository

BeanBag at present is a prototype of a CORBA Component Repository that could be extended in the following ways.

5.4.2.1 Exact Interface Matching

At present the exact interface-matching module of BeanBag matches the component methods, base classes and interfaces. In order for an exact interface match to occur many components of an interface have to match. These were outlined in Section 4.7.1. Matching parameter types and member variables could be implemented in the next version of the system.

5.4.2.2 Adding a Class Loader

BeanBag uses Reflection to query the remote interfaces of an EJB Bean. This implies that the EJB-JAR archive must be on the `CLASSPATH` of the Server. However, it would be inconceivable to expect the user to add the path of the EJB-JAR to the `CLASSPATH` each time a new component was to be inserted, and to re-register the BeanBag server.

A Class Loader should be added to the BeanBag system, to dynamically load the classes that are needed. From parsing the .ejbml files the name of the remote class can be found, and then the remote interface class can be loaded by the class loader, so that the Reflection API can be used to retrieve the remote interface. [URL8] describes Class Loaders in more detail.

5.4.2.3 Transactions

Inserting a component into the BeanBag system results in a number of database tables being updated. BeanBag should utilise transactions to ensure that if there is an error while inserting data into one or more tables, that the insert operation is aborted, else when the component was inserted again, BeanBag would complain that it has been inserted previously.

5.4.3 Using OCL to describe the pre- and post- conditions

In the current version of BeanBag, pre- and post- conditions are represented as strings in XML tags. The scope of the project did not include condition parsing, however a future version of the system should implement this module to facilitate the retrieval of components that satisfy a pre- or post- condition. This allows the user to extract more semantic information about a component.

In a future version of the system, pre- and post- conditions, invariants and other semantic information about methods, could be represented in OCL [see Section 2.5.2]. A module could be added to match the user requirements for methods with the OCL expressions describing component behaviour.

This is an example of how the post-condition for the `birthdayHappens()` method is represented in OCL.

```
Person::birthdayHappens()
      post : age = age@pre + 1
```

The OCL for the post-condition can be represented in the XML tags for methods described in Section 3.4.2. The XML for the `birthdayHappens` () method in the BeanBag system would look like this:

```
<method id='birthdayHappens'>
      <parameter name='age' type='integer'></parameter>
      <post-condition>age = age@pre + 1</post-condition>
</method>
```

5.4.4 Searching on the Documentation

BeanBag could be extended to enable the storage of developers' component documentation and design documents. If the JavaDoc was stored in the component, BeanBag could be extended to automatically extract parts of the documentation and store them in the database, so that developers could query this data. This would also apply to the UML design documents, if they were deployed with the component. This documentation would be useful from a semantic perspective to developers and component assemblers.

5.4.5 The Oracle XML SQL Utility for Java

New technologies also influence the direction that BeanBag may take in the future. XML processing and schema languages are always progressing. Oracle's DMBS Oracle8i includes XML parsers for Java, C, C++ and PL/SQL. The DBMS product can return query results as XML and load data from an XML document into a database table or view. This means that the XML file does not have to be parsed by a SAX Parser.

The Oracle XML SQL Utility for Java performs the following:

- Generates XML documents from SQL queries
- Writes XML documents into database tables
- Supports W3C XML 1.0 Recommendation

Using the XML SQL Utility, a BeanBag client could perform this query:

```
    SELECT beanName FROM Properties WHERE keyword = ‘e-commerce’;
```

and the result returned from the database would be in XML form:

```
<?xml version=”1.0”?>
```

```
<ROWSET>
      <ROW id=”1”>
            <beanName>FinanceBean</beanName>
      </ROW>
      <ROW id=”2”>
            <beanName>BankingBean</beanName>
      </ROW>
</ROWSET>
```

BeanBag could use style sheets to display these results to the user.

The ability for the XML SQL Utility to write XML documents to the database brings more benefits to BeanBag. The current database includes a simple table *Names* that maps unique identifiers to component names. The column names are `id` and `name`. To insert a row into this table in SQL looks like:

```
      INSERT INTO Names (id, name) VALUES (10101, ‘BankingBean’);
```

If this were to be represented in XML it would look like the following XML file, which includes its DTD and one bean name and id.

```
<?xml version = “1.0”?>
<!DOCTYPE ROWSET [
<!ELEMENT ROWSET (ROW)*>
<!ELEMENT ROW (ID, NAME)>
<!ELEMENT ID (#PCDATA)>
<!ELEMENT NAME (#PCDATA)>
]>
<ROWSET>
<ROW>
<ID>10101</ID >
<NAME>BankingBean</NAME>
</ROW>
</ROWSET>
```

The database table `Names` Meta Data is:

ID	VARCHAR(10)
NAME	VARCHAR(20)

The XML SQL API is then used to store the XML file in the database.

Had this technology been available during this project, it would have eliminated the process of parsing the XML file and saving the elements and attributes in the database. The database tables could have been designed in conjunction with the DTD to allow for this utility to insert the data automatically.

5.4.6 Implementing an XML Properties Editor

In BeanBag, components are deployed with their Developer Description, i.e. in the case of EJBs, the `xml/component.xml` file. Developers need to populate this file themselves, therefore it would be useful to have a GUI interface for it. Technologies such as the XML Editor Maker from Alphaworks IBM is a Java tool that generates visual editors from XML schemas, for instance DTDs. Editor Maker reads the DTD and produces the corresponding Java classes for a GUI Editor automatically.

The current version of EditorMaker is available from the Alphaworks website. During the implementation phase of this system it was installed, but the current version has some class referencing inconsistencies with its sister component, XML BeanMaker.

5.5 Summary

In this chapter, the BeanBag modules were evaluated against the requirements outlined in Chapter 3. An example of how to insert a component property into the description was given to show how the XML component descriptions are extensible. The query module was evaluated and sample queries that are implemented were listed.

The final section comprises of the conclusions, and outlines exactly what was implemented in the system. Certain functionality was not included in the system, and reasons for this are given. The BeanBag system could develop as a CORBA server or as an EJB component, and both these avenues are explored in the future developments section.

6. Bibliography

[BW] Anne Thomas. (1998). Borland/Inprise Enterprise JavaBeans WhitePaper

[CCM99] OMG. The CORBA Component Model, Final submission http://www.omg.org/cgi-bin/doc?orbos/99-02-05

[Chap97] David Chappell, Chappell & Associates. (1997) The Next Wave, Component Software Enters the Mainstream. A Rational Whitepaper.

[COM+99] JP Morgenthal. (July 1999). Microsoft COM+ will Challenge Application Server Market. Microsoft COM+ Whitepaper, July 1999.

[DFFLC98] Alin Deutsch, Mark Fernandez, Daniela Florescu, Alon Levy, Dan Suciu. XML-QL: A Query Language for XML. QL'98.

[EJB1.0] Enterprise JavaBeans Specification 1.0 http://java.sun.com/products/ejb/docs10.html

[EJB1.1] Enterprise JavaBeans Specification 1.1 http://java.sun.com/products/ejb/docs.html

[Flan97] David Flanagan. (1997). Java Examples in a Nutshell. O'Reilly.

[Flan97_2] David Flanagan. (1997). Java In A Nutshell. O'Reilly.

[Geor99] Nektarios Georgalas. (1999). A Framework that uses Repositories for Information Systems and Knowledge Integration. Proceedings of the ASSET99 Symposium on Application-Specific Systems and Software Engineering Technology, IEEE Computer Society, Dallas, Texas, 24-27 March 1999

[HHK98] Ali Hamie, John Howse, Stuart Kent. (1998). Interpreting the Object Constraint Language. Proceedings of Asia Pacific Conference in Software Engineering, IEEE Press.

[Hub99] Richard Hubert. (May 1999). CORBA Components Vs EJB Styles, Setting Standards for Distributed Architecture. Component Strategies.

[IKK98] Hiroshi Ishikawa, Kazumi Kubota, Yahuhiko Kanemasa. A Query Language for XML Data. QL'98.

[Kie98] Don Kiely. (April 13, 1998). The Component Edge. InformationWeek

Issue 677.

[Kin98] Joseph R. Kiniry. (1998). The Specification of Dynamic Distributed Component Systems. OOPSLA ’98.

[Kin99] Joseph R. Kiniry. (1999). Leading to a Kind Description Language: Thoughts on Component Specification. Presented at the COOTS'99 Advanced Topics Workshop On Validating the Composition/Execution of Component-based systems.

[MR99] Mark Reinhold, An XML Data-Binding Facility for the Java Platform, 30 July 1999

[Muh96] M. Muhlhauser. Special Issues in Object–Oriented Programming ECOOP96 Workshop Reader.

[NH99] Simon North, Paul Hermans. (February 1999). SAMS Teach Yourself XML in 21 Days.

[Oracle99] Oracle XML SQL Utility for Java. (July 1999). Release Notes for 1.1.0.

[OSQL] The Oracle8 SQL Server Reference

[OYM] Mika Ohtsuki, Norihiko Yoshida and Akifumi Makinouchi. (1996) A Distributed Repository for Object-Oriented Software Components. 3rd Asia-Pacific Software Engineering Conference (APSEC '96).

[QJHF98] Wu Qiong, Chang Jichuan, Mei Hong, Yang Fuqing. (1998). JBCDL: An Object-Oriented Component Description Language. Proceedings of the Technology of Object-Oriented Languages and Systems-Tools.

[RMH99] Richard Monson-Haefel. (June 1999). Enterprise JavaBeans. O’Reilly.

[SK] Salman Khan. Accessing Oracle from Java. Whitepaper from Oracle Corp.

[SLR98] David Schach, Joe Lapp, Jonathan Robie, Querying and Transforming XML. QL’98.

[Szy98] Clemens Szyperski. (1998). Beyond Object-Oriented Programming. Addison-Wesley.

[URL1] Softlab Whitepaper on Enabler Technology Overview.

http://www.softlabna.com/Products/Enabler/WhitePaper/ena_tech.html

[URL2] The Enterprise JavaBeans Specification Version 1.1 Public Release

http://java.sun.com/products/ejb/docs.html

[URL3] UML Object Constraint Language Specification, Version 1.1, September 1997

http://www.rational.com/uml/resources/documentation/index.jtmpl

[URL4] IBM article on the Object Constraint Language

http://www.software.ibm.com/ad/standards/ocl.html/

[URL5] The JavaBean's FAQ
http://java.sun.com/beans/FAQ.html

[URL6] Mark Johnson. (1998). *A beginner's guide to Enterprise JavaBeans, JavaWorld.*
http://www.javaworld.com/javaworld/jw-10-1998/jw-10-beans.html

[URL7] Mike Day. (1998). *Contrasting JavaBeans and Enterprise JavaBeans, A Developer's Round Table Discussion.* IBM Online Library.

http://www.ibm.com/java/education/javabeans-enterprise-javabeans.html

[URL8] The Java Class Loader API
http://java.sun.com/products/jdk/1.0.2/api/java.lang.ClassLoader.html

[URL9] The Java Bug Parade
http://developer.java.sun.com/developer/bugParade/bugs/4040920.html

[URL10] The Swing Connection
http://java.sun.com/products/jfc/tsc/index.html

[URL11] The CORBA Component Model

http://www.omg.org/news/pr99/9_02a.html

[URL12] IONA Knowledge Base article on oneways
http://www.iona.com/online/support/kb/OrbixWeb/articles/1074.398.html

[URL13] Dr. GUI on Components, COM and ATL, Microsoft Corp. 1999
http://msdn.microsoft.com/isapi/msdnlib.idc?theURL=/library/welcome/dsmsdn/msdn_drguion020298.htm

[URL14] Microsoft Repository SDK Version 2.1b. 1999
http://msdn.microsoft.com/repository/downloads/sdk/default.asp

[URL15] Trisha Gorman. (1998). 20 questions on XML
http://builder.cnet.com/Authoring/Xml20/index.html

[URL16] Jay Greenspan, Introduction to XML for HotWired, 13 October 1998
http://www.hotwired.com/webmonkey/98/41/index1a.html?tw=xml

[URL17] Lisa Rein. (March 1999). *The Quest for an XML Query Standard.*
http://www.xml.com/xml/pub/1999/03/quest/index.html

[URL18] The Lore Database Management System

http://WWW-DB.Stanford.EDU/lore/

[URL19] An article posted on comp.databases.oracle.misc

http://x47.deja.com/getdoc.xp?AN=207304091&CONTEXT=93540261
4.411041870&hitnum=5

[URL20] An article posted on borland.public.delphi.database.sqlservers

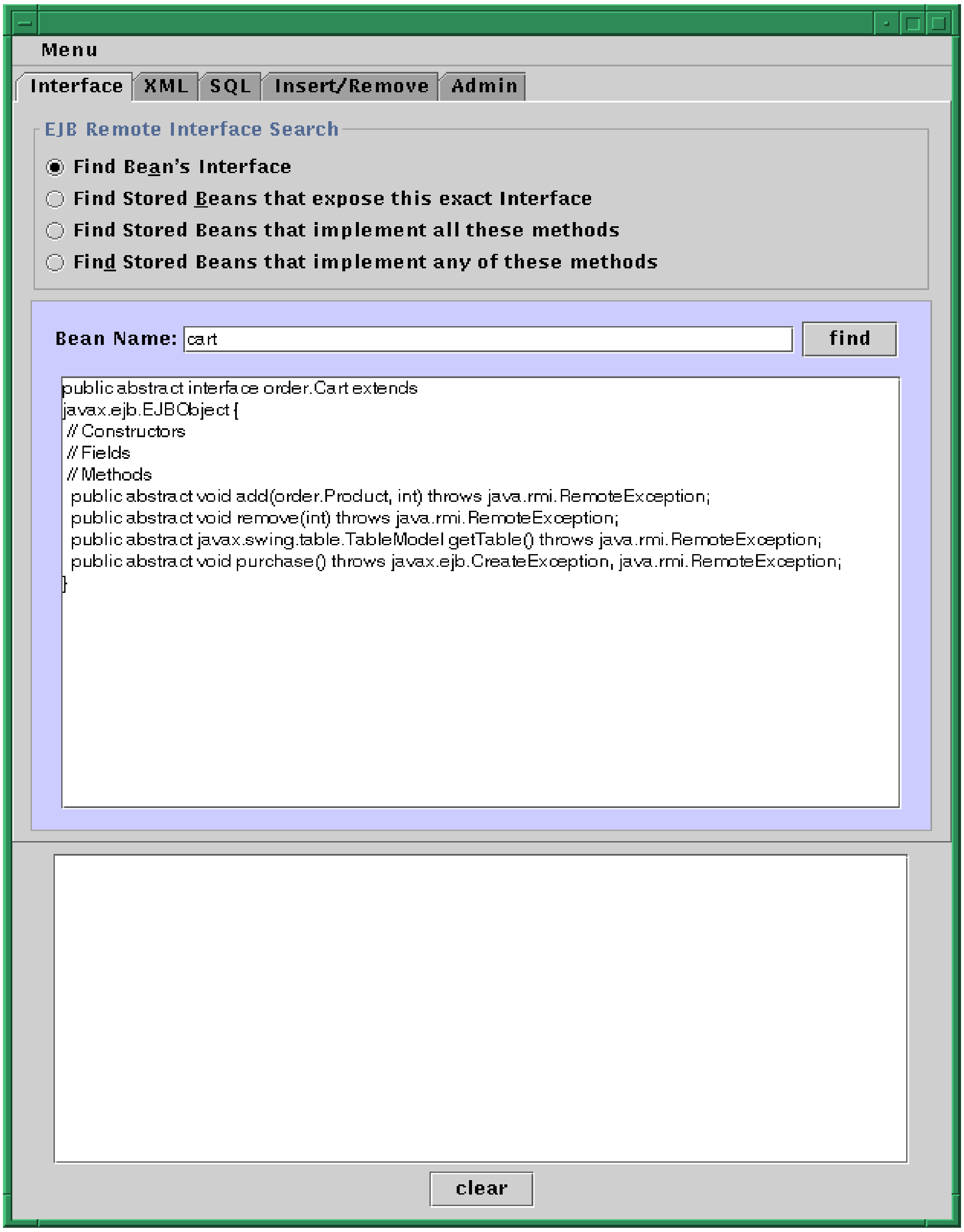
Oracle8 - maximum open cursors exceeded

http://x42.deja.com/getdoc.xp?AN=506493851&CONTEXT=93446794
4.1664876594&hitnum=3

[URL21] Simon Ritter. *Enterprise JavaBeans: Answers to every developer's top questions*. IBM Online Library.
http://www.software.ibm.com/developer/library/ejbsun/Appendix

7. Appendix

7.1 Appendix A: Screen Shots

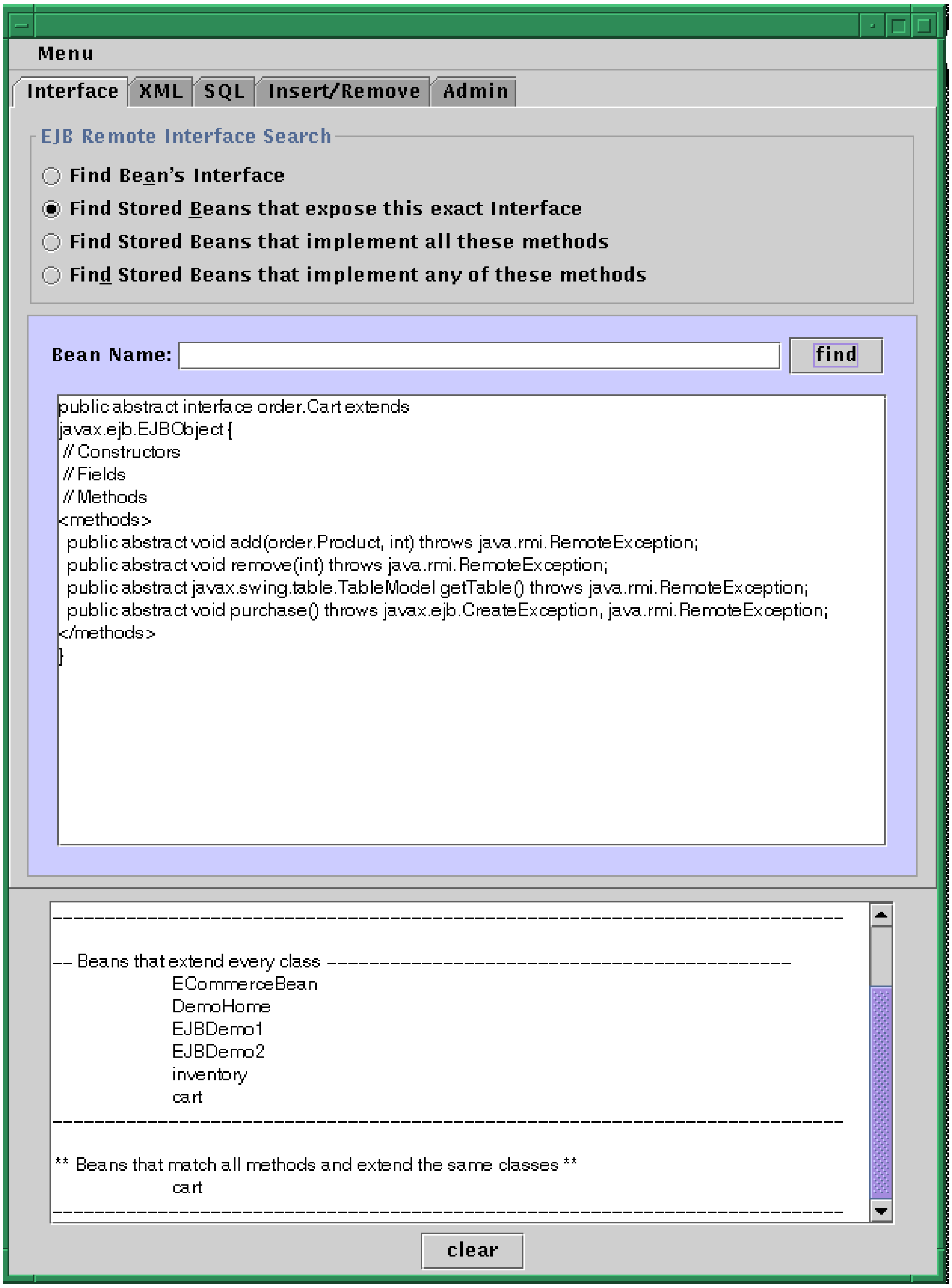


Find Bean's Interface [✓]
Find Stored Beans that expose this exact Interface
Find Stored Beans that implement all these methods
Find Stored Beans that implement any of these methods

Bean Name: cart

```
public abstract interface order.Cart extends
javax.ejb.EJBObject {
 // Constructors
 // Fields
 // Methods
  public abstract void add(order.Product, int) throws java.rmi.RemoteException;
  public abstract void remove(int) throws java.rmi.RemoteException;
  public abstract javax.swing.table.TableModel getTable() throws java.rmi.RemoteException;
  public abstract void purchase() throws javax.ejb.CreateException, java.rmi.RemoteException;
}
```

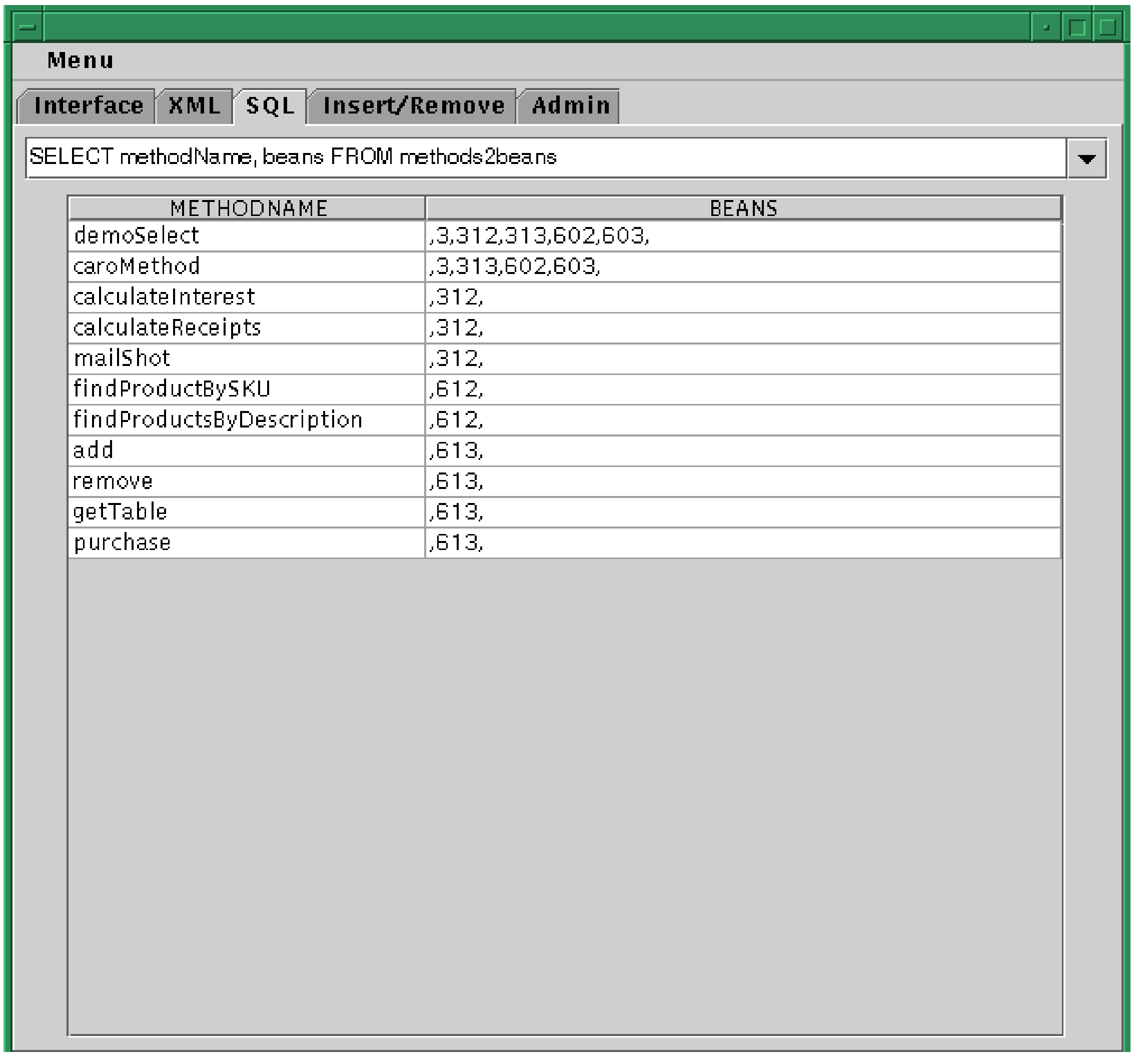
Figure 7-1 Retrieving the Remote Interface of a Component


Menu
Interface
XML
SQL
Insert/Remove
Admin
EJB Remote Interface Search
Find Bean's Interface
[✓ Find Stored Beans that expose this exact Interface]
Find Stored Beans that implement all these methods
Find Stored Beans that implement any of these methods
Bean Name:
find
clear

```
public abstract interface order.Cart extends
javax.ejb.EJBObject {
 // Constructors
 // Fields
 // Methods
<methods>
  public abstract void add(order.Product, int) throws java.rmi.RemoteException;
  public abstract void remove(int) throws java.rmi.RemoteException;
  public abstract javax.swing.table.TableModel getTable() throws java.rmi.RemoteException;
  public abstract void purchase() throws javax.ejb.CreateException, java.rmi.RemoteException;
</methods>
}
```

```
--------------------------------------------------------------------------------

-- Beans that extend every class ---------------------------------------------
                  ECommerceBean
                  DemoHome
                  EJBDemo1
                  EJBDemo2
                  inventory
                  cart
--------------------------------------------------------------------------------

** Beans that match all methods and extend the same classes **
                  cart
--------------------------------------------------------------------------------
```

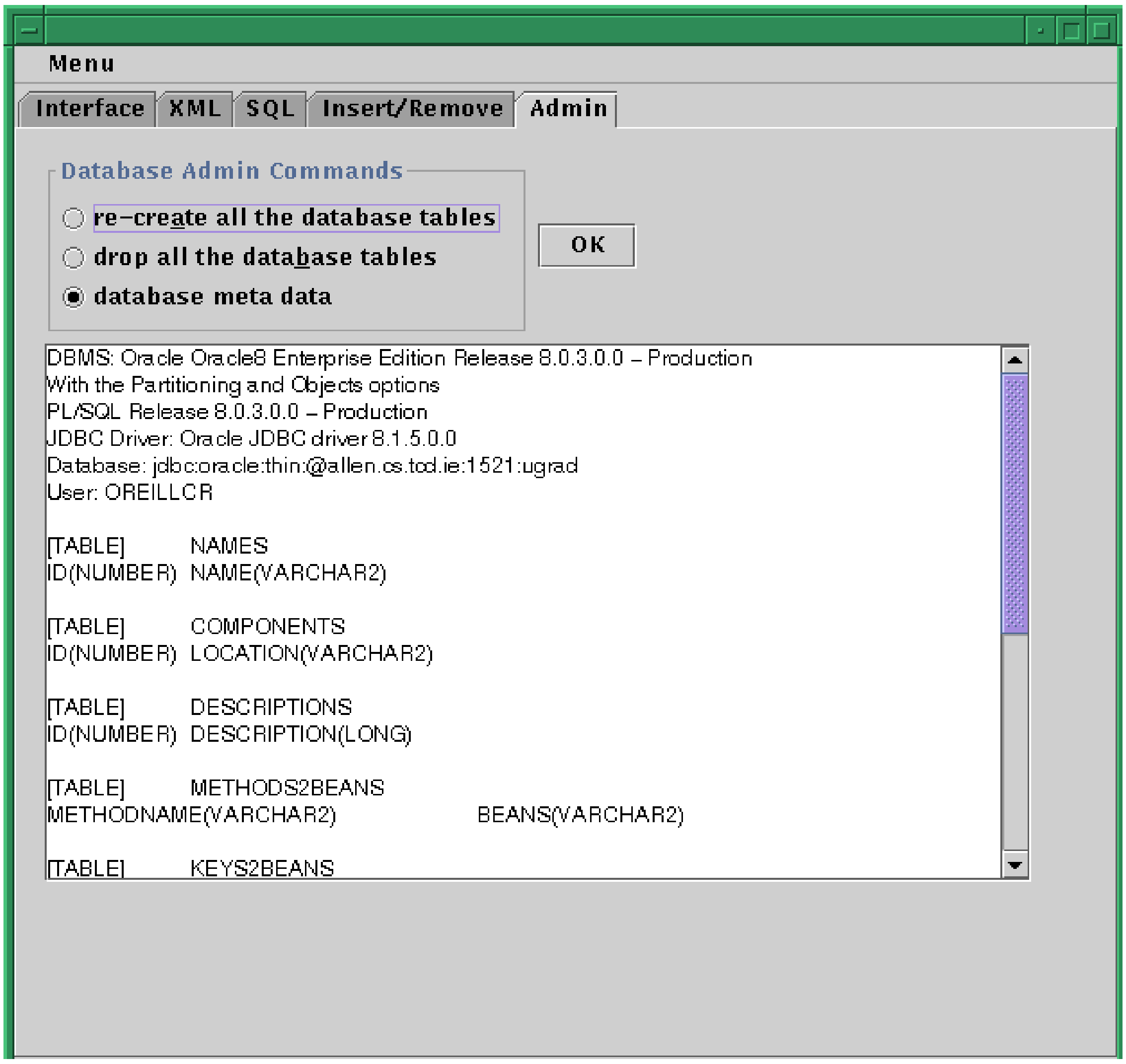
Figure 7-2 Matching the Interface of a Component with one in the Database



```
SELECT methodName, beans FROM methods2beans
```

METHODNAME	BEANS
demoSelect	,3,312,313,602,603,
caroMethod	,3,313,602,603,
calculateInterest	,312,
calculateReceipts	,312,
mailShot	,312,
findProductBySKU	,612,
findProductsByDescription	,612,
add	,613,
remove	,613,
getTable	,613,
purchase	,613,

Figure 7-3 The methods2beans Indexed Table



re-create all the database tables
drop all the database tables
[● database meta data]

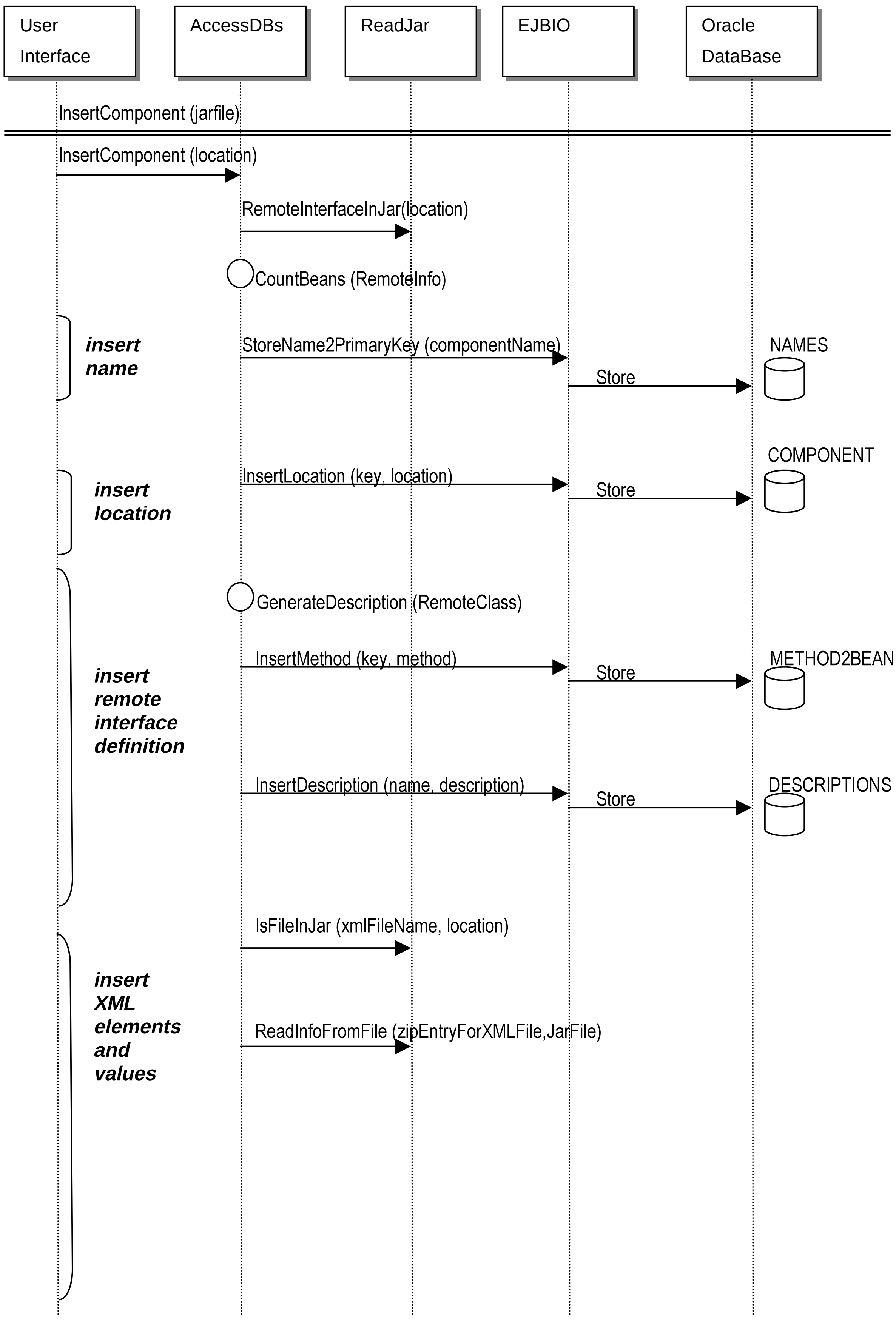
```
DBMS: Oracle Oracle8 Enterprise Edition Release 8.0.3.0.0 – Production
With the Partitioning and Objects options
PL/SQL Release 8.0.3.0.0 – Production
JDBC Driver: Oracle JDBC driver 8.1.5.0.0
Database: jdbc:oracle:thin:@allen.cs.tcd.ie:1521:ugrad
User: OREILLCR

[TABLE]       NAMES
ID(NUMBER)  NAME(VARCHAR2)

[TABLE]       COMPONENTS
ID(NUMBER)  LOCATION(VARCHAR2)

[TABLE]       DESCRIPTIONS
ID(NUMBER)  DESCRIPTION(LONG)

[TABLE]       METHODS2BEANS
METHODNAME(VARCHAR2)              BEANS(VARCHAR2)

[TABLE]       KEYS2BEANS
```

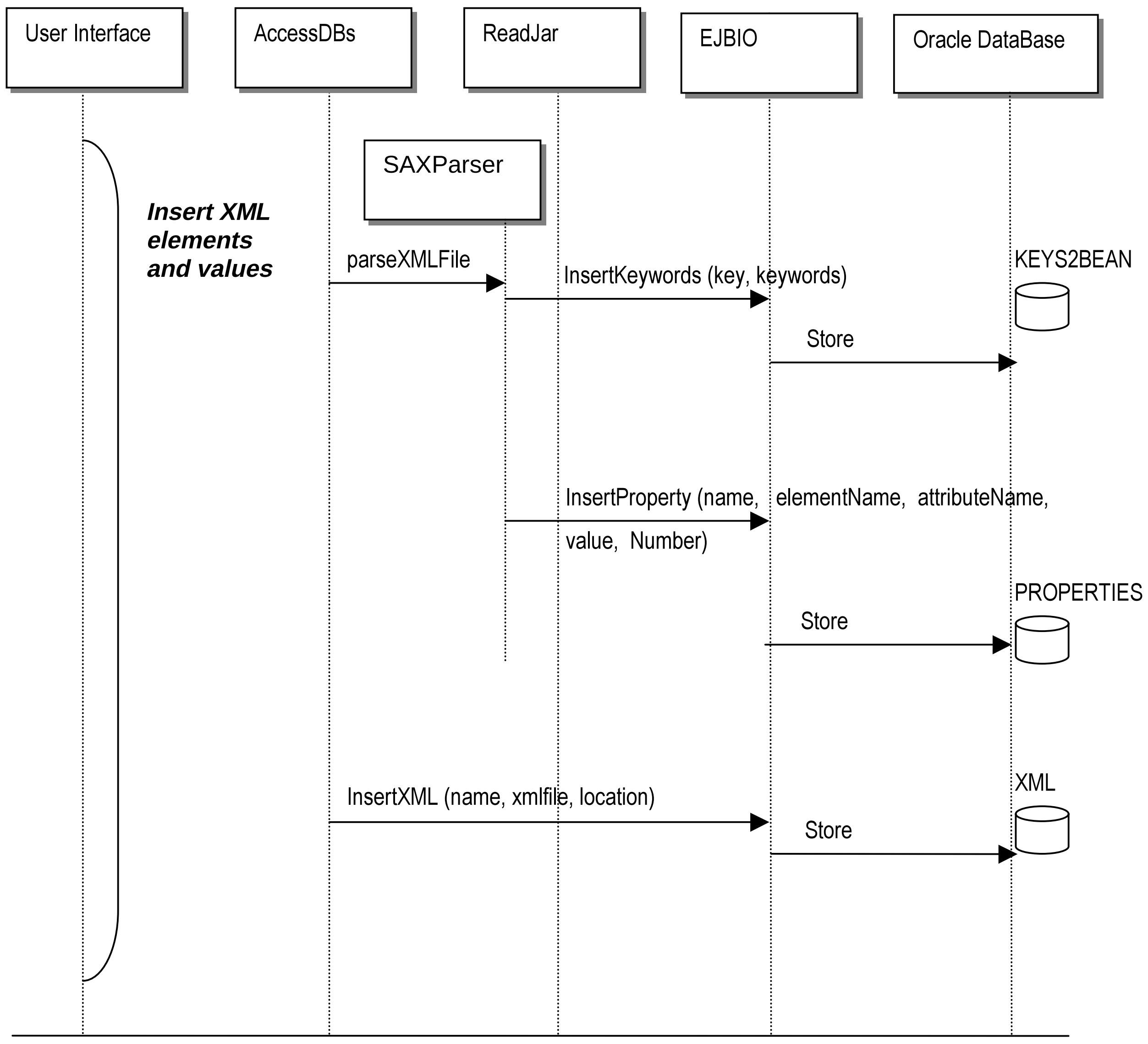
Figure 7-4 Retrieving Table & Driver Meta Data from the Database

7.2 Appendix B: The BeanBag IDL

```
// forward declaration
interface repository;

// factory that will create a new repository object for each client
interface repositoryFactory {
      repository newRepository();
};

// this is the callback object. When the sql is finished on the
// server it notifies the client
typedef sequence<string> strArray;
interface callback {
      // return the column names of the result set
      oneway void columnNames(in strArray oneD);
      // return the results
      oneway void results(in strArray twoD);
};

// the main repository object that will interface with the database
interface repository {
      // thrown this exception if there is an error on the server side
      // this way the client knows what is happening with the database
            exception db_error {string reason;};

      // Methods corresponding to the Interface Tab
      string retrieveDesc(in string componentName) raises (db_error);
      strArray retrieveBeansWithMethods(in strArray methods,
            in string bool) raises (db_error);
      strArray retrieveBeansThatExtend(in strArray extend,
            in string bool) raises (db_error);

      // Methods corresponding to the XML Tab
      oneway void retrieveProperties(in string componentName,
                              in callback obj);
      oneway void retrieveBeansFromProperties(in string whereClause,
                              in callback obj);
      // Methods corresponding to the SQL Tab
      oneway void executeSQL(in string sql, in callback obj);

      // Methods corresponding to the Insert/Remove Tab
      void insertComponent(in string location) raises (db_error);
      void deleteFromTable(in string beanName) raises (db_error);

      // Methods corresponding to the Admin Tab
      strArray printMeta() raises (db_error);
      void createTables() raises (db_error);
      void dropTables() raises (db_error);
      void closeUpShop() raises (db_error);
      void queryData(in string tableName) raises (db_error);

      string retrieveLocation(in string componentName)
                  raises (db_error);
      string retrieveXML(in string componentName)
                  raises (db_error);
};
```

7.3 Appendix C: BeanBag Sequence Diagram


User Interface
AccessDBs
ReadJar
EJBIO
Oracle DataBase
InsertComponent (jarfile)
InsertComponent (location)
RemoteInterfaceInJar(location)
CountBeans (RemoteInfo)
insert name
StoreName2PrimaryKey (componentName)
Store
NAMES
COMPONENT
insert location
InsertLocation (key, location)
Store
GenerateDescription (RemoteClass)
insert remote interface definition
InsertMethod (key, method)
Store
METHOD2BEAN
InsertDescription (name, description)
Store
DESCRIPTIONS
IsFileInJar (xmlFileName, location)
insert XML elements and values
ReadInfoFromFile (zipEntryForXMLFile,JarFile)


User Interface
AccessDBs
ReadJar
EJBIO
Oracle DataBase
SAXParser
Insert XML elements and values
parseXMLFile
InsertKeywords (key, keywords)
KEYS2BEAN
Store
InsertProperty (name, elementName, attributeName, value, Number)
PROPERTIES
Store
XML
InsertXML (name, xmlfile, location)
Store