Interoperable components across multiple component architectures

Ravi S. Banda
West Virginia University

Follow this and additional works at: https://researchrepository.wvu.edu/etd

Recommended Citation
https://researchrepository.wvu.edu/etd/901

This Thesis is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself. This Thesis has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.
Abstract

A component is a pre-fabricated, pre-tested software object that provides clients with a well-defined set of services. Component software requires an architecture through which any developer or vendor can deliver a component at any time and have that component become immediately useful to applications on any given system. A system that supports component software must therefore support a generic service abstraction that is, an architecture that defines how all types of components appear and how they are manipulated. As part of the thesis various component architectures COM, JavaBeans and CORBA have been studied and the interoperability issues between the components developed using these architectures has been discussed. A Migration tool has been developed to assist the developer in migrating an ActiveX control to a platform independent JavaBean. The tool gives a list of all ActiveX controls in the Windows registry and developer can select any particular ActiveX control to be converted to a JavaBean. The tool gives a skeleton code in Java for a JavaBean, which has properties and events similar to that of original ActiveX control. JavaBeans have been developed for making Graphical User Interface applications and Database applications like Forms and Reports. This further emphasizes that component-based development fosters rapid development of highly customizable applications.
Acknowledgments

I wish to express my gratitude and appreciation to my advisor Dr. Srinivas Kankanahalli for providing me with excellent guidance and encouragement. I would also like to thank my committee members Dr. V. Jagannathan and Dr. Sumitra Reddy for their help and guidance.

I would like to thank my parents, my sister and my cousins Lalitha and Raghu for their encouragement and support.

I am thankful to Cristian Goina for his guidance and to my friends Avinash Kalgi, Chethan Gubbi, Sharath Simha, Srivatsan Kannan, Sudarshan Hebbale for their encouragement and support.
# Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Software Components</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2.1 Object Orientation</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2.2 Granularity</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2.3 Component Types</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>2.4 Visual/Non-Visual</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Component Architectures</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>3.1 Component Object Model</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>3.2 Distributed Component Object Model</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>3.3 JavaBeans Component Model</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>3.4 Enterprise JavaBeans</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>3.5 CORBA</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>3.5.1 The OMG Object Model</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>3.6 Need for Interoperability</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>Migration Assistant for ActiveX Controls</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>4.1 Java Native Interface</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>4.2 Registry Viewer</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>4.2.1 Registry</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>4.3 Control Viewer</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>4.3.1 ITypeInfo interface</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>4.4 Code Generator</td>
<td>32</td>
</tr>
</tbody>
</table>
5 JavaBeans for building GUI applications 35
6 Database Components 39
  6.1 DataControlBean 40
  6.2 Forms Wizard 41
  6.2.1 Design View 42
  6.2.2 Simple Form Wizard 43
  6.2.3 Structure 44
7 Help System 46
  7.1 HTML Parser 46
  7.2 Help Tree 47
  7.3 Help Contents 47
  7.4 Help Browser 47
  7.5 Structure 48
8 Conclusions and Future Work 49
  8.1 Conclusions 49
  8.2 Future Work 49

Bibliography 50
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Structure of Control Viewer</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>Structure of OLEControlViewer</td>
<td>30</td>
</tr>
<tr>
<td>6.1</td>
<td>Structure of Forms Wizard</td>
<td>44</td>
</tr>
<tr>
<td>7.1</td>
<td>Structure of Help System</td>
<td>48</td>
</tr>
</tbody>
</table>
Chapter 1. Introduction

'Components' is the slogan going around the software industry today. Component-based development is traditionally defined as the process of building systems by way of combination, aggregation and integration of pre-engineered and pre-tested software objects. This approach provides three benefits: reusability of software, control of complexity and control of change. Many of the specific benefits of component-based development, such as rapid-application development, increased software quality, and improved maintainability and modifiability are directly related to the notion of assembling applications from reusable software.

The earliest discussions of component technology were limited to compound document technology such as Microsoft's OLE and IBM's OpenDoc. Soon afterwards the focus was exclusively on Visual Basic and VBX controls. Although these controls were well known examples of software components they were of a poor standard because they were inherently 16-bit oriented. They were replaced by 32-bit OCX controls and later by ActiveX controls as the defacto industry standard for software components in Windows environment. Today, encapsulated software objects such as ActiveX controls and JavaBeans are industry standards for software components.

Component software requires an architecture which supports a generic "service abstraction" - that is, an architecture that defines how all types of components appear and how they are manipulated. This allows the components developed by various developers to interact without any problems. In addition, the architecture must be extensible, so that a new component category can be introduced without having to revise the architecture. This is the biggest problem of creating an extensible service architecture.
This Thesis focuses on the development of component software and studies the currently existing architectures COM, JavaBeans and CORBA. There are lots of differences between these architectures, thus making the interoperability between components very difficult. Any company, which has invested in one of these technologies, has to spend lot of effort and time to redirect their software toward a different technology. There is a necessity for migration tools, which will assist the developers in moving their effort from one technology to other with a minimum effort. Some of the tools are Sun's ActiveX Bridge for JavaBeans, which allows JavaBeans to be converted to ActiveX controls, IBM's Migration Assistant to convert ActiveX controls to JavaBeans. As part of the Thesis a migration tool for converting ActiveX controls to JavaBeans was developed. The various chapters of this work concentrate on the software components, component models, migration tool, JavaBeans for GUI applications, Database beans, Help System for Chicory, Conclusions and Future work that can be done to enhance the features of migration tool.
Chapter 2. Software Components

A component is a pre-fabricated, pre-tested software object that provides clients with a well-defined set of services. Each component is a distinct entity that can be defined and described outside the context of any particular software package. It is typically decoupled from other components, processes, services and applications. Components are pieced together in various ways to realize complete, integrated software systems. Component-based development also fosters rapid development of highly customized applications. Equally importantly, most applications built with components will be highly “customizable”. Depending on their support for inheritance, components can offer limited customizability (property editing), or they can be fully customizable by subclassing [1].

One method to understand the universe of components is to examine the attributes of the various types of component software.

- **Object-orientation**
  - Encapsulation
  - Polymorphism
  - Inheritance

- **Granularity**
  - Fine-Grained
  - Larger-Grained

- **Component Types**
  - Technical components
  - Business components
2.1 Object Orientation

One attribute of component software centers on how object-oriented it is. The notion that encapsulation is required for component-based development is commonly agreed. Encapsulated software objects limit interaction to clearly defined public interfaces. An object’s behavior cannot be changed except through the proper interface calls. There are differences over the issue of inheritance and to a lesser degree on polymorphism when applied to components. Polymorphism is a characteristic of objects that can provide different functionality (methods) depending on the messages sent to them. Inheritance is the property that allows child objects to be created from parent objects, with the child object inheriting functionality from the parent. However support for inheritance and polymorphism is not critical for the definition of components.

2.2 Granularity

Components can be classified based on their size relative to the systems they are building. Fine-grained components are typically small in size and have the applicability across a wide range of application types. Fine-grained components usually take the form of desktop-bound visual controls. In contrast, large-grained components, as their name implies, are larger in size, but necessarily more limited in their applicability across a broad range of application types. Large-grained components are typically closely related to an organization’s business model, and are much more server-centric. The term
“business object” is often used to describe large-grained, server-centric components that map to common business entities and processes.

2.3 Component Types

Components can also be characterized according to their function within applications. Some components support the technical infrastructure of a given system. User interface controls or inter-application communication components provide examples of this type of component. Other components, which relate to an organization’s business model, are more tightly aligned to the workings of the business.

2.4 Visual/Non-Visual

Visual components are software components that have a visual representation that requires physical space on the display surface of a parent application. Parent applications are sometimes more generally referred to as containers. The motivation for a component to be Non-Visual is so it can be used to build server applications.

To sum up, a broader, more inclusive definition of software components is software objects providing some type of known service, or specifications capable of creating such objects, that can be used in combination with other components to build systems via a well defined interface [2].

A correct and stable component system results when its components are placed in architecture that accomplishes the goals of the system under development and they work reliably in the target environment.

A software component architecture enables the building of applications by assembling pre-existing software components. Software component architectures are also referred to as Software component models, Component architectures, and Component frameworks.
A component model refers to a suite of development tools, a library of specific components and an architecture that supports a general concept of operation.

Three models dominate discussion and development at present

- COM/DCOM from Microsoft
- JavaBeans and Enterprise JavaBeans from Sun
- CORBA from Object Management Group

These models are discussed in detail in the next chapter.
Chapter 3. Component Architectures

Component software requires an architecture through which any developer or vendor can deliver a component at any time and have that component become immediately useful to applications on any given system. To be used successfully, component software requires that applications always check on what components exist when they need them, instead of assuming there is only a limited set. When a new component is added to the system, it should become instantly available to all applications, even those that are already running. A system that supports component software must therefore support a generic "service abstraction"—that is, an architecture that defines how all types of components appear and how they are manipulated. In addition, the architecture must be extensible, so that a new component category (as opposed to an implementation of an existing type) can be introduced without having to revise the architecture. This is the problem of creating an extensible service architecture. In other words, the architecture must expect that new component types, or categories, will be defined later on. The other big problem that such an architecture must solve is that of "versioning." It turns out that the first definition of a component type is easy to manage, as is the first implementation of any particular component. The difficulty comes in managing revisions to the designs and the implementations over time [1].

A component model refers to a suite of development tools, a library of specific components and an architecture that supports a general concept of operation.

Three models dominate discussion and development at present

- COM/DCOM from Microsoft
- JavaBeans and Enterprise JavaBeans from Sun
• CORBA from Object Management Group

3.1 Component Object Model

COM is a component architecture from Microsoft. The Component Object Model provides a means to address problems of application complexity and evolution of functionality over time. The Component Object Model is an object based programming model designed to promote software interoperability; that is, to allow two or more applications or components to easily cooperate with one another, even if they are written by different vendors at different times, in different programming languages, or if they are running on different computers running different operating systems. To support its interoperability features, COM defines and implements mechanisms that allow applications to connect to each other as software objects. In other words, COM, like a traditional system service API, provides the operations through which a client of some service can connect to multiple providers of that service in a polymorphic fashion. But once a connection is established, COM drops out of the picture. COM serves to connect a client and an object, but once that connection is established, the client and object communicate directly without having to suffer overhead of being forced through a central piece of API code.

COM is not a prescribed way to structure an application; rather, it is a set of technologies for building robust groups of services in both systems and applications such that the services and the clients of those services can evolve over time. In this way, COM is a technology that makes the programming, use, and uncoordinated/independent evolution of binary objects possible. COM is not a technology designed primarily for making
programming necessarily easy; indeed, some of the difficult requirements that COM accepts and meets necessarily involve some degree of complexity. However, COM provides a ready base for extensions oriented towards increased ease-of-use, as well as a great basis for powerful, easy development environments, language specific improvements to provide better language integration, and pre-packaged functionality within the context of application frameworks [3] [4].

This is a fundamental strength of COM over other proposed object models: COM solves the "deployment problem", the versioning and evolution problem where it is necessary that the functionality of objects can incrementally evolve or change without the need to simultaneously and in lockstep evolve or change all existing clients of the object. Objects and services can easily continue to support the interfaces through which they communicated with older clients as well as provide new and better interfaces through which they communicate with newer clients.

To solve the versioning problems as well as providing connection services without undue overhead, the Component Object Model builds a foundation that:

- Defines a binary standard for interoperability
- Performs versioning
- Is language independent
- Provides distributed capabilities

**Interoperability**

COM enables calls to be made between components by specifying a binary standard for such calls. The standard specifies how a component constructs the v-tables for its interfaces, and how the client calls interface functions indirectly, through the v-tables.
COM’s binary standard allows clients to safely call a component's interface methods, without concern for the component's implementation language.

- **Versioning**

A good versioning mechanism allows one system component to be updated without requiring updates to all the other components in the system. Versioning in COM is implemented using interfaces. The COM design completely eliminates the need for things like version repositories or central management of component versions. When a software module is updated, it is generally to add new functionality, or to improve existing functionality. In COM, you add new functionality to your COM component by adding support for new interfaces. Because the existing interfaces don't change, other components that rely on those interfaces continue to work. Newer components that know about the new interfaces can use those newly exposed interfaces.

- **Language Independent**

COM is language independent. By specifying a binary object standard (as opposed to a source code standard, such as C++), COM enables calls to be made between components written in different programming languages (C, C++, Small Talk, Java and so on). Components can be implemented in any language that supports the creation of v-tables. They can be called from any language that supports indirect calls through function pointers. Typically, the various components in the system have been written in many different languages.

- **Distributed Capabilities**

The binary standard allows COM to intercept an interface call to an object and make instead a remote procedure call (RPC) to an object that is running in another process or
on another machine. A key point is that the caller makes this call exactly as it would for an object in the same process. The binary standard enables COM to perform inter-process and cross-network function calls transparently.

Although there is, of course, more overhead in making a remote procedure call, no special code is necessary in the client to differentiate an in-process object from an out-of-process object. This means that as long as the client is written from the start to handle RPC exceptions, all objects (in-process, cross-process, and remote) are available to clients in a uniform, transparent fashion.

The Component Object Model defines several fundamental concepts that provide the model's structural underpinnings. These include:

- A binary standard for function calling between components.
- A provision for strongly typed groupings of functions into interfaces.
- A base interface providing:
  - A way for components to dynamically discover the interfaces implemented by other components.
  - Reference counting to allow components to track their own lifetime and delete themselves when appropriate.
- A mechanism to identify components and their interfaces uniquely, worldwide.
- A "component loader" to set up component interactions and, additionally (in the cross-process and cross-network cases), to help manage component interactions.

In this context we need to mention some architectural frameworks built on top of COM like OLE and ActiveX.
OLE is an architectural framework, built on top of the Component Object Model, that supplies a set of object-based services to clients. OLE is comprised from several distinct technologies that use each other's services to provide an object-enabling system. OLE controls are OLE based components. This can be summed up as follows: COM is the specification. OLE is a set of services supplied by objects that conform to the specification.

ActiveX controls is a technology based on OLE, with its primary target originally Internet and intranet platforms. The ActiveX control specification is a lot more flexible than the OLE control specification, and ActiveX controls are not required to implement all the interfaces OLE controls have to implement. Therefore, ActiveX controls can be smaller, faster, and more efficient than OLE controls.

3.2 Distributed Component Object Model

Distributed COM is an extension to COM that allows network-based component interaction. While COM processes can run on the same machine but in different address spaces, the DCOM extension allows processes to be spread across a network. With DCOM, components operating on a variety of platforms can interact, as long as DCOM is available within the environment.

3.3 JavaBeans Component Model

JavaBeans is the platform-neutral, component architecture for the Java application environment. It is the ideal choice for developing or assembling network-aware solutions for heterogeneous hardware and operating system environments--within the enterprise or across the Internet. JavaBeans extends Java's "Write Once, Run Anywhere" capability to "Write once, run anywhere, reuse everywhere". In fact, JavaBeans takes interoperability a
A well developed software component technology should fully encourage code to be written once and not require rewrites to add or improve functionality. Adhering to the promise of "write once", JavaBeans provides a practical means of adding and improving functionality in an existing code base without reworking the original code. The goal of writing JavaBeans once, in addition to making sense in terms of development resources, also makes perfect sense in terms of version control. This encourages developers to incrementally make changes to components instead of rewriting significant portions from scratch. The result is a steady progression of functionality, which in turn dictates a more consistent evolution of a component through increasing versions [5].

The statement "run anywhere" boils down to the requirement that JavaBeans components to be cross-platform. For JavaBeans cross-platform support comes easily because it is based on Java. The "run anywhere" statement doesn't just refer to JavaBeans components executing on different platforms, however, but also to execute across distributed network environments.

The statement "reuse everywhere" refers to the capability for JavaBeans to be reused in many different scenarios including other applications, other components, documents, web sites and application builder tools. This meets the primary goal of software components in general, which is code reuse.

The primary design goals for JavaBeans are summarized by the following list of requirements for JavaBeans components:

- Compact and easy to create and use
• Fully portable
• Built on the inherent strength of Java
• Support flexible design-time component editors
• Leverage robust distributed computing mechanisms

The above requirements have been met in the following ways

• **Simple and Compact**

The first requirement of JavaBeans to be very compact is based on the fact that JavaBeans components often will be used in distributed environments in which entire components might be transferred across a low-bandwidth Internet connection. Clearly, components must be as compact as possible to facilitate a reasonable transfer time. The second part of this goal relates to the ease with which the components are built and used. Imagining components that are easy to use is not such a stretch, but creating a component architecture that makes building components easy is a different issue altogether.

JavaBeans components are based largely on the class structure already in use with traditional Java programming. Even though the goal is for JavaBeans components to be as compact as possible, this is no way a limitation toward creating complex and potentially bulkier components should the need arise.

• **Portable**

The JavaBeans API coupled with platform-independent Java system on which it is based altogether comprise the platform-independent solution. The end result will be reusable components that work on all platforms.
• **Leveraging Java Strengths**

The existing Java architecture already offers a wide range of benefits easily applied to components. One of the important features of Java is its built-in class discovery mechanism, which enables objects to interact with each other dynamically at run time. This results in a system in which objects can be integrated with each other independently of their respective origins or development history. This is the big advantage for JavaBeans as this functionality is already provided whereas other component architectures have had to implement complex mechanisms to achieve the same result. Another example of JavaBeans inheriting Java functionality is persistence, which is the capability for an object to store and retrieve its internal state. Persistence is handled automatically in JavaBeans by way of the serialization mechanism already present in Java.

• **Application Builder Support**

Another design goal of JavaBeans relates to design-time issues and how developers build applications using JavaBeans components. The JavaBeans architecture includes support for specifying design-time properties and editing mechanisms to facilitate visual editing of JavaBeans components. The result is that developers are able to use visual application builder tools to assemble and modify JavaBeans components in a seamless fashion. This feature alone will usher in the use of professional application builder tools and significantly boost the productivity of application developers.

• **Distributed Computing Support**

Although it is not a core element of JavaBeans architecture, support for distributed computing is a major issue with JavaBeans. Because distributed computing requires
relatively complex solutions attributed to the complex nature of distributed systems, JavaBeans leverages the use of external distributed approaches based on need.

3.4 Enterprise JavaBeans

In this context the new Enterprise JavaBeans component architecture needs to be mentioned. Enterprise JavaBeans is a component architecture for creating scalable, multitier, distributed applications, and it makes possible the creation of dynamically-extensible application servers. The basic idea behind Enterprise JavaBeans is to provide a framework for components that may be "plugged in" to a server, thereby extending that server's functionality. Enterprise JavaBeans is similar to the original JavaBeans only in that it uses some similar concepts.

The goals for Enterprise JavaBeans are as follows

- EJB is designed to make it easy for developers to create applications, freeing them from low-level system details of managing transactions, threads, load balancing, and so on. Application developers can concentrate on business logic and leave the details of managing the data processing to the framework. For specialized applications, though, it's always possible to get "under the hood" and customize these lower-level services.

- The EJB Spec defines the major structures of the EJB framework, and then specifically defines the contracts between them. The responsibilities of the client, the server, and the individual components are all clearly spelled out. A developer creating an Enterprise JavaBean component has a very different role from
someone creating an EJB-compliant server, and the specification describes the responsibilities of each.

- EJB aims to be the standard way for client/server applications to be built in the Java language. Just as the original JavaBeans (or Delphi components, or whatever) from different vendors can be combined to produce a custom client, EJB server components from different vendors can be combined to produce a custom server. EJB components, being Java classes, will of course run in any EJB-compliant server without recompilation. This is a benefit that platform-specific solutions can't hope to offer.

- Finally, the EJB is compatible with and uses other Java APIs, can interoperate with non-Java apps, and is compatible with CORBA [6].

### 3.5 CORBA

CORBA specifies a system which provides interoperability between objects in a heterogeneous, distributed environment and in a way transparent to the programmer. Its design is based on OMG Object Model.

#### 3.5.1 The OMG Object Model

The OMG Object Model defines common object semantics for specifying the externally visible characteristics of objects in a standard and implementation-independent way. In this model clients request services from objects (which will also be called servers) through a well-defined interface. This interface is specified in OMG IDL (Interface Definition Language). A client accesses an object by issuing a request to the object. The request is an event, and it carries information including an operation, the object reference
of the service provider, and actual parameters (if any). The object reference is an object
name that defines an object reliably.

The components of CORBA are as follows:

- **Object Request Broker**

  The most important piece of the Object Management Architecture is the ORB. The ORB
  is the only portion of CORBA that must be present in order to build a CORBA-compliant
  application. Many ORBs are shipped without any of the CORBA Services or CORBA
  Facilities, and you must create (or purchase) the Business Objects yourself. However,
  without the ORB, a CORBA application cannot function.

  The most visible function of a CORBA ORB is to respond to requests from your
  application or from another ORB. During the life-cycle of your running CORBA
  application, your ORB may be asked to do many different things, including:

  - Look up and instantiate objects on remote machines
  - Marshal parameters from one programming language (such as C++) to another
    language (such as Java)
  - Handle security across your machine's local boundary
  - Retrieve and publish metadata on objects on the local system for another ORB
  - Invoke methods on a remote object using static method invocation described by a
    downloaded stub
  - Invoke methods on a remote object using dynamic method invocation
  - Automatically start objects that aren't currently up and running
  - Route callback methods to the appropriate local object that it is managing
The great advantage about the ORB is that nearly all of the implementation details for all of these duties are hidden from the software developer. Simply providing the appropriate "hooks" in your code to initialize the ORB and register your application with the ORB opens your application up to a vast galaxy of distributed objects.

- **Internet Inter-ORB Protocol (IIOP)**

IIOP is the protocol that ORBs use to communicate over TCP/IP networks. IIOP implementation is required for an ORB to be fully compliant with CORBA.

- **Interface Description Language (IDL)**

IDL is used to specify the interface between the client ORB and the server ORB. The IDL interface specification is then compiled via an IDL-to-implementation language compiler to produce stub and a skeleton code. Ideally, the IDL will enable any ORB to communicate with other vendors' ORBs. In practice, this isn't always the case, but between language-specific versions of a single vendor's ORB the vendor's IDL can be expected to be 100 percent compatible.

- **Business Objects**

Business Objects are what you as the applications developer produce. OMG defines business objects as high-level representations of concepts that exist in a business domain [7].

**3.6 Need for Interoperability**

There are lots of differences between these architectures, thus making the interoperability between components very difficult. Any company, which has invested in one of these technologies, has to spend lot of effort and time to redirect their software toward a different technology. There is a necessity of migration tools, which will assist the
developers in moving their effort from one technology to other with a minimum effort.

Some of the tools are Sun's ActiveX Bridge for JavaBeans, which allows JavaBeans to be converted to ActiveX controls, IBM's Migration Assistant to convert ActiveX controls to JavaBeans and Visual Edge's Object Bridge for COM/DCOM-CORBA communication. The next chapter will focus on the tool developed by me to migrate ActiveX controls to JavaBeans.
Chapter 4. Migration Assistant for ActiveX controls

JavaBeans is quickly becoming the component technology of choice for software developers as they move from a focus on desktop OS specific applications to solutions based on the industry standard network-computing environment - Java. With its open architecture, platform independence, network affinity, and its natural extension of existing Java skills, JavaBeans has gained momentum as developers are delivering JavaBean components for use in application assembly tools and e-business application solutions.

For many corporations and developers, their current component investment has been directed at the proprietary, desktop optimized ActiveX (formerly OCX) technology on Windows platforms. With the networked computing solutions, IS organizations and independent software developers must be careful not to develop components that are limited to specific operating systems, which is counter to the basic principle of choice in network computing solutions. Therefore, developers are moving their desktop component investments to the network, using JavaBeans, to leverage the opportunities offered by e-business.

In addition, many of the capabilities that are so exciting to JavaBeans developers are not available on ActiveX, or are incomplete. There is, therefore, a need for a set of conversion conventions and tools designed to convert desktop ActiveX components into JavaBeans components. The resulting JavaBeans will be usable in both in new network savvy e-business applications as well as traditional desktop applications, including ActiveX containers. Developers and customers benefit from the advantages offered by JavaBeans by simply leveraging their current investments in ActiveX [8].
The JavaBeans Migration Assistant for ActiveX contains an automated tool that uses the ActiveX information, available in the registry or a disk file, to create a set of methods, properties and enums that an equivalent JavaBean would implement to achieve the same function. Data types are mapped from Windows-specific types to their equivalent Java types.

The tool wouldn't have been possible without the use of Java Native Interface (JNI)[9].

4.1 Java Native Interface

The JNI is a native programming interface. It allows Java code that runs inside a Java Virtual Machine (VM) to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly [10].

Programming to Java Native Interface involves

- Identifying the work that has to be native

- Creating a Java class which acts as a wrapper to access the native methods. The class should contain empty methods identified by the keyword 'native'.

- Generating a header file for the Java class so that the functions can be implemented on the native side. This is done using 'javah' tool supplied with JDK.

- The native files should include the header file generated by 'javah' tool and give implementation for the methods defined in the header file.

- Building a Dynamic Linked Library (DLL) from the compiled and linked native files

- Loading the Dynamic Linked Library in the Java program so that the native functions are accessible to Java programs.
Native methods are loaded with the System.loadLibrary method. We may use a single library to store all the native methods needed by any number of classes, as long as these classes are to be loaded with the same class loader. The VM internally maintains a list of loaded native libraries for each class loader. But care should be taken that native library names are chosen in a way that minimizes the chance of name clashes.

If the underlying operating system does not support dynamic linking, all native methods must be prelinked with the VM. In this case, the VM completes the System.loadLibrary call without actually loading the library. The programmer can also call the JNI function RegisterNatives() to register the native methods associated with a class. The RegisterNatives() function is particularly useful with statically linked functions [11].

The JNI offers a set of standard interface functions. Using these interface functions, you can call JNI functions from your native method code to do such things as access and manipulate Java objects, release Java objects, create new objects, call Java methods, and so on.

The tool is mainly composed of three components

- Registry Viewer
- Control Viewer
- Code Generator

4.2 Registry Viewer

Every ActiveX control is registered in the registry of the machine. Some introduction has to be given to the Registry so that it's easy to know how to identify ActiveX controls from such a large repository of information.
4.2.1 Registry

The system registry is the Windows 95 and Windows NT central data repository. Sometimes referred to as the registration database, it is the primary configuration database for application, user and computer-specific information. The registry is implemented as a hierarchical structure, much like a directory or structured storage tree. The hierarchy is based on elements called 'keys'. Keys contain other keys, called 'subkeys'. On 32-bit platforms there are four pre-defined keys that serve as the root keys for the entire registry. All application and system-defined keys are subkeys of at least one of the root keys [12].

The root keys are named:

- HKEY_CLASSES_ROOT
- HKEY_LOCAL_MACHINE
- HKEY_USERS
- HKEY_CURRENT_USER

All the information about the COM objects including ActiveX controls are stored under HKEY_CLASSES_ROOT, in the predefined key CLSID. The CLSID is the key where COM CLSIDs are stored. Each class ID is a 16-byte number that identifies a distinct COM server object. The registry entry for an ActiveX control is slightly different and has to provide some mandatory keys. The subkeys are

- Control (Optional)
  
  ActiveX controls put the Control key in the registry to identify themselves as ActiveX controls. This attribute is used by the Registry Viewer to get a list of all the controls available in the registry
• InProcServer32 (Mandatory)

The InProcServer32 points to the control's .OCX file.

• Insertable (Optional)

The Insertable key is included on all objects that can be embedded in a compound document.

• ProgID (Optional)

ProgID provides a useful, human-readable name for the control. The value of the ProgID key is used by the Control Viewer to create an instance of the COM object and get access to its methods and variables.

• TypeLib (Mandatory)

TypeLib points to the control's type library. A type library is a group of type information structures and the means to navigate through those structures. Each and every control has a type library associated with it. Type libraries contain the descriptions of object classes, interfaces, dispatch interfaces and typedefs. This Type library is used by the Control Viewer to get information about the enumeration objects and events supported by the control.

This ends our discussion about the registry.

Viewing the controls is handled by the ControlViewer class, which has a native method that gets the information about the controls. The native method goes through the registry and gets the CLSID, ProgID, InProcServer32, Version Free ProgID, TypeLib and name attributes of the control. The information about the control is stored in a RegisteredControl object. The native method returns an array of RegisteredControl objects. The RegistryViewer gets the array and displays the controls in an hierarchical
structure. The structure of the RegistryViewer component can be shown in the following figure.

![Diagram of the structure of the RegistryViewer component]

**Figure 4.1 Structure of Control Viewer**

The controls are sorted by their name and presented to the user in a tree. When the user selects a control the information about the control will be displayed in a side panel. The RegistryViewer also provides the option of running the RegistryEditor program from its menu and toolbar so that the user can see the CLSID's of the controls being presented. If the user wants to browse a control to see the properties, methods and events supported by it, he has to double click on the tree node representing a control. This brings up the next component of the Migration Tool i.e. the ControlViewer.
4.3 Control Viewer

To understand the functioning of the Control Viewer we need to know about the characteristics of an ActiveX control as to how it exposes its properties, methods and events. Every ActiveX control is a COM object, so it implements the COM's primary interface IUnknown. The IUnknown is used for two things.

- **Life Cycle Management**
  
  IUnknown controls a COM server's lifetime in memory.

- **Interface Access**
  
  IUnknown knows about all of the other interfaces the server implements. It also grants client access to the other interfaces.

Any client wishing to get to know about the ActiveX control needs to get a pointer to the IUnknown interface of the control. Through this pointer he can query about the other interfaces that are being supported by the control. To get a pointer to the IUnknown interface the COM object needs to be instantiated. We use the ProgID of the control to call the CoCreateInstance method to instantiate the COM object and get the pointer to the IUnknown interface.

OLE Automation is the mechanism that OLE controls or ActiveX controls use to expose their properties, methods and events to the outside world. To do this in a uniform way, OLE controls implement the standard interface IDispatch. By invoking the method QueryInterface through the IUnknown pointer on the control we get the pointer to the IDispatch interface. In addition to providing property and method access, IDispatch also provides property and method type information. This information includes property and method names, ID numbers, the name and type of all method parameters, method return
types, etc. To access this information we call GetTypeInfo method on the IDispatch. This method returns a type information object that the controller uses to obtain type data [13].

**4.3.1 ITypeInfo interface**

This is the main interface that provides all the information about the properties and methods of the control apart from giving access to the TypeLibrary in which it is located. The TypeLibrary information can be used to know about the events and enumeration objects supported by the control. The ITypeInfo interface provides the following important functions

- **GetContainingTypeLib**
  Retrieves the containing type library and the index of the type description within that type library.

- **GetTypeAttr**
  Retrieves the TYPEATTR structure that contains the attributes of the type description like the number of methods and variables.

- **GetFuncDesc**
  Retrieves the FUNCDESC structure that contains information about a specified function.

- **GetVarDesc**
  Retrieves a VARDESC structure that describes the specified variable.

- **GetNames**
  This returns the names of the formal parameters of a function or the name of the variable.
GetDocumentation

Retrieves the documentation string, the complete Help file name and path, and the context ID for the Help topic for a specified type description.

Since these functions are implemented in C++ care should be taken to release memory when we are done with the functions. ITypeInfo interface provides three functions that do the job for us.

- ReleaseTypeAttr
  Releases a TYPEATTR previously returned by GetTypeAttr.

- ReleaseFuncDesc
  Releases a FUNCDESC previously returned by GetFuncDesc

- ReleaseVarDesc
  Releases a VARDESC previously returned by GetVarDesc

Using the above functions we can get all the information about the control. There are various other structures used by the ITypeInfo interface to give access to information like the type of a method, whether it is a 'set' or a 'get' or a simple 'invoke' method. The types of the parameters that has to used when calling the functions whether it be a integer or a float or a String or any other object.

We created wrappers for the above mentioned interfaces and their methods and using Java Native Interface these methods were implemented using C++ and the results were passed between Java and C++. The overall structure of the classes used and their relationship in the ControlViewer is given
Figure 4.2 Structure of OLEControlViewer
When the user selects a node in the ControlViewer corresponding to a control, we instantiate a new OLEControlViewer class and pass the ProgID of the control to the class. The ProgID is used to locate the control and get the CLSID of the control. The OLEControlViewer class instantiates a OLEBrowser class with the ProgID. The main function of the OLEBrowser class is to locate the CLSID of the control and create the COM object and get pointers to the IUnknown and IDispatch interfaces. OLEBrowser uses JavaOLE and JavaGUID classes to get the pointers to IUnknown and IDispatch classes. Using these pointers, we call the function GetTypeInfo on IDispatch interface to retrieve the type info object.

Once we get the TypeInfo object we get its TypeLibrary. From the TypeLibrary, using the functions GetTypeInfoCount and GetTypeInfo we can get access to all the other TypeInfo objects in the library. The GetTypeInfo function returns a TypeInfo object and from that we can get the TYPEKIND structure that tells us about the TypeInfo objects i.e. whether it is an Enumeration type or a Dispatch Interface type. Currently we are identifying the events by reading the name of the TypeInfo object and checking whether it contains the keyword 'Event'. If the TypeInfo object is an enumeration type the OLEControlViewer assigns the job of introspecting it to HandleEnumeration object. If the object is an 'event' type it is handled by HandleEvent object. If the object is a dispatch type it is handled by HandleDispatch object. These objects use the various functions in the ComITypeInfo class to introspect the TypeInfo objects and get to know about the properties, methods and any default values of the variables and the parameters of the functions, their types and return values.
The OLEControlViewer displays the results to the user classified in a tree structure with Methods/Properties, Events and Enumeration type as nodes.

An OLEControlTreeNode represents each leaf node under the three categories. If the node is under properties/methods it contains a list of the variables if there and complete signatures of functions with the return type, function name, names of formal parameters and types of the parameters. If the node is under 'Events' it contains the methods with the return type, names of formal parameters and types of the parameters. If the node is under 'Enumeration' it contains all the enumeration constants their types and their values.

When the user selects a node all the values will be displayed on a display panel.

4.4 Code Generator

Once the user is done with viewing all the methods/properties, enumeration objects and events he can select the 'Generate' button the toolbar to generate Java code for the control. The code generation typically involves copying all the functions and variables to a Java file and taking care of the type mappings between Java and C++. The code generated is only a skeleton giving the outline of the functions and the user has to fill up the methods with the code from the ActiveX control. As Java and C++ code are very similar the user has to make only minimal changes when copying the code from C++ to Java [14].

The main TypeInfo object will be the main Java class that extends the java.awt.Component class. All the methods and properties are copied to the Java class directly. For the properties to adhere to the JavaBeans component model 'set' and 'get' methods are added which give access to the properties. Every enumeration object is placed in a separate Java file. The enumeration object is similar to an interface in Java.
The constants with their types and values are copied to the Java file. While every attempt has been made to avoid collisions with methods defined in base classes, generated methods may collide with methods already defined in Java base classes such as java.awt.Component or EventObject that have different signatures. If this happens the user has to change the name of the generated method. Stock events in the control are mapped to stock events in the Component class.

A sample code will look like this

```java
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.beans.*;

// --------------------------------------------------------------------------------------
// CCounter
// Dispatch interface for CoolCounter Control
// --------------------------------------------------------------------------------------
public class CCounter extends Component
{
    public void Add( int HowMuch )
    {
    }
    public void Subtract( int HowMuch )
    {
    }
    public void AboutBox( )
    {
    }
    // Getters
    public int getDigitCount() { return m_DigitCount; }
    public int getValue() { return m_Value; }
    public int get_Value() { return m__Value; }
    //Setters
    public void setDigitCount( int DigitCount )
    {
        m_DigitCount = DigitCount;
    }
    public void setValue( int Value )
    {
        m_Value = Value;
    }
```
public void set_Value( int _Value )
{
    m__Value = _Value;
}

// Member variables
private int m_DigitCount;
private int m_Value;
private int m__Value;

This is just a part of the code. The code shows the methods that are used by the control and the properties exposed by it. Set and Get methods are provided for the properties to adhere to JavaBeans component model. The user has to fill the body of the methods from the ActiveX control code. As most of the controls are written in C++, which is similar to Java the user need to make only minor changes. The code is to be compiled and the class files are to be grouped in a Jar file.
Chapter 5. JavaBeans for building GUI Applications

Chicory™ is an Integrated Software Application Development Environment that supports the entire software application development lifecycle. The design philosophy behind Chicory™ is to support multi-paradigmatic programming using components and component frameworks. It not only supports visual programming metaphors to assemble applications using components but also provides frameworks and their application programming interfaces (APIs) to enable the application programmer to customize the framework to suit his/her application. Using these components the user can build a Graphical User Interface in a relatively short time [15].

The visual components used in Chicory™ are JavaBeans, which extend the functionality provided by the JFC Swing components.

Building a GUI in Chicory™ involves the following steps

1. Selecting the container
   
   The container can be a Frame, Panel or an Applet

2. Drag and Drop the GUI components in the container

3. Change the properties of the beans to suit the needs of the application
   
   The properties can be visual like Color, Font etc. or non-visual like name etc.

4. Generate the Java code for the application

   The components were developed using Swing classes. The advantage of Swing components is their uniform look and feel on all platforms. The look and feel is a property so it can be changed to reflect the user's needs on various platforms. This allows to have a Windows Look and Feel for a component on a Motif platform.
A wide range of components to suit the needs of the users has been developed by Chicory team. A brief description of the various components and how they can be used is given as follows

- **ChicoryButton**
  
The Button is a simple control that generates an action when the user clicks on it.

- **ChicoryCheckBox**
  
The CheckBox has only two states 'on' or 'off'. When the user clicks on a checkbox its state changes and generates an action event.

- **ChicoryRadioButton**
  
The RadioButton has also two states 'on' or 'off'. When the user clicks on a radio button its state changes and generates an action event.

- **ChicoryToggleButton**
  
The ToggleButton works like a Button that stays depressed when toggled on. When toggled off there is no difference between a ToggleButton and an ordinary Button.

- **ChicoryLabel**
  
A Label provides an easy way of putting an unselectable text in a GUI.

- **ChicoryProgressBar**
  
The ProgressBar component allows the user to display the status of an operation.

- **ChicoryScrollBar**
  
The ScrollBar can be used as a control for ScrollPanes. ScrollBars allow the user to choose exactly which part of the region is visible.
• **ChicorySlider**
  A slider works like a scrollbar; however it adds the ability to display major and minor tick marks.

• **ChicoryStatusBar**
  The StatusBar is an extension of the Label except that its border is different and it appears to be raised. The user can set the text of the StatusBar which indicates the state of an operation.

• **ChicoryList**
  The List component provides an area containing selectable text items (one per line). Lists can allow single selection or multiple selections at a time. When the user selects an item the List generates an action event.

• **ChicoryComboBox**
  The ComboBox provides a menu-like list of choices accessed by a distinctive button. The user presses the button to bring up the "menu", and then chooses one of the items. When the user chooses an item, the ComboBox generates an action event.

• **ChicoryTextField**
  A TextField displays a selectable and editable text in a single line.

• **ChicoryPasswordField**
  A PasswordField is similar to a TextField only that the text visible or the text input by the user is not displayed openly. The text is masked by '*' character. This component can be used when the users are entering their passwords.

• **ChicoryTextArea**
  A TextArea displays a selectable and editable text in multiple lines.
- ChicoryTree
  The Tree component can be used to represent hierarchical data. The Tree has to be placed in a ScrollPane to completely view all the elements of the tree.

- ChicoryTable
  The Table component can be used to present large amount of data in rows and columns. The Table has to be placed in a ScrollPane to completely view all the rows and columns.

All these components are provided in the ToolPallette. The user has to click on an icon representing the component and click on the GraphicDesigner. The GraphicDesigner is the place where the user drops the components. Once the component is dropped the properties corresponding to the component are displayed in the PropertySheet. The user can manipulate the properties and if they are visual the changes can be seen immediately in the GraphicDesigner. From the PropertySheet the user can also link the events generated by the components to various methods in the other components. After the GUI is made the user can generate the code for the GUI.
Chapter 6. Database Components

The advantage of Database Components is that they will provide information from the database with a minimal amount of coding. This approach will enable the users to concentrate on the business logic of the application and also speed the development of the application.

As part of Chicory database application development the database components have been developed so that the user can assemble a Form in a short time. The Database Components are just like GUI components except that they are database aware, in other words the components have some extra properties like the column in a database table they are attached to, the type of the column, the URL of the database [16].

All these properties are placed in a DBComponentIfc interface. The interface also contains some other methods like editable, data, maxvalue, minvalue and precision value.

To be a database component the above mentioned interface has to be implemented.

The various components developed are

- **DBCheckBoxBean**
  
  This component will enable two different values to be displayed. This component will make binary information more informative to the user.

- **DBComboBoxBean**
  
  This component will provide the user with way to show a selection of choices. The component provides a menu-like list of choices accessed by a distinctive button. The user presses the button to bring up the "menu", and then chooses one of the items.
- **DBListBean**

  This component is similar to the ComboBox bean except that it presents all the data to the user and he can choose any one of them.

- **DBLabelBean**

  This component presents the data in an uneditable format in other way it's read-only.

- **DBTextFieldBean**

  This component is the widely used one to present data. The user can edit the data in the component.

- **DBTextAreaBean**

  This component can used to present a large amount of data that can't fit in a single line.

- **DataControlBean**

  This is the heart of the applications built by the database components.

### 6.1 DataControlBean

Every database application should have one DataControlBean. This bean first gets all the remaining database components in the container and queries about their columns, column types and the source they are connected to. It is important that all the components have the same source. The properties of the DataControlBean are the URL it is connected to, the table name, the driver it has to use to contact the database, the user name and the password to get the data from the table. The DataControlBean builds a SQL statement based on the column of the remaining components. The user can also set a filter criteria for the SQL statement. It instantiates the driver and connects to the database using the URL, the username and the password. It executes the SQL statement at the server and
gets back the results. All the results of the query are stored in the DataControlBean. The DataControlBean calls the setData method on each of the components and sets the respective data from the results it got from the database. The DataControlBean is a visual control and provides the following services. It has buttons that allow the user to browse through the data. It has a 'next' button to go to the next record, 'previous' button to see the previous record, last button to see the last record, 'first' button to see the first record, 'add' button which allows the user to add a new record, 'delete' button to delete a record. All the changes made to the data are instantly noted and respective UPDATE, INSERT and DELETE statements are built using the data in the components. The DataControlBean also provides two more buttons 'Rollback' and 'Commit'. Using the 'Rollback' button the user can forego all the changes that have been made and get back the old data. The 'Commit' button takes all the UPDATE, INSERT and DELETE statements and executes them at the server making the changes permanent. If there are any exceptions to the changes that are going to be made the user is informed though a message box about the probable cause of the exception [17].

6.2 Forms Wizard

Chicory provides a FormsWizard as part of its Database application development. The use of FormsWizard is to build forms very easily and very fast for ready use. To use the FormsWizard the user has to create or open a new Database Project. There are two steps in creating a new Database project

1. Login to a remote database

   The user has to supply the URL of the database, user name, password and the driver that has to be used to set up the connection.
2. Get schemas and catalog information from the database.

To create a new Form the user can select the Database menu and click on 'Create Form..' item. A Dialog box will popup asking for the schema that has to be used for connecting to the database. The various schemas for which the information has been collected will be presented to the user. The user has to select a schema. The next Dialog box gives the users the option of creating the Form using 'Design View' or 'Simple Wizard'.

6.2.1 Design View

The Design View comes up with a tool palette containing the database components, a Graphic Designer where the user can drag and drop components and format the layout of the form and a property sheet to view the properties of the various components present in the Graphic Designer. The Graphic Designer comes up with a DataControlBean whose properties have already been set using the information obtained while logging on to the database. This includes the driver name, user name, password, table name and URL. The user can set the filter criteria for the SQL statement that is going to be built. The user has to click on the component icon on the tool palette and click on the Graphic Designer. With this action a component is added to the Graphic Designer and its properties are reflected in the property sheet. The important properties of the component that has to be set are the column name, column type and URL of the database. The URL property is important as the DataControlBean uses this to identify the component and in building the SQL statement. Once all the components are added to the Graphic Designer and their properties are correctly set the user can click on the 'Start' button located on the top of the Graphic Designer. This will call a method findDBCComponents on the DataControlBean which will first find all the database components in the container, gets their column
names and their types and builds an SQL statement. If the filter criterion is set the string is concatenated to the SQL string. The DataControlBean instantiates the driver class and using the user name and password sets up connection with the database server and executes the query. The results are fetched back and stored in the DataControlBean. From these results the DataControlBean sets the data in each component with respect to their column name. The DataControlBean allows adding, deleting or modifying the records. After any change the user has the option of committing the changes or rollback the changes that have been made. The FormWizard also gives a code generation utility, which allows the user to generate Java code for the form. After generating the code it can be compiled and run as an independent program.

### 6.2.2 SimpleFormWizard

When the user selects this wizard a FieldSelector dialog box will come up which shows the columns in the table that has been selected. The user can select all the fields or some of his choice from the table and click on the 'Next' button. This brings up a tool palette, a Graphic Designer with components corresponding to the fields that have been selected in the previous step. The Graphic Designer also has the DataControlBean with its properties pre-set using the login information provided by the user while logging on to the database. The user can straight away click on the 'Start' button which will call a method findDBComponents on the DataControlBean which will first find all the database components in the container, gets their column names and their types and builds an SQL statement. If the filter criterion is set the string is concatenated to the SQL string. The DataControlBean instantiates the driver class and using the user name and password sets up connection with the database server and executes the query. The results are fetched
back and stored in the DataControlBean. From these results the DataControlBean sets the
data in each component with respect to their column name. The user can add new records
or delete records or change the existing records. Once the changes are made the user has
the option of committing all the changes or roll back the changes he has made. He can do
this through two buttons 'Commit' and 'Rollback' provided with the DataControlBean.
The Form is associated with the Database project, but if one wants to use the form
independently there is a code generation utility that generates the code for the Form. The
user has to compile and run the code to view the Form.

6.2.3 Structure

The structure of Forms Wizard in Chicory along with the classes used is given below

![Figure 6.1 Structure of Forms Wizard](image)
The Form Designer extends ChicoryTool so that it can be added to Chicory's MDI area. The FormDesigner contains the DatabaseForm, which extends GraphicDesign so that components can be dragged from ToolPalette and dropped in DatabaseForm. Whenever a new Form is created, a new node is added to the Forms node in the DatabaseManager tree. The Form node contains all the components in the DatabaseForm as its child nodes. The Form node can be clicked to get the FormDesigner to the front, and if the Form is closed a new FormDesigner is brought up. If the user right clicks on the DatabaseForm a popup menu will be displayed with the options of deleting a component or generate Java code for the form. DatabaseFormCodeGenerator class takes care of the code generation.
Chapter 7. Help System

The fast development of computer products raises a problem to both users and developers. Being a user, facing a new product, certainly one wants to learn it as quickly as possible. The most efficient way is to use the build-in help system. A good help system will make your work quicker and easier. Being a developer, you must make sure that your product is not only useful but also easy to learn. The help system deals with the process of creating a Java and HTML based system for Chicory. The help system should be able to provide an access to the user manual for Chicory and Java API files for a quick reference while building applications using Chicory. It has the following features

- A friendly user interface
- Well organized document management
- Powerful and easy to use functions

The following are the components of the help system

7.1 HTML Parser

We have two parsers. The first one is for parsing Chicory user manual documentation to build and serialize the help tree with intermediate nodes representing the chapters and leaf nodes that represent the individual topics. The second parser is to be directed to the documentation directory of the JDK and it goes through all the directories recursively and builds the Java API help files in a tree and serializes the tree. The parser also builds three lists containing the classes, interfaces and packages found in the Java documentation. These lists are loaded when Chicory is started. The lists can be viewed in the HelpContents frame. The user can select an item in the list to view the documentation pertaining to that.
7.2 Help Tree

The tree is shown in the ProjectManager window, which is accessible to the user. The tree contains two nodes. The first node 'Chicory Help' contains the user manual documentation and the second node 'Java Docs' contains the Java API documentation. The user has to select the topic he is interested in and double click on the node representing that topic. This will bring up the HelpBrowser that displays the documentation page. It's important to note that when the user changes the location of original Java documentation files he has to update the help tree by using the 'Generate Java API help files' menu item in Help menu of Chicory.

7.3 Help Contents

The help contents provide the user with the choice of viewing documentation based on classes, packages and interfaces. The user can select an item and click on 'display' button to view documentation pertaining to that item. As mentioned above if the user changes the location of Java documentation he has to run the parser again to rebuild the lists of classes, packages and interfaces.

7.4 Help Browser

The Help Browser is just like any other web browser, which displays HTML contents. The main component of the Help Browser is the JEditorPane, which allows viewing of HTML contents. JEditorPane supports hyper links, which allow the user to move from one page to the other. When the user wants to see any documentation the respective html file is displayed on the browser. The browser also keeps track of the pages that have been visited by the user so that he can switch between various documents. The browser has a
text field, which gives the current URL of the file being viewed. A new URL can be typed in to go to a different page. The users can also bookmark the pages he wants to visit frequently as it saves time in typing the full URL of the page. From the browser the user can switch back to help contents by clicking on the 'search' button [18].

7.5 Structure

The structure of Help System in Chicory is given as follows. The TreePanel that contains the HelpTree is instantiated in the ProjectManager. The ResourceManager instantiates the HelpContents and the HelpBrowser. The actionPerformed method in the TreePanel handles the mouse clicks on the nodes of the tree and brings up the HelpBrowser with the specific topic. In the same way when an item is selected in the HelpContents and action is performed on the 'display' button the HelpBrowser is brought up with the specified topic. There is a callback from HelpBrowser wherein the user can go back to the HelpContents from the browser itself. The HelpBrowser has a reference to HelpContents through ResourceManager.

![Diagram showing the structure of Help System]

Figure 7.1 Structure of Help System
Chapter 8. Conclusions and Future Work

8.1 Conclusions

This thesis explored the advantages of using components in an application development and identifies the need for migration tools to convert existing components from one architecture to other. The Migration Tool for converting ActiveX controls to JavaBeans was developed as part of this thesis is very helpful in achieving the above goal. The tool allows the ActiveX developers to easily migrate their development efforts to JavaBeans while preserving their investment in - and adding value to - any previous work based on ActiveX. The JavaBeans developed for building GUI applications and Database applications further show that component based development also fosters the rapid development of highly customizable applications.

8.2 Future Work

The tool can be modified in future to provide an option for the developer while converting ActiveX controls to JavaBeans. The option can be generating a 100% pure JavaBean or a bean that acts like a wrapper for the ActiveX control, this typically involves invoking native methods from the JavaBean.

The tool can make a CORBA server which mirrors the control and communicate with the COM objects through a bridge. Currently there is COM/DCOM - CORBA bridge offered by VisualEdge company.
Bibliography


http://msdn.microsoft.com/developer

[3] COM Specification, Microsoft Corporation,  
http://www.microsoft.com/com/comdocs.asp


[8] JavaBeans Migration Assistant for ActiveX, Sun Microsystems  
http://www.javasoft.com/beans/ActiveXtools.html


[13] Microsoft Developers Network Online Library,  
http://premium.microsoft.com/msdn/library

[14] Mark Davis, Porting C++ to Java,  


[18] Java Help Software, Sun Microsystems
   http://www.javasoft.com/products/javahelp


[22] Mark Wutka, Hacking Java: The Java Professional's Resource Kit

    Lorensen, Object Oriented Modelling and Design, Prentice Hall


VITA

Name: Ravi S. Banda
Date of Birth: 21 June, 1975
Place of Birth: Vijayawada, India

Education:

M.S in Computer Science, West Virginia University
Date of Graduation: May 1999

B.S in Mechanical Engineering, Nagarjuna University, India
Date of Graduation: May 1996
APPROVAL OF EXAMINING COMMITTEE

Dr. Sumitra Reddy

Dr. V. Jagannathan

Dr. Srinivas Kankanahalli

(Committee Chairman)

Date _______________________