A DIGITAL LIBRARY COMPONENT ASSEMBLY ENVIRONMENT

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMUPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Linda Eyambe
January 2005

Supervised by
Hussein Suleman

Abstract

Digital libraries (DLs) represent an important evolutionary step towards accessing structured digital information. DLs are often built from scratch or by using proprietary monolithic software that is often difficult to customise and extend to meet changing requirements. Researchers are beginning to realise that this is not an ideal solution and as a result, are creating component suites and accompanying protocols to encourage the creation of modular DLs. Despite the introduction of component models, it is not immediately apparent how they can be seamlessly assembled to produce diverse, yet fully functional, component-based digital library systems without knowledge of the underlying protocols.

This dissertation presents a graphical user interface and its associated framework for creating DL systems from distributed components, consequently shielding DL architects from the complexity of using components models and taking advantage of the inherent benefits of the component programming paradigm. The framework introduced in this dissertation was designed to be generic enough to be adopted for the assembly of a variety of component-based systems over and beyond the digital library community.

After being tested on over thirty inexperienced users and modelling a number of existing DL systems, graphically assembling distributed components has been shown to be a viable approach to simplify the creation of modular DLs from a pool of heterogeneous components.

Acknowledgements

There are a number of people without which this work would not have been possible. At the top of the list, I would like to thank Dr. Hussein Suleman, my supervisor, not only for his guidance, support, encouragement and constant availability throughout the course of this study, but mostly for the feedback filled with useful suggestions.

I would like to thank my father for affording me the opportunity to pursue my studies further; my friends and the rest of my family for being constant source of support.

Thanks to the members of the Advanced Information Management (AIM) lab for arming with me with useful knowledge that helped to successfully complete this work. Further thanks go to my fellow Masters students, who will be sorely missed next year, for providing an atmosphere filled with fun and laughter that made work seem less daunting.

Lastly, I wish to thank David Moore and Stephen Emslie for the contribution they made towards certain parts of this work.

I acknowledge that all references are accurately recorded and that, unless stated, all work contained herein is my own.

This work was partially supported by SA-NRF grant No. 2054030.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Digital libraries (DLs), originally termed electronic libraries, are the result of the meshing of two communities: library professionals, including librarians and publishers, and computer scientists, including their poster child, Internet developers. Until recently, these two communities had very little interaction. Even now, it is not uncommon to find librarians who know little of digitisation or computer scientists who are unfamiliar with the tools of librarianship. The introduction of the Web has brought about greater collaboration between these two very different disciplines in a quest to facilitate the access to, and dissemination of, information.

Integrating the basic ideas from conventional libraries with the opportunities presented by computers and networks has resulted in better information delivery than was present in the past. From a single personal computer, users are able to access ever-available information stored on computers around the world without leaving the comforts of their homes. It is important to note that not all online information accessed in this manner is necessarily from a digital library. The following section attempts to differentiate a digital library from electronic databases and searching facilities.

1.1 What is a Digital Library?

There exist several definitions for a digital library which are often formulated during the course of digital library research projects. The definitions are consequently influenced by the people involved in those projects, their understanding of the nature of libraries and also by the nature of those projects. Borgman (1999) suggests that the research community's definitions serve to focus attention on research problems and to expand the community of interest around those problems, while the library community's definitions focus on practical challenges involved in transforming library institutions and services. It is therefore not surprising that the term "digital library" means different things to different people but is meaningless to most. Perhaps the most comprehensive definition of a digital library which emphasise both the technical and service aspects is the one given by Gladney et al. (1994):

> *"A digital library is an assemblage of digital computing, storage, and communications machinery together with the content and software needed to reproduce, emulate, and extend the services provided by conventional libraries based on paper and other material means of collecting, cataloguing, finding, and disseminating information. A full service digital library must accomplish all essential services of traditional libraries and also exploit the well-known advantages of digital storage, searching and communication."*

From this definition, one can assume that a digital library should at the very least consist of managed and systematically organised data, replicating the fundamental ideas of conventional libraries, yet exploiting the capabilities presented by the electronic media.

1.2 Designing Digital Libraries

Rapid technological advances continue to dictate the pace at which digital libraries have been able to develop, resulting in low-cost equipment and ever-simplified DL software, ultimately enabling users from all walks of life to create and maintain their own electronic collections. However, off-the-shelf monolithic packages are often unsuitable for more complex projects, as they do not always contain the services required to meet the needs of certain user communities. Other DL development efforts involve resorting to custom-built software with intensive and time consuming design, implementation and testing cycles. The Multimedia Information System (MMIS) (IBM, 2005), designed and developed by IBM, is an example of a DL system custom-built for the specific needs of the users of the Hong Kong Central Library. An undertaking of this nature is nontrivial and resource intensive, often resulting in monolithic systems that: are built from scratch, consequently reinventing the wheel; decrease in maintainability and extensibility as the complexity of such systems rises; and take little notice of future deployment requirements.

To avoid the pitfalls associated with monolithic DLs, and because it is widely accepted as good software engineering practice to adopt some form of component model wherever possible in creating systems, several researchers have investigated a component-based approach to DL design, as can be observed from the emergence of modular DL systems such as the OpenDLib project (Castelli et al., 2004) and DSpace (Tansley et al., 2003). These DL systems, and others, will be described in later sections. Component frameworks, however, have failed to be embraced by the DL community because in several of the projects that adopted component models, the components communicate using non-standard protocols, making modification a complex process.

One of the most recent additions to the growing pool of component frameworks is the Open Digital Library (ODL) project. The ODL project was founded on the basic premise that DLs can be built as a network of components instead of as monolithic systems by relying entirely on current and emerging DL interoperability standards (Suleman, 2002). One of such DL interoperability standards, and perhaps the most widely accepted within the DL community, is the Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH) (Lagoze and Van de Sompel, 2001). This simple protocol enables heterogeneous DLs to simply and efficiently exchange metadata about digital resources.

The ODL model has been tested in several real worlds practical DLs such as those run by the Networked Digital Library of Theses and Dissertations (NDLTD) (Fox, 1999), Computer Science Teaching Centre (CSTC, 2004) and Journal on Educational Resources in Computing (Fox, 2002), and has shown promise. With the toolkit available from the ODL project, varied digital libraries can now be constructed with minimal programming experience. These components are configured by means of command-line configuration scripts, which suffices for DL researchers with simple scenarios, but is inadequate for general and widespread use.

This research investigates the use of a simple visual interface to facilitate the process of creating component-based DLs based on the principles of ODL and the OAI-PMH. Using graphical user interfaces to assemble systems is not a new concept and has been successfully applied to other domains, as later sections will demonstrate. The hypothesis is that, armed with an easy-to-use tool, ordinary users with limited technical knowledge will be able to create and manage their own modular DL systems based on current interoperability standards, with little effort. This research is therefore arising in response to a need for simplified modular digital library creation, and a method of encouraging the development of digital libraries by inexperienced users.

1.3 Motivation

In addition to the points raised above, this research is further motivated by the following factors:

- There are very few component-based DL systems on the market. Later sections review some of these systems in more depth but, in general, systems such as Greenstone (Witten, 2003) – developed as part of the New Zealand National Project which focuses on the construction and presentation of heterogeneous collections of documents – and Koha (Blake et al., 2000) offer a small finite set of customisable services, with a limited number of possible workflows.

- Existing systems are limited in their ability to construct a DL from a pool of heterogeneous components. The components often communicate using non-standard protocols or resort to assembling components in a disparate or ad-hoc manner, resulting in problems of adaptability and interoperability. The proponents of the ODL component suite and associated framework address this problem by applying its specialised version of the OAI-PMH to a wide range of components, thus improving interoperability at component level.

- The extent to which visual development environments have evolved makes it rather surprising that, as of yet, very little research has gone into creating one of such systems for digital library components. The failure to adopt the drag and drop techniques applied to popular visual programming environments such as Borland Delphi (Borland Software Corporation, 2004) and Microsoft Visual Studio (Microsoft, 2004a) to the creation of DLs can probably be attributed to the fact that component technology in the digital library discipline is still fairly new and few standard components exist.

- This research introduces a framework for creating an open visual development environment (VDE) that enables DL systems to be constructed from components that come from a wide variety of vendors, and can be easily integrated to provide a developer with an ideal solution. Open VDEs are characterised by components or tools originating from multiple vendors, but working together without requiring extensive integration efforts by the application developer. These modified components are then reusable, allowing complex and highly functional DLs to be created quickly and simply.

- A graphical user interface results in technical details of the OAI-PMH and complexities of component customisation being transparent to the user, hence lowering the bar for DL creation.

- Due to the fact that the design of the proposed system relies solely on instantiating, or using instances of, previously installed components, the level of technical competence required to get a digital library up and running is greatly reduced.

- Although the ODL approach has largely been embraced by the DL community, as can be observed from its extensive use in popular digital libraries such as NDLTD (Fox, 1999), and CSTC (2004), it is not without a few shortcomings. During user testing of the ODL components, Suleman (2002) outlined a number of issues that arose that could be eliminated with the introduction of a Graphical User Interface.

1.4 Aims

The aims of this research were:

- To enable ordinary users (non-technologists or experts) to create a digital library from a suite of components. It will simplify the way digital libraries are created, effectively placing digital libraries within the reach of non-technologists or librarians.
- To aid in eradicating, or at least greatly reducing, several problems (e.g., typographical errors) that occur during manual configuration of component-based DL systems. This painstaking manual process seems redundant especially with the proliferation of visual development environments within and beyond the Web Services development community.
- To investigate the applicability of the visual composition environment to real-world, large-scale digital libraries.
- Because the system is designed to be generic, it can be applied to other development communities, for example Web Service developers, with little modification.

1.5 Methodology

A number of components were created or modified to provide a pool of independent remotely configurable DL building blocks. Next, a test system was constructed based on a client-server architecture to permit a subset of the pool of components to be assembled graphically on the client with the resultant DL instantiated on one or more servers.

An initial evaluation was then performed by a small number of test users. Based on the results of that initial study, the system was then tweaked before undergoing a second more comprehensive testing cycle. The results of the final test were analysed and then conclusions were drawn.

Three existing digital library systems varying in complexity and requirements were then modelled to demonstrate the flexibility and real-world applicability of the graphical assembly environment.

1.6 Dissertation Organisation

Chapter 1 introduces the reader to digital libraries, the motivation for this research and its aims.
Chapter 2 presents the background to this research, with a discussion on component-based systems in general and digital library components and systems specifically.
Chapter 3 provides information on related work done simultaneously with this one by collaborators.

Chapter 4 explains the approach taken to produce a graphical user interface for assembling distributed digital library components.

Chapter 5 demonstrates how the graphical user interface was used to model three existing digital library systems.

Chapter 6 presents the evaluation methodology and the results of the user tests.

Chapter 7 presents concluding remarks and recommendations for future work.

Chapter 2

Background

2.1 Introduction

Over the years, the software development industry has seen a gradual shift from command-line to graphical user interfaces, from standalone to distributed applications and from monolithic to modular systems. This evolution can, in part, be attributed to technological advancements that manifest as faster and more reliable networks, clearly defined component interfaces and increased software usability and reusability.

The digital library arena has made several attempts at keeping abreast with the changing times. There have been efforts made towards modularising DLs such as the Dienst conceptual architecture for digital libraries (Lagoze and Davis, 1995) based on the Dienst protocol (Davis et al., 2000), and the OpenDLib project (Castelli et al., 2004). Other DL projects such as DSpace (Tansley et al., 2003) provide extensible and customisable component-based software to support the management and dissemination of digital material. Integrating the lessons learned from these projects with the basic principles of software development suggests that components are an integral part of the solution to obtaining simple, scalable and interoperable systems. Often, the existence of suitable component models alone is not sufficient to ensure that applications that use those components are easily developed and deployed. Integrated Development Environments (IDEs) such as Microsoft's Visual Studio .NET (Microsoft, 2004a) have abstracted the process of creating diverse systems from components even further by introducing highly integrated but flexible platforms that enable developers to build end-to-end business solutions that can leverage existing architectures and applications.

This chapter examines some of these IDEs after introducing the reader to component programming and the popular component models in use today. Because this research places emphasis on applying visual and component programming concepts to the design of digital library systems, there will be an overview of typical digital library components, the DL systems created from these components and DL systems in general.

2.2 Software Components

Traditionally, large blocks of handcrafted code were assembled into monolithic applications. Software reuse was obtained by linking with software libraries obtained either from third parties, or created in-house from scratch. A major disadvantage of this approach is that software boundaries are frequently not well thought out. This can lead to internal code dependencies that make the monolithic application difficult to modify and to maintain.

Components, on the other hand, are designed with standard, clearly defined interfaces that tend to protect them from changes in the software environment outside their boundaries. Applications are composed (assembled) at run-time from components selected from a component pool. Because components communicate only through well-defined interfaces, when an application needs to be modified, a single component can be edited (or exchanged for a similar component), without fear of disturbing the other components that make up the application.

The last decade has shown that object-oriented technology alone is not enough to cope with the rapidly changing requirements of present-day applications. Although object-oriented methods encourage the development of rich models that reflect the objects in the problem domain, this does not necessarily mean they yield software architectures that can be easily adapted to changing requirements. They have also been very successful for implementing and packaging components, but offer only limited abstractions for flexibly connecting components and explicitly representing architectures. Schneider and Nierstrasz (1999) suggest that visual application builders and scripting languages go a step further than object oriented frameworks since they incorporate important concepts needed for component-based application development, such as higher-level abstractions for composing components. These visual application builders generally focus on a specific application domain and offer a collection of reusable components tailored to their application domain. However, due to their restriction to specific domains, Schneider et al. argue that visual application builders and scripting languages are not flexible enough for general-purpose component-based development and lack a well-understood formal foundation.

2.2.1 Composing Components

Schneider et al. indicate that component-based software development has always been driven by an underlying component framework. A component framework offers a predefined set of reusable and plug-compatible components and defines a set of rules for how components can be instantiated, adapted and composed. In order to successfully plug components together, it is necessary that the interface of each component matches the expectations of the other components and that the "contracts" between the components are well-defined. Therefore, component based application development depends on adherence to restricted, plug-compatible interfaces and standard interaction protocols.

Garlan et al. (1995) suggest that it may be necessary to adapt the behaviour of components in order to compose them. Such adaptations are needed whenever components have to be used in systems that they have not been designed for. They warn that adaptations of this kind, however, are often nontrivial and considerable glue code may be needed to reuse components coming from different frameworks. This is because, in general, glue code is rather application specific and cannot be reused in different settings, unless well-understood glue abstractions can be used.

Piccola (Schneider et al., 1999), a small component composition language was created in an attempt to formalise the rules that govern component composition. Other research projects, such as the Vienna Component Framework (VCF) (Oberleitner et al., 2003), have directed their efforts at supporting interoperability and composability of components across different component models, a facility that is lacking in existing component models.

2.2.2 Distributed Components

Distributed computing extends object-oriented programming methods by allowing objects to be distributed across a heterogeneous network so that these distributed components interoperate as a unified whole. Components are typically designed for distribution across networks for use on multi-vendor and multi-platform computing systems and, as such, standard interfaces and communication methods are important. Component models provide the basis for inter-service communications and component integration. Web sites can offer sophisticated services for users by performing interactive tasks that involve calls by one server to multiple other servers. These multi-tiered environments allow tasks to be broken up into different services that run on different computers. Services such as application logic, information retrieval, transaction monitoring, data presentation and management may run on different computers that communicate with one another to provide end users with a seamless application interface. The client-server architecture is most commonly used for leveraging the power of distributed computing. Orfali et al. (1996) indicate that in client-server computing the client, typically a PC, provides the graphical interface, while the server provides access to shared resources such as databases. Component-like objects allow for the creation of client-server systems by assembling "live blobs of intelligence and data" (Orfali et al., 1996) in an infinite number of Lego-like arrangements. Orfali et al. predicted that this would have a huge impact on the client-server use of applications, as the distinction between what is a client and what is a server will blur and machines will be both at the same time. Seemingly, that prediction was accurate, as nowadays client-server applications have become subdivided into self-managing components that can play together and roam across networks and operating systems resulting in applications that are no longer split across client and server lines.

However, distributing applications over networks is not without challenges. In addition to the increased security risk due to the increase in potential points of attack, distributing components over networks leads to another interesting problem. In a standalone system, components run as a unit in the memory space of the same computer. If a problem occurs, the components can easily communicate information about that problem with one another. But if components are running on different computers, they need a way to communicate the results of their work or problems that have occurred, hence the need for components to adhere to the specifications of a component model with a well-defined communication infrastructure.

2.3 Component Models

Sheldon (2001) suggests that standard component models and inter-component communication architectures are critical in furthering the use of component technology on the Web. Some of the most common component models are CORBA, EJBs and Microsoft's COM. This section discusses these and other component models and technologies that are used for objects to communicate in a distributed environment.

2.3.1 ActiveX

ActiveX (not to be confused with ActiveX Controls) refers to a loosely defined set of COM-based technologies (Chappel and Linthicum, 1997). The term first grew out of Object Linking and Embedding (OLE), a technology for creating compound documents. In the next release of OLE, OLE2, the OLE architects' desire to provide a more general mechanism for allowing one piece of software to provide services to another resulted in the Component Object Model (COM)(Microsoft, 2004b). Very quickly, the term COM was used in technologies that had absolutely nothing to do with compound documents. Microsoft had a general infrastructure technology, but no brand name. Thus, ActiveX was born.

The family of COM technologies includes COM+, Distributed COM (DCOM) and ActiveX Controls (Microsoft, 2004b). COM is Microsoft's component software architecture developed primarily for Windows. It is the foundation upon which OLE and ActiveX are based, and provides a means to re-use code without requiring re-compilation. A straightforward way to think about COM is as a packaging technology, a group of conventions and supporting libraries that allows interaction between different pieces of software in a consistent, object-oriented way. In COM, a component is a platform-specific binary file that compliant applications and other components can utilise. Programs incorporating a component's services never have access to its internal data structure, but instead include pointers to its standardised interface. Thus, it is possible for components to interact with each other regardless of how they work or what languages they are written in.

COM's first incarnation assumed COM objects and their clients were running on the same machine (although they could still be in the same process or in different processes). From the beginning, however, COM's designers intended to add the capability for clients to create and access objects on other machines. DCOM is a protocol that enables software components to communicate directly over a network in a reliable, secure and efficient manner. Previously called "Network OLE," DCOM is designed for use across multiple network transports, including Internet protocols such as HTTP. DCOM is based on the Open Software Foundation's DCE-RPC specification (OSF, 1997) and works with both Java applets and ActiveX components through its use of COM.

COM+ is an enhanced version of COM that provides better security and improved performance. It is the merging of the COM and Microsoft Transaction Server (MTS) programming models with the addition of several new features. Following COM and DCOM, MTS was designed to provide server-side component services and to fix some of DCOM's deficiencies, e.g., how it handles security issues, and the complete lack of a component management and configuration environment. Microsoft (2004b) explains that developers use MTS to deploy scalable server applications built from ActiveX components, focusing on solving business problems instead of on the programming application infrastructure by combining the features of a transaction processing (TP) monitor and an object request broker into an easy-to-use package. MTS delivers the plumbing, including transactions, scalability services, connection management and point-and-click administration; providing developers with the easiest way to build and deploy scalable server applications for business and the Internet. COM+ unifies COM, DCOM and MTS into a coherent, enterprise-worthy component technology (Visokey et al., 2000).

2.3.2 CORBA

The Common Object Request Broker Architecture (CORBA) (OMG, 2004) middleware technology is a set of specifications designed to support object-oriented distributed computing in a platform and language independent manner. CORBA is similar in purpose to DCOM, but while

DCOM is a proprietary technology, CORBA was devised by an assembly of over 800 corporations in the computing industry known collectively as the Object Management Group (OMG). Language independence is possible in CORBA using the Interface Definition Language (IDL). IDL allows all objects to be described in the same manner provided there is a bridge between the native language and the IDL.

At the core of COBRA is the object request broker (ORB). ORBs are an alternative to remote procedure calls (RPCs) and message-oriented middleware, with the added benefit that the ORB itself is capable of locating other objects that can service requests. The ORB manages marshalling requests, establishes a connection to the server, sends data and executes the request on the server side. Objects simply specify a task to perform, but the location of the object that can satisfy the request is not important. The end user sees applications as being seamless, even though services and data may be coming from many places on the network. ORBs from different vendors communicate using the Internet Inter ORB Protocol (IIOP), which has been part of the CORBA specification since version 2.

The CORBA 3.0 specification defines CORBA Beans, which combines features of CORBA and Enteprise JavaBeans (with additional support for XML) into a model that can compete against the entrenched DCOM model.

Because DCOM is a Microsoft technology, little consideration is given to view points outside Microsoft, while CORBA is always open to improvements by other professionals who can input on the model. Arena (2003) suggests that one of the implications of this is that an evolving model such as CORBA seems to be dynamic changing over time while DCOM seems to be more static and unchanging over time.

CORBA and DCOM also differ with the platforms they are particularly suited to. With DCOM being created by Microsoft, the DCOM model is more designed towards distributed applications in Windows environments. CORBA on the other hands appears to be more suited for problems which involve having to deploy a distributed application across multiple platforms.

2.3.3 Enterprise JavaBeans

Enterprise JavaBeans (EJBs) (Sun Microsystems, 2004a) are discrete bodies of Java code with fields and methods to implement modules of business logic. These software components can be manipulated and integrated visually in a builder tool to create applets and applications, usually within the J2EE framework (Sun Microsystems, 2004b). Sun describes J2EE as a collection of specifications for Web Services, business objects, data access and messaging. This collection of Application Programming Interfaces (APIs) defines the way in which Web applications communicate with the servers that host them. The J2EE platform provides a component-based approach to the design, development, assembly and deployment of enterprise applications. The J2EE specification defines the following application components: application client components, EJB components, Java Servlets and JavaServer Pages (JSP) components (also called Web components), and Applets.

EJBs generally reside within a container on an application server. The implementation of an EJB consists of Java classes that are deployed in the container. Clients use an enterprise bean's home and remote interface to invoke its methods. The home interface defines methods to create or to look up component instances, while the remote interface provides access to a given instance. To interact with an EJB component, the client first obtains a reference to the bean's home interface, which the client can use to create a new component instance of the bean or to look up an existing one. Both of these operations return a reference that implements the remote interface. EJBs can implement different concepts: entity beans model business concepts that are represented in database tables; session beans model a workflow and thus implement a particular task; message-driven beans are similar to session beans but work in message-oriented middleware settings.

Unlike other component models, EJBs do not support events. Additionally, there is no standardised means to find out at runtime if a component uses a particular service provided by the EJB application server.

2.3.4 Web Services

Web Services represent an important step for building distributed applications on any operating system that supports communication over the Internet. The precise definition of a Web Service is a much-debated subject and definitions like "any piece of software that makes itself available over the Internet and uses a standardized XML messaging system" (Cerami, 2002) only help to further blur the issue. However, for most Web Services, all that is needed is a combination of XML, HTTP and a messaging protocol such as SOAP. In general, a Web Service should have a public interface defined in XML grammar describing all the methods available to clients and specifying the signature for each method. Currently, interface definition is accomplished via the Web Service Description Language (WSDL). Locating or publishing a Web Service on the Internet is most commonly achieved via Universal Description, Discovery, and Integration (UDDI). What all this leads to is the creation of distributed systems that facilitate program-to-program interaction as opposed to the traditional user-to-program interaction mode.

Clients sometimes access services using a tightly coupled, distributed computing protocol, such as DCOM, CORBA or Remote Method Invocation (RMI). Systinet (2003) explains that while these protocols are very effective for building a specific application, they limit the flexibility of the system, and do not operate effectively over the Web. Tight coupling limits the reusability of individual services, as each of the protocols is constrained by dependencies on vendor implementations, platforms, languages or data encoding schemes that severely limit interoperability. The Web Services architecture takes all the best features of the service-oriented architecture and combines it with the Web. The Web supports universal communication using loosely coupled connections. Web protocols are completely vendor-, platform- and language-independent. The resulting effect is an architecture that eliminates the usual constraints of DCOM, CORBA, or RMI. Web Services support Web-based access, easy integration and service reusability.

2.4 Integrated Development Environments

Visually-oriented mouse-driven development systems are making the esoteric work of programming accessible to the masses. In the old days (pre-nineties) the predominant way to develop programs on the PC was to run compilers or Make files from the system command line (Petzold, 1992). Since then, several Integrated Development Environments (IDEs) have not only found their way onto the desktops of programmers, but non-programmers alike because they are designed to tackle the problem of bringing computing faculties to people who do not have extensive computer training by using visual (i.e., non-linear) representations in the programming process.

Shu (1989) attributes the proliferation of IDEs to the fact that people generally think in pictures and textual programming languages have proven to be difficult for many people to learn to use effectively. Furthermore, Shu indicates that some applications are very well suited to graphical development approaches, e.g., scientific visualisation and simulation creation. Davidson (2004) explains that one of the advantages of being able to graphically assemble components is that it allows for greater understanding of requirements of a project when used during rapid prototyping since an application can be assembled right in front of the customer in a collaborative manner.

Many products are being released on the market that allows the user to graphically assemble components in order to create applications, essentially sidestepping the inherent complexities of many object-oriented programming languages. IDEs such as Borland Delphi allow systems to be created by assembling local components such as ActiveX Controls and OLE, while others such as Microsoft's Visual Studio, within the .NET framework, allow systems to be created from components accessible over a network. Presently, not only is it possible to visually design applications using familiar affordances, but also interoperability and scalability can now be infused into all aspects of the system right from the start by exploiting the capabilities of distributed computing.

Component assembly in Java was introduced with the BeanBox (Johnson, 1997) and commercialised with Java Studio (Nourie, 2004). M7's Application Assembly Suite (Johnson, 2004) is yet another Java solution to visual application development. To visually wire and deploy applications, it provides repository-based access to persistent business objects and a graphical environment and tools. Applications can then be quickly assembled without necessarily requiring a developer to write any code. Because other non-visual Java IDEs offer full control over application creation, including performance tuning and integration with other component frameworks, M7 can be integrated with several other IDEs including Eclipse and Borland's JBuilder. Still in the Java domain, BEA's WebLogic Workshop (BEA, 2004) is M7's nearest application-assembly rival. However, the most recent M7 offering has a few benefits over BEA. For one, WebLogic Workshop only provides support for the WebLogic application server, whereas M7 supports other application servers, such as IBM's WebSphere and JBoss Group's self-named server. Applications such as the WebLogic Workshop enable J2EE applications to be created quickly without having to enter the maze of J2EE documentation by automatically taking care of interfaces and connections. The BEA assembly solution allows developers to create Web services, whereas M7 only allows externally created Web services to be added to its repository for use in applications. Although M7 may be good at shielding junior developers from Java's complexity, Biggs (2003) is adamant that it misses the functionality required to give experienced Java programmers an edge.

A recent component assembly project proposes a component-based application development tool named the COBALT Assembler (Lee et al., 2003), which supports the design and implementation of EJB component assembly by using a plug-and-play technique based on the architecture style. Because EJBs do not support component assembly using a plug-and-play technique due to the hard-wired composition at the code level, Lee et al. suggest that an architecture for EJB component assembly be defined abstractly and any inconsistencies between the system architecture and its implementation are then eliminated at the implementation level. The proponents of the COBALT Assembler first define the system architecture with the Architecture Description Language (ADL) and then wrapper and glue code are generated for the assembly. After any inconsistencies between the architecture and its implementation are resolved, the assembled EJB components are deployed in an application server as a new composite component.

Several systems rely on object-oriented technology to provide reusable code modules that are accessible in an IDE as icons. Some of these IDEs contain modules that are customised through menus and then connected by lines that describe the interaction between them (Petzold, 1992).

But there are only a few IDEs that compose components in the manner described by Petzold. Using arrows during the design of an application to describe the relationships among its constituent components is more prevalent in modelling software such as Rational Rose (IBM, 2004) and Microsoft's Visio (Microsoft, 2004d). Here, different structures are used to represent the various objects and the relationship between two components or processes is typically represented using lines or arrows.

The Control and Coordination of Complex Distributed Services (C3DS) (SIRAC, 2001) research project focused on developing a single IDE that supports component and task assembly by using arrows to establish the connections between the components or tasks, and provided tools for the entire system development life cycle, i.e., analysis through to deployment and maintenance. Unfortunately, the project was dissolved before achieving its goals.

The Automated Learning Group (ALG, 2004) has over the last few years, developed the Data to Knowledge (D2K) application environment for data mining. D2K is a rapid, flexible data mining and machine learning system that integrates analytical data mining methods for prediction, discovery and deviation detection, with data and information visualisation tools. It offers a visual programming environment that allows users to connect programming modules together to build data mining applications and supplies a core set of modules, application templates and a standard API for software component development. Similar to our proposed IDE, D2K also supports distributed computing.

The D2K modules take 0 or more inputs and produces 0 or more outputs. The D2K development team refers to this simple concept as a "data flow paradigm". Data simply flows through an itinerary, enabling the execution of modules. Each module performs some service then exits. The similarity of the D2K system to the product of this research is compelling. The difference lies in the fact that the D2K system targets a different problem domain, namely data mining.

From the systems overviewed in this section, it is apparent that IDEs are not a novel solution to creating systems by assembling components from a component pool. Although different domains employ differing methods to create modular systems, graphically assembling components can be seen as a popular approach to building component-based systems.

2.5 Digital Libraries

2.5.1 Overview

The IDEs discussed in the previous section are predominantly dedicated to assembling applications consisting of components that implement one or more of the component models described in Section 2.3. Modular DLs however, are often assembled from proprietary components, i.e., components that are not implemented as bona fide Web services or using any other well known component model. Consequently, no research could be found that addresses the problem of creating systems by assembling heterogeneous digital library components.

Without a widely accepted component model that can be adopted when creating componentised DL systems, one big issue remains—interoperability. Component interoperability becomes a pressing issue if the intention is to create a GUI that will use heterogeneous components in order to build distributed DL systems. Despite component interoperability being a pertinent concern in terms of this research, the interoperability dilemma that most plagues the DL community is interoperability between digital library systems wanting to exchange metadata.

The ensuing sections delve further into the digital library arena by outlining the part played by the OAI in improving digital library interoperability, the ODL project's contribution to digital library componentisation and interoperability at the component level before culminating with a discussion on other current digital library components and systems.

2.5.2 The OAI and OAI-PMH

To address the need for interoperability amongst DLs, a group of representatives from various existing archives gathered in Santa Fe, Mexico, USA in October 1999 to discuss issues related to pre-print servers. The Santa Fe Convention, as it was originally known, quickly evolved into the Open Archives Initiative (OAI). A steering committee was appointed and given the responsibility of overseeing the OAI's development and promoting use of its work. The central theme of the steering committee's first meeting was the establishment of recommendations and mechanisms to facilitate cross-archive value-added services (Van de Sompel and Lagoze, 2000).

After the first meeting which mostly involved participants from the E-prints community, further meetings were held that invited participation from a broader audience including publishers and researchers. After much consultation, version 1.0 of the OAI Protocol for Metadata Harvesting (OAI-PMH) was released in January 2001 (Van de Sompel and Lagoze, 2001). This protocol promised to provide a simple mechanism for DLs to interoperate effectively. Modifications in the underlying schema standard resulted in a minor upgrade of the OAI-PMH to version 1.1. In June 2002, version 2.0, a more stable version than its predecessors, was released to enhance generality.

The simplicity of the OAI-PMH led to its acceptance within the DL community, and as a result, a large number of digital library projects have taken, or are taking, steps towards adding OAI capabilities to their systems.

In a nutshell, the OAI-PMH provides an application-independent interoperability framework based on metadata harvesting (Van de Sompel and Lagoze, 2001). There are two classes of participants in the OAI-PMH framework:

– Data Providers administer systems that support the OAI-PMH as a means of exposing metadata; and
– Service Providers use metadata harvested via the OAI-PMH as a basis for building value-added services.

Interoperability enables data providers to communicate with the providers of services, and hence allows archives to share their metadata. As was the intent of component-based programming, service providers are able to provide highly specialised services, which are built on the well-defined interfaces provided by data providers. With a simple yet effective interoperability protocol such as the OAI-PMH, providers of data can focus on improving and maintaining the quality of their data with the knowledge that there are specialised, high quality service components that can be seamlessly integrated to form a DL system.

2.6 Digital Library Components

As previously mentioned there are two classes of DL components: Repositories that hold the information about the digital object or resource, and services that allow users to view or modify the resource. Discussed below are some of the more common DL services.

2.6.1 Typical Services Provided by DL Components

2.6.1.1 Searching

Searching, as the term implies, usually involves looking through indexed metadata for words that match the query. Some search components offer more sophisticated searching mechanisms such as Boolean queries and proximity queries. An example of such a search engine is Lucene (Goetz, 2000) which is discussed in a later section in more depth. IRDB (Suleman, 2002) is a simpler search service which implements the ODL-Search interface, also discussed shortly.

2.6.1.2 Cross-Archive Searching

In contrast to simple searching described above, cross archive searching enables the service provider to harvest metadata in one or more formats from multiple remote archives and index the data according to collection, set or specific fields within the metadata. An example of a component that implements cross-archive searching is Arc (Liu et al., 2001), developed by the Old Dominion University Digital Library Research Group.

2.6.1.3 Annotations

Annotations are augmentations to existing documents and usually require the construction of a separate database for that purpose. A typical implementation of an annotation service involves linking the data in the annotation database to the record that is being annotated in the source archive. ODL's Annotate is an example of a component that provides annotation services.

2.6.1.4 Filtering

Several researchers are focusing on automating the creation and maintenance of user profiles and applying these profiles to information retrieval (Nabil et al, 1997). Users indicate a set of interests, and objects corresponding to those interests are presented. Filtering tends to imply passive mechanisms, but software agents are often used for a more proactive approach i.e., improving over time as they record additional user actions and reactions.

2.6.1.5 Browsing

Unlike searching, a browsing service often requires that the metadata contain fields with controlled vocabularies that can be used to build categories within which the objects may be placed. With some browsing components, these categories may be predefined.

2.6.2 OAI-Compliant Components

There is a large body of research dedicated to components developed for experimental purposes and a similarly large amount for those developed for production DL systems. Selecting the right components requires an innate understanding of the features and capabilities of these components. A lot of emphasis has been placed on improving searching and retrieving of heterogeneous resources and presenting those resources in a uniform manner. With the emergence of the OAI-PMH as a standard to resolve DL interoperability problems, several components have been designed to work with OAI-compliant archives and render services on metadata that is harvested using the OAI-PMH.

2.6.2.1 Searching services

Arc (Liu et al., 2001), one of several components developed at Old Dominion University (ODU), prides itself as being one of the first searching services based on the OAI protocol. Arc harvests metadata from several OAI compliant archives, normalises them and stores them in a relational database. The transition of the Networked Computer Science Technical Reference Library (NCSTRL, 2004) from a Dienst-based system with federated search algorithms to an OAI-compliant one has seen Arc being used as the metadata harvesting search engine replacement. Arc has also been extended and incorporated into the Archon digital library (Maly et al., 2002), also part of the ODU digital library project.

NASA Technical Report Server (Nelson et al., 2003) is a publicly available metadata harvesting-based search service originally based on distributed searching (federation). Its mission is to collect, archive and disseminate NASA aerospace information, but offers users the possibility of obtaining scientific and technical information from sites external to NASA. Both Arc and NTRS use hierarchical harvesting— the ability for a service to act as a provider of data to other services.

The University of Michigan (UM) digital library group developed a broad and generic information retrieval service called OAIster (Hagedorn, 2003), for information about publicly available digital library resources provided by the library research community. The service was built through a collaboration that relies on the University of Illinois's metadata harvester (Cole et al., 2002). In their partnership with UIUC, UM developed mechanisms to regularly export and transform the harvested data to a system that is enhanced for information retrieval. The service encompasses as broad a collection of resources as possible, regardless of subject matter.

In addition to OAIster, my.OAI (2004) and Perseus (Mahoney, 2001) are a few other searching services that also cover all registered OAI data providers. My.OAI allows users to personalise the appearance of their interface, and it allows users to customise the service to suit individual interests.

2.6.2.2 Other Service Providers

DP9 (Liu et al., 2002) employs an alternative approach to facilitating information retrieval of data contained in archives by providing an OAI service provider for Web crawlers. DP9, created as an ODU research project, is an open source gateway service that allows general search engines like Google to index OAI-compliant archives. Indexing OAI collections via an Internet search engine is often difficult because Web crawlers cannot access the full contents of an archive, are unaware of the OAI and cannot handle XML content very well. This problem is solved by providing consistent URLs for repository records, and converting them to OAI queries against the appropriate repository when the URL is requested. This allows search engines to index the resources contained within OAI compliant repositories. As DP9 is a gateway service, it does not cache the OAI records and only forwards any request to a corresponding OAI data provider. Its quality of service is highly dependent on the availability of OAI data providers. However, Celestial (2004) was designed to complement DP9 by caching metadata records and increasing data availability by offloading work from individual archives.

A component offering something other than the promise of more efficient indexing and searching algorithms is the Kepler project (Maly et al., 2001). The Kepler concept is to give any user, by means of a self-contained, self-installing software system, the ability to have a personal, portable data provider or "archivelet". An archivelet has the tools to let the user publish a report as simply as it is to post to a Website, yet have a fully OAI-compliant digital library that can be harvested by a service provider. The Kepler designers' vision is to provide tools and software for communities to easily deploy digital libraries that are customised for their needs, can be populated, managed and are "open" for development of future services.

The Digital Library Technologies group at the National Centre for Supercomputing Applications (NCSA, 2003) pledged a continuing effort to develop components of a new infrastructure for building large-scale digital libraries of distributed, heterogeneous digital information objects. The components developed enable digital information to be used across and within communities by providing user-configurable tools for formatting, translating, publishing, indexing and searching data and metadata. Their tools are designed to interoperate using standard protocols such as the OAI-PMH or the ISO-standard Z39.50 (ANSI, 1995) protocol for information retrieval. Their component suite currently focuses on Components for Constructing Open Archives (COCOA) and Grammatical Understanding Kernel (Grunk) (Plutchak et al., 2002).

The framework, in its simplest case, works by connecting a JDBC-compliant SQL database to Open Archives in-a-Box (OAIB, 2004), which is a small Java servlet module developed as part of the COCOA framework. Briefly, OAIB is an application for exporting metadata stored in a relational database over OAI-PMH, enabling data providers to have their data accessible to the growing world of OAI-compliant tools such as harvesters, union catalogues and metadata processors. OAIB also provides a configuration wizard, enabling most OAIB features to be controlled with minimal programming.

Grunk is a Java class library for extracting structured metadata from semi-structured text formats. It is designed to make it easy to rapidly develop applications that extract information from undocumented, unsupported or just difficult to use text file formats. Typically, new data can be received periodically and imported into a database, using Grunk to regularise partially formatted fields. OAIB then connects to the database and formats records according to the OAIB configuration file. Other service components can then query this data.

2.6.3 Non-OAI Components

The following section discusses other components that can be used as part of a digital library. Because these components were not fundamentally designed to operate in an OAI environment, they must be supplied with adequate interfaces. The two components mentioned below are both indexing and searching components.

2.6.3.1 Lucene

Lucene (Jakarta, 2004) is a high-performance, full-featured text search engine written entirely in Java and is suitable for nearly any application that requires full-text search. Both indexing and searching features make up the Lucene API. Lucene was started by Doug Cutting as an independent project and it became an official Jakarta project in September 2001.

2.6.3.2 Swish-E

The Swish-E (Simple Web Indexing System for Humans—Enhanced) (Tennant et al., 2002) search engine is a descendant of SWISH, which was created in 1994 by Kevin Hughes. SWISH was transferred in 1996 to the UC Berkeley Library to perfect and add features, and the result was licensed under the GPL and renamed Swish-E.

Swish-E offers a unique combination of features that make it attractive for this DL component assembly research. Some of Swish-E's offerings include a fast and robust toolkit with which to build and query indices, is well documented, undergoes active development and bug fixes and includes a Perl interface (Tennant et al., 2002).

2.6.3.3 Discussion

Constructing a digital library by integrating non-OAI components such as the above two with ODL components using a graphical user interface will enable a variety of complex and highly functional DLs to be created quickly and simply. In general, service providers are beginning to offer increasingly sophisticated services and are proliferating in spite of the inherent bias towards data providers in the OAI community. Currently, there are a lot more data providers than service providers but the competitive market originally envisioned for service providers is beginning to emerge.

2.7 Digital Library Systems

Thus far, the emphasis has been on individual components. The following section shifts the focus to the various ways in which these components (and others) fit together as part of a larger framework.

2.7.1 The Open Digital Library Project

There have been several open inter-component communication protocols that enable DLs to be built as component-based systems. One of the front-runners and is the Open Digital Library (ODL) project, on which this research is largely based.

The ODL project was created to demonstrate that DLs can be built as a network of inter-connected components that communicate using protocols based on the OAI-PMH instead of as monolithic systems. By using the OAI-PMH, widely used in the DL community due to its simplicity and understandability, interoperability can become an integral part of the entire DL system.

Although the OAI-PMH has lowered the complexity of metadata transfer among vastly different DLs, the development of high quality services that operate over those repositories is still a challenge. With the ODL project, the provision of services can be as simple as the provision of data, effectively facilitating the development, management and interoperability of DLs.

Table 2.1 below, obtained from Open Digital Libraries (2002) by Suleman, tabulates the ODL components, their interface protocols, as well as a brief description of the function of each component.

Name of component	Description	Interface Protocol
DBUnion	Multiple data source merger	ODL-Union
IRDB	Search engine	ODL-Search
DBBrowse	Category-based browser	ODL-Browse
WhatsNew	Tracker for recent entries	ODL-Recent
Box	Dumb archive supporting submit and retrieve	ODL-Submit
Thread	Threaded annotation engine	ODL-Annotate
Suggest	Recommender system	ODL-Recommend
DBRate	Ratings manager	ODL-Rate
DBReview	Peer review workflow manager	ODL-Review

Table 2.1 – ODL components, descriptions and protocols

Figure 2.1 demonstrates how a digital library can be created by inter-connecting a number of ODL components. These components all support one of the ODL protocols in Table 2.1 above.

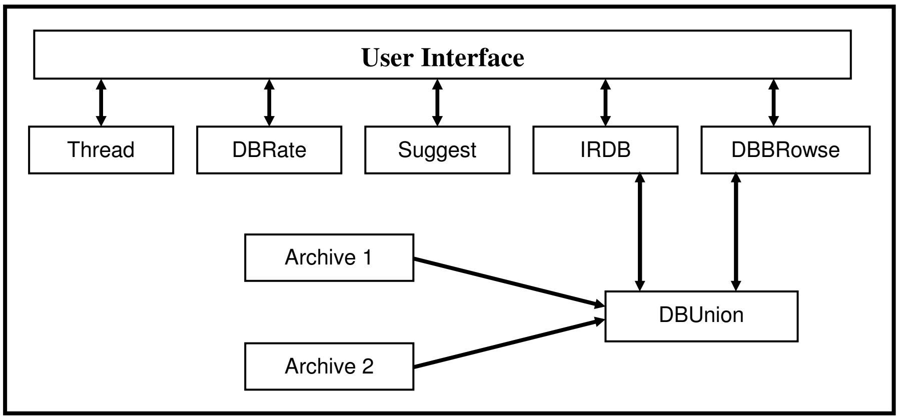


Figure 2.1 – A sample DL from a number of ODL components

During user testing of the ODL components, Suleman (2002) outlined a number of issues that arose. Mentioned below are those that this research attempts to address:

- Confusion over baseURLs: A baseURL is the URL that makes a component accessible via the WWW. Some participants expressed some confusion regarding which baseURL to use. Since all the baseURLs were similar, it was not obvious.
- Typographical errors: Entering URLs by hand resulted in many typographical errors and users found themselves having to retype the same, often long, URL all over again.
- Connectivity errors: Some components failed to work as they were expected to simply because they had been connected to a wrong component, once again due to the similarity of the baseURLs.

In addition to the fact that the issues raised above are all surmountable, there are a number of other factors supporting the use of ODL components as the focal point of this research:

- They are independent, self-contained objects that can be configured for use in a variety of situations.
- They are extensible, and to some degree, portable.
- They are based on a simple and understandable protocol. These characteristics are inherited by the components.
- Layering components does not adversely affect the performance of the resultant systems.

While the OAI-PMH was introduced by the OAI (Van de Sompel and Lagoze, 2000) in an attempt to resolve DL interoperability matters, the ODL project was a definitive step towards addressing component interoperability issues by applying the basic philosophy of the OAI-PMH (Lagoze and Van de Sompel, 2001) to individual components. Hence, there finally is a component model built on top of well-established standards that can be a foundation on which to develop a visual digital library development environment.

2.7.2 5SL – A conceptual design model

Some DL design attempts focus on conceptual design models such as 5SL, which is “a language for declarative specification and generation of domain–specific digital libraries” (Fox and Gonçalves, 2002). 5SL is based on the 5S formal theory for digital libraries and enables high-level specification of DLs in five complementary dimensions, which are the Stream, Structural, Spatial, Scenario and Societal models. The practical feasibility of this model was demonstrated by the presentation of a 5SL digital library generator for the MARIAN digital library system (Fox et al., 2002).

Kelapure et al., (2003) presented methods for the development, implementation, and deployment of a generic DL generator that can be used by DL designers to semi-automatically produce tailored services from the 5SL model, thus bridging the gap between the DL models and system implementation. In future, Kelapure et al. aim to reduce the manual intervention required by DL designers during the DL development life cycle.

The 5SL model facilitates the process of DL construction and maintenance by using a generator to automatically create DLs for a particular application and user community. Before the generation can take place, the model also requires the DL designer to formalise a conceptual description of the library using the 5SL language concepts. This requires a considerable level of know-how and expertise which is often possessed only by computer scientists and/or experts in the field of digital libraries, and is often required but overlooked for complex DLs.

2.7.3 Other Component-based DL Systems

Some modular DL systems were made according to the notion of individually defined services located anywhere on the Internet, and when combined, these services constitute a digital library. OpenDLib (Castelli et al., 2002) was built as a distributed digital library, made of components with open inter-component protocols. These services can be distributed over different servers, replicated or, if necessary, centralised. The functionality of the OpenDLib digital library includes the storage of and access to multimedia and multilingual resources, cross-language search and browsing, user registration and personalised information dissemination of new incoming documents.

DSpace (Tansley et al., 2003) is a digital institutional repository that captures, stores, indexes, preserves and redistributes the intellectual output of a university’s research staff in digital formats. Developed jointly by MIT Libraries and Hewlett-Packard, DSpace is freely available to research institutions worldwide as an open source system that can be customised and extended. DSpace is designed for ease-of-use, with a Web-based user interface that can be customised for institutions and individual departments.

The DSpace system is organised into 3 layers, each consisting of a number of components: the application layer, business logic and the storage layer. The storage layer is responsible for the physical storage of metadata and content, while the business logic layer deals with managing the content of the archive, users of the archive, authorisation and workflow. The application layer contains components that communicate with the outside world, such as the user interface. Some of the services the DSpace system offers are: searching (based on the Lucene search engine), browsing, the ability to retrieve new resources by means of its "history" service, and also a "workflow" service that manages the peer-review of submitted documents. DSpace will be discussed further in Chapter 5.

The Flexible Extensible Digital Objects Repository Architecture, better known as simply Fedora (Payette et al., 2002), is an open software toolkit that gives organisations flexible tools for managing and delivering their digital content. The core of Fedora consists of a digital object model that supports multiple views of each digital object and allows for relationships to be established between the various digital objects. All the functions (services) within Fedora are exposed as Web services. This reliance on standard protocols such as WSDL and XML ensures that it is able to function as an extensible and generic repository substrate upon which many kinds of applications can be created. Applications that are built on Fedora include: digital asset management systems; institutional repositories, digital libraries and content management systems, to name a few.

Archon (Maly et al., 2002) is yet another project from the Old Dominion University DL Research group. Metadata in the Archon system is obtained from a federation of physics collections using the OAI-PMH and the Arc search engine via its in-built harvester. Archon also possesses the ability to harvest from non-OAI archives such as Emilio. Archon implements services that are specially tailored for the physics community, such as new services to search and browse equations and formulae as well as a cross-archive citation service to integrate heterogeneous collections.

Some research has been invested in simplifying certain aspects of digital library creation, specifically, building of collections. The Greenstone Digital Library Software (Witten, 2003) is the product of such research. Greenstone provides a way of building and distributing digital library collections, opening up new possibilities for organising information and making it available over the Internet or on CD-ROM (Witten and Bainbridge, 2002). Produced by the New Zealand Digital Library project, Greenstone is intended to lower the bar for construction of practical digital libraries, yet at the same time leave a great deal of flexibility in the hands of the user.

In accordance with the maxim "simple things should be easy, complex things should be possible", Greenstone enables new users to quickly put together standard-looking collections from a set of source documents that may be HTML, Word, PDF or any of a number of other formats (Witten et al., 2001). Greenstone has been used to build many digital library collections from different countries, with different sorts of source materials**.**

There are two components central to the design of the Greenstone runtime system: "receptionists" and "collection servers." From a user's point of view, a receptionist is the point of contact with the digital library. It accepts user input, analyses it and then dispatches a request to the appropriate collection server (or servers). This locates the requested piece of information and returns it to the receptionist for presentation to the user. Collection servers act as abstract mechanisms that handle the content of the collection, while receptionists are responsible for the user interface.

The Web interface to the Greenstone collections can be easily customised to suit personal needs. However, without an advanced understanding of C++, the language it was developed in, extensions to add support for services not included in the Greenstone package are more complicated.

2.7.4 Personalised Services

Some DL systems can be distinguished from others due to the personalised nature of their service offerings. Cyclades (Fischer, 2003) is an OAI-compliant DL system composed of a federation of independent but interoperable services. Cyclades harvests metadata, indexes it and stores it in a local database. It includes mechanisms for dynamically structuring the harvested data into meaningful (from some community's perspective) collections. Cyclades differs from OpenDLib and DSpace in that it offers a filtering service to support information filtering on the basis of individual user profiles and the community the user belongs to. It also includes a facility to provide recommendations about new published articles within a working community based on the user's profile and services to support collaboration among members of communities and project groups such as related links, textual annotations and ratings.

2.7.5 Semantic Structuring

Torii, an implementation of the TIPS project, uses a different approach to disseminating information. TIPS is based on the idea that further progress in information distribution and scientific publishing on the Web requires the implementation of more extensive semantic structure in the documents that are exchanged; a unified, desktop-like, Web access to different archives, journals and services to manage information; and the availability of better information retrieval and filtering techniques (Bertocco, 2001).

The TIPS project is based on the concept of multi-layered documents (a document with its full set of annotations) and dynamic access. Like Cyclades, the TIPS project supports filtering according to user profiles. An essential element to the TIPS project that differentiates it from the rest is the fact that unlike most other search engines, it adds semantic structure to the data such that the search engine (powered by Okapi) can distinguish between a search for a book by John Smith, a book about John Smith and a book about a book by John Smith.

Torii is a portal that provides end users with a single gateway to personalised information and comprehensive support for their work, giving access to the tools they need in their everyday work, ideally replacing the user's desktop environment. It integrates dynamic access to a wide variety of data formats, allowing users to share, manage and maintain information from one central user interface, organising access to information all the while retaining location transparency of the data.

Torii implements services such as iArchive, which facilitates access to arXiv—a pre-print archive; icite, which is a citation harvesting system; and m2db which is a service on the multimedia archive that allows you to upload, download and retrieve multimedia documents using the OAI-PMH. Furthermore, Torii is accessible via the Wireless Application Protocol (WAP) (Fabbrichesi, 2002).

2.7.6 Augmented DLs

Goh and Leggett (2000) are of the opinion that all digital libraries should provide services that encompass not only searching, browsing and retrieval, but an entire range of services that support the user's digital scholarship from task inception to task completion. They subscribe to the notion that the users of research material (patrons) go through a cycle of Acquiring, Structuring, Authoring and Publishing (ASAP). This results in a class of digital libraries known as patron-augmented digital libraries (PADLs) that provide acquiring, structuring, authoring and publishing services to patrons. A patron-augmented digital library is one which contributes to the evolution of a library's holdings by augmenting the holdings to meet specific needs through new artifacts such as documents, annotations or other organisational structures via the support services offered by the PADL. UM's NaviQue system (Furnas and Rouch, 1998) is an example of a system offering, in addition to browsing and searching, the ability to author text, as well as organise material on the NaviQue workspace using direct manipulation to clarify their conceptualisations and coordinate their search activities.

NaviQue's notion of personal workspace has also been applied in the construction of MiBiblio 2.0 (Reyes-Farfán and Sánchez, 2003) where users can personalise their workspaces by choosing the resources and services they need. It enables users to organise, classify and manage their workspaces including resources from any of the federated libraries. Results can be kept in personal spaces and organised into categories using a drag-and-drop interface.

2.7.7 Discussion

Several DL systems, including those mentioned in this section, lay claims to extensibility. This may be so, but at what cost? Inserting a new component into a pre-packaged application will most likely require the presence of a specialist in that environment. Without the introduction of a standard and widely accepted inter-component interoperability protocol, possibly layered on existing standards like the OAI-PMH, this will continue to remain the case. The ODL project has attempted a solution to this problem and met some success. Building upon the success of ODL, this research hopes to increase the extensibility of DL systems by trivialising the construction of ODL/OAI compliant component-based DL applications.

2.8 Summary

This chapter has served as a brief overview of ongoing and finished work related to this research as well as some of the standards upon which this work lays its foundation. There was a brief introduction into software components followed by an overview of the most popular component models. A review of popular IDEs that assemble applications based on those component models ensued. The OAI's attempt to resolve the interoperability dilemma with its Protocol for Metadata Harvesting was discussed along with the ODL components and their associated framework. Finally, the chapter concluded with other DL systems, their features and the components they employ.

Chapter 3

Background: The Blox Toolkit

3.1 Introduction

In the relatively short period of time since production DL systems first appeared on the market, several different approaches have been utilised by different communities in creating their DLs. Castelli et al. (2003) note that many DLs of the future will be networked systems hosted by servers belonging to supporting institutions, with distributed or replicated services on more than one server. This new vision therefore encourages the design of systems that are capable of supporting this kind of interaction. Castelli et al. further note that in order to design extensibility into any DL system, these systems must be based on open architectures. Systems based on open architectures are systems that explicitly promote interoperability, both internally and externally, as well as ease of modification and extension. Following this premise, it is therefore necessary to design DLs in a way that makes it possible to add a new service to an existing DL without having to rebuild the entire system.

As described in Chapter 2, the OpenDLib (Castelli et al., 2002) is one of a number of systems that allow designers to create DLs with open architectures that support plug-and-play expansions. However, the approach taken in this research differs fundamentally from other systems, including those discussed in Chapter 2, because in addition to applying the concept of an open architecture in a somewhat different manner, it attempts to exploit the advantages of traditional visual environments by using a Graphical User Interface (GUI). This allows the user to have a high level conceptual view of the system under construction, and a more intuitive method of assembling different service components by using simple drag and drop techniques made popular by several modelling systems.

This chapter introduces the reader to the design and implementation of a framework for assembling distributed components, including its architecture, the design and use of a Graphical User Interface for assembling components and the server that manages these components.

The work undertaken during the course of this research initially involved the participation of two honours students (Moore et al., 2003). These students developed certain aspects of the framework in partial fulfilment of their honours degrees. This chapter details their contribution i.e., a generic client (GUI) that can be used to assemble different component types and the server that manages these components. The next chapter highlights my involvement, including the changes made to the basic system to tailor it to digital libraries and how all the pieces fit together in an integrated component assembly framework.

3.2 Design considerations

During the design of the system, several factors were taken into consideration. Firstly, the emphasis of this project was on creating an architecture that allows users to visually instantiate and assemble distributed components remotely. As a result of this, it was necessary to use existing components that could easily be adapted to function effectively in this environment, rather than design special purpose components from scratch. The ODL components (Suleman, 2002), as previously discussed, are a set of self-contained independent digital library components that interact using specialised versions of the OAI-PMH, and are interconnected by means of command line configuration scripts. Creating a framework that allows ODL components to be interconnected using a GUI rather than the traditional command-line method was in itself an attractive prospect. However, restricting the range of the components that can be assembled with the GUI to only one source of components seems unnecessarily limiting. An architecture was needed that treats components in a uniform manner, ultimately facilitating the introduction of any new component into the component pool as long as it complies with the interfacing protocol discussed in subsequent sections.

Another important consideration during the implementation of the component assembly environment was the use of current communication standards such as XML (Bray et al., 2000), XML Schemas (Fallside, 2001) and SOAP (Mitra, 2003). Rather than create custom protocols for the different communication requirements, it was considered preferable to build on top of existing and well-established standards.

3.3 Architecture

Increasingly, people with limited IT skills are becoming involved in projects that often require a higher degree of computer knowledge than they currently possess. Website hosts are beginning to realise this, and are consequently facilitating the process of performing certain popular tasks.

Bravenet (2004), for example, allows users to include components such as hit counters or guest books in their Web pages after registering with them at no cost. This is achieved by requesting that users fill in certain configuration details in order to create an instance of the associated component, which can then be referenced from their website. With this method, several complex components can be included into Web pages with relative ease. In addition to the obvious drawback of the often distracting advertising that accompanies these types of components, one has less control over the component instance and, as such, one cannot modify the component beyond the configuration fields provided. Furthermore, as the component is not physically located on one's computer, if something should happen to the server hosting the instance, everything may be lost.

In spite of its disadvantages, remotely instantiating components is still a very attractive prospect, especially when considering the overhead incurred when using the ODL components e.g., during the installation phase, in order to ensure that all the requirements to run the components are met. Simply instantiating a component on a server machine that already meets all the component's requirements would obviate the need for specialised software. The loss of valuable services in large production DL projects composed of several distributed components can be prevented by using only those components located on trusted servers.

For this research, a client-server approach was chosen. The client-server software architecture is widely known to be versatile, because of its inherent message-based and modular infrastructure that is intended to improve usability, flexibility, interoperability and scalability. Because the intent was to use distributed components in an environment that encourages those very properties, a client server architecture seemed like the obvious choice.

Previously, to create a digital library using ODL components, one had to download and install all of the desired components, then configure each one individually. This process may not be so straightforward if one's computer does not possess all the required software to run the components, such as a Web server, MySQL and required Perl modules. But with a client-server architecture, one can connect to one or more servers hosting components without having to worry about installing them. Furthermore, with a client-server approach, one can make use of component instances created by others. It should therefore be possible to use an existing DL comprising, for example, an archive and search engine instantiated by someone else and augment that DL with a different user interface.

Figure 3.1 presents a broad overall view of the different elements in this client-server model:

- A graphical user interface (the client), which will be the only part of the system directly visible to the user, with the ability to communicate with the servers.
- A server daemon, which handles all communication between the components and the GUI.
- An interface to those components that both parties (the components and component server) understand.

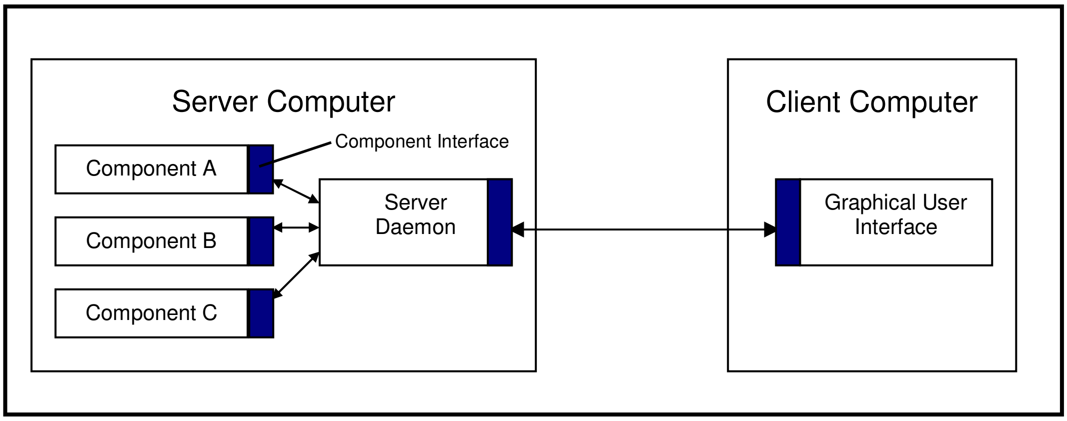


Figure 3.1 – An overview of the system architecture

Although Figure 3.1 demonstrates the client and server as two separate entities, this design offers a lot of flexibility, as they could potentially be located on the same computer. Figure 3.2 illustrates how this design can also be employed in a distributed environment by allowing a single client to communicate with several remote servers hosting components and, conversely, servers may communicate with any number of clients.

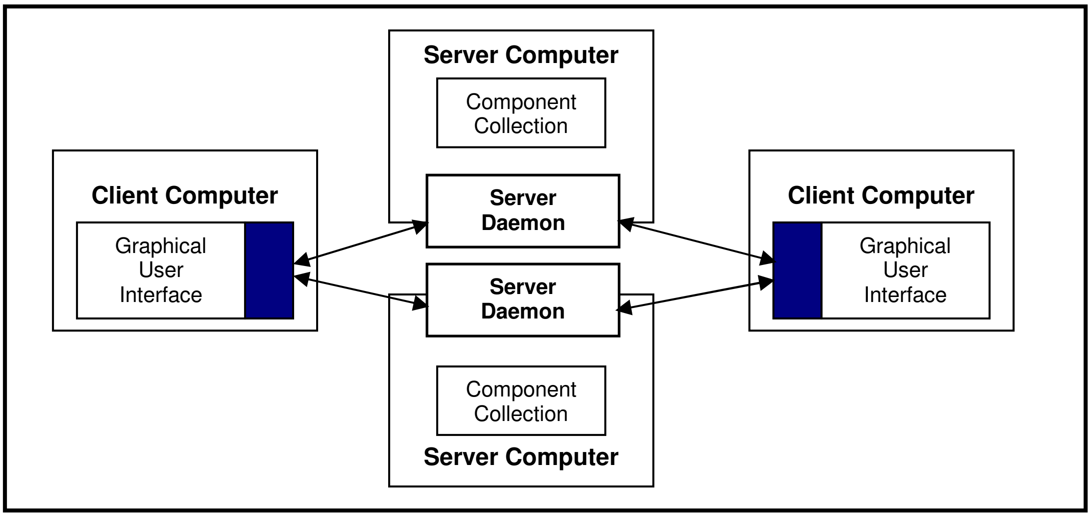


Figure 3.2 – Clients interacting with multiple servers and servers interacting with multiple clients

3.4 The Graphical User Interface

3.4.1 Overview

A Graphical User Interface (GUI) was created to allow users to specify the components they would like to include in the digital library and the details of their configuration. This enables the user to interact with distributed components in an effortless manner by making use of point and click, and drag and drop capabilities. The GUI (named Blox) was implemented using wxPython a Python port of wxWindows (Smart et al. 2005), and hence can run on any operating system due to the portable nature of wxWindows.

Creating a digital library by laying out desired components and then connecting them to one another allows users to have a conceptual view of the system under construction in a way that was previously not possible. Users are given the option of assembling their DL systems from any of the standard component types available or, alternatively, they may use previously instantiated components and are consequently not required to enter any configuration information for those components. This gives new meaning to the phrase "component re-use", since systems are not only built from reusable component templates, but also from existing component instances. A component template is labelled as a "type" in the user interface. A type in the context of this research is a skeleton for a component, and simply provides the list of fields to be filled in order for a component to be successfully configured. A configured component is displayed as a component "instance" in the user interface as depicted in Figure 3.3.

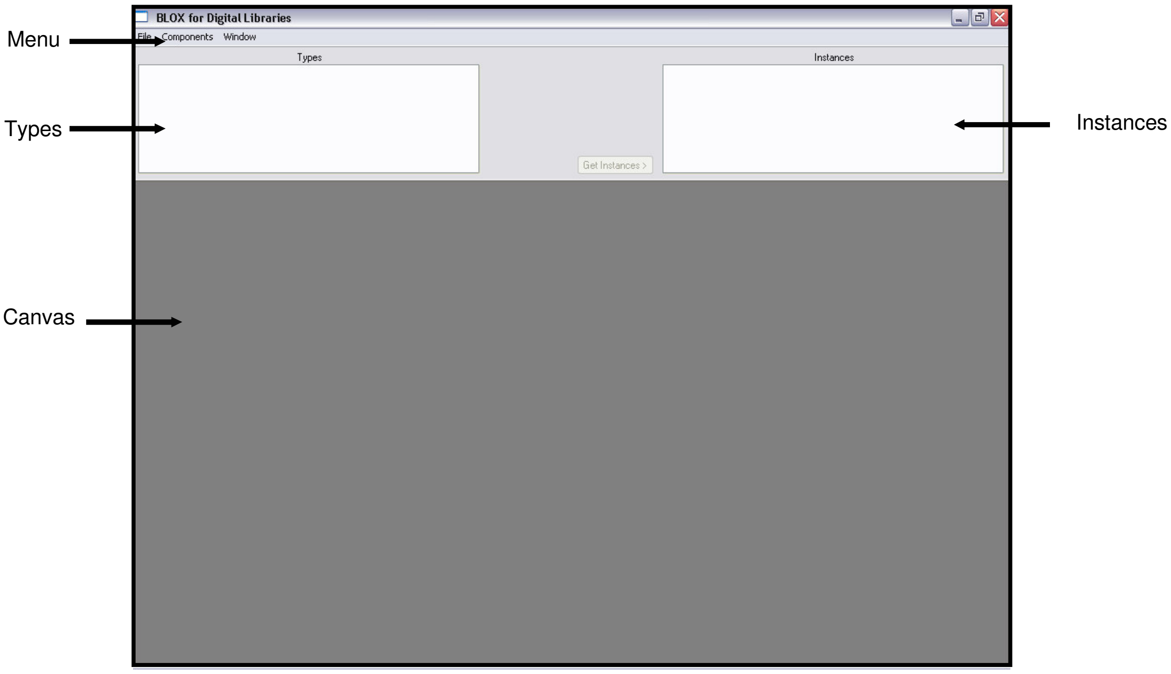
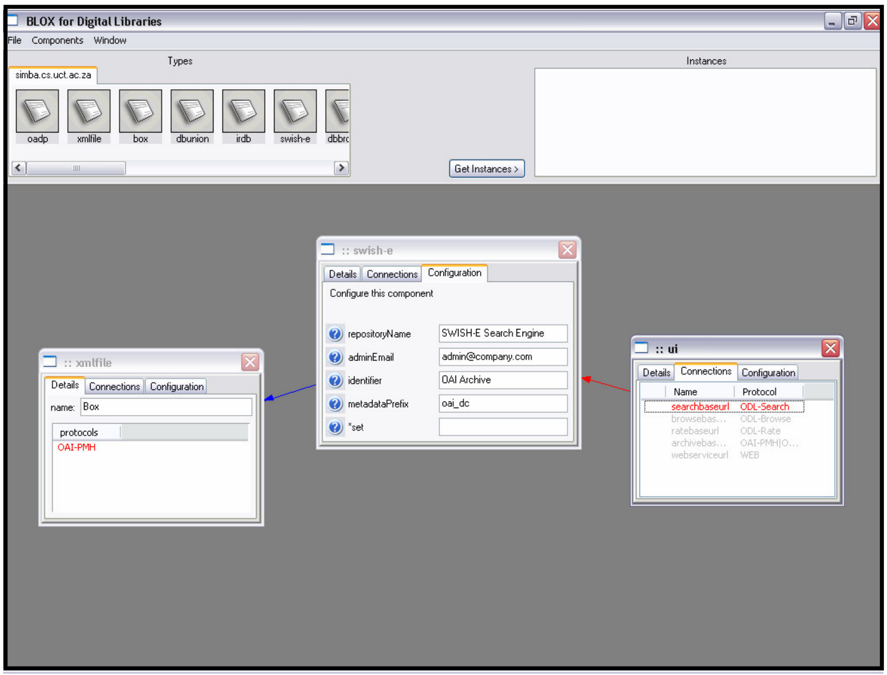


Figure 3.3 – Snapshot of the initial Blox Graphical User Interface

In addition to containers for the types and instances, the GUI consisted of a menu bar and a canvas or workspace where components are assembled. There is also a button labeled "Get Instances", which retrieves all the instances hosted on the specified server or servers.

Components are grouped in the types and instances panels according to the servers on which they are located. This is essential as users may make decisions on which components to use based on their location. To achieve this result, components are represented as icons within notebook pages, with each page of the notebook denoting a different server. Users can then locate a server by selecting a specific notebook page.

To use a component type or instance, the component is simply dragged onto the canvas. This creates a window which represents the component. Users are then able to manipulate the component window in the same way they would the windows of their operating system. This allows users to apply their previous experiences in order to intuitively interact with the components when performing typical actions like resizing and closing component windows. Figure 3.4 displays a simple digital library created by arranging three components on the canvas.



name: Box

protocols
OAI-PMH

Configure this component

repositoryName: SWISH-E Search Engine
adminEmail: admin@company.com
identifier: OAI Archive
metadataPrefix: oai_dc
*set:

Name	Protocol
searchbaseurl	ODL-Search
browsebas...	ODL-Browse
ratebaseurl	ODL-Rate
archivebas...	OAI-PMH|O...
webserviceurl	WEB

Figure 3.4 – A simple DL designed with the Blox System

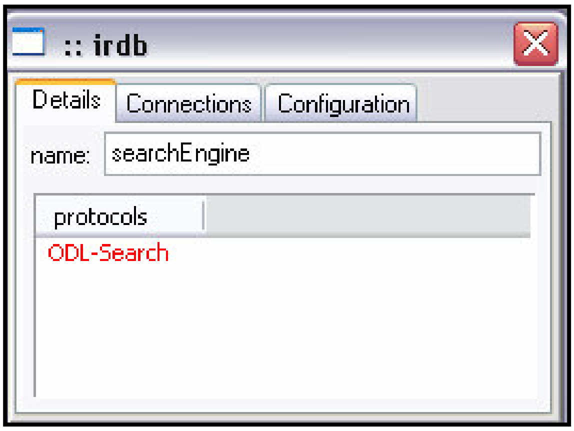
The components depicted in Figure 3.4 expose different notebook tabs, each having a specific function which will be described in subsequent sections. The leftmost window in the figure above, the XMLFile archive, exposes the Details tab; the middle window, the Swish-E search engine, exposes the Configuration tab, and rightmost window, the user interface component, exposes the

Connections tab. The various interconnections between components can be observed from the coloured arrows linking the components.

3.4.2 The Component Window

Each component window contains a notebook consisting of three tabs – the details tab, the connections tab and the configuration tab.

3.4.2.1 Details Tab



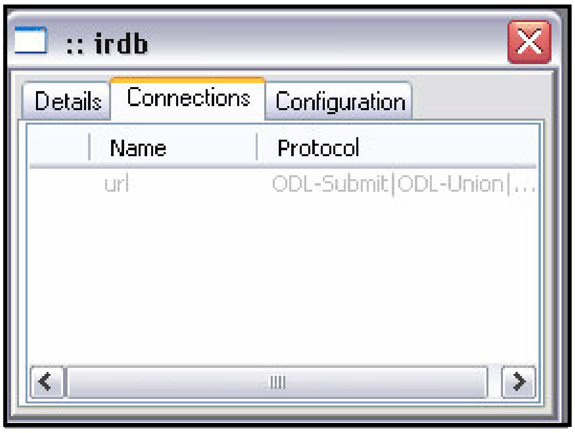
name: searchEngine

protocols
ODL-Search

Figure 3.5 – A component window displaying the details tab

The details tab, displayed in Figure 3.5, allows the component to be named, and displays the protocol that a client may use to communicate with the component.

3.4.2.2 Connections Tab

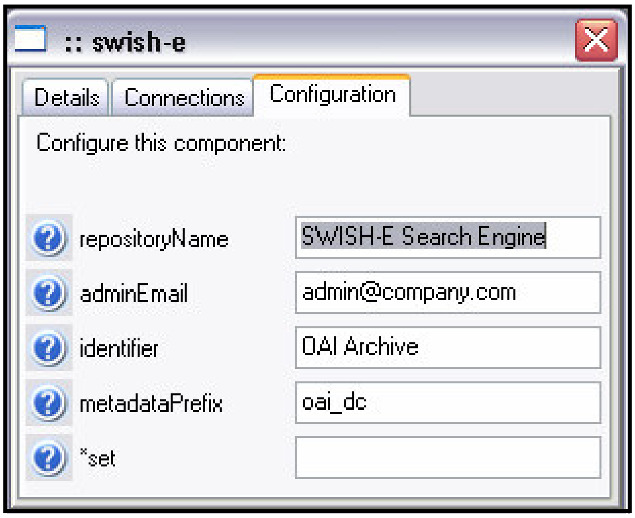


Name	Protocol
url	ODL-Submit|ODL-Union|...

Figure 3.6 – A component window displaying the connections tab

The connections tab (Figure 3.6) displays a list of the names of all the possible connections that can be made from that component and the protocols that the server component should implement for that connection to be valid. To create a connection between two components, the name of the connection is dragged into the connection tab of the other component. If the connection is valid (i.e., the two components are compatible), a coloured line is drawn between the two components and the same colour is used to highlight the name of that connection. This action is synonymous to entering a baseURL in the manual configuration method, but using the GUI requires no knowledge of what the actual baseURL is. Actual baseURL values are automatically entered by the GUI behind the scenes. Using colours to differentiate the different connections is beneficial in the instance where one component is connected to many others. Selecting a connection name and pressing the delete key deletes a connection.

3.4.2.3 The Configuration Tab



Configure this component:

repositoryName	SWISH-E Search Engine
adminEmail	admin@company.com
identifier	OAI Archive
metadataPrefix	oai_dc
*set	

Figure 3.7 – A component window displaying the configuration tab

The configuration tab (Figure 3.7) displays a form containing the configuration information for a component. Numerical fields are represented by a text control whose value can be incremented and decremented by using the arrows arranged next to the control. Drop down lists are used in instances where one of a finite set of values are permissible, and fields supporting “string” values are represented by simple text boxes. Most of the fields contain default values originating from the schema that defines that type of component. Later sections will expand more on the component schemas. Sometimes, some of the fields contained in the form are arranged in repeatable groups. Clicking on the plus sign next to these repeatable groups creates another copy of that group. For example, if a component may be configured to harvest from multiple archives, clicking on the plus button creates more fields so details of the additional archives can be entered.

3.4.3 Managing Components

When using remote components, a user can specify the server on which the components they would like to use are located by selecting “Server Manager” from the file menu. Figure 3.8 demonstrates the Server Manager dialog box. Users then have to specify one or more server/port pairs which host the components they would like to include in the digital library.

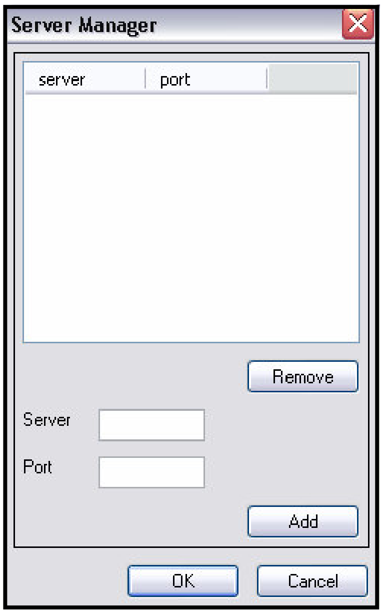


Figure 3.8 – Server Manager dialog box

Selecting "New project" from the file menu indicates that the user would like to start a new project. This action results in a dialog box being presented to the user so they can select one of the available handlers. Handlers are used by the GUI to identify the kind of components the user would like to use, and by so doing, requests the right type of components from the server. For example, a server may host Python components, DL components and others, but by selecting a specific handler, only those types of components are returned to the GUI by the server. This permits handlers to be written for any kind of component system, thus creating a truly generic and open system.

After a DL system has been assembled on the canvas and the configuration details of all its constituent components have been filled, the user may decide to publish the system by selecting "Publish" from the Components menu. This action results in the extraction of the instance description from each component, which is then compiled into a Component Connection Language (CCL) file. The CCL is an XML-based language that describes a component-based system. The CCL will be discussed in detail in Section 4.5 of the next chapter. This CCL file is then sent off to the CCL Resolver, an independent module located on the client computer, dedicated to ensuring all the components described in the CCL are properly configured. The CCL Resolver was designed to operate independently of the GUI so components could be configured while leaving the user free to perform other activities in the GUI. The CCL Resolver extracts instance descriptions from the CCL and forwards them to the Client Communications module one by one, so they can be sent off to the relevant server or servers for processing. If the response from a server to Client Communications indicates that an error occurred during the configuration of a component, the error is relayed back to the GUI and displayed to the user. They can then correct the error and attempt to republish the system.

3.4.4 The CCL Resolver

The CCL Resolver's main function is to accept a CCL document, extract its constituent components and forward them to the client communications layer so that each can be sent off to the relevant server and configured. The CCL Resolver extracts all the information required to configure a component such as the server, port, handler and instance description for each component in the CCL. Because the nature of the DL components are such that the successful configuration of one depends on another, the connection tags in the CCL are examined in order to determine the dependencies. A dependency tree is created so that components are configured only when all the components they depend on (i.e., connect to) have been configured first. For example, an attempt to configure a search engine to harvest from a non-existent archive will result in an error returned by the search engine component, so it is important to ensure that the archive is successfully configured before attempting to configure the search engine. Any errors that occur halt the configuration process, and the errors returned by the component are displayed to the user.

3.5 The Blox Server

3.5.1 Overview

The Blox client (GUI) was designed to communicate with one or more Blox servers locally or remotely located. The server acts as an intermediary between the Blox client and the remotely hosted components, by receiving requests from the client, forwarding those requests to the appropriate components, and returning a response. The server was designed in a manner that permits simultaneous communication with multiple clients. Figure 3.9 illustrates the layout of the Blox server as a subset of the entire architecture.

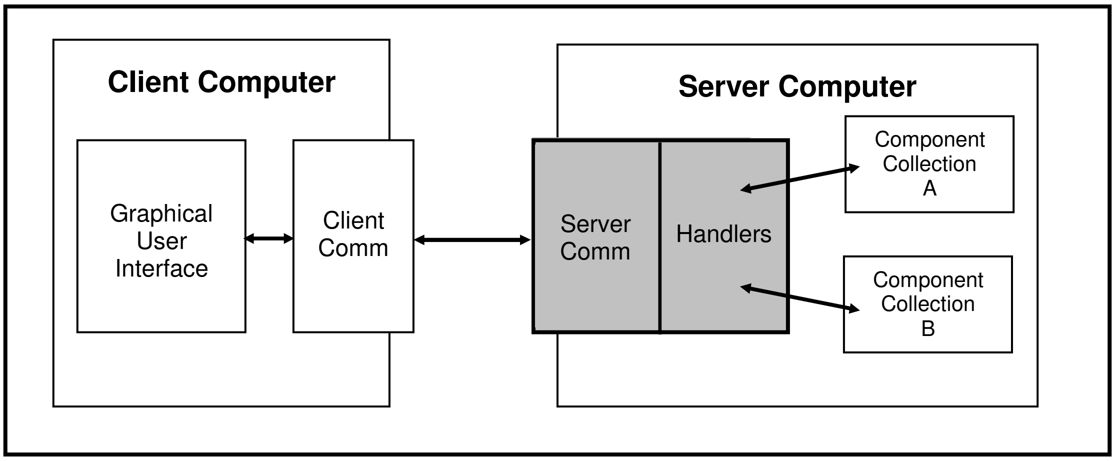


Figure 3.9 – The architecture with the server shaded

3.5.2 Server Structure

The server contains two separate parts that interface to provide the necessary functionality to the components it manages and the GUI — the server side communications component and component specific handlers as is depicted in Figure 3.9.

The server was designed to transport XML documents from one or more clients to the components. This is achieved by listening for messages from the server-side communications component, interpreting these messages and forwarding them to the relevant handlers. Handlers deal with the administration of the types of components they were designed to manage. That is, if the server has to deal with, say, DL components and Python components (Python templates providing specific functionality), handlers would have to be written for both component suite types and registered with the server. Handlers are discussed in more depth in Section 3.5.5.

The server and all its constituent components were implemented in Python (Van Rossum, 2005). Since there are no visual aspects to the server, there was no need for the libraries provided by the wxPython API, as was the case for the Blox client. The entire project involved a fair amount of XML parsing, and Xerces 2.0 was used for that purpose along with the Python wrapping library, pirxx 1.3.

3.5.3 Server Communications

A major feature of the communications protocol involved not using a request/response architecture with timeouts. The main reason for this is because some components (such as the configuration of certain ODL components) might require a considerable amount of time to complete the required activities before they can return a response. This adds some complexity to the communication procedure, as there is no guarantee the components will return a response, yet a response must eventually be returned to the client to prevent users from waiting indefinitely. No server-side solution to this dilemma has been found as yet. One can only transfer the responsibility to the components in the hope that they are implemented in a foolproof manner that guarantees a response.

Effective communication between clients and servers was achieved by using SOAP, a platform independent, XML-based standard, over SMTP. The communication between the client and server involves the use of several SMTP message sequences to deal with component-specific information and handler information.

The client (by its communication layer) is required to send numbered messages via the SMTP "Message-Id" header. The server also numbers responses in the same manner, with the client-specified number included in the “ In-Reply-To” header. This enables the client and server to engage each other in a conversational manner, if required. The server knows where replies should be sent by using the “Reply-To” header in the format port@server.

3.5.4 Server Interaction with the client

The functionality the server is required to provide to the client is tabulated in Table 3.1, along with the request and response formats that should be implemented to provide that functionality.

Functionality	Requests	Responses
Obtain a list of handlers	GetHandlers	GetHandlersResponse
Obtain a list of component types	SubscribeTypes	SubscribeTypesResponse
Obtain a list of component instances	SubscribeInstances	SubscribeInstancesResponse
Obtain a specific component instance, identified by a given name	GetInstance	GetInstanceResponse
Receive and dispatch configuration information for a component	ConfigureComponent	ConfigureComponentResponse

Table 3.1 – Server interface illustrating possible request types and corresponding responses

As can be seen from Table 3.1, the GetHandlers request retrieves all the handler instances available, including their names and descriptions, and the handler type it is, in the form of a GetHandlersResponse message. The client can obtain component types by issuing SubscribeTypes request. Instances are retrieved using SubscribeInstances, and requesting a specific instance is done via the GetInstance request. Components are configured by means of the ConfigureComponent request, with a response supplied via the ConfigureComponentResponse response. If an error should occur during a transaction, for instance, if there is a request for a component instance that does not exist, an error is reported using SOAP’s generic error reporting mechanism.

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <soap:Body>
    <m:GetInstance xmlns:m="http://cs.uct.ac.za/dmoore/blox">
      <HandlerName>myHandler</HandlerName>
      <Name>ComponentA</Name>
    </m:GetInstance>
  </soap:Body>
</soap:Envelope>
```

Figure 3.10 – Example of using the GetInstance request

Figure 3.10 demonstrates how a request can be made for a specific component instance by encoding the handler and component name in SOAP using the GetInstance request. The response from such a request is encoded using the GetInstanceResponse message (Figure 3.11), which includes the instance description of the component that was requested enclosed in “Instance” tags.

```
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
 <soap:Body>
   <m:GetInstanceResponse xmlns:m="http://cs.uct.ac.za/dmoore/blox">
     <Instance>
        The instance goes here…
     </Instance>
   </m:GetInstanceResponse>
 </soap:Body>

</soap:Envelope>
```

Figure 3.11 – Example of the response to a GetInstance request

3.5.5 Server Interaction with Components

In addition to the section that controls communication with clients, the server is also required to have a plug-in system so that it can deal with different types of components. This functionality is implemented by means of Handlers. Handlers need to interface with the rest of the server components and, as such, are required to conform to a standard Handler Interface.

3.5.5.1 Handlers Overview

It was important to have an extensible structure that allows handlers to be easily added and registered with the server. This was achieved by having a configuration file (an XML document) containing the name of the handlers and the respective scripts with associated arguments that can be loaded at run-time. Therefore, after writing a handler, all that is needed is to add it to the server configuration file. Handlers in turn have their own XML-based configuration file containing all the root directories of the components they manage.

Figure 3.12 contains an excerpt of the Blox server configured with a handler that deals with ScriptConfigurable components, i.e. the ODL components.

```
<?xml version="1.0"?>
<ServerConfig>
  <Handlers>
        <Handler>
                <Module>ScriptConfigurableServer</Module>
                <Class>ScriptConfigurable</Class>
                <Arguments>
                        <Argument>ScriptConfigurable.cfg</Argument>
                </Arguments>
        </Handler>
  </Handlers>
</ServerConfig>
```

Figure 3.12 – Blox server configured with the OAI/ODL Handler

As can be observed from Figure 3.12, each handler specified in the server configuration file contains three elements—the module that provides the functionality to deal with the components associated with that handler, the class of handler, and finally, the arguments to be passed when instantiating the handler.

3.5.5.2 Handler Interface

In order to provide all the required functionality to the client, the handler interface involves seven functions:

- SubscribeTypes
- SubscribeInstances
- GetInstance with the instance name as a parameter
- ConfigureComponent
- GetHandlerInstance which returns an XML document defining this instance of a Handler
- GetHandlerName which returns the name of the handler
- GetHandlerType which returns a URI representing the Handler type

Handlers work by examining the Handler configuration file for the directories of the components and executing specific scripts. The scripts that should be available to the handlers are defined in Section 4.6. Determining the script to execute is based on the function that was called from the handler interface. For example, the SubscribeInstances function in the handler interface will execute the ListInstances script from the directories registered in the handler configuration file and then returns a result (in this case, a list of all component instances) back to the server.

To demonstrate that all kinds of component-based systems can be assembled with the GUI, Python templates were created and a Python Handler that implements the Handler interface was created to manage these components. Users were then able to design simple programs from the Python components.

3.6 Summary

This section described the architecture of the framework as well as the client and server that was developed by two honours students to allow remote components to be assembled using a graphical user interface. It also described how using handlers allow various component suites to be assembled without modification to the client or server source code.

Chapter 4

Design

4.1 Introduction

The previous chapter presented a broad overview of the component assembly environment and discussed the basic GUI and server features required for remote visual component assembly. This chapter describes how the GUI and server created by two honours students was augmented to provide additional functionality, and tailored for digital library components, as well as how the client and server interact with components by means of a new component interface specification.

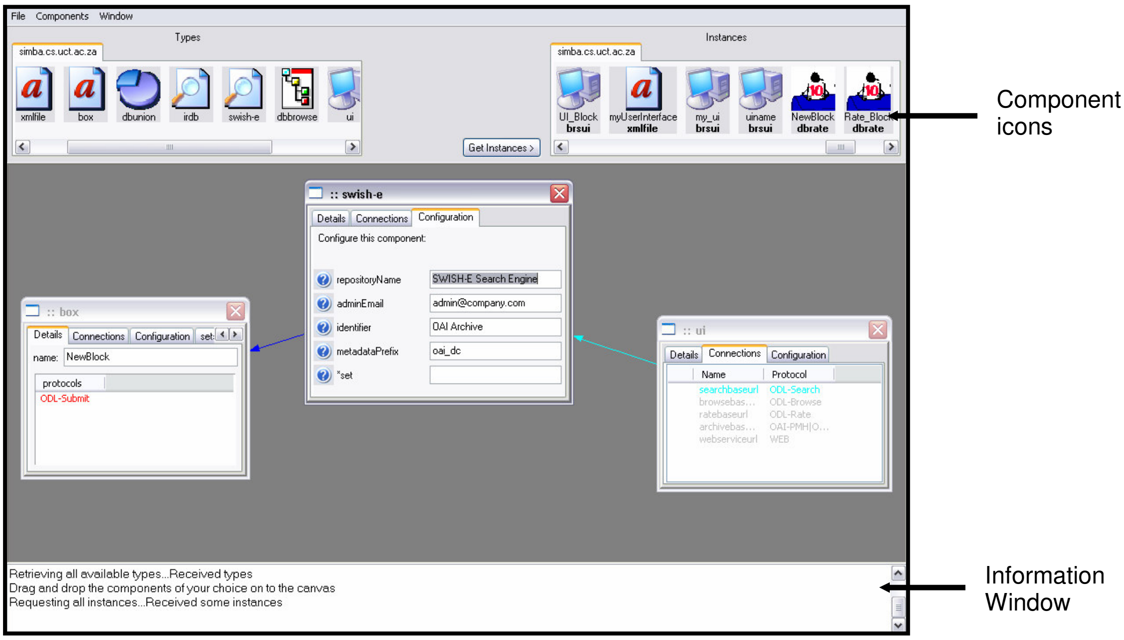
4.2 Blox+

4.2.1 Overview

The Blox client and server went through a number of modifications. This resulted in Blox being renamed to Blox+. This section outlines the alterations to the basic client and server.

4.2.2 Informational Window

After an initial evaluation of the GUI, it was observed that users did not always feel they knew what they system was doing in response to their input. Therefore, an information window was added to the GUI, which informs the user of the status of the system, as can be observed in Figure 4.1.



Configure this component:

repositoryName	SWISH-E Search Engine
adminEmail	admin@company.com
identifier	OAI Archive
metadataPrefix	oai_dc
*set	

name: NewBlock

protocols
ODL-Submit

Name	Protocol
searchbaseurl	ODL-Search
browsebas...	ODL-Browse
ratebaseurl	ODL-Rate
archivebas...	OAI-PMH|O...
webserviceurl	WEB

```
Retrieving all available types...Received types
Drag and drop the components of your choice on to the canvas
Requesting all instances...Received some instances
```

Figure 4.1 – Blox+: Blox client after modifications

In addition to the introduction of an information window providing tips, instructions and informing the user of what the system is doing, Figure 4.1 shows that the components have all been supplied with different icons based on the kind of component.

4.2.3 Sets and Metadata Prefixes Tab

Each component window initially had three tabs – details, connections and configuration. A fourth tab – the sets and metadata prefix tab (sets/mdps), displayed in Figure 4.2, was introduced to provide information required for the configuration of certain digital library components.

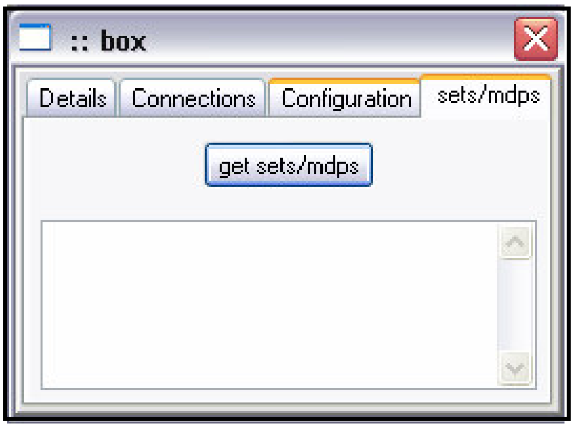
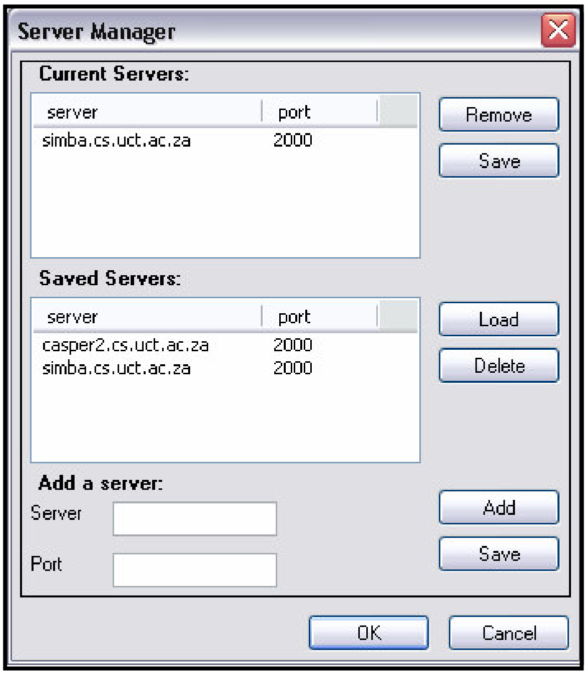


Figure 4.2 – The sets and metadata prefixes (“sets/mdps”) tab

This tab only appears in the component window of OAI-compliant archive components. Records in an OAI-compliant archive are organised into sets or categories, and stored in many formats denoted by different metadata prefixes. Clicking on the "get sets/mdps" button retrieves all the sets and metadata prefixes within that archive. This information is useful if, for example, the user would like to harvest only records with a specific metadata prefix within a specific set. Knowledge of this information is essential if those requirements are to be included in the configuration details of the component doing the harvesting. However, to obtain the set and metadata prefix details, the component must first be configured on the hosting server. Thus, clicking the "get sets/mdps" button publishes that component, and all the components that it connects to. For example, if an ODL-Union component connects to two archives, and the "get sets/mdps" button of the ODL-Union component is clicked, the ODL-Union component as well as the two archives are published, but not the search engine that connects to the ODL-Union component. The implementation of this was achieved by requiring that archives return information on the sets and metadata prefixes they possess in response to a ConfigureComponent request.

4.2.4 Server Manager

Selecting "Server Manager" from the File menu results in the user being presented with a dialog box (Figure 4.3) requesting the user to enter the servers and corresponding ports they would like to connect to. Allowing multiple servers enables users to create systems composed of distributed components. The main addition here was the ability for a user to save and load previously used servers.



Current Servers:

server	port
simba.cs.uct.ac.za	2000

Saved Servers:

server	port
casper2.cs.uct.ac.za	2000
simba.cs.uct.ac.za	2000

Add a server:

Server

Port

Figure 4.3 – Server Manager dialog box

The user may add a server by either entering the server and port in the text fields provided, or by double-clicking on one of the previously saved server values.

4.2.5 Open/Save DL

Digital Libraries (and indeed all component systems assembled by the GUI) are described using an XML-based language known as the Component Connection Language (CCL). The CCL is expanded upon in Section 4.5. Selecting "Open DL" from the File menu results in a typical "Open File" dialog box prompting the user to select a CCL file. The component instances that are described in the CCL file are loaded onto the canvas. These instances are also requested from their corresponding servers in order to be able to inform the user if the DL described in the CCL contains instances that have been modified or deleted from their hosting servers. This ensures that users are aware that they may need to re-configure their DL even if they made no changes. Additionally, the component types of the instances described in the CCL are also retrieved in order to be able to create and populate the various component tabs.

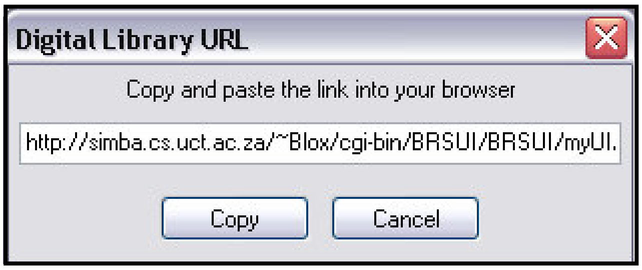
DLs can be saved by simply selecting "Save DL" from the File menu. This causes a typical "Save As" dialog box to be presented to the user, prompting the user to supply a file name. Each component instance is then converted to an XML instance description (the XML representation of a component instance) and stored in a *.CCL file where * is the user-specified project name.

4.2.6 Clear Canvas

This File menu item is used to delete all the components from the canvas, so a new DL system can be assembled.

4.2.7 View Digital Library

Once a DL has been published, the "View Digital library" menu item is enabled if that DL system contains a user interface instance. Users are presented with a dialog box (Figure 4.4) containing the URL of the DL user interface component, which can then be copied and pasted into their preferred browser.

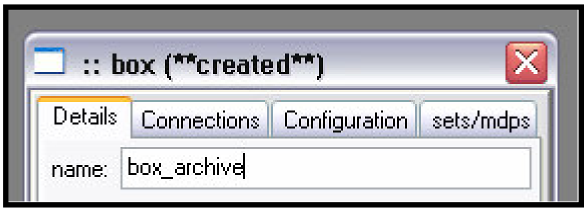

Digital Library URL

Copy and paste the link into your browser

http://simba.cs.uct.ac.za/~Blox/cgi-bin/BRSUI/BRSUI/myUI.

Figure 4.4 – Digital Library URL dialog box

However, if the successfully published DL system contains no user interface instance, they are simply informed that the DL has been created successfully. The digital library's user interface baseURL can also be obtained by going to "View Digital Library" on the Component menu. All of the successfully published components are indicated by including the text "(**created**)" on the title bar of the component window, as can be seen in Figure 4.5.



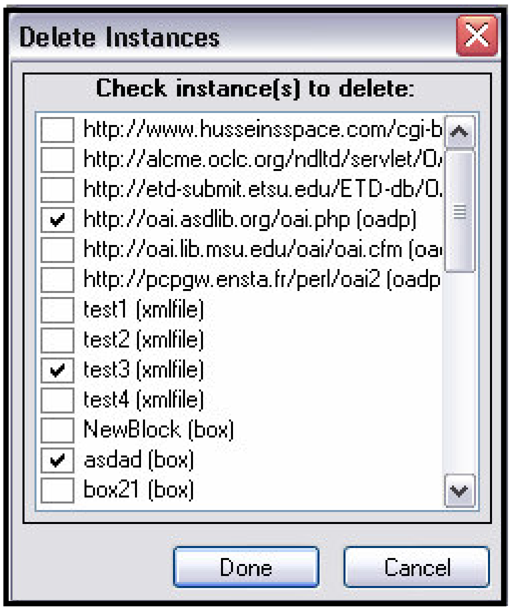
name: box_archive

Figure 4.5 – Title bar of a successfully created component

If no errors are reported and the DL system contains a user interface instance, the user is then asked if they would like to view the resultant DL. If they opt to view the DL, they are presented with a dialog box with the URL of the user interface component, which can then be copied and pasted into their preferred browser in order to load the digital library's user interface.

4.2.8 Delete Instances

To supply users with more control over the remote components they create, they have the ability to delete component instances from the servers via the "Delete Instances" menu item in the Components menu.



Check instance(s) to delete:

- ☐ http://www.husseinsspace.com/cgi-b
- ☐ http://alcme.oclc.org/ndltd/servlet/O
- ☐ http://etd-submit.etsu.edu/ETD-db/O
- ☑ http://oai.asdlib.org/oai.php (oadp)
- ☐ http://oai.lib.msu.edu/oai/oai.cfm (oa
- ☐ http://pcpgw.ensta.fr/perl/oai2 (oadp
- ☐ test1 (xmlfile)
- ☐ test2 (xmlfile)
- ☑ test3 (xmlfile)
- ☐ test4 (xmlfile)
- ☐ NewBlock (box)
- ☑ asdad (box)
- ☐ box21 (box)

Figure 4.6 – Delete Instances dialog box

A dialog box containing a list control with all the instances appears prompting the user to select the instances to delete. The GUI then transmits a DeleteInstance request to the relevant servers. The Blox client augmented to provide the functionality to deal with this new request type. After receiving DeleteInstanceResponse message, the deleted instances are then removed from the instances panel if the server indicates the request was successfully completed. The delete instances dialog box with some instances marked for deletion is displayed in Figure 4.6.

The major addition to the server was introducing the DeleteInstance request and corresponding DeleteInstanceResponse message type. This allows the users to delete instances existing on the hosting server via the GUI. Consequently, any handler created has to implement this interface as well.

4.2.9 Client Communications

Every time information has to be passed from the GUI to the server, such as a request for all component types or instances, or the response from such a request has to be relayed back to the GUI, the client communications layer is called upon to handle all the necessary communication activities. The original intent was for the user interface and the server to communicate asynchronously, so sending a request to the server still leaves the GUI free for the user to perform other activities. This was implemented by introducing an event-based system where responses from servers are received by the client communications layer and then registered with the GUI so they can be handled. However, after the preliminary testing phase, it was discovered that users found it more comfortable to wait for a response from the server than be in control of the user interface without some form of assurance that a response was forthcoming. Hence, although the underlying implementation is still asynchronous, to the user it appears to be synchronous—users have to wait for the server to return a response before they can move on to another task. This effect is achieved by transforming the default arrow-shaped cursor into an hourglass while a response is pending. However, this means that in the configuration of DL systems that take several minutes, the user is left waiting and incapable of doing anything else until a response is returned.

4.3 Creating a Digital Library – Client Perspective

The following section describes the sequence of activities in a typical usage scenario of the GUI.

After starting up the system, if a user would like to create a DL from distributed components, they would:

1. Specify the servers hosting the components by selecting "Server Manager" from the File menu.

2. Start a new project by selecting "New Project" from the File menu.
3. Select the DL handler from the list of available handlers. All the DL components types are then retrieved from the specified servers.
4. Drag and drop the desired components onto the canvas.
5. Name all of their components via their Details tab.
6. Customise the components by filling in all the configuration information, typically by modifying default values via the Configuration tabs.
7. Select the Connection tabs for all the components and create relevant connections by dragging and dropping connection names.
8. Publish their system by selecting "Publish" from the Components menu.
9. View the resultant DL by pasting the link provided into their Internet browser.

Additionally, users may want to:

- Use existing instances by retrieving all available instances via the "Get Instances" button.
- Retrieve the sets and metadata prefixes for a component via the "sets/mdps" tab.
- Delete certain instances by selecting "Delete Instances" from the Components menu.
- Use the "Open DL" menu item to continue the design of a DL system by loading a saved DL project.
- Save a DL project by selecting "Save DL".

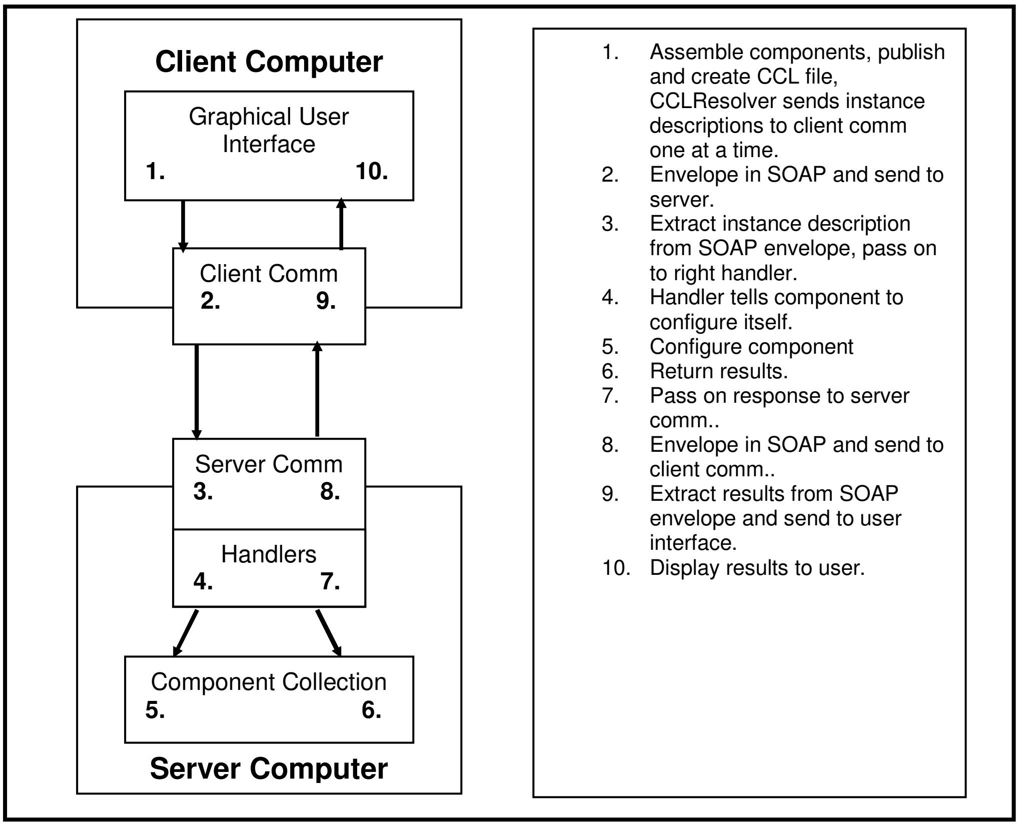
4.4 Creating a Digital Library – Server Perspective

To improve understanding of the server functions, the following section outlines what happens in the server during the process of creating a digital library with Blox+.

- The user selects "New Project" in the GUI. This results in a call to GetHandlers. The server discovers what handlers have been registered by examining its configuration file. A GetHandlersResponse message is returned containing all the handlers.
- The user selects a handler from the list of available handlers. This automatically results in a call to SubscribeTypes on all servers supporting that handler. A response is returned and the GUI displays all the component types.
- The user clicks on the "Get Instances" button. A call to SubscribeInstances is made with the current handler as an argument. The call is forwarded to the right handler and all instances are retrieved and sent back to the client.
- The user assembles components as desired.
- The user selects "Publish" in the GUI. A CCL file is created and a call to ResolveCCL in the GUI is made. ResolveCCL (via the client communication layer) then forwards the components contained in the CCL one at a time to the server. The server identifies the

handler for the component, which in turn calls ConfigureComponent, the component is configured and a response returned.

- The user selects "Delete Instances". A call to DeleteInstances is made, once again the right handler for those instances are identified and results returned (i.e., if the components were successfully deleted or not).



1. Assemble components, publish and create CCL file, CCLResolver sends instance descriptions to client comm one at a time.
2. Envelope in SOAP and send to server.
3. Extract instance description from SOAP envelope, pass on to right handler.
4. Handler tells component to configure itself.
5. Configure component
6. Return results.
7. Pass on response to server comm..
8. Envelope in SOAP and send to client comm..
9. Extract results from SOAP envelope and send to user interface.
10. Display results to user.

Figure 4.7 – Communications process between client and server

Figure 4.7 illustrates the steps that occur during the publishing process and other typical communication activities.

4.5 The Component Connection Language (CCL)

During the design of the Blox system, an important consideration was how to define a system made up of a number of components. A language was developed to represent the conceptual model of a system consisting of a collection of independent components. In this case, it is the conceptual model of a digital library. This XML-based language, referred to as the Component Connection Language (CCL), contains the instance descriptions of the connected components, the server and port on which the components were installed as well as information required to reconstruct their graphical representation in the GUI. Figure 4.8 illustrates a sample CCL file consisting of 3 interconnected DL components.

```
<?xml version="1.0"?>
<ln:CCL xmlns:ln="http://simba.cs.uct.ac.za/ccl">
        <instance>
                <handler>testODL</handler>
                <instanceDescription>
                        <name>component1</name>
                        <description>
                                ...instance description goes here...
                        </description>
                </instanceDescription>
                <position>
                        <start>
                                <xpos>73</xpos>
                                <ypos>134</ypos>
                        </start>
                        <height>200</height>
                        <width>268</width>
                </position>
        </instance>
        <instance>
                <handler>testODL</handler>
                <instanceDescription>
                        <name>component2</name>
                        <description>
                                ...instance description goes here...
                        </description>
                </instanceDescription>
                <position>
                        <start>
                                <xpos>391</xpos>
                                <ypos>142</ypos>
                        </start>
                        <height>200</height>
                        <width>268</width>
                </position>
        </instance>
        <instance>
                <handler>testODL</handler>
                <instanceDescription>
                        <name>component3</name>
                        <description>
                                ...instance description goes here...
                        </description>
                </instanceDescription>
                <position>
                        <start>
                                <xpos>691</xpos>
                                <ypos>138</ypos>
                        </start>
                        <height>200</height>
                        <width>268</width>
                </position>
        </instance>
        <connection>
                <from>component2</from>
                <to>component1</to>
        </connection>
        <connection>
                <from>component3</from>
                <to>component2</to>
        </connection>
</ln:CCL>
```

Figure 4.8 – A sample CCL file

Whenever a digital library project is saved by the GUI, it is saved as a CCL document. This CCL file can then be retrieved and reloaded at a later date to continue designing the DL. The CCL also permits existing DL systems to be easily modified.

As can be seen in Figure 4.8, each component instance is encapsulated in “instance” tags. The last part of the CCL is the “connection” tag. When the user connects two components in the GUI, their names are included in a “from” and “to” pair depending on the transfer of data between the components. Therefore, three connections result in three “connection” tags each containing “from” and “to” tags. Within the “instance” tag is the name of the handler that controls that component (handlers were discussed in Section 3.5.5 of Chapter 3), an “InstanceDescription” tag containing the name of the instance and its description (i.e., configuration information) and an optional “position” tag that is used to graphically represent that component in the GUI.

4.6 Component Interface

4.6.1 Overview

Sometimes components have to be utilised in an environment other than the one they were initially created for. Such is the case with the ODL components and all the other components used in this research. The ODL components were originally designed to be configured by running a Perl script via the command-line. Normally, a user had to run a “configure” script and was then prompted for configuration information along the way. This method of configuration was clearly inappropriate in a visual environment, and so a different system had to be devised in order to configure these components. The matter is further complicated due to the fact that the components may be distributed and the calling method may not possess the rights or know-how required to modify a component remotely.

To resolve this problem, each component required to function effectively with the Blox system had to conform to a standard interface specification. The first consideration when dealing with the components was how to represent a component and its required configuration details. A type description was used for this purpose. A type description describes the information required to successfully configure a component—XML Schemas were used for this. This decision was motivated by the fact that an instance of an ODL component is represented as an XML document. It then makes sense for a type description to be the Schema representation of that XML document and an instance of that Schema to be the component’s instance description. Also, the GUI can easily verify if the configuration information supplied by the user is valid by simply validating an instance description against its type description before sending off the information to be configured. Figure 4.10 shows a component’s type description and Figure 4.9 shows a corresponding instance description. The type description, wherever possible, contains default values that the GUI uses to populate the component's configuration tab. Some of these values, such as database details, are very specific to the server hosting the component. So, to host ODL components that can be instantiated remotely by others, the type description would have to be modified to reflect the details specific to that server.

```
<?xml version="1.0"?>
<irdb xmlns="http://simba.cs.uct.ac.za/irdb"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://simba.cs.uct.ac.za/irdb
                         http:// http://simba.cs.uct.ac.za/irdb.xsd">
        <repositoryName>the Rep name</repositoryName>
        <adminEmail>root@cs.uct.ac.za</adminEmail>
        <database>DBI:mysql:Demo</database>
        <dbusername>root</dbusername>
        <table>stuff</table>
        <archive>
                <identifier>mytest</identifier>
                <url>http://talc.cs.uct.ac.za/cgi-bin/OAI-
                     XMLFile/XMLFile/mytest/oai.pl</url>
                <metadataPrefix>oai_dc</metadataPrefix>
                <interval>86400</interval>
                <interrequestgap>10</interrequestgap>
                <overlap>1</overlap>
                <granularity>second</granularity>
        </archive>
</irdb>
```

Figure 4.9 – An instance description for the IRDB component

```
<?xml version="1.0"?>
<schema targetNamespace="http://simba.cs.uct.ac.za/box"
xmlns="http://www.w3.org/2001/XMLSchema" xmlns:odl="http://simba.cs.uct.ac.za/box"
elementFormDefault="qualified" attributeFormDefault="unqualified">
        <annotation>
                <documentation>
         Configuration for Box Archive implementation
      </documentation>
                <appinfo>
                    <begbaseurl>http://simba.cs.uct.ac.za/~Blox/cgi-bin/ODL-Box-
1.2/Box/</begbaseurl>
                    <endbaseurl>box.pl</endbaseurl>
                    <protocols><protocol>ODL-Submit</protocol></protocols>
                </appinfo>
        </annotation>
        <element name="box">
        <complexType>
                <sequence>
                        <element name="repositoryName" type="string" default="Box OAI
archive">
                                <annotation>
                                        <documentation>Full name of the component for
identification purposes</documentation>
                                </annotation>
                        </element>
                        <element name="adminEmail" type="string">
                                <annotation>
                                        <documentation>Email to which problems or
comments will be reported</documentation>
                                </annotation>
                        </element>
                        <element name="database" type="string">
                                <annotation>
                                        <documentation> name of database</documentation>
                                </annotation>
                        </element>
                        <element name="dbusername" type="string" minOccurs="0">
                                <annotation>
```

```
                                        <documentation>username to connect to 
database</documentation>
                                </annotation>
                        </element>
                        <element name="dbpassword" type="string" minOccurs="0">
                                <annotation>
                                        <documentation>password to connect to 
database</documentation>
                                </annotation>
                        </element>
                        <element name="recordlimit" type="integer" default="1000">
                                <annotation>
                                        <documentation>Number of records displayed before 
issuing a resumption token (integer)</documentation>
                                </annotation>
                        </element>
                        <element name="archiveId" type="string" default="test">
                                <annotation>
                                        <documentation>Archive Identifier</documentation>
                                </annotation>
                        </element>
                        <element name="allowread" type="string">
                                <annotation>
                                        <documentation>Names (or IP addresses) of those 
machines that may use the OAI/ODL interfaces to read from the archive </documentation>
                                </annotation>
                        </element>
                        <element name="allowwrite" type="string">
                                <annotation>
                                        <documentation>Names (or IP addresses) of those 
machines that may use the OAI/ODL interfaces to write from the archive 
</documentation>
                                </annotation>
                        </element>
                        <element name="populate" default="yes">
                                <annotation>
                                        <documentation>Would you like
                                        to populate the Box archive
                                        with test data? (yes,no)</documentation>
                                </annotation>
                                <simpleType>
                                <restriction base="string">
                                             <enumeration value="yes"/>
                                             <enumeration value="no"/>
                                </restriction>
                                </simpleType>
                        </element>

                </sequence>
        </complexType>
        </element>
</schema>
```

Figure 4.10 – The type description for the Box component

The component interface supports five service requests, implemented as scripts, outlined in Table 4.1. The entire interface specification document is supplied in Appendix A.

Interface Element	Function
Autoconfig	Receives an instance description (XML document) and instance name and then uses that data to create a component instance
GetInstance	Takes an instance name and returns its instance description
GetType	Returns the type description of that component
ListInstances	Returns all instance descriptions of a particular component type
removeInstance	Takes an instance name and deletes the specified instance

Table 4.1 – The interface of a component if required to function with the Blox system

The scripts were all implemented in Perl and stored in each component's root directory. All the components and their root directories are then registered with the server, in the Handler's configuration file.

4.6.2 autoconfig

The function of this interface is to configure a component based on the instance description retrieved from standard input. This instance information is processed and errors are sent to standard output in the format outlined in Figure 4.11.

```
<?xml version="1.0"?>
<autoconfig xmlns="http://simba.cs.uct.ac.za/autoconfig"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://simba.cs.uct.ac.za/autoconfig
http://simba.cs.uct.ac.za/autoconfig.xsd">
        <configured>no</configured>
        <error>Missing configuration Name</error>
        <error>Missing Database Table Prefix name</error>
        <error>Incorrect format for Admin email</error>
</autoconfig>
```

Figure 4.11 – Example of the XML output of autoconfig indicating at least one error occurred

In the absence of errors (Figure 4.12), for a new instance, the configuration information is stored in a new directory with the name given when autoconfig was called, or an old directory overwritten in the case of modifying an existing instance.

```
<?xml version="1.0"?>
<autoconfig xmlns="http://simba.cs.uct.ac.za/autoconfig"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://simba.cs.uct.ac.za/autoconfig
http://simba.cs.uct.ac.za/autoconfig.xsd">
        <configured>yes</configured>
        <errors></errors>
</autoconfig>
```

Figure 4.12 – Sample autoconfig output indicating no errors occurred

4.6.3 getType

The getType interface returns to standard output an XML schema with the type description for a particular component to which the instance description must validate. It takes no arguments. It was implemented by retrieving the type description from a folder labelled "type" situated in the component's root directory. Strictly speaking, as long as the getType interface can somehow locate a type description, it is not imperative to have a type folder. However, for consistency, all the components used in this research were supplied with a "type" folder containing the components' type descriptions.

4.6.4 listInstances

A call to the listInstances interface returns a list of all the instances of that component type. This interface element is most useful when a specific instance is required but its actual name is unknown, and also for discovering the available instances on a given server. The output of listInstances (displayed in Figure 4.13) is a list of all the instance descriptions of that component. The instances are displayed in alphabetical order, with the name of the instance preceding each instance description.

```
<?xml version="1.0"?>
<listInstances xmlns="http://simba.cs.uct.ac.za/listInstances"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://simba.cs.uct.ac.za/listInstances
                          http://simba.cs.uct.ac.za/listInstances.xsd">
<name>TEST 1 </name>
<instance> ...test1's instance description goes here...</instance>
<name>TEST2</name>
<instance> ...test2's instance description goes here...</instance>
<name>TEST3</name>
<instance> ...test3's instance description goes here...</instance>
<listInstances>
```

Figure 4.13 – An example output of the listInstances interface

4.6.5 getInstance

The getInstance interface accepts an instance name and retrieves the instance description for the component instance matching the given name.

4.6.6 removeInstance

The removeInstance interface accepts an instance name and deletes the component instance. It returns a “1” signalling success, or a “0” indicating the instance could not be deleted. This may happen when an instance is being used by another process at the moment of the delete attempt.

4.7 Interfacing ODL components

Table 4.2 outlines the ODL components that have been equipped with the auto-configurable interface.

Component	**Description**
DBUnion	Harvests metadata from multiple sources and merges them to form a single archive
IRDB	Simple search engine
DBBrowse	Allows for browsing through metadata using defined fields
Box	Simple database-driven archive that allows records to be submitted and retrieved
DBRate	Allows for the rating of resources
OAI-XMLFile	Archive that stores metadata records as XML or text files
DBReview	Manages peer reviewing of submitted resources
Thread	Allows resources to be annotated

Table 4.2 – ODL components used in this research

The ODL components outlined in Table 4.2 were all interfaced to function in the Blox+ environment. This was achieved by creating an XML Schema for each of them to ensure that they all possess a type description, and the type descriptions were then placed in a folder labeled “type” in the root directory of each component. For the purposes of this project, a component’s root directory is the directory where instances are stored. Autoconfig scripts were created so that components receive configuration details from the instance description supplied when autoconfig is called rather than from command-line prompts, as was the case with the “configure” scripts.

Figure 4.11 and Figure 4.12 display a sample XML output from the autoconfig scripts for most components. However, certain components, such as archives, provide additional information when no errors occurred during the configuration process. Figure 4.14 below displays the output of autoconfig when an archive is configured.

```
<?xml version="1.0"?>
<autoconfig xmlns="http://simba.cs.uct.ac.za/autoconfig"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://simba.cs.uct.ac.za/autoconfig
http://simba.cs.uct.ac.za/autoconfig.xsd">
        <configured>yes</configured>
        <info>
             <set>math</set>
             <set>physics</set>
             <mdp>oai_dc</mdp>
             <mdp>vra_core</mdp>
        </info>
</autoconfig>
```

Figure 4.14 – Example of the XML output of the autoconfig response when no errors occurred

Notice the introduction of the “info” tag. This optional tag allows additional component-specific information to be sent to the GUI. In this case, this takes the form of a number of sets and metadata prefixes that can then be displayed to the user.

If no errors occurred during the configuration of a component, the instance description is placed in a folder bearing the name supplied when autoconfig was called. If a component instance already bearing that name exists in the root directory, the instance description of the existing component is examined to check if it contains the same information as the new one. If it does, autoconfig terminates and returns no errors. However, if the new instance description supplied differs from the previous one, the old directory is overwritten and all relevant database tables deleted, but only if autoconfig deems the details contained in the instance description to be valid.

To further simplify the configuration of a component, certain fields such as “table prefix,” required by database-driven components and previously supplied by the user, are now automatically generated by autoconfig and inserted into the instance description.

Applying the specification for getType to the ODL components was trivial. A call to getType simply returns the schema located in the "type" folder. ListInstances involved examining all the subfolders of the component's root directory to determine if it was an instance folder, and then returning all the instance descriptions found, in the format detailed in Figure 4.13. GetInstance for the ODL components involved locating a component instance by the given name and then returning its instance description or an error if there is no such instance. Lastly, removeInstance simply deletes the instance folder with the given name as well as all the database tables that instance may have created.

4.8 Interfacing Non-ODL components

Any component can be interfaced in this manner in order for it to be able to communicate with the Blox server. To demonstrate this premise of extensibility, other components had to be similarly interfaced. However, interfacing with the server is not sufficient to ensure that any system created from those components will function as intended. The ODL components all communicate with one another using several special-purpose OAI-PMH-like protocols. Some other DL component with its own proprietary protocol may interface with the server, but might not function properly in conjunction with the ODL components. Figure 4.15 demonstrates the protocols involved when several ODL components combine to form a DL system.

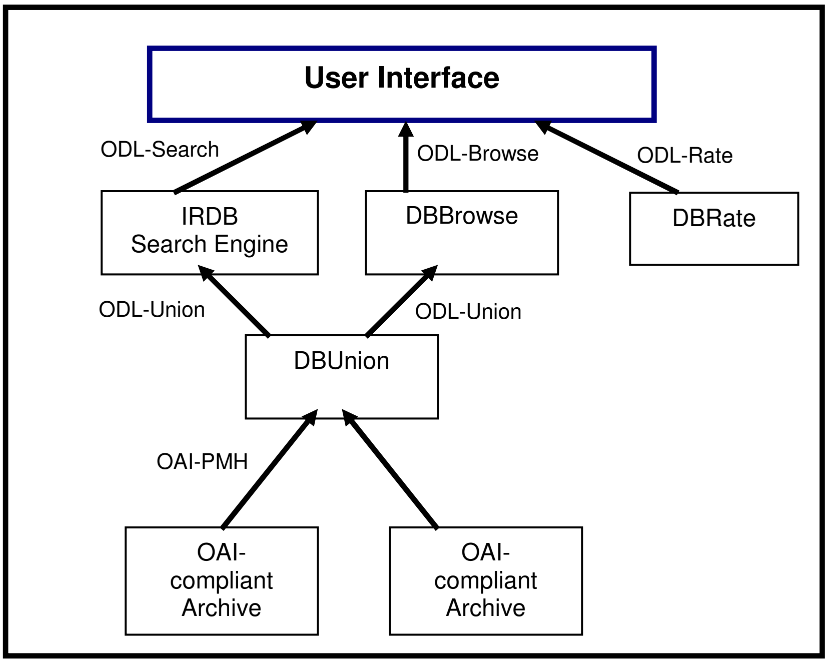


Figure 4.15 – A sample DL system consisting of several ODL components

Replacing the IRDB search engine component in the DL system depicted in Figure 4.15 requires that the new component understands the ODL-Union protocol, and the user interface understands its interfacing protocol. The conclusion derived from this observation is that for a component to function effectively amongst the ODL components, it must implement one of the ODL service protocols. Therefore, to demonstrate that other components could be interfaced to work with the server using the interfacing specification provided in Section 4.6, yet have that component function in conjunction with the ODL components, a new search engine component (Swish-E) was modified to emulate IRDB (i.e., implement the ODL-Search protocol).

The DBRate component, depicted in Figure 4.15, interacts directly with only the user interface. This suggests that other non-ODL components (components not supporting any of the ODL service protocols) can be introduced into the same system as long as they interact only with the user interface and the user interface can communicate with them. This led to the development of two other components. Firstly, a generic user interface that can be configured to accept a number of components, and a bulletin board (phpBB), which only interacts with that generic user interface. The following sections discuss the implementation of the Swish-E search engine, the phpBB bulletin board and the user interface component.

4.8.1 Swish-E

As mentioned in the background chapter, the Swish-E (Simple Web Indexing System for Humans—Enhanced) (Tennant, 2002) search engine, a descendant of SWISH, offers a unique combination of features that make it attractive for this DL component assembly research and a suitable alternative to ODL's IRDB. Some of Swish-E's offerings include: a fast and robust toolkit with which to build and query indices; good documentation; and active development and bug fixes. Swish-E is an executable that was designed to receive its configuration information as command-line arguments. This was unsuitable for this visual component assembly environment and so modifications had to be made in order for it to conform to the interface specification described in Section 4.6.

Indexing and searching with Swish-E involves creating a configuration file containing information on what to index and other indexing related directives, actually performing the indexing, and then searching the indexed data with specified search queries.

In order to make Swish-E a suitable DL search engine that works with the ODL components, it was modified to accept data transported using the OAI-PMH. This was achieved by specifying in the Swish-E configuration file that the intent was to use its PROG function. The Swish-E PROG function allows an external program to be run in order to obtain data to be indexed. The baseURL of the archive to be harvested from, metadata prefix, and optional set values are passed to the external program, which then harvests the records from the archive and stores it in a format that can be indexed by Swish-E. A search request to, and corresponding response from, a Swish-E instance was made to comply with ODL-Search protocol. So in effect, Swish-E was transformed into an ODL component in order for it to interface with other ODL components.

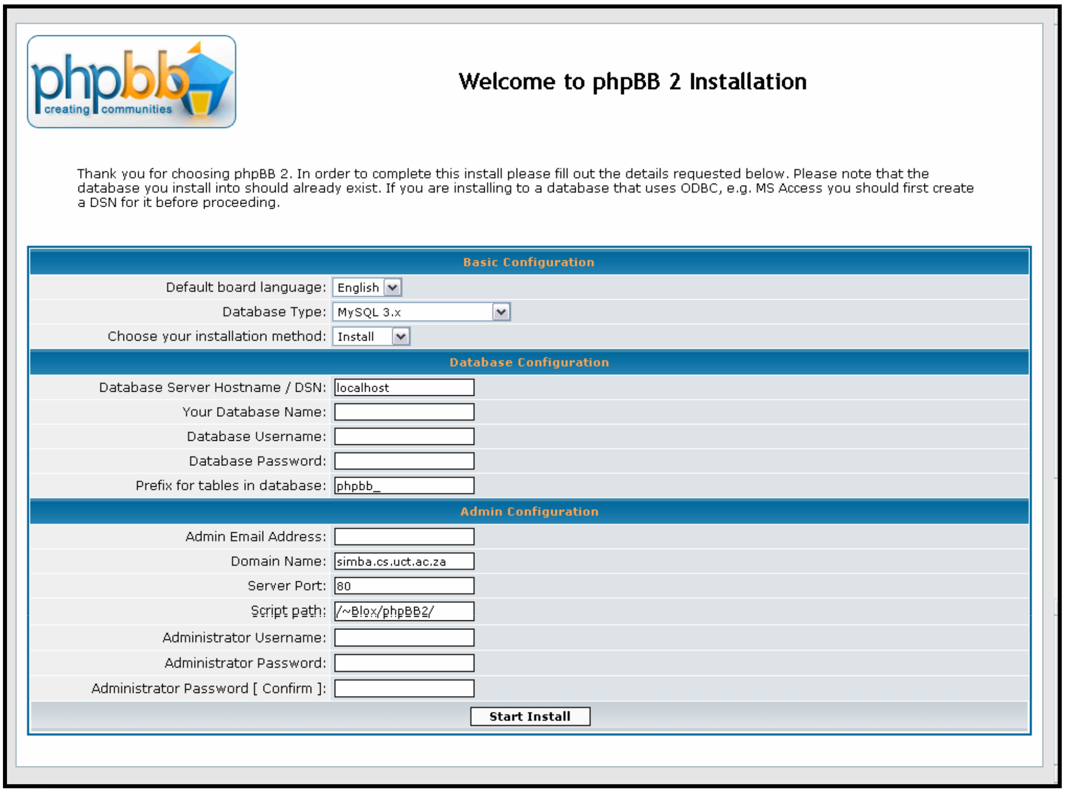
The autoconfig interface for Swish-E accepts an instance description containing the required information for an instance of the Swish-E search engine to be created. Swish-E's autoconfig interface creates a Swish-E configuration file, runs the external program so that the data to be indexed can be obtained with parameters obtained from the instance description, and then indexes the data. Autoconfig returns an error if indexing could not be performed or if any of the configuration details defined in the instance description were invalid.

The implementation of the getType, listInstances, getInstance and removeInstance interfaces was similar to the interfacing of the other ODL components outlined in Table 4.2 and described in Section 4.7.

4.8.2 PHPBB

The second non-ODL component interfaced was phpBB (2004) — a popular PHP-driven bulletin board. For phpBB to be included in a DL system, an instance of it must be created and that instance referenced in the DL user interface. The purpose of including phpBB in the component suite used in this experiment was not because it is imperative for a digital library to contain a bulletin board, but rather to demonstrate the possibility of augmenting the component suite with virtually any component that makes sense to include in a digital library.

PhpBB version 2.0.8 was used in this research, and was originally designed to receive its configuration information from an online form as depicted in Figure 4.16.



Welcome to phpBB 2 Installation

Thank you for choosing phpBB 2. In order to complete this install please fill out the details requested below. Please note that the database you install into should already exist. If you are installing to a database that uses ODBC, e.g. MS Access you should first create a DSN for it before proceeding.

Basic Configuration	
Default board language:	English
Database Type:	MySQL 3.x
Choose your installation method:	Install
Database Configuration	
Database Server Hostname / DSN:	localhost
Your Database Name:	
Database Username:	
Database Password:	
Prefix for tables in database:	phpbb_
Admin Configuration	
Admin Email Address:	
Domain Name:	simba.cs.uct.ac.za
Server Port:	80
Script path:	/~Blox/phpBB2/
Administrator Username:	
Administrator Password:	
Administrator Password [Confirm]:	

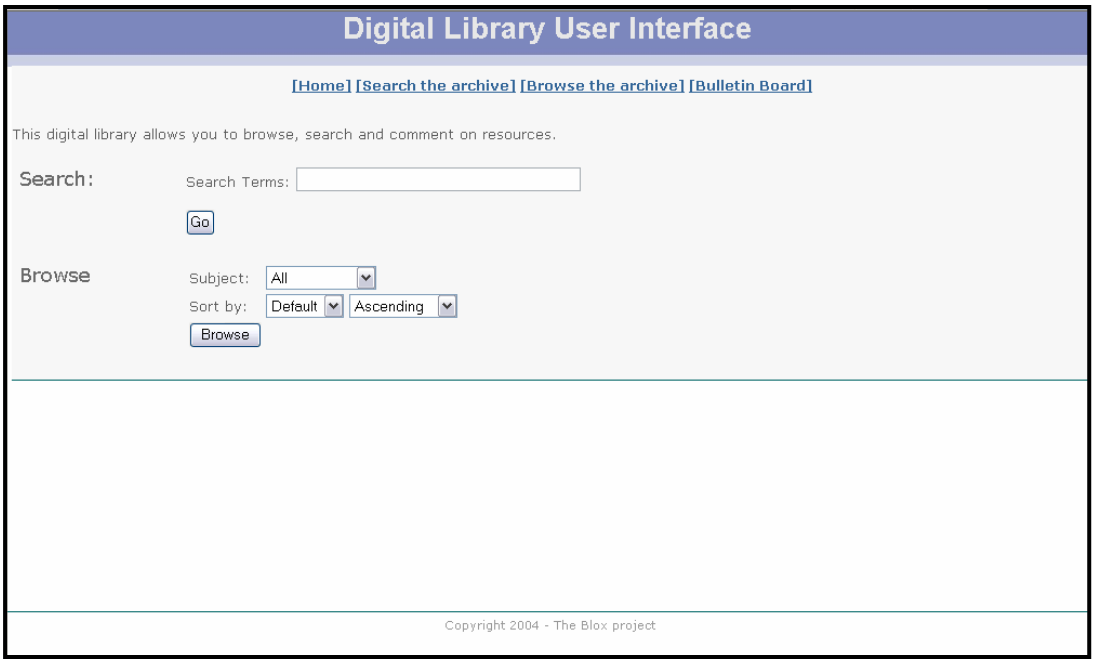
Figure 4.16 – The online configuration form for phpBB

For it to be included in the pool of components available to the GUI, it also needed to be properly interfaced according to the specification, that is, configuration had to be done by passing to autoconfig an instance description containing the configuration details.

Creating the autoconfig interface for phpBB involved modifying phpBB's original install file to accept configuration values from an XML file (the instance description) rather than the online form, and also ensuring that the results from autoconfig conforms to the specification. The other interfaces were created similarly to the ODL components.

4.8.3 A Digital Library User Interface

The original ODL component suite contains a simple user interface designed to work with the ODL-Search protocol. This UI provides an online textbox in which search keywords can be entered. This suffices for simple DLs; however, users may wish to incorporate other services such as browsing and rating into a single user interface. Furthermore, as the UI was not in itself a component, it was not reusable and each digital library required a separate installation of the UI. Therefore, a UI component was created to provide more interaction between users and the DL systems they design without requiring the user to be aware of any communication or implementation protocols like the OAI-PMH or HTML. This new UI component was made to interface with the ODL-Search, ODL-Browse, ODL-Rate, ODL-Recent, ODL-Annotation, ODL-Submit and ODL-Review service protocols. In addition to communicating with components that interface using the aforementioned protocols, the UI had to be in itself a fully configurable component with the ability to interface with additional components (such as phpBB) without requiring users to write any code. This is consistent with the aim of providing a simple way of creating DLs from distributed components. Figure 4.17 demonstrates an instance of the UI component configured to communicate with DBBrowse, DBRate and Swish-E, which interface using the ODL-Browse, ODL-Rate and ODL-Search protocols respectively, as well as phpBB. Figure 4.18 illustrates the interface to the DBRate – the result of clicking on one of the listed search query results.



Digital Library User Interface

[Home] [Search the archive] [Browse the archive] [Bulletin Board]

This digital library allows you to browse, search and comment on resources.

Search: Search Terms:

Go

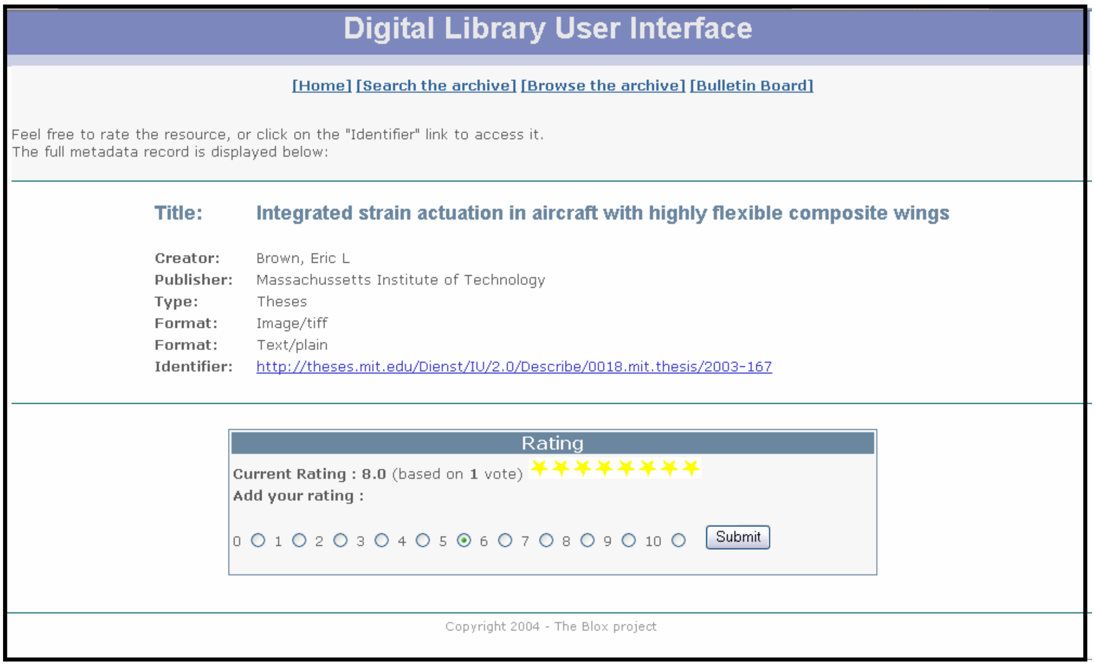
Browse Subject: All

Sort by: Default Ascending

Browse

Copyright 2004 - The Blox project

Figure 4.17 – An instance of the DL user interface component



Digital Library User Interface

[Home] [Search the archive] [Browse the archive] [Bulletin Board]

Feel free to rate the resource, or click on the "Identifier" link to access it.
The full metadata record is displayed below:

Title: Integrated strain actuation in aircraft with highly flexible composite wings

Creator:	Brown, Eric L
Publisher:	Massachussetts Institute of Technology
Type:	Theses
Format:	Image/tiff
Format:	Text/plain
Identifier:	http://theses.mit.edu/Dienst/IU/2.0/Describe/0018.mit.thesis/2003-167

Rating

Current Rating : 8.0 (based on **1** vote) ★★★★★★★★

Add your rating :

0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ● 6 ○ 7 ○ 8 ○ 9 ○ 10 ○

Copyright 2004 - The Blox project

Figure 4.18 – The UI interface to DBRate

Configuring the UI allows the user to specify details such as the text to be displayed on the header, footer and title bar, and also allows user to specify the components they would like to connect to the UI. Additionally, a connection can be made to any component with a baseURL that does not require any special processing by the UI as is the case with the phpBB component. This was implemented by including an optional but repeatable "service" tag in the UI's instance description. The service tag contains a "url" and "label" tag where the URL of the service to connect to can be specified along with the label that should be displayed in the UI. Each label is displayed as a link to the corresponding URL, aligned from left to right at the top of the page. This can be observed in Figure 4.18, where the phpBB instance's label is displayed as "Bulletin Board".

Interfacing the UI component according to the specification involved creating an autoconfig interface that accepts an instance description with the component's configuration details, creating a UI instance based on those details, and returning the results of the configuration process to standard output. The other interfaces (getType, listInstances, getInstance and removeInstance) were once again similar to the ones described for the ODL components in Section 4.7.

4.8.4 External Archives

When configuring certain ODL components, such as the search or browse engines, using the command line, one would normally have to supply the baseURL of the archive they are trying to connect to. However, because one of the reasons the GUI was designed was to prevent the user from having to type long baseURLs, a new method was required to provide the baseURLs of external archives, i.e., archives located anywhere on the Web. A new component called Open Archives Data Provider (OADP) was created, which contains the list of OAI-compliant archives available on the OAI website. The user simply selects the desired archive and links it to some other component such as a search engine. The OADP can be seen as a black box with exactly the same external interfaces as the other ODL components.

To implement the OADP component, a script was written that periodically downloads the list of existing OAI-compliant archives available from the OAI website and stores it in the component. Since no archive is actually being created, using the OADP component typically involves a call to the listInstances interface, which returns a list of all the instances of archives that are stored in the OADP component, formatted according to the listInstances specification. Users are not compelled to only use the archives available from the list. Users may use any archive of their choice by configuring OADP with the baseURL of the new archive. Autoconfig for OADP then tests the baseURL to ensure that it is in fact a valid archive, adds it to the list if it is, and returns a list of the sets and metadata prefixes belonging to that archive if the archive exposes its sets and metadata prefixes according to the OAI-PMH. Autoconfig usually returns a positive response, unless the archive provided in the instance description does not exist in the list of OAI compliant archives and cannot be added to the list because, upon inspection, the baseURL provided is deemed to be invalid.

The OADP component, used in conjunction with the Blox+ system, has been essential in enabling users to discover, and have access to, popular OAI-compliant repositories, without requiring users to remember long URLs.

4.9 Summary

This chapter provided an overview of the modifications done to the Blox Graphical User Interface and Server, and how it can be used to assemble and, to some extent, manage distributed digital library components. Subsequent sections described how creating a standard component interface facilitates the exposure of, and communication with, distributed components that comply with the specification. The implementation of the interface standard to some ODL components was discussed, and other components were created/modified and interfaced to demonstrate the applicability of the interface standard to a wide variety of components, hence creating a distributed and truly extensible component assembly framework.

Chapter 5

Case Studies

5.1 Introduction

One of the assertions made during the course of this work was that the Blox+ system made it possible to model various component-based digital library systems, differing in requirements, by simply selecting the appropriate components and linking them up. This section demonstrates how this was applied to three existing digital libraries varying in complexity and functionality.

5.2 NDLTD

5.2.1 Overview and Architecture

The Networked Digital Library of Theses and Dissertations (NDLTD) is an organisation that consists of a number of participating institutions, many of whom maintain their own collections of electronic theses and dissertations. The NDLTD union catalogue project's main aim is to improve graduate education through more effective and faster sharing of electronic resources (Fox, 1999).

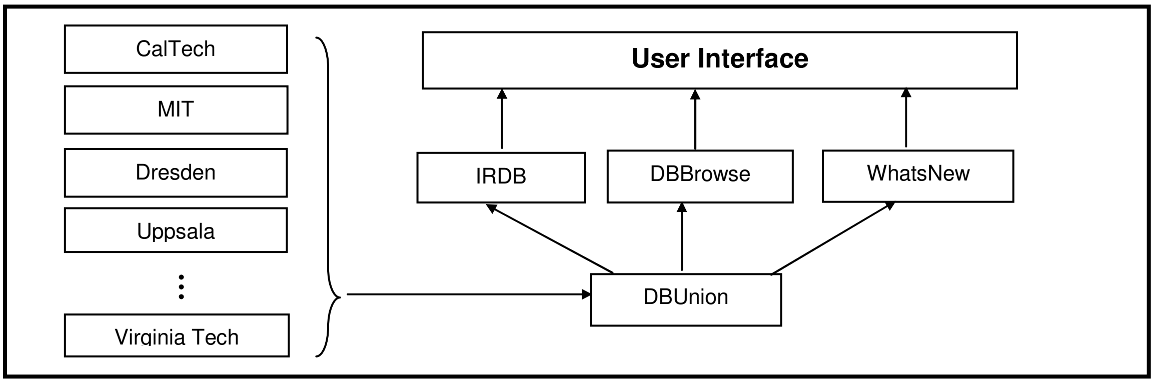
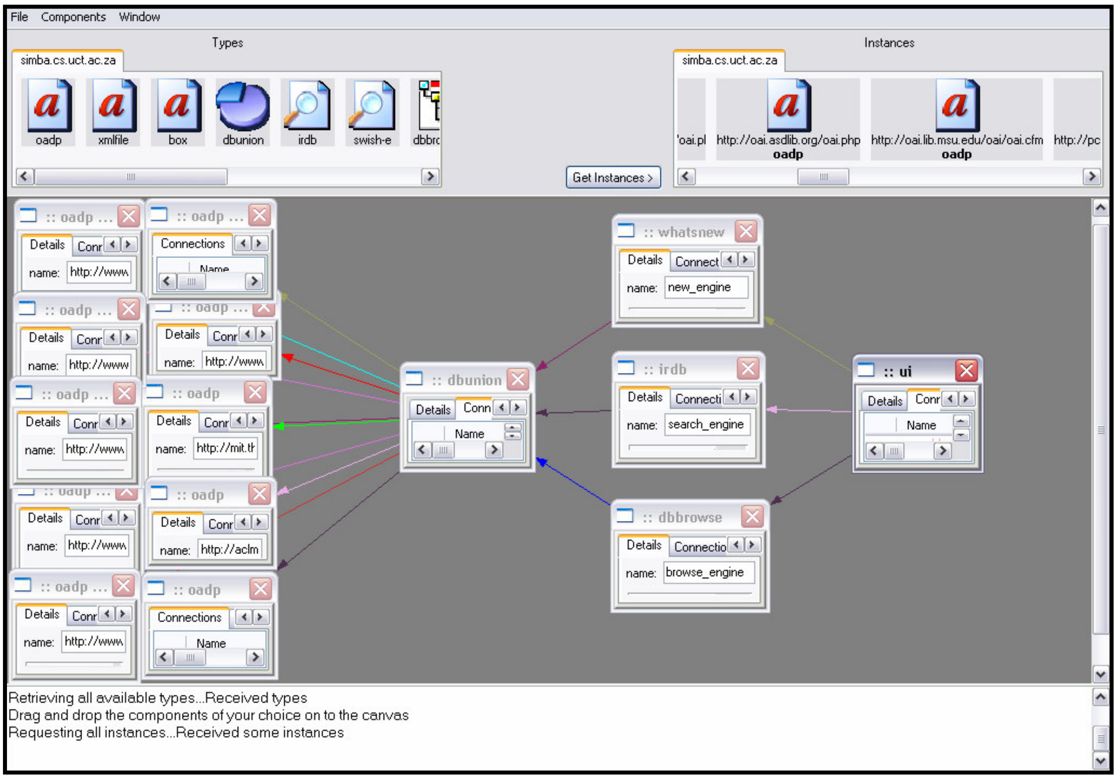


Figure 5.1 – Overview of the NDLTD architecture

The NDLTD architecture displayed in
Figure 5.1 consists of a number of ODL components. Metadata from participating institutions are periodically harvested and merged into a single archive by means of the DBUnion component. The IRDB and DBBrowse service components allow users to respectively search and browse through harvested data, hence serving as a centralised means of accessing the metadata collected in several institutional repositories. The WhatsNew component layers on top of DBUnion to service the User Interface with the most recent additions to the union archive.

5.2.2 Implementation

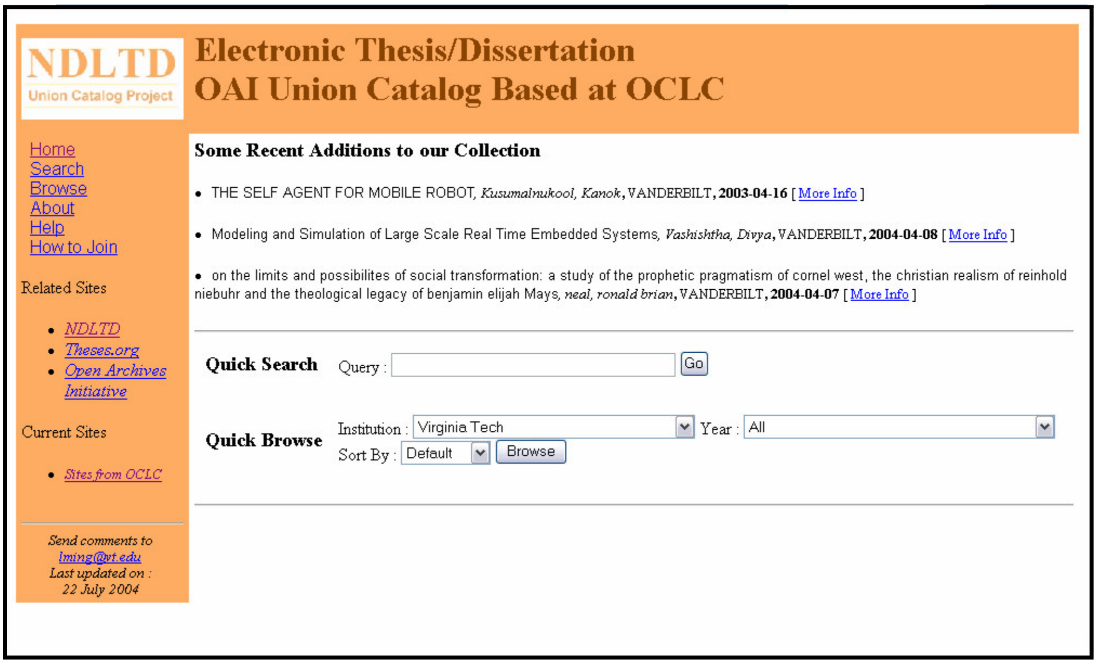
Figure 5.2 illustrates how the NDLTD system was created using ODL components. The archives are displayed on the left, each connected to the DBUnion component, while DBBrowse, IRDB and WhatsNew are linked to the DBUnion component and the UI component.



Retrieving all available types...Received types
Drag and drop the components of your choice on to the canvas
Requesting all instances...Received some instances

Figure 5.2 – Creation of NDLTD using Blox+

The specific functionality accessible through the DL’s Web interface once the DL system is published depends on the components that were included during its design. In the case of this particular DL system, the start up page displays the most recent entries into the archive component (the archive in question in this instance is the DBUnion component) as well as browsing and searching capabilities. Figure 5.3 illustrates the original NDLTD user interface, while Figure 5.4 illustrates the Web interface generated with Blox+.



Electronic Thesis/Dissertation
OAI Union Catalog Based at OCLC

Home
Search
Browse
About
Help
How to Join

Related Sites

- NDLTD
- Theses.org
- Open Archives Initiative

Current Sites

- Sites from OCLC

Send comments to
lming@vt.edu
Last updated on :
22 July 2004

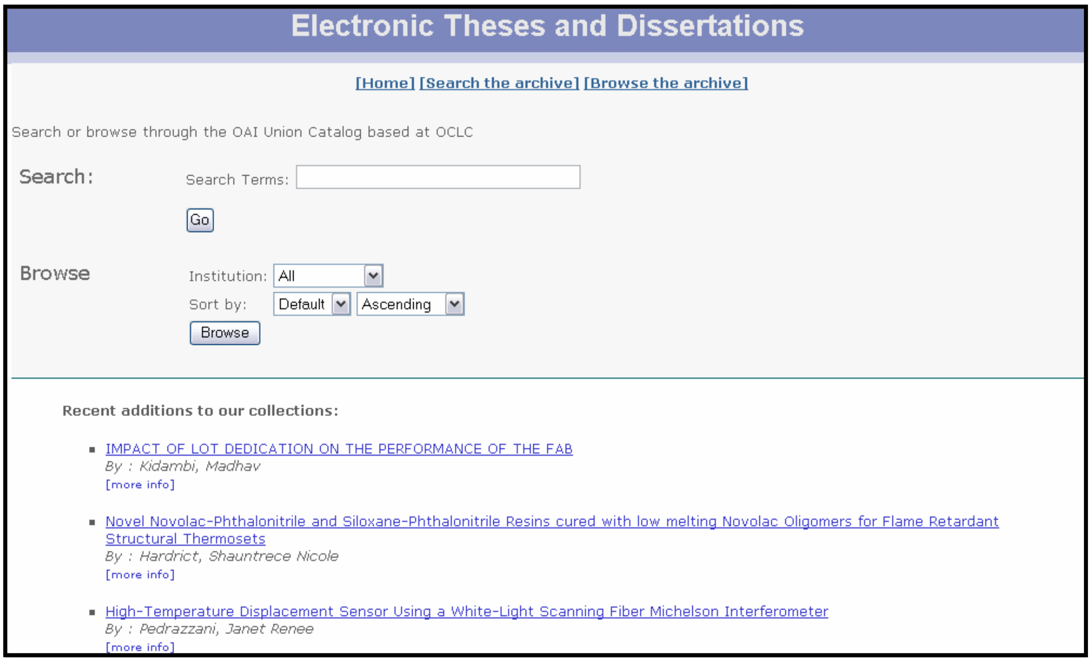
Some Recent Additions to our Collection

- THE SELF AGENT FOR MOBILE ROBOT, *Kusumalnukool, Kanok*, VANDERBILT, **2003-04-16** [More Info]
- Modeling and Simulation of Large Scale Real Time Embedded Systems, *Vashishtha, Divya*, VANDERBILT, **2004-04-08** [More Info]
- on the limits and possibilites of social transformation: a study of the prophetic pragmatism of cornel west, the christian realism of reinhold niebuhr and the theological legacy of benjamin elijah Mays, *neal, ronald brian*, VANDERBILT, **2004-04-07** [More Info]

Quick Search Query : Go

Quick Browse Institution : Virginia Tech Year : All
Sort By : Default Browse

Figure 5.3 – The NDLTD OAI Union Catalogue User Interface



Electronic Theses and Dissertations

[Home] [Search the archive] [Browse the archive]

Search or browse through the OAI Union Catalog based at OCLC

Search: Search Terms:
Go

Browse Institution: All
Sort by: Default Ascending
Browse

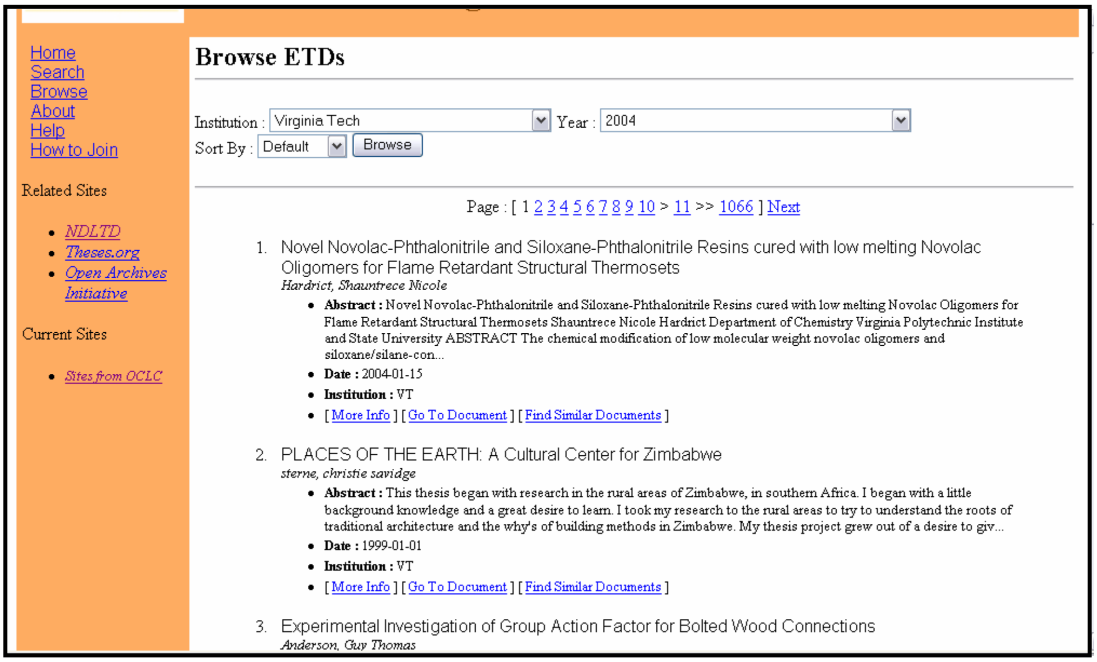
Recent additions to our collections:

- IMPACT OF LOT DEDICATION ON THE PERFORMANCE OF THE FAB
  By : Kidambi, Madhav
  [more info]
- Novel Novolac-Phthalonitrile and Siloxane-Phthalonitrile Resins cured with low melting Novolac Oligomers for Flame Retardant Structural Thermosets
  By : Hardrict, Shauntrece Nicole
  [more info]
- High-Temperature Displacement Sensor Using a White-Light Scanning Fiber Michelson Interferometer
  By : Pedrazzani, Janet Renee
  [more info]

Figure 5.4 – The remodelled version of NDLTD OAI Union Catalogue

The focus here was to demonstrate that the functionality of the NDLTD system could be duplicated by Blox+ and not to ensure that the two looked identical. The layout and colour scheme could be modified by supplying the user interface with a different CSS stylesheet.

The output of a typical browse operation with the NDLTD system is illustrated in Figure 5.5, and the modelled version is depicted in Figure 5.6. The UI component automatically generates the categories available for sorting and browsing based on the configuration of the DBBrowse component.



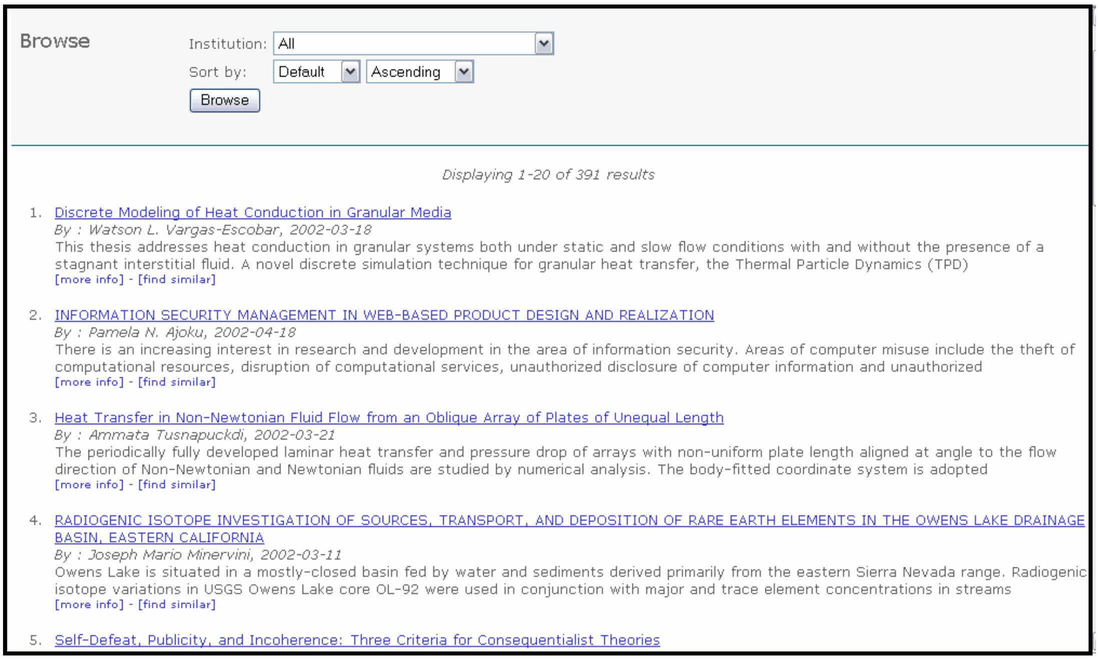
Browse ETDs

Institution : Virginia Tech Year : 2004
Sort By : Default

Page : [1 2 3 4 5 6 7 8 9 10 > 11 >> 1066] Next

1. Novel Novolac-Phthalonitrile and Siloxane-Phthalonitrile Resins cured with low melting Novolac Oligomers for Flame Retardant Structural Thermosets
   Hardrict, Shauntrece Nicole
   - **Abstract :** Novel Novolac-Phthalonitrile and Siloxane-Phthalonitrile Resins cured with low melting Novolac Oligomers for Flame Retardant Structural Thermosets Shauntrece Nicole Hardrict Department of Chemistry Virginia Polytechnic Institute and State University ABSTRACT The chemical modification of low molecular weight novolac oligomers and siloxane/silane-con...
   - **Date :** 2004-01-15
   - **Institution :** VT
   - [More Info] [Go To Document] [Find Similar Documents]
2. PLACES OF THE EARTH: A Cultural Center for Zimbabwe
   sterne, christie savidge
   - **Abstract :** This thesis began with research in the rural areas of Zimbabwe, in southern Africa. I began with a little background knowledge and a great desire to learn. I took my research to the rural areas to try to understand the roots of traditional architecture and the why's of building methods in Zimbabwe. My thesis project grew out of a desire to giv...
   - **Date :** 1999-01-01
   - **Institution :** VT
   - [More Info] [Go To Document] [Find Similar Documents]
3. Experimental Investigation of Group Action Factor for Bolted Wood Connections
   Anderson, Guy Thomas

Figure 5.5 – Output from a browsing operation on the NDLTD OAI Union Catalogue



Institution: All
Sort by: Default Ascending

Displaying 1-20 of 391 results

1. Discrete Modeling of Heat Conduction in Granular Media
   By : Watson L. Vargas-Escobar, 2002-03-18
   This thesis addresses heat conduction in granular systems both under static and slow flow conditions with and without the presence of a stagnant interstitial fluid. A novel discrete simulation technique for granular heat transfer, the Thermal Particle Dynamics (TPD)
   [more info] - [find similar]
2. INFORMATION SECURITY MANAGEMENT IN WEB-BASED PRODUCT DESIGN AND REALIZATION
   By : Pamela N. Ajoku, 2002-04-18
   There is an increasing interest in research and development in the area of information security. Areas of computer misuse include the theft of computational resources, disruption of computational services, unauthorized disclosure of computer information and unauthorized
   [more info] - [find similar]
3. Heat Transfer in Non-Newtonian Fluid Flow from an Oblique Array of Plates of Unequal Length
   By : Ammata Tusnapuckdi, 2002-03-21
   The periodically fully developed laminar heat transfer and pressure drop of arrays with non-uniform plate length aligned at angle to the flow direction of Non-Newtonian and Newtonian fluids are studied by numerical analysis. The body-fitted coordinate system is adopted
   [more info] - [find similar]
4. RADIOGENIC ISOTOPE INVESTIGATION OF SOURCES, TRANSPORT, AND DEPOSITION OF RARE EARTH ELEMENTS IN THE OWENS LAKE DRAINAGE BASIN, EASTERN CALIFORNIA
   By : Joseph Mario Minervini, 2002-03-11
   Owens Lake is situated in a mostly-closed basin fed by water and sediments derived primarily from the eastern Sierra Nevada range. Radiogenic isotope variations in USGS Owens Lake core OL-92 were used in conjunction with major and trace element concentrations in streams
   [more info] - [find similar]
5. Self-Defeat, Publicity, and Incoherence: Three Criteria for Consequentialist Theories

Figure 5.6 – Browsing with the modelled version of NDLTD OAI Union Catalogue

5.3 CITIDEL

The Computing and Information Technology Interactive Digital Educational Library (CITIDEL) was built by a consortium of several universities as the part of the National Science, Technology, Engineering and Mathematics Digital Library (NSDL) (Lagoze et al., 2002a) in order to "serve the computing education community in all its diversity and at all levels" (Fox et al., 2002) to the end of providing integrated access to all related collections. This makes it possible for contributors to retain ownership and control of their material, while making them widely available through CITIDEL browsing and searching operations. Visitors are able to contribute to CITIDEL in a number of ways:

- Commenting on existing resources
- Developing a collection and making it a member collection of CITIDEL
- Submitting a resource to be archived at the CITIDEL site
- Developing class plans, laboratory exercises, projects, homework assignments, and other course elements and publishing them in CITIDEL

All this goes to ensure that the individuals and institutions access, and contribute to, CITIDEL collections for the benefit of others.

5.3.1 Architecture

From the start, CITIDEL was designed to be a componentised system that could easily evolve to keep up with changing requirements. CITIDEL's current architecture consists of a number of components and code modules that together provide the required functionality to access metadata-serving archives including, but not limited to, NCSTRL, CSTC, ACM and NSDL. Figure 5.7 illustrates a simplified view of the current CITIDEL architecture.

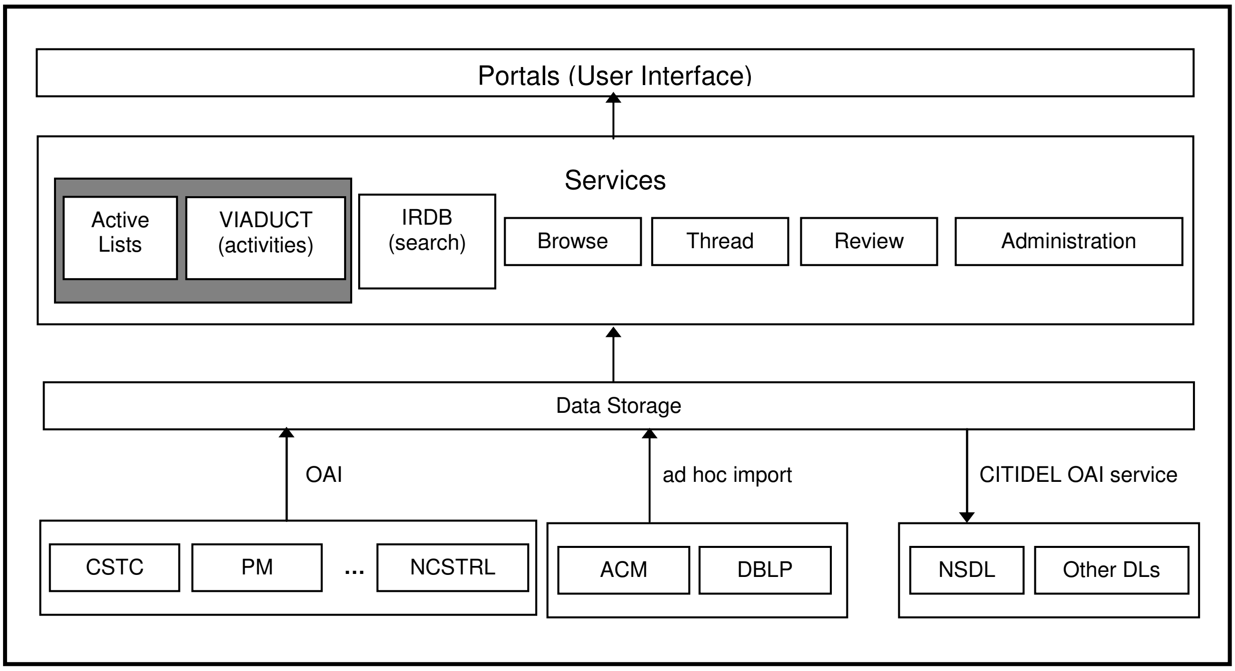


Figure 5.7 – Overview of the CITIDEL architecture

CITIDEL uses components from multiple vendors. The only ODL components it includes are the IRDB and Thread components, which provide searching and annotation services respectively. The shaded area represents two specialised CITIDEL services: The CITIDEL Active Lists service allows the user to describe and sequence resources from CITIDEL, annotate this sequence, and then organise them into ordered or unordered lists, guided paths, or slideshows to create engaging and interactive Web pages. The Virginia Instructional Architect for Digital Undergraduate Computing Teaching (VIADUCT) service, an adaptation of the Instructional Architect (IA) (2005), is a lesson-plan oriented tool for making instructional activities. An instructional activity can be any type of activity that would be useful in teaching. Resources that are useful for an assignment, such as background reading or references, are then located from the CITIDEL collection and attached to the activities to create lesson plans. CITIDEL implements a relatively simple peer-review workflow. Submitted items are either accepted or rejected. Upon acceptance they immediately become part of the CITIDEL collection and other services can operate on this new data. Resources are harvested into CITDEL from a number of archives using the OAI-PMH whenever possible; otherwise, it uses different import methods to obtain data from non OAI-compliant archives such as ACM and DBLP.

5.3.2 Implementation

In modelling CITIDEL, the emphasis was placed on simulating the provision and dissemination of information activities that occur through the CITIDEL Web portal. This involved ensuring that all users can access resources by entering a search query, or browsing by subject or source collection. Additionally, authenticated users should be able to submit a new item to the CITIDEL collection. The basic design of the CITIDEL system was achieved by augmenting the NDLTD model to include login, submission and threading services. Authentication was achieved by connecting a Box component (which stores user information) to the user interface, while the ability to comment on any given resource was made possible with the inclusion of the ODL Thread component. Figure 5.8 displays the conceptual design of the CITIDEL model that was implemented using Blox+, with "A" representing the CITIDEL member collections.

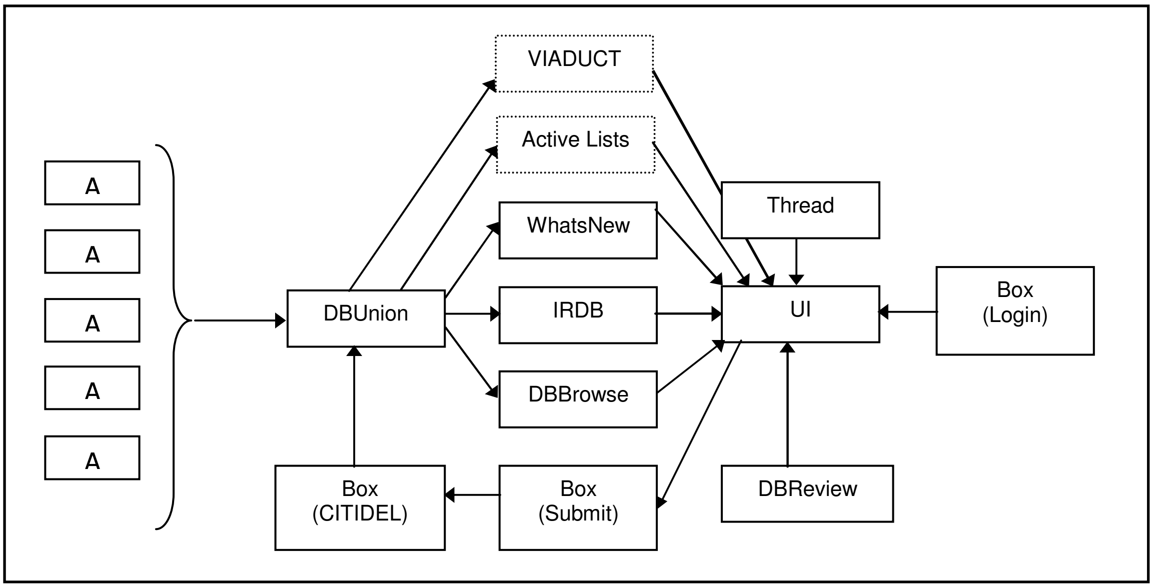


Figure 5.8 – Overview of the CITIDEL model conceptual architecture

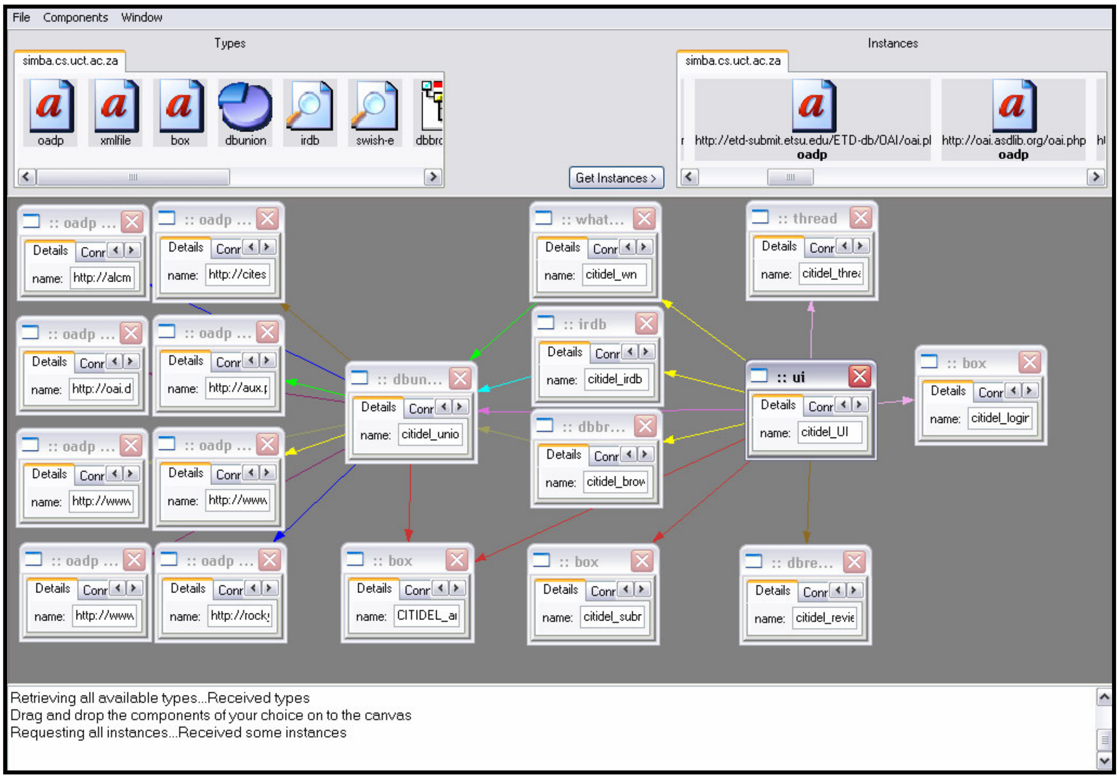
The diagram above displays a WhatsNew component, even though there was no indication of such a component in Figure 5.7. This is because the administrators of CITIDEL manually update the Web interface with all the new updates to the CITIDEL system. These updates are not limited to resources, but also include new Web pages added to the site, activities, tutorials, infrastructure related improvements, etc. But for simplicity, the WhatsNew component was used to display only the most recent resources.

The Active Lists and VIADUCT services are represented in Figure 5.8 as dotted lines because they were not included in final implementation of the CITIDEL model. This was because the ability to apply the Autoconfigure Interface Specification to non-ODL and non-OAI components has already been demonstrated by interfacing PhpBB and the Swish-E search engine as was described in Section 4.8, and as such were considered beyond the scope of this research. Incorporating these two CITIDEL services in the model could be accomplished by undertaking the following steps:

1. Interfacing the components according to the specification in Section 4.8.

2. Ensuring that they are OAI-PMH compliant in order for them to interact with the archive (DBUnion) and the UI to permit resource discovery.
3. Configuring the UI to indicate that these components require authentication before full access can be granted, and ensuring that session state information can be transmitted from the UI to the components.
4. Because these components will have their own Web interface, for consistency, it is important to ensure that the look and feel of the UI is reflected in the Web interface of the components.

New submissions are placed into a Box component and notifications are sent to the DBReview component. When a submission is accepted, it is moved to another Box component (labelled "CITIDEL" in Figure 5.8) after which it becomes part of the CITIDEL collection. The DBReview component is constantly updated to reflect changes in the resource's status. Figure 5.8 illustrates how the conceptual architecture above was represented in Blox+.



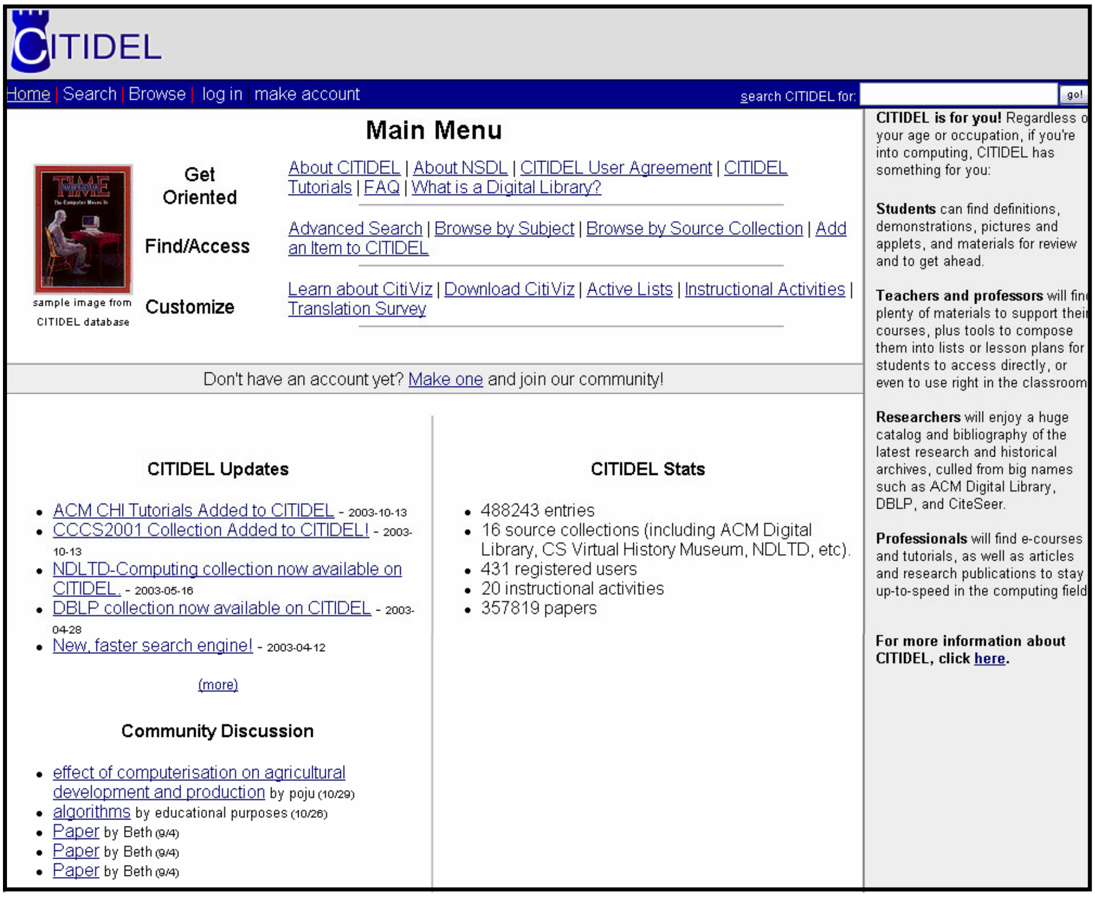
:: oadp ... name: http://alcm
:: oadp ... name: http://cites
:: what... name: citidel_wn
:: thread name: citidel_threa
:: oadp ... name: http://oai.d
:: oadp ... name: http://aux.p
:: dbun... name: citidel_unio
:: irdb name: citidel_irdb
:: ui name: citidel_UI
:: box name: citidel_logir
:: oadp ... name: http://www
:: oadp ... name: http://www
:: dbbr... name: citidel_brow
:: oadp ... name: http://www
:: oadp ... name: http://rocky
:: box name: CITIDEL_ai
:: box name: citidel_subr
:: dbre... name: citidel_revie

Retrieving all available types...Received types
Drag and drop the components of your choice on to the canvas
Requesting all instances...Received some instances

Figure 5.9 – Designing CITIDEL using Blox+

The same user interface used to create NDLTD was used in modelling CITIDEL. Connecting a Box component to the UI's 'usersbaseurl' field ensures that the UI component is aware that specific tasks, such as submitting a new resource, require that users be authenticated first. Submitted resources are first stored in a Box component connected to the 'submitbaseurl' field of the UI and to DBUnion component which periodically harvests those new submissions. Figure 5.9 only shows 8 archives being harvest into the Union component because only 8 of CITIDEL's member collections expose their metadata via the OAI-PMH.

Figure 5.10 depicts the home page of CITIDEL's web interface and Figure 5.11 illustrates the modelled version.



Main Menu

sample image from CITIDEL database

Get Oriented About CITIDEL | About NSDL | CITIDEL User Agreement | CITIDEL Tutorials | FAQ | What is a Digital Library?

Find/Access Advanced Search | Browse by Subject | Browse by Source Collection | Add an Item to CITIDEL

Customize Learn about CitiViz | Download CitiViz | Active Lists | Instructional Activities | Translation Survey

Don't have an account yet? Make one and join our community!

CITIDEL Updates

- ACM CHI Tutorials Added to CITIDEL - 2003-10-13
- CCCS2001 Collection Added to CITIDEL! - 2003-10-13
- NDLTD-Computing collection now available on CITIDEL. - 2003-05-16
- DBLP collection now available on CITIDEL - 2003-04-28
- New, faster search engine! - 2003-04-12

(more)

Community Discussion

- effect of computerisation on agricultural development and production by poju (10/29)
- algorithms by educational purposes (10/26)
- Paper by Beth (9/4)
- Paper by Beth (9/4)
- Paper by Beth (9/4)

CITIDEL Stats

- 488243 entries
- 16 source collections (including ACM Digital Library, CS Virtual History Museum, NDLTD, etc).
- 431 registered users
- 20 instructional activities
- 357819 papers

CITIDEL is for you! Regardless o your age or occupation, if you're into computing, CITIDEL has something for you:

Students can find definitions, demonstrations, pictures and applets, and materials for review and to get ahead.

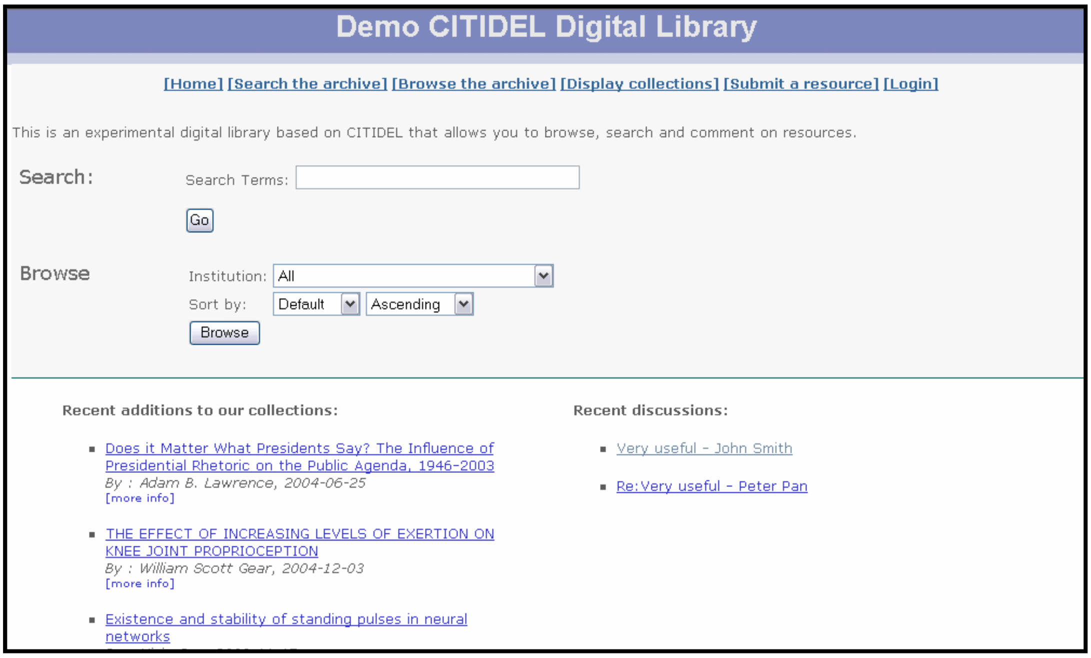
Teachers and professors will fin plenty of materials to support thei courses, plus tools to compose them into lists or lesson plans for students to access directly, or even to use right in the classroom

Researchers will enjoy a huge catalog and bibliography of the latest research and historical archives, culled from big names such as ACM Digital Library, DBLP, and CiteSeer.

Professionals will find e-courses and tutorials, as well as articles and research publications to stay up-to-speed in the computing field

For more information about CITIDEL, click here.

Figure 5.10 – CITIDEL's Web interface



Demo CITIDEL Digital Library

[Home] [Search the archive] [Browse the archive] [Display collections] [Submit a resource] [Login]

This is an experimental digital library based on CITIDEL that allows you to browse, search and comment on resources.

Search: Search Terms: [ ] Go

Browse Institution: All
Sort by: Default Ascending
Browse

Recent additions to our collections:

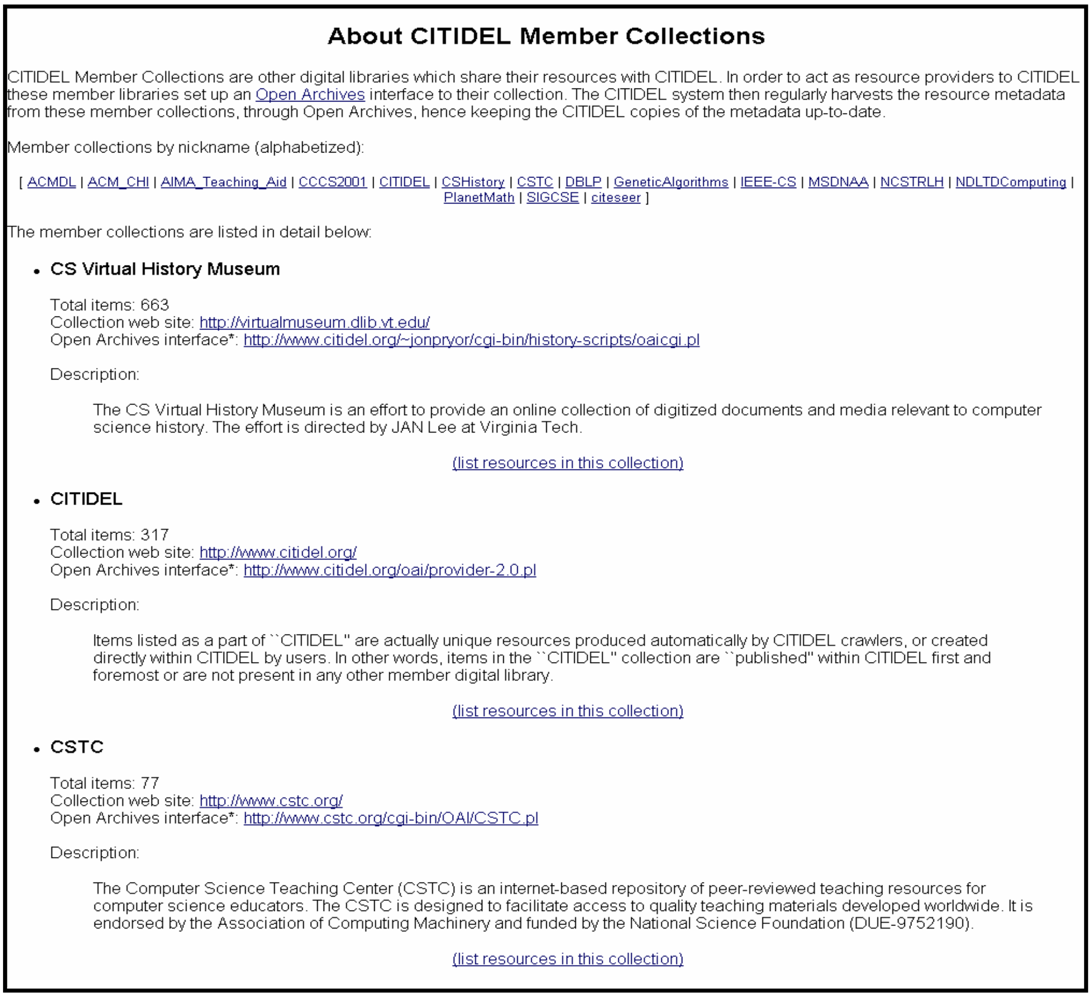
- Does it Matter What Presidents Say? The Influence of Presidential Rhetoric on the Public Agenda, 1946-2003
  By : Adam B. Lawrence, 2004-06-25
  [more info]
- THE EFFECT OF INCREASING LEVELS OF EXERTION ON KNEE JOINT PROPRIOCEPTION
  By : William Scott Gear, 2004-12-03
  [more info]
- Existence and stability of standing pulses in neural networks

Recent discussions:

- Very useful - John Smith
- Re:Very useful - Peter Pan

Figure 5.11 – The modelled version of CITIDEL

On CITIDEL's home page, the most recent discussions are available, which when clicked, direct the user to a page displaying the resource's metadata and comments in the form of annotations. Users are then able to reply to a comment or post a new one. Clicking on "Browse by Source Collection" on the index page results in a page displaying all available collections, including information pertaining to those collections and a link allowing users to search or browse through that particular collection. This is illustrated in Figure 5.12 and the model equivalent in Figure 5.13.



About CITIDEL Member Collections

CITIDEL Member Collections are other digital libraries which share their resources with CITIDEL. In order to act as resource providers to CITIDEL these member libraries set up an Open Archives interface to their collection. The CITIDEL system then regularly harvests the resource metadata from these member collections, through Open Archives, hence keeping the CITIDEL copies of the metadata up-to-date.

Member collections by nickname (alphabetized):

[ACMDL | ACM_CHI | AIMA_Teaching_Aid | CCCS2001 | CITIDEL | CSHistory | CSTC | DBLP | GeneticAlgorithms | IEEE-CS | MSDNAA | NCSTRLH | NDLTDComputing | PlanetMath | SIGCSE | citeseer]

The member collections are listed in detail below:

- **CS Virtual History Museum**

  Total items: 663
  Collection web site: http://virtualmuseum.dlib.vt.edu/
  Open Archives interface*: http://www.citidel.org/~jonpryor/cgi-bin/history-scripts/oaicgi.pl

  Description:

  > The CS Virtual History Museum is an effort to provide an online collection of digitized documents and media relevant to computer science history. The effort is directed by JAN Lee at Virginia Tech.

  (list resources in this collection)

- **CITIDEL**

  Total items: 317
  Collection web site: http://www.citidel.org/
  Open Archives interface*: http://www.citidel.org/oai/provider-2.0.pl

  Description:

  > Items listed as a part of ``CITIDEL'' are actually unique resources produced automatically by CITIDEL crawlers, or created directly within CITIDEL by users. In other words, items in the ``CITIDEL'' collection are ``published'' within CITIDEL first and foremost or are not present in any other member digital library.

  (list resources in this collection)

- **CSTC**

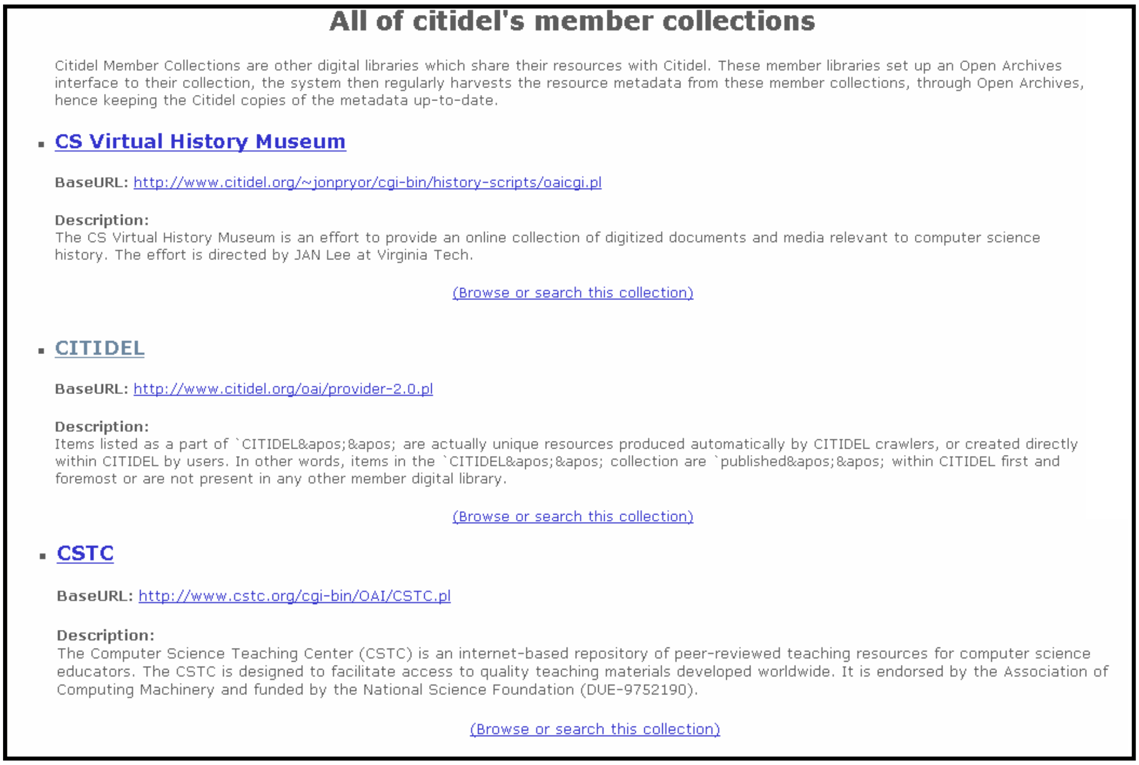
  Total items: 77
  Collection web site: http://www.cstc.org/
  Open Archives interface*: http://www.cstc.org/cgi-bin/OAI/CSTC.pl

  Description:

  > The Computer Science Teaching Center (CSTC) is an internet-based repository of peer-reviewed teaching resources for computer science educators. The CSTC is designed to facilitate access to quality teaching materials developed worldwide. It is endorsed by the Association of Computing Machinery and funded by the National Science Foundation (DUE-9752190).

  (list resources in this collection)

Figure 5.12 – Browsing through collections with CITIDEL



All of citidel's member collections

Citidel Member Collections are other digital libraries which share their resources with Citidel. These member libraries set up an Open Archives interface to their collection, the system then regularly harvests the resource metadata from these member collections, through Open Archives, hence keeping the Citidel copies of the metadata up-to-date.

- CS Virtual History Museum

  BaseURL: http://www.citidel.org/~jonpryor/cgi-bin/history-scripts/oaicgi.pl

  Description:
  The CS Virtual History Museum is an effort to provide an online collection of digitized documents and media relevant to computer science history. The effort is directed by JAN Lee at Virginia Tech.

  (Browse or search this collection)

- CITIDEL

  BaseURL: http://www.citidel.org/oai/provider-2.0.pl

  Description:
  Items listed as a part of `CITIDEL&apos;&apos; are actually unique resources produced automatically by CITIDEL crawlers, or created directly within CITIDEL by users. In other words, items in the `CITIDEL&apos;&apos; collection are `published&apos;&apos; within CITIDEL first and foremost or are not present in any other member digital library.

  (Browse or search this collection)

- CSTC

  BaseURL: http://www.cstc.org/cgi-bin/OAI/CSTC.pl

  Description:
  The Computer Science Teaching Center (CSTC) is an internet-based repository of peer-reviewed teaching resources for computer science educators. The CSTC is designed to facilitate access to quality teaching materials developed worldwide. It is endorsed by the Association of Computing Machinery and funded by the National Science Foundation (DUE-9752190).

  (Browse or search this collection)

Figure 5.13 – Browsing through collections with the CITIDEL model

5.4 DSpace

5.4.1 Overview

As has been discussed in previous chapters, DSpace is a customisable digital library system that allows users to capture and describe digital works using a custom workflow process. DSpace is a joint development effort by MIT Libraries and Hewlett-Packard (HP) in order to produce a freely available extensible digital library system to research institutions worldwide.

5.4.1.1 Data Model

For modelling purposes, it is important to understand the way in which content is organised in DSpace. At the highest level, content is organised into communities that correspond with organisational bodies such as departments, labs, research centres or schools. Each of these communities is organised into collections of logically related material. These collections, in turn, consist of items that are logical groupings of content and metadata that make sense to archive as a single unit. Each item can comprise of several files called bitstreams.

5.4.1.2 Submission

A major aspect of DSpace is its content submission procedure. DSpace implements the concept of personal workspaces to assist in the process of submitting new resources. The DSpace Web interface guides users through the submission process through a series of interactive steps. Users can commence the process for several resources simultaneously, save the current states of those submissions and return at a later date to complete the process. Once the steps have been completed and a resource is committed to the DSpace archive, the system initiates the review workflow associated with the collection the user submitted to. Each community determines the various workflows for reviewing the resources submitted to its constituent collections. DSpace has several types of users who participate in the process of resource creation and review. All users can perform browsing and searching operations on the DSpace repository, but only registered users can submit new resources by including the URL of the resource, or by uploading the relevant files. Table 5.1 tabulates the various types of users and their roles.

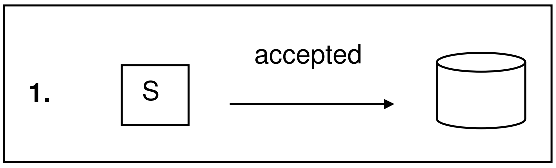
Role	Permissions
Submitter	Submit metadata and upload files
Reviewer	Review content of all files submitted to a collection Accept or reject submissions Cannot edit metadata
Coordinator	Can edit metadata of all submissions to the collection they have been assigned Can accept or reject all submissions to collection
Metadata Editor	Can edit metadata of all submissions to collection Submission automatically becomes part of DSpace after this step

Table 5.1 – Submission and peer review participants

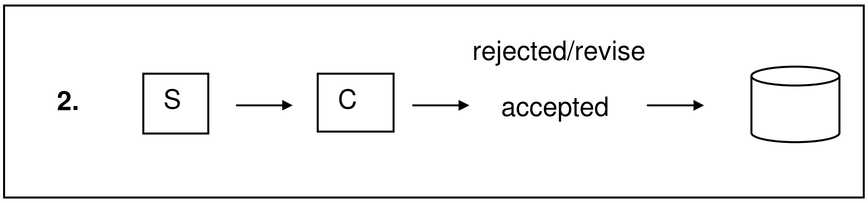
5.4.1.3 Workflow

DSpace communities can specify one of five possible workflows for their collections. The different workflows are illustrated below with S representing the Submitter, C, the Coordinator, R, the Reviewer and E, the Metadata Editor.

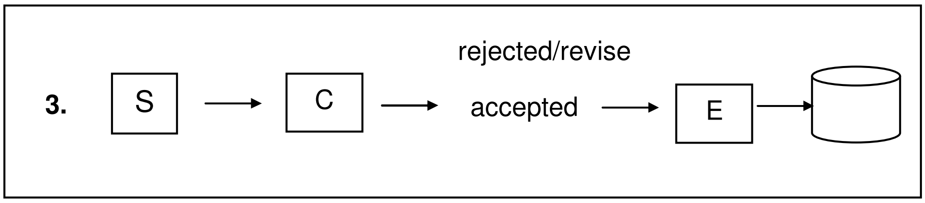
In the simplest case (1.), all submissions are directly accepted into the collection.



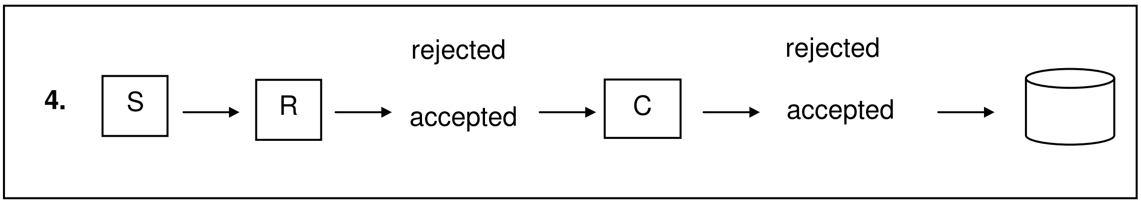
In the Submitter-Coordinator (2.) workflow submissions are accepted, rejected or revised by a collection's Coordinator before being accepted into the collection.



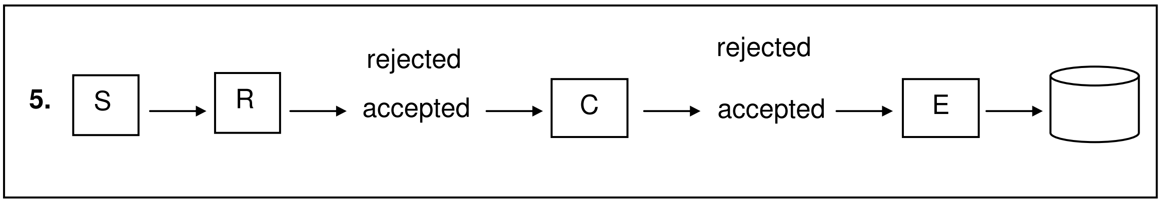
The third workflow builds on top of the second workflow by including a metadata editor at the last stage of the process whose sole responsibility is to edit the accepted resource's metadata.



The fourth possible workflow includes a reviewer who writes a review for the submission and then decides whether to accept or reject the submission. Acceptance moves the resource on to the next phase of the workflow where it may still be rejected by the coordinator.



The final and most complex workflow for DSpace augments the fourth to incorporate the Metadata Editor.



At each stage notifications are sent to the parties who are directly concerned informing them of their duties. Participants of the reviewing process are able to perform the relevant activities through the Web interface.

5.4.2 Architecture

In addition to the submission service, DSpace contains a number of other services that are essential for resource discovery. The Lucene search engine component enables users to search the entire DSpace repository or a specific collection. DSpace also allows users to browse their communities' resources by date, title and author. Results can also be limited to specific collections. Figure 5.14 illustrates the architecture of DSpace adapted from Tansley et al. (2003).

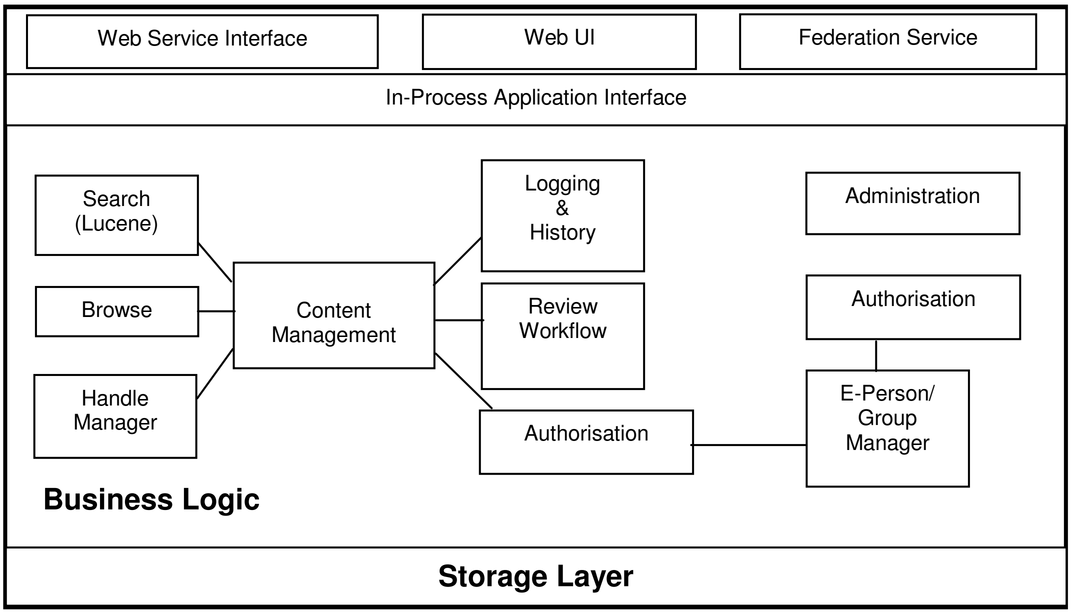


Figure 5.14 – DSpace architecture

The E-Person component of DSpace is responsible for managing information about users. This, along with the authorisation component, ensures that only authenticated users are able to use the submission, email notification (subscription) or administration services. The handle manager ensures that each DSpace resource, collection and community is supplied with a persistent identifier to provide a storage and location independent mechanism for locating any given resource. It does so by using the CNRI Handle System (CNRI, 2005).

5.4.3 Implementation

In modelling DSpace, emphasis was placed on demonstrating the possibility of modelling DSpace's concept of communities, collections and the various different workflows by using the Blox+ system.

The User Interface component plays a large role in determining the nature and complexity of the digital libraries generated by the Blox+ system. The UI was augmented to allow submissions to be made to specific communities and collections. Hence for a UI to function effectively in this multi-purpose manner, it is essential to be easily able to switch the UI from a basic NDLTD-like configuration to a DSpace-like system by simply toggling certain configuration fields. The conceptual design of DSpace using ODL components and a UI component is depicted in Figure 5.15.

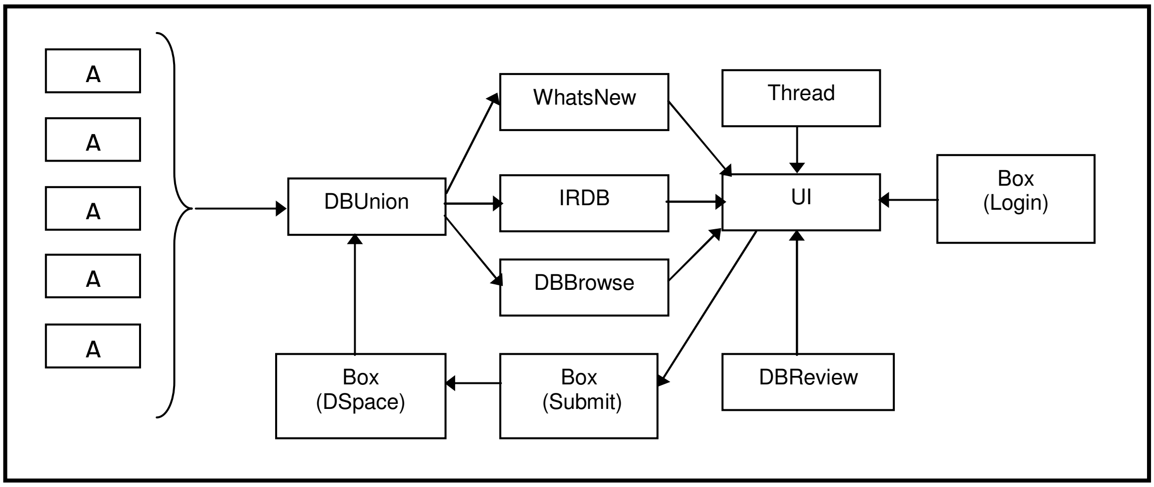


Figure 5.15 – Conceptual design of DSpace using ODL components

Figure 5.15 shows a similar conceptual architecture to the one created for CITIDEL. The main difference is that, for DSpace, the UI is configured to permit the creation of multiple collections. When a user creates a new collection, they can then assign one of five workflows to the collection. Information about a collection, its workflow and the various communities are maintained in an XML file. As can be observed from the conceptual architecture diagram for DSpace, there is a Box component (labelled 'DSpace') where accepted resources are stored, another Box component (labelled 'Submit') where all submissions are initially stored, and a DBReview component that manages the workflows associated with the various collections. Once a resource has gone through the reviewing process and is eventually accepted, it is submitted to the DSpace archive. This in turn is connected to the DBUnion component so accepted resources can be discovered through the various information retrieval methods such as the browse and search services. Reviews for any given resource could be stored in a different Box component, but to reduce the complexity associated with modelling DSpace in this component assembly environment and limit the number of components required, reviews were stored in the same component as the resource, in this case the 'Submit' component and the reviewing engine notified of where the review has been stored.

Once a resource is submitted, the UI determines what collection it was submitted to, identifies the workflow associated with that collection and submits a relevant transaction to the reviewing engine. Participants in the reviewing process are able to log on via the DL's Web interface and their responsibilities are immediately visible. Submitters can similarly log on and view the status of their submissions.

Submitting to DSpace involves multiple steps. These include:

- Selecting a collection
- Describing the content item by adding metadata and keywords
- Uploading the file(s)
- Verifying the submitted item
- Accepting the DSpace license
- Finding the submitted item in a workflow

The UI component currently contains no upload functionality. Resources are assumed to exist somewhere on the Internet and hence users can only include the URL where the resource is located. Additionally, verifying a submitted item using various hashing algorithms was considered beyond the scope of this research, as all that is needed is a more powerful UI component. Hence submitting an item with the DSpace model involves selecting a community/collection and entering the resource's metadata, and the ability to subsequently track the status of that resource at any given moment as it progresses through the workflow associated with that collection.

The UI component enables the DL architect to provide the details for the administrator of the system. This information is used to initialise the DL by creating the first section in the DBReview component, as well as the administrator's login for the Web interface. Once the system is created, the administrator can then login and assign other administrators to the system, or other users acting in various capacities such as editors and reviewers. The DBReview component enables these roles to be limited to specific sections such a particular community, collection or resource.

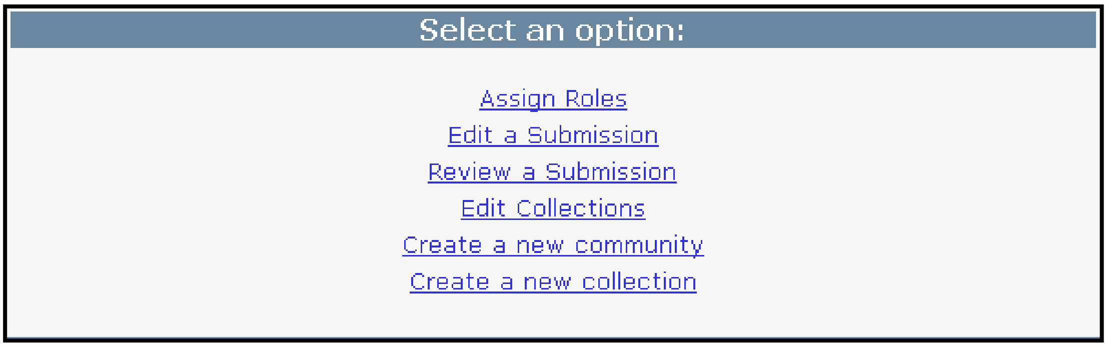
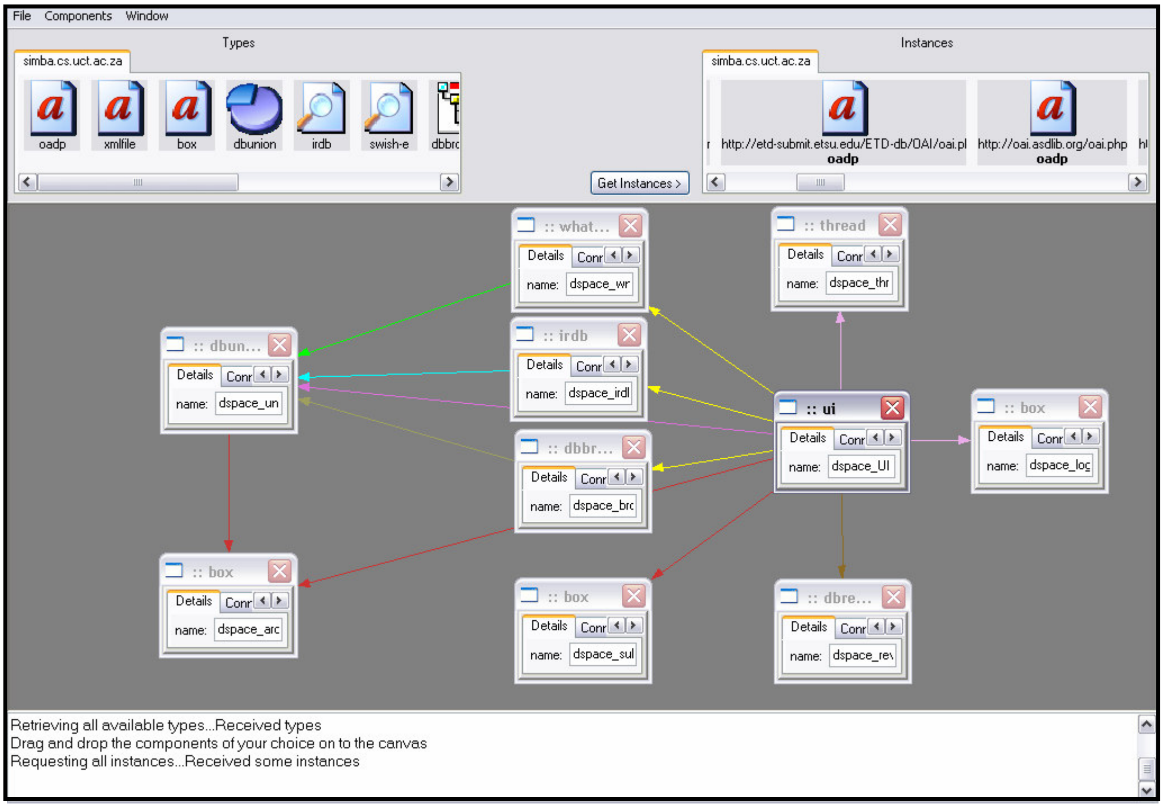


Figure 5.16 – UI Admin Options

Administrators of the system are able to login and perform other actions such as edit the details of communities, including their homepages or descriptions. Figure 5.16 illustrates the different options available to the system administrator.

Figure 5.17 illustrates how the conceptual design depicted in Figure 5.15 is represented using Blox+.



name: dspace_wr

name: dspace_thr

name: dspace_un

name: dspace_irdl

name: dspace_UI

name: dspace_log

name: dspace_brc

name: dspace_arc

name: dspace_sul

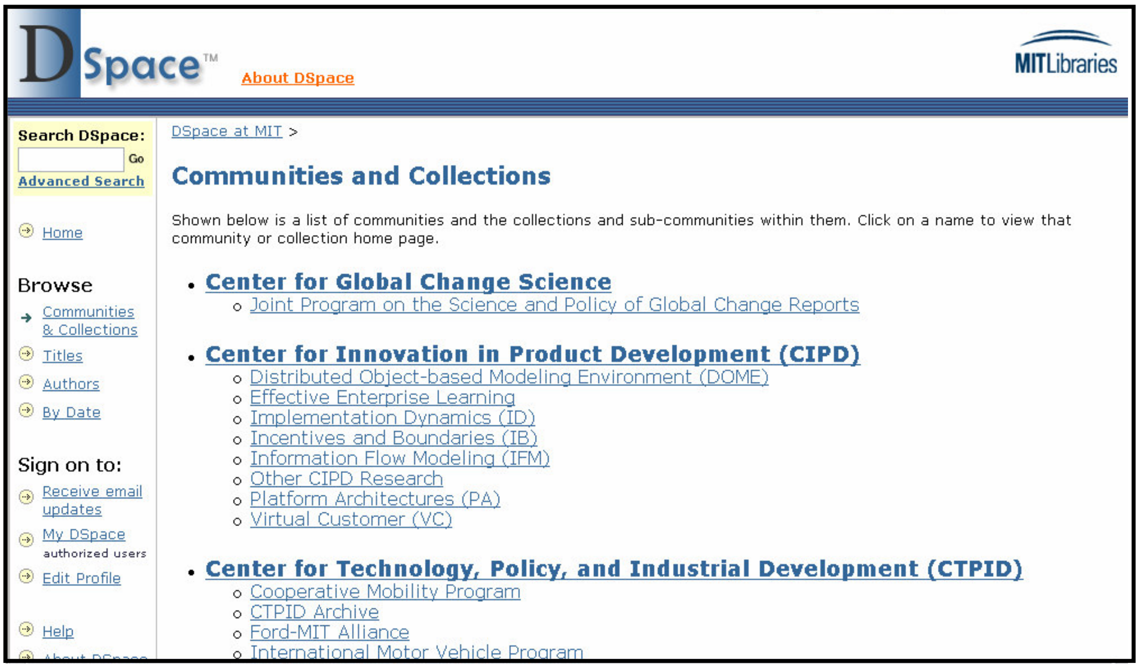
name: dspace_rev

```
Retrieving all available types...Received types
Drag and drop the components of your choice on to the canvas
Requesting all instances...Received some instances
```

Figure 5.17 – Designing DSpace using Blox+

DSpace exposes its resources using the OAI-PMH and is able to maintain the structure of its collections via the protocol’s “sets” functionality. This is reproduced in the model by ensuring that the set structure is maintained when accepted resources are harvested into the Union archive in Figure 5.17 above.

Figure 5.18 shows various communities and collections of a DSpace implementation, while Figure 5.19 shows the equivalent in the DSpace model.



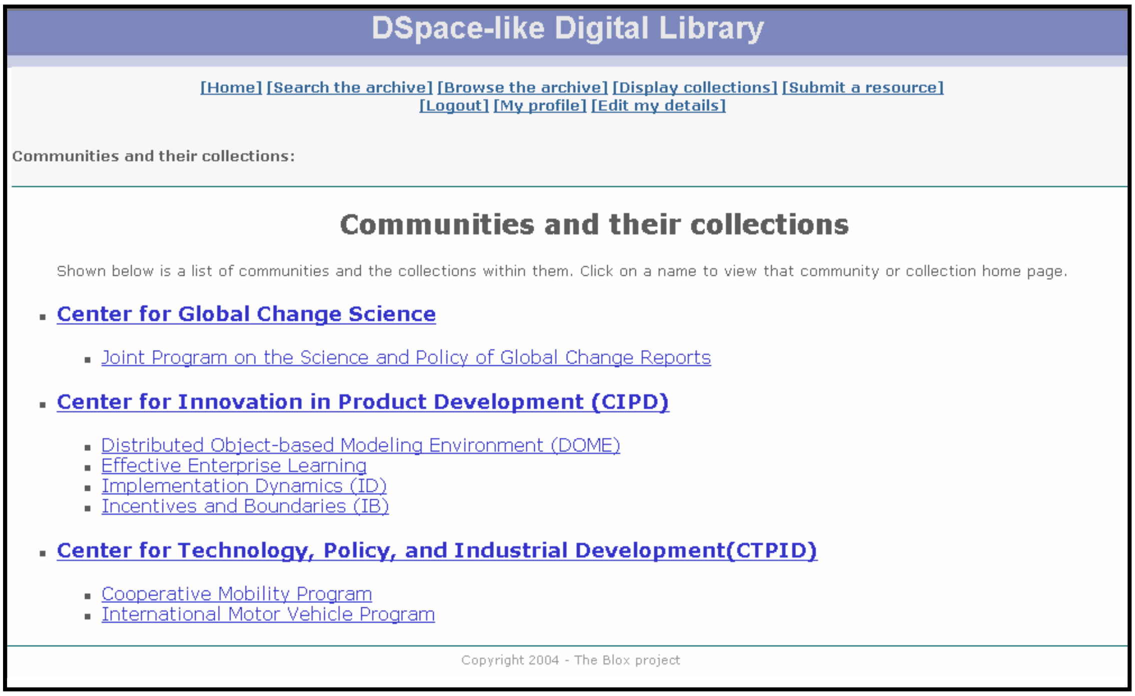
DSpace at MIT >

Communities and Collections

Shown below is a list of communities and the collections and sub-communities within them. Click on a name to view that community or collection home page.

- Center for Global Change Science
  - Joint Program on the Science and Policy of Global Change Reports
- Center for Innovation in Product Development (CIPD)
  - Distributed Object-based Modeling Environment (DOME)
  - Effective Enterprise Learning
  - Implementation Dynamics (ID)
  - Incentives and Boundaries (IB)
  - Information Flow Modeling (IFM)
  - Other CIPD Research
  - Platform Architectures (PA)
  - Virtual Customer (VC)
- Center for Technology, Policy, and Industrial Development (CTPID)
  - Cooperative Mobility Program
  - CTPID Archive
  - Ford-MIT Alliance
  - International Motor Vehicle Program

Figure 5.18 – DSpace communities and collections


DSpace-like Digital Library
[Home] [Search the archive] [Browse the archive] [Display collections] [Submit a resource]
[Logout] [My profile] [Edit my details]

Communities and their collections:

Communities and their collections

Shown below is a list of communities and the collections within them. Click on a name to view that community or collection home page.

- Center for Global Change Science
  - Joint Program on the Science and Policy of Global Change Reports
- Center for Innovation in Product Development (CIPD)
  - Distributed Object-based Modeling Environment (DOME)
  - Effective Enterprise Learning
  - Implementation Dynamics (ID)
  - Incentives and Boundaries (IB)
- Center for Technology, Policy, and Industrial Development(CTPID)
  - Cooperative Mobility Program
  - International Motor Vehicle Program

Copyright 2004 - The Blox project

Figure 5.19 – DSpace model communities and collections

5.5 Summary

This chapter has demonstrated that modelling systems vastly differing systems with varying levels of complexity is possible by using the Blox+ system and remotely configurable components. Given the UI is the user's only point of contact with the DL, the UI's configuration is often what distinguishes one system from another. Systems such as NDLTD, which only has the basic resource discovery components, to systems like DSpace, which contains more complicated resource management and dissemination processes, can be designed and deployed using the same GUI environment and techniques.

Chapter 6

Evaluation

6.1 Introduction

Previous chapters illustrated how a Graphical User Interface could be implemented that enables users to assemble digital library systems from independent distributed components. This chapter discusses how this framework was tested in a controlled environment, employing users with differing backgrounds in order to evaluate the success of introducing a graphical user interface to simplify the assembly of digital library components in general and ODL components in particular, and to test its ability to provide users with a more effective mechanism for component management.

6.2 Pilot Study

6.2.1 Overview

Before undertaking the user testing, a pilot study was conducted to identify unforeseen difficulties that might be encountered during the experiment and the issues that needed to be urgently addressed. This initial test was carried out by five computer science students, their observations were noted, and changes to the system were implemented. The pilot study followed a similar format to the main test and is therefore not going to be discussed here. The major difference between the pilot study and the final test was that, in the pilot study, the users were given a brief handout (Appendix B.2) containing background information on the experiment they were about to carry out. Although it was a short handout, it became obvious that users were reluctant to spend a few minutes reading the material. Therefore, as will be discussed shortly, in the final experiment users were given a brief verbal overview which included key concepts like ODL, OAI-PMH, the Blox+ system and what they would be required to do in the experiment.

6.2.2 Outcome

Although the pilot study affirmed that users were able to use the system in the manner intended, some issues were raised that had to be overcome before proceeding with the final experiment. These issues were predominantly HCI considerations such as the choice of icons, aesthetics of the GUI and giving the users a better sense of control. To overcome these issues, some of the icons were modified to facilitate identification of the component they represented and the cursor was changed into an hourglass when the system was busy carrying out the user's instructions. During the course of the pilot study, it became apparent that users disliked having to fill in several configuration fields. The components were subsequently modified to reduce the number of fields by supplying some configuration values behind the scenes (such as "tableprefix"- the name of the table where the component's database data is stored).

6.3 Methodology

6.3.1 Overview

The tests had to be conducted in a manner that replicates typical usage scenarios expected from the Blox+ software. Therefore, the following points were taken into consideration:

- The users were not required to install the components used during the experiment, neither were they required to install and configure the Blox+ server.
- Users were not expected to possess any previous knowledge of digital libraries or their associated protocols.
- The users were expected to have basic computer skills, such as the ability to drag and drop, and the ability to copy and paste.

The subjects were first instructed to fill in a pre-test questionnaire (Appendix B.1) in order to obtain some background information on them, including their knowledge of DLs, their familiarity with graphical modelling environments and DL-related technologies.

Subsequently, users were given a five minute overview of the following:

- What DLs are
- What the experiment was about
- What the Linux commands meant and how to use them
- What they were required to do during the experiment
- The functions of the components they were going to use during the experiment
- The OAI-PMH, specifically the Identify request they were required to use on the command line in order to test if the components had been properly configured

Users then had to follow step-by-step instructions (Appendix B.3) in order to configure the simple three component digital library displayed in Figure 6.1, first using the command line, and then using the Blox+ system. Users were requested to first build the digital library using the command line so a comparison could be made between the two approaches.

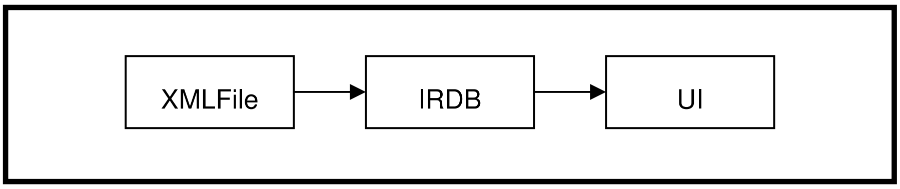


Figure 6.1 – Simple digital library

6.3.2 Command-line Configuration

The XMLFile component was already configured, and data inserted into it, leaving users with only the task of testing it to see if it was functioning properly by navigating their way through the directories to the right XMLFile instance, and then issuing a command-line OAI-PMH Identify request to it.

Users then had to configure the IRDB component to harvest from the XMLFile instance. Finally, the users had to configure the UI component to use the IRDB instance they created, and then view the resulting DL in their browser.

6.3.3 GUI Configuration

Using the Blox+ GUI to create the simple DL in Figure 6.1, users were required to perform the following steps:

- Start up the Blox+ system
- Select the appropriate server
- Obtain the component types by starting a new project
- Request for instances
- Obtain the specified XMLFile instance and drop it onto the canvas
- Drag and drop the IRDB and UI component types onto the canvas
- Name their components
- Modify some of the configuration fields
- Create the required connections
- Publish the system
- View the resultant DL in their browser

After assembling the digital library depicted in Figure 6.1 using the command line and then with the Blox+ system, the users were then required to modify a previously created digital library, depicted in Figure 6.2, by using the "Open DL" functionality provided by the Blox+ GUI. Users had to first view the digital library in Figure 2, add the DBRate component to it, and then view the resulting DL in order to confirm the changes they made. The result of modifying the DL system in Figure 6.2, by augmenting it with the DBRate component, is depicted in Figure 6.3.

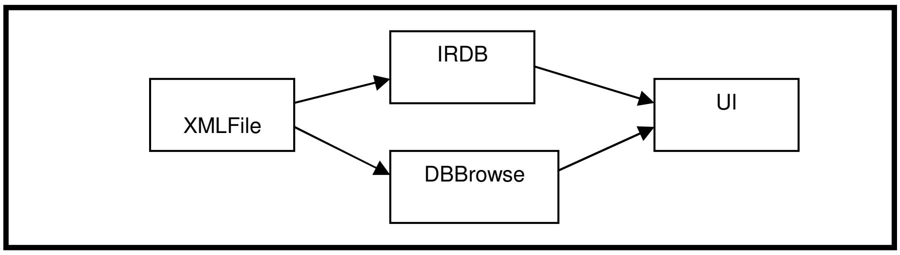


Figure 6.2 – Digital Library users were required to modify

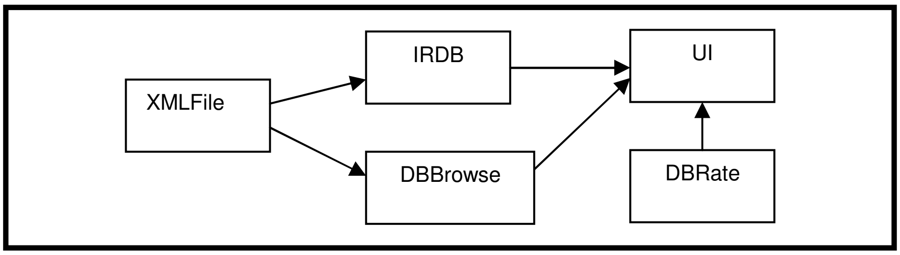


Figure 6.3 – Result of adding the DBRate component

Finally, the users were required to fill out a questionnaire (Appendix B.5) to obtain feedback on their experience.

6.4 Results

Thirty-four students participated in the experiment and the time it took them to complete it ranged from 25 minutes to over an hour. The large time difference to complete the experiment can be attributed to the fact that the students were from differing backgrounds, with some having little or no prior exposure to Linux systems, as can be observed from the results of the background questionnaire tabulated in Table 6.1 below.

Question	Responses	
Program and current year of study	**By level**	
	Undergrad	
	1st year	8
	2nd year	21
	3rd year	4
	Postgrad	
	Masters	1
	By Degree	
	BSc	25

	BCom BEng BusSci BSocSci MSc				1 1 5 1 1
Major	Computer Science Other				23 11
Have you ever used modeling software or component-based development systems such as RationalRose, Microsoft Visual Studio, Delphi or Visual Basic?	Yes No				18 16
Have you ever used a Unix/Linux operating system before?	Yes No				21 13
Have you installed a CGI web-server script/application before (e.g., website guestbook, website counter)?	Yes No				4 30
Do you know what a digital library is?	Yes No				3 31
Have you ever installed a digital library before?	Yes No				0 34
	Excellent	Good	Adequate	Poor	Clueless
How would you rate your understanding of the functioning of web services? (involving UDDI, WSDL etc)		3	8	10	13
How would you rate your proficiency with ODL Components?			1	8	25
Rate your familiarity with the OAI Protocol for Metadata Harvesting.			1	6	27

Table 6.1 – Results obtained from background questionnaire

Table 6.2 summarises the results obtained from the final questionnaire. This questionnaire required that users indicate the extent to which they agreed or disagreed with the statements presented according to a 5-point scale. The last four questions were free-form questions and as such, have not been included in Table 2, but have all been documented in Appendix C. The free form questions are:

- What was the most difficult part of the whole exercise?
- What in the entire exercise did you least understand?
- State anything you would add, subtract or otherwise do differently to the system.
- Enter any additional comments

Questions	**Responses**				
	Strongly Agree	**Agree**	**Neutral**	**Disagree**	**Strongly Disagree**
I understand how the Blox system functions.	2	16	12	4	
I understand the concept of types and instances.	3	19	9	3	
Based on previous knowledge, and the knowledge gained from conducting this experiment, I think it is a good idea to be able to graphically assemble systems from distributed components.	18	13	2	1	
I found the interface consistent with my previous experiences with Visual IDEs.	5	11	13	4	1
I would consider using ODL and OAI components if I need to build a system with requirements similar to the exercise.	13	13	4	4	
I would use a system such as Blox to create digital libraries if I had the need to create one.	12	16	6		
I would use a system such as Blox to connect and configure other types of components (e.g. other	9	18	7		

Web Services, or COM or CORBA objects) if it had support for them.					
Modifying an existing digital library was easy with the Blox system	13	15	6		
When I created a digital library using Blox, it produced the digital library I expected it to produce.	19	10	5		
The interface responded as I expected it to.	14	13	6	1	
I found it easy to use Blox to accomplish the tasks.	11	17	6		
I found it easy to apply my previous GUI experiences with Blox.	16	8	7	2	1
I like the idea of representing a component as a window.	19	10	5		
I found using Blox to create a digital library easier than manually using OAI/ODL Components.	19	7	8		
I found using Blox to create a digital library faster than manually using OAI/ODL Components.	21	10	3		

Table 6.2 – Questions and responses obtained from the user-testing questionnaire

6.5 Discussion

One of the aims of this research was to encourage users with no knowledge of the OAI-PMH, Web Services and other DL-related technologies to get a fully functional digital library up and running with little effort. With this in mind, users were selected based on their willingness to participate in the experiment rather than their computer prowess. But as the experiment progressed, it became increasingly apparent that their ability to perform the tasks required of them was greatly influenced by their background and previous experiences. Thirty-three of the thirty-four participants were undergraduates from various faculties, with a third of the students not having computer science as one of their majors. Half of the users had never used modelling software before, and a third had never used a Linux command prompt. Hence the stage was set to test the level of skill required to successfully use the ODL components, and even more importantly, as far as this research is concerned, to use the Blox+ system.

It was rather interesting to note that although none of the respondents had ever installed a digital library before, and only three admitted to knowing what a digital library was, nine of the participants indicated that they were somewhat familiar with the ODL components with seven admitting familiarity with the OAI-PMH. This can perhaps be attributed to the fact that the respondents were not aware of the specific nature of ODL and the OAI-PMH to digital libraries. The users probably assumed the OAI-PMH was a protocol they must have come across at some point while using the Internet.

6.5.1 Understandability

The responses to the question concerning whether they understood how the Blox+ system functions and if they had grasped the concept of types and instances were divided. This suggests that three minutes presenting all the key concepts required for novices to create a digital library using the command line and with Blox+ may not have been sufficient. This is further demonstrated by comments such as:

- "[I didn't understand] the digital library concept. More research on the subject would help improve understanding" and
- "ODL – Still not sure about it".

Not understanding the difference between types and instances affected the way they were able to create the DL, as users were not sure when to select a component from the types panel or from the instances panel. Incidentally, most of the users who could not differentiate between a type and an instance were not computer science students. However, the four users who indicated they do not understand how the Blox+ system functions agreed that Blox+ produced the digital library they expected it to produce, and that it was easy to use. This only goes to demonstrate their ability to follow step-by-step instructions, but the outcome may not have been the same had they been simply told to create a three component digital library. Thus, to improve the understandability of the Blox+ system, users need to be properly coached verbally or through an on-line help system, on not only how to use the interface, but also on the basic activities that take place behind the scenes for a DL to be created.

6.5.2 Usability and Simplicity

There were two recurring issues users had with the command-line configuration method. These issues were confusion over which baseURL to use and the fact that the baseURL had to be entirely retyped if a typographical error was made—issues the introduction of a GUI was meant to address. From the free-form comments users made, it became apparent that the users felt these issues were properly addressed when they realised dragging a baseURL from one component window to another in the GUI was equivalent to typing a baseURL in on the command line. Additionally, the fact that users did not have to navigate through directories using Linux commands, or enter several configuration parameters helped to ensure that none of the users disagreed with the fact that Blox+ was easier to user than using the command line to configure the components. Without making a comparison between the two configuration methods, most of the users agreed that Blox+ was easy to use in accomplishing the tasks, with none disagreeing and six of the users choosing to remain neutral. Upon closer investigation, it was discovered all six of the users who chose to remain neutral on the matter, either remained neutral, disagreed or strongly disagreed with the suggestion that it was easy to apply Blox+ to their previous GUI experiences and if Blox+ was consistent with their experience with other Visual IDEs. This suggests that their unfamiliarity with the Blox+ User Interface and its approach to connecting components may have been a mitigating factor.

6.5.3 Speed

Most of the users were in agreement that modifying an existing digital library with Blox+ was a simple task. Three of the six users who remained neutral were the only three users who remained neutral on whether Blox+ was faster than using the command line to configure components. The fact they had to wait for about 15 seconds for the digital library to load before modification may have been a mitigating factor. The rest of the respondents either agreed or strongly agreed that Blox+ was faster than using the command line. The Blox framework ensures that all components are supplied with defaults such that they can work without further modification, and users were spared from having to spend time entering baseURLs, hence making Blox+ appear faster than using the command line. This experiment used three simple components with a small set of data, so users were not required to wait for long periods of time for harvesting to take place, as would typically be the case if large archives were used. Users would perhaps have had a different view had that been the case.

6.6 Summary

When users were asked if they would use the ODL/OAI components should they need to build a system with similar requirements to the system they constructed during the experiment, eight of the respondents were skeptical. This can be attributed to the fact that there was very little mention of what ODL components were and how they differed from other types of components, and as such those users did not feel particularly inclined to agree with the statement. Those who did agree probably did so because they were satisfied with the digital library created from those components and not necessarily because they understood what the ODL component framework was about.

In general, the overall feedback was positive since most users agreed that not only would they use Blox+ to create their digital library should they need one, but would also consider using Blox+ to create other component-based systems if it had support for them. After assembling the three-component digital library with the command-line and then with the GUI, users were able to better appreciate the differences between the two approaches, with the vast majority indicating that they thought it was a good idea to be able to assemble component-based systems with a graphical user interface. However, the results may have been different if the users were required to create highly complex DL systems.

A number of users expressed their appreciation for the Blox+ system as is, with comments such as:

- “This is the new system to use in future”.

- “I think that the Blox program is very good, personally I knew nothing about components and stuff before but with this program it has made me understand”.
- “Brilliant idea!”
- “Using the windows and the drag-and-drop was MUCH simpler and user-friendly than the manual part”.
- “I don’t know what other systems are like, but I would advise people to use Blox”.

Also, a number of useful suggestions were made. All the comments are available in Appendix C, but the more interesting ones have been outlined below:

- “A built-in preview of the digital library as it appears in the browser”.
- “Types and Instance ‘blocks’ on top of GUI could be better spread and thus easier to view”.
- “A help button for troubleshooting, guidelines and stuff”.
- “People should be told to create ui->irdb->xmlfile – i.e. work from left to right (it’s more intuitive)”.
- “Automatically open the default web browser instead of copy/paste, and more clearly defined links between components in the GUI”.
- “Pictures on the digital library User Interface”.
- “’Get Instances’ took a while, but I didn’t see an indication that it was doing something”.

The Blox+ system allows a user to view their digital library if it contains a user interface component. Users are required to copy the link provided and paste it in their browser. Several users felt it would have been preferable for Blox+ to automatically display the DL UI in their default browser rather than be required to copy and paste. There were also several requests for online help. The asynchronous method of communication resulted in users not being sure when communication had ended as was observed in the case of the response to the ‘Get Instance’ request. Subsequent versions of Blox could be modified to address those issues.

6.7 Conclusion

This chapter discussed the evaluation of the Blox+ GUI and also, though indirectly, the underlying infrastructure. Several conclusions can be drawn from performing this exercise:

- There is an easier way to create DL systems that abstracts the technical details, such as the OAI-PMH, from the user.
- Users found the Blox+ user interface easy to use.
- They would consider using it in future DL development efforts.
- There are still a few issues that could be improved upon.
- Users without basic computer skills would have some difficulty understanding how to use the GUI.

Chapter 7

Conclusion and Future Work

Creating digital libraries is a non-trivial endeavour, especially for people who have no experience in the field. There are several packages available for this purpose, but most come with inaccurate promises of simplicity and extensibility. The framework presented in this dissertation focuses on equipping users with a GUI framework that will ultimately facilitate the process of designing component-based distributed systems, by using drag and drop techniques to connect components together in the manner that they see fit.

Though the ODL components were used as the reference points for this work, other components were similarly interfaced to demonstrate the generality and wide-scale applicability of the Autoconfigure Interface Specification. This chapter presents concluding remarks on composing components graphically, some of the concerns that surfaced, issues encountered, the temporary solutions adopted to resolve the issues that arose during the design and testing stages of this work, and finally, some recommendations for future work.

7.1 Artefacts

Unlike most other DL packages on the market, the component assembly environment is simply a platform on which components can be assembled in an ad-hoc manner. Users are able to connect to a server hosting components, link the components together and publish the system to simply and quickly generate a DL using any component as long the component complies with the Autoconfigure Interface Specification.

To this end, a framework was devised that comprises of: a Graphical User Interface, where the component assembly takes place; a server, to handle the communications between the components and the GUI; and the Autoconfigure Interface Specification that determines how the various components are manipulated. This all leads to a multi-platform, language independent component model since the components are only accessed through their external interfaces.

A language called the Component Connection Language (CCL) was developed, which describes any given system made up of a selection of components. Thus, with only a CCL, any given DL system can be easily modified or reproduced.

7.2 Outcomes

Chapter 6 presented the methodology used to evaluate the product of this research. Based on a detailed analysis of the results from the user tests that involved the participation of over thirty users, it was discovered that the users found the system easier and faster than the alternative, i.e. using the command-line to create systems from a number of ODL components. The users also expressed that they would consider using Blox+ should they ever need to create a DL system. Although most of the test users had no prior experience with designing digital libraries, their responses were mostly favourable indicating that this method of designing modular DLs suitably addressed most of the issues raised during the manual configuration of ODL components.

Chapter 5 described how three existing DL systems were modelled to demonstrate how this component assembly environment could move beyond the theoretical nature of research projects to find a place amongst production systems.

In keeping with the aims of this research outlined in chapter one, the user tests and modelling of existing DL systems have demonstrated that:

- Novice users are able to simply and quickly graphically construct a digital library from a pool of components, effectively demystifying the process of creating digital libraries.
- It is possible to reduce the frustration experienced by users when using the command-line to assemble digital library components by reducing the amount of typing required by using a graphical user interface.
- The framework proposed in this dissertation is applicable to a wide variety of components over and beyond the digital library arena, resulting in a truly generic component assembly environment.
- The methods suggested can be used to create production systems

The intention is that this novel and generic component assembly framework will encourage ordinary users to experiment with building and maintaining their own DL collections, ultimately facilitating and promoting the sharing of digital information.

7.3 Future Work

7.3.1 User Interface

There are several important related issues that could be addressed by future research.

To simplify the process of creating digital libraries, the user interface component is essential in ensuring that users do not have to create one from scratch. Yet, it is probably the component that will require the most modification and hand-coding in order for it to adequately service the DL's target user-community.

The subject of future research could be to reduce the amount of coding required to incorporate new functionality into the UI to cater for different types of components. Some researchers have attempted generating portions of HTML code such as HTML forms by interpreting XML Schemas that describe the fields required in the form. This option was explored to some extent by the collaborators in this research (Moore et al., 2003) to create the forms that hold the components' configuration fields. A more comprehensive attempt was made by Suleman (2002) by creating MDEdit – a Perl module, which transforms Schemas into HTML forms. However, both of these implementations only cater for a subset of the Schema language and do not cover the entire HTML vocabulary.

Presently, the UI accommodates different permutations of all the components used during the course of this research. However, the UI is rather rigid in that the output from a component is placed in a very specific position in the UI. For example, if a WhatsNew component is connected to the UI, the results of recent entries into the archive are displayed at the bottom of the UI window. What is needed is a method indicating the position and appearance of the component in UI, possibly by visually modelling the UI in a GUI, without coming into direct contact with any HTML code. The UI's presentation is currently serviced by a cascading style sheet (CSS), so giving the UI a different look and feel is just a simple matter of selecting a different style sheet. However, with regards to presentation, CSS style sheets are limited in what they are able to achieve. By visually modelling a UI, in conjunction with XSL style sheets, one could, for instance, model a UI to have its navigation bar at the side rather than at the top of the page. Some of these UI issues are currently being addressed by a colleague, Kevin Feng.

7.3.2 Component Dependencies

One problem encountered during the course of this research was how to represent the dependencies between two components. Components are generally designed to be self-sufficient blocks of code that encapsulate a specific functionality. But sometimes, certain components require information that is outside their immediate boundaries in order to complete the configuration process. To illustrate this, consider configuring ODL's DBUnion component to only harvest metadata from a specific set belonging to some archive. Doing this using the command line requires that the DBUnion component sends a request to the archive to provide all its sets via the OAI-PMH ListSets request. Once the sets are returned and displayed, the user only needs to select the desired set and then the configuration of DBUnion can proceed. Components are configured one at a time and the user is able to interact with the previously instantiated components. Using the GUI for this is a trickier proposition as configuration information is supplied for all the components that make up the system before being sent off to the server for batch configuration. The workaround adopted to resolve this dilemma was to ensure that all archive components return information on the sets and metadata prefixes they contain in the response to a 'configure' request. That is, as soon as an archive is configured, it returns its sets and metadata prefixes. With this approach, one can configure a single archive component and use the information generated from the configuration response to configure others. This workaround is suitable for this particular scenario, but to deal with other similar situations, one solution could be to scrap batch configuration altogether and configure components individually as is done using the command line.

Dependencies do not only occur between components. Sometimes dependencies may occur within a single component. A component may want to present certain fields for the user to enter configuration information, or expose different information, based on the value of another configuration field in the same component. For example, if a DL designer indicates that they want users to be able to log on to the digital library, only then should they be presented with fields so that the initial administrative username and password can be entered. Currently, users are presented these fields regardless of whether they wish to include a login component or not. The content of these fields are subsequently ignored if no login component is specified. Future work could attempt a solution by using a form generator based on a larger subset of XML Schema than most currently available tools.

7.3.3 Security

Security is an important aspect of all network applications, but even more so for distributed systems. This is because distributed applications present an even greater threat since more computers and components are involved and each one is liable to attack. Attacks in distributed applications can have several outcomes: unauthorized release of information, modification of information or denial of service.

Access control plays an important role, to limit the possibility of any of the aforementioned scenarios happening. As far as this research is concerned, there are two main areas that need to be considered – access to the digital library via the Web interface, and access to the remote components that make up the digital library.

Securing the digital library by restricting access to certain functions can be achieved by authenticating users before they can complete certain tasks. The UI component created in this research allows DL designers to indicate that they would like users to be authenticated; the generated UI instance then provides forms for logging in and the registration of new users. Users can then maintain a profile on the DL system. Session state is maintained with the use of cookies. With this mechanism, attackers are limited to brute force attempts on passwords. Being aware of the security threats and taking proactive steps towards reducing the possibility of them occurring is the only way to run secure systems in a networked environment.

As it stands now, users only have limited control over the component instances created. Once a component is instantiated, the instance can be modified or deleted by anyone, which is obviously undesirable. What is needed is for a security infrastructure to be put in place that will restrict access to the component's interface. One way to accomplish this is for users to be authenticated by the GUI at startup so that only owners of component instances are able to modify or delete a component's instance on the server. Furthermore, it may be useful for the owners of instances to be able to declare their instance as "hidden" so that it cannot be discovered by the listInstances interface.

References

ALG, 2004. *Automated Learning Group* [online].
Available: http://alg.ncsa.uiuc.edu/do/index [11 November 2004].

ANSI, 1995. *Z39.50* [online]. Available: http://www.loc.gov/z3950/agency/markup/markup.html [13 December 2004].

Arena M. 2003. *Object Orientated Analysis & Design: Object Orientated Framework/Model Comparisons* [online]. Available: http://www.ucc.asn.au/~marena/oomodels.pdf [17 January, 2005].

Arms, W. 2000. *Digital Libraries*. Cambridge, MA: MIT Press.

Arms W.Y., Fox, E., Narum J. and Hoffman, E. 2002. NSDL: From prototype to production to transformational national resource. *Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries*, Portland, OR, 14-18 July 2002, 368-368. New York, NY, USA: ACM Press.

BEA, 2004. *BEA WebLogic Workshop 8.1* [online]. Avalaible: http://www.bea.com/-framework.jsp?CNT=index.htm&FP=/content/products/workshop [17 January, 2005].

Bertocco, S. 2001. Torii, an Open Portal over Open Archives. *HEP Libraries Webzine*, 1(4) June 2001.

Biggs, M. 2003. Developers get a shield from Java complexity. *JavaWorld* [online]. Available: http://www.javaworld.com/javaworld/jw-06-2003/jw-0606-iw-m7.html [13 December 2004].

Blake, R. and Hamilton-Williams, R. 2000. *The Koha story* [online]. Available: http://koha.org/about/story.html [14 December 2004].

Borland Software Corporation, 2004. *Borland Delphi* [online].
Available: http://www.borland.com/delphi/index.html [5 June, 2003].

Borgman, C.L., 1999. What are digital libraries? Competing visions. *Information Processing and Management*, 35(3): 227-43.

Bravenet, 2004. *Bravenet.com: Web tools for Webmasters* [online]. Available: http://www.bravenet.com/ [13 December 2004].

Bray, T., Paoli, P., Sperberg-McQueen, C.M., Maler, E. and Yergeau, F. (eds) 2000. *Extensible Markup Language (XML) 1.0 (Third Edition), W3C* [online]. Available: http://www.w3.org/TR/REC-xml/ [13 December 04].

Breeding, M. 2002. Understanding the Protocol for Metadata Harvesting of the Open Archives Initiative. *Computers in Libraries,* 22(8): 24-30.

Canter, S. 1992. Programming Without Words. *PC Magazine,* 11(19):31.

Castelli, D. and Pagano, P. 2002. OpenDLib: A Digital Library Service System. *Proceedings of the 6th European Conference on Research and Advanced Technology for Digital Libraries*, Rome, Italy, September 2002, 292–308. London, UK: Springer-Verlag.

Castelli, D. and Pagano, P. 2003. A System for Building Expandable Digital Libraries. *Proceedings of the 3rd ACM/IEEE-CS joint conference on Digital Libraries*, Houston, Texas, 24-28 June 2003, 335-345. Washington, DC, USA: IEEE Computer Society.

Celestial, 2004. *Celestial* [online]. Available: http://celestial.eprints.org/ [11 November 2004].

Cerami, E., 2002. *Web Services Essentials*. O'Reilly & Associates.

Chappel, D. and Linthicum, D.S. 1997. *ActiveX Demystified* [online]. Available: www.byte.com/art/9709/sec5/sec5.htm [11 November 2004].

Chowdhury, G. & Chowdury, C. 2003. *Introduction to Digital Libraries*. London, UK: Facet Publishing.

CNRI, 2005. *CNRI Handle System 5.3* Available: http://www.handle.net/hs_manual/index.html [10 October 2005].

Cole, T.W., Kaczmarek, J., Marty, P.F., Prom, C.J., Sandore, B. and Shreeves, S.L. 2002. Now that we've found the ‘Hidden Web’ what can we do with it? The Illinois Open Archives Initiative Metadata Harvesting experience. *The Proceedings of MW2002 and Museums and the Web 2002:*

Selected Papers, Boston, Massachusetts, 17-21 April 2002, 63-72. Pittsburgh, PA: Archives and Museum Informatics.

CSTC, 2004. *Computer Science Teaching Center* [online]. Available: http://www.cstc.org
[10 June, 2003].

Davis, J.R. and Lagoze, C. 2000. NCSTRL: Design and Deployment of a Globally Distributed Digital Library. *Journal of the American Society for Information Science*, 51(3):273-280.

D2K, 2003. *Data to Knowledge* [online]. Available: http://www.d2k.com [4 June, 2003].

Fabbrichesi, M. 2002. TORII: Access the Digital Research Community. *Proceedings of the AH'2002 Workshop on Personalization Techniques in Electronic Publishing*, Málaga, Spain, May 2002, 71-76.

Fallside, D.C. 2001. *XML Schema Part 0: Primer Second Edition* [online]. Available: http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/ [13 December 2004].

Fischer, G. and Fuhr, N. 2003. CYCLADES: User Services for Open Archives. *Proceedings of the conference on Worldwide Coherent Workforce and Satisfied Users*, Oldenburg, Germany, 17 – 19 September 2003.

Fox, E.A. 1999. Networked Digital Library of Theses and Dissertations. *Proceedings of DLW15*, Nara, Japan: ULIS. July 1999 [online]. Available: http://www.ndltd.org/pubs/dlw15.doc [11 Nov 2004].

Fox, E.A. 2002. *Networked Digital Library of Theses and Dissertations* [online]. Available: http://www.ndltd.org [15 Dec 2004].

Fox, E.A. and Cassel, L. 2002. *Journal of Educational Resources in Computing* [online]. Available http://www.acm.org/pubs/jeric/ [15 March 2004].

Fox, E., Knox, D. Cassel, L., Lee, J.A.N, Pérez-Quiñones, M., Impagliazzo, J. and Giles, C.L. 2002. *CITIDEL: Computing and Information Technology Interactive Digital Educational Library* [online]. Available http://www.citidel.org [15 December 2004].

Fox, E. and Gonçalves, M. 2002. 5SL-A language for Declarative Specification and generation for Digital Libraries. *Proceedings of the 2nd Joint conference on Digital libraries*, Portland, OR, 14-18 July 2002, 263 - 272 . New York, NY, USA: ACM Press.

Furnas, G.W. and Rauch, S.J. 1998. Considerations for Information Environments and the NaviQue Workspace. *Proceedings of the 3rd ACM International Conference on Digital Libraries*, 23-26 June 1998, Pittsburgh, PA, USA, 79-88. ACM.

Garlan, D., Allen, R. and Ockerbloom, J. 1995. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, 12(6):17-26.

Goetz, B. 2000. The Lucene searcFh engine: Powerful, flexible and free, *JavaWorld* [online]. Available: http://www.javaworld.com/javaworld/jw-09-2000/jw-0915-lucene.html [13 December 2004].

Goh, D. and Leggett, J. 2000. Patron-augmented digital libraries. *Proceedings of the fifth ACM conference on Digital libraries*, San Antonio, Texas, 2-7 June 2000, 153-163. New York, NY, USA: ACM Press.

Hagedorn, K. 2003. OAIster: A 'No Dead Ends' OAI Service Provider. *Library Hi Tech*, 21(2):170-181.

IBM, 2004. *Rational Rose Data Modeller* [online]. Available: http://www-306.ibm.com/-software/awdtools/developer/datamodeler/ [13 December 2004].

IBM, 2005. *IBM Built the New Generation Digital Library for Hong Kong* [online]. Available: http://www-306.ibm.com/software/success/cssdb.nsf/CS/MCAG-5TEKBQ?OpenDocument &Site= [10 October 2005].

IA, 2005. *Instructional Architect* [online]. Available: http://ia.usu.edu/index.php [10 October 2005].

Jakarta, 2004. *Jakarta Lucene* [online]. Available: http://jakarta.apache.org/lucene/docs/-index.html [13 December 2004].

Johnson, M. 1997. The BeanBox: Sun's JavaBeans test container. *JavaWorld* [online]. Available: http://www.javaworld.com/javaworld/jw-09-1997/jw-09-beanbox.html [13 December 2004].

Johnson, J., 2004. *M7 Application Assembly Suite. Websphere* Journal [online]. Available: http://www.sys-con.com/WebSphere/articleprint.cfm?id=323 [13 December 2004].

Kelapure, R., Gonçalves, M. and Fox, E. 2003. Scenario-Based Generation of Digital Library Services. *Proceedings of the 7th European Conference on Research and Advanced Technology for Digital Libraries*, Trondheim, Norway,17-22 August 2003, 263-275. Springer.

Lagoze, C. and Van de Sompel, H. 2001. *The Open Archives Initiative Protocol for Metadata Harvesting* [online]. Available: http://www.openarchives.org/OAI/openarchivesprotocol.html [6 July 2003].

Lagoze, C. and Davis, J.R. 1995. Dienst - An Architecture for Distributed Document Libraries. *Communications of the ACM*, 38(4):47.

Latimore, D. 1995. IDEs foster open real-time systems. *Electronic Engineering Times*, 2(865):70.

Liu, X., Maly, K., Zubair, M., and Nelson, M.L. 2001. Arc: An OAI Service Provider for Cross-Archive Searching. *Proceedings of the 1st ACM/IEEE-CS joint conference on Digital Libraries*, Roanoke, VA, 24–28 June 2001, 65-66. New York, NY, USA: ACM Press.

Liu, X., Maly, K., Zubair, M. and Nelson, M.L. 2002. DP9: An OAI Gateway Service for Web Crawlers. *Proceedings of the 2nd ACM/IEEE-CS Joint Conference on Digital Libraries*, Portland, OR, 14-18 July 2002, 368-368. New York, NY, USA: ACM Press.

Mahoney, A. 2001. Tools for students in the Perseus digital library. *CALICO Journal*, 18(2):269-282.

Maly, K., Zubair, M. and Liu, X. 2001. Kepler - An OAI Data/Service Provider for the Individual, *D-Lib Magazine*, 7(4) [online]. Available: http://www.dlib.org/dlib/april01/maly/04maly.html [17 December, 2005].

Maly, K., Zubair, M., Nelson, M., Liu, X., Anan, H., Gao, J., Tang, J. And Zhao, Y. 2002. Archon - A Digital Library that Federates Physics Collections. *Proceedings of the 6th International Conference on Dublin Core and Metadata for e-Communities (DC-2002: Metadata for e-Communities: Supporting Diversity and Convergence)*, Florence, Italy, October 2002, 27-34. Firenze University Press.

Microsoft, 2004a. *Microsoft Visual Studio* [online]. Available: http://msdn.microsoft.com/-vstudio/ [13 December 2004].

Microsoft, 2004b. *COM: Component Object Model Technologies* [online]. Available: http://www.microsoft.com/com/default.mspx [13 December 2004].

Microsoft, 2004c. *Getting Started in .Net* [online]. Available: http://www.microsoft.com/-net/basics/whatis.asp [13 December 2004].

Microsoft, 2004d. *Visio 2003 Product Overview* [online]. Available: http://www.microsoft.com/-office/visio/prodinfo/overview.mspx [13 December 2004].

Mitra, N. 2003. *SOAP Version 1.2 Part 0: Primer* [online]. Available: http://www.w3.org/-TR/2003/REC-soap12-part0-20030624/ [13 December 2004].

Moore, D., Emslie, S. and Suleman, H. 2003. *BLOX: Visual Digital Library Building*, Technical Report No. CS03-20-00, Department of Computer Science, University of Cape Town.[online]. Available: http://pubs.cs.uct.ac.za/archive/00000075/01/BLOX_research_paper.pdf [13 December 2004].

MyOAI, 2004. *my.OAI* [online]. Available: http://www.myoai.com/ [13 December 2004].

Nabil, R.A., Holowczak, H., Halem, M., Nand, L. and Yesha, Y. 1997. *IEEE Digital Library Technical Committee* [online]. Available: http://cimic3.rutgers.edu/ieee_dltf.html [11 November 2004].

NCSA, 2003. *National Centre for Supercomputing Applications* [online]. Available: http://www.ncsa.uiuc.edu/ [8 July 2003].

NCSTRL, 2002. Networked *Computer Science Technical Reference Library* [online]. Available: http://www.ncstrl.org/ [5 December 2004].

NDLTD, 2003. *Networked Digital Library of Theses and Dissertations* [online]. Available: http://www.ndltd.org/ [11 November 2004].

Nelson, M.L., Rocker, J. and Harrison T.L. 2003. OAI and NASA Scientific and Technical Information. *Library Hi-Tech*, 21(2).

Nourie, D. 2004. *Java Studio Creator: An IDE to Create Web Applications* [online]. Available: http://java.sun.com/developer/technicalArticles/WebServices/jscoverview/ [8 August 2003].

OAI, 2000. *Open Archives Initiative* [online]. Available: http://www.dlib.org/dlib/february00/vandesompel-oai/02vandesompel-oai.html [8 August 2003].

OAIB, 2003. *Open Archives in a Box (OAIB)* [online]. Available: http://dlt.ncsa.uiuc.edu/oaib/ [21 June 2003].

OMG, 2004 *Object Management Group* [online]. Available: http://www.omg.org/ [13 December 2004].

Oberleitner, J., Gschwind, T. and Jazayeri, M. 2003. The Vienna Component Framework enabling composition across component models. *Proceedings of the 25th International Conference on Software Engineering*, Portland, Oregon, May 2003, 25-35. Washington, DC, USA: IEEE Computer Society.

Orfali, R., Harkey, D. and Edwards, J. 1996. *The Essential Distributed Objects Survival Guide*. New York: John Wiley & Sons.

Payette, S. and Staples, T. 2002. The Mellon Fedora Project: Digital Library Architecture Meets XML and Web Services. *Research and Advances Technology for Digital Technology: 6th European Conference, ECDL 2002, Rome, Italy, September 16-18, 2002. Proceedings.* 406-421. Springer 2002.

Petzold, C. 1992. More than just a pretty face. *PC Magazine*, 11(11):195-212.

Plutchak, J., Futrelle, J. and Gaynor, J. 2002. Components for Constructing Open Archives. *Proceedings of the 2nd ACM/IEEE-CS Joint Conference on Digital Libraries*, Portland, OR, 14-18 July 2002, 403-403. New York, NY, USA: ACM Press.

Reyes-Farfán, N. and Sánchez, J.A. 2003. Personal spaces in the context of OAI. *Proceedings of the 3rd ACM/IEEE-CS Joint Conference on Digital Libraries*, Houston, Texas, 24-28 June 2003, 182-183. Washington, DC, USA: IEEE Computer Society.

Schneider, J. and Nierstrasz, O. 1999. Components, Script and Glue. *Software Architectures – Advances and Applications*, 4:13-25. Springer-Verlag.

Sheldon, T. 2001. *COM (Component Object Model)* [Online]. Available: http://www.linktionary.com/c/com.html [10 August 2004].

Shu, N. 1989. Visual Programming: Perspectives and Approaches. *IBM Systems Journal*, 28(4):525-547.

SIRAC, 2001. *SIRAC : Distributed Systems for Cooperative Applications* [online]. Available: http://www.inria.fr/recherche/equipes/sirac.en.html [13 December 2004].

Smart, J., Roebling, R., Zeitlin, V. and Dunn, R. 2005. *wxWidgets 2.6.2: A portable C++ and Python GUI toolkit* [online]. Available: http://www.wxwindows.org/ [11 October 2005].

Suleman, H. 2002. *Open Digital Libraries* (Phd Thesis). Blacksburg, VA USA: Virginia Tech.

Suleman, H. and Fox, E.A. 2001. A Framework for Building Open Digital Libraries. *D-Lib Magazine*, 7(12) [online]. Available: http://www.dlib.org/dlib/december01/suleman/-12suleman.html [8 July, 2003].

Sun Microsystems, 2004a. *J2EE Enterprise JavaBeans Technology* [online]. Available: http://java.sun.com/products/ejb/ [11 November, 2004].

Sun Microsystems, 2004b. *Java 2 Platform, Enterprise Edition (J2EE)* [online].
Available: http://java.sun.com/j2ee/ [11 November, 2004].

Systinet Corporation, 2003. *Web Services: An Introductory Guide for ISVs* [online]. Available: http://cramsession.bitpipe.com/detail/RES/1044980238_368.html [11 November, 2004].

Szyperski, C. 1999. Components and Objects Together. *Software Development*, 7(5), May 1999.

Tansley, R., Bass, M., Stuve, D., Branschofsky, M., Chudnov, D., McClellan, G. and Smith, M. 2003. The DSpace institutional digital repository system. *Proceedings of the 3rd ACM/IEEE-CS Joint Conference on Digital Libraries*, Houston, Texas, 24-28 June 2003, 87-97. Washington, DC, USA: IEEE Computer Society.

Tennant, R. 2002. *Simple Web Indexing System for Humans – Enhanced* [online]. Available: http://swish-e.org/ [13 December 2004].

References

Van de Sompel, H. and C. Lagoze (2000) The Santa Fe Convention of the Open Archives Initiative, *D-Lib Magazine, 6*(2) [online]. Available http://www.dlib.org/dlib/february00/vandesompel-oai/02vandesompel-oai.html [8 August 2003]

Van Rossum, G. 2005. *Python Library Reference* [online]. Available: http://docs.python.org/lib/lib.html [11 October 2005].

Visokey, P., Hong, Q. and Mulyani, Y. 2000. *COM/DCOM & COM+: A Primer on the Evolution of a Microsoft Development Environment* [online].
Available: http://csis.pace.edu/~ctappert/cs616-02/pres-com.doc [11 November, 2004].

Witten, I. 2003. Examples of Practical Digital Libraries Collections: Built Internationally Using Greenstone, *D-Lib Magazine*, 9(3) [online].
Available: http://dlib.org/dlib/march03/witten [12 July, 2003].

Witten, I., Loots, M., Trujillo, N. and Bainbridge, D. 2002. The promise of digital libraries in developing countries. *The Electronic Library*, 20(1):7-13.

Witten, I., Bainbridge, D. and Boddie, S. 2001. Power to the people: end-user building of digital library collections. *Proceedings of the 1st ACM/IEEE-CS Joint Conference on Digital Libraries*, Roanoke, VA, 24–28 June 2001, 94-103. New York, NY, USA: ACM Press.

Appendix A

ODL/OAI Interface Specification

Version 1.0

A.1 Introduction

For component to function effectively with the visual development environment currently being constructed, the interface for such an interaction must be clearly defined. This document is intended to provide a quick and easy understanding of the manipulation of remotely located digital library components. Numerous examples are used throughout the document to illustrate the basic features of the interfaces and how they may be accessed.

A.2 Conventions used in this document

Directories, files, actual input and output are written in `Courier.` Examples of commands to be executed in this document are demonstrated using Perl scripts.

A.3 Component Interface

The following section outlines the interface of an auto-configurable component.

A.3.1 Root directory

The root directory of a component is defined as the directory containing the configuration files for the component. Minimally, the root directory should contain the following interfaces/scripts:

- autoconfig
- getType
- getInstance
- listInstances

- removeIntstance

A folder named `type` may be present containing the type description.

The root directory may also contain instances of a component and a `template` folder. In addition to the autoconfig interface, a configure interface may also be present for manual configuration.

A.3.2 autoconfig

The function of this interface is to automatically configure a component based on instance data retrieved from standard input. This instance information is processed and errors are sent to standard output. In the absence of errors, for a new instance the configuration information is stored in a new directory, or an old directory rewritten in the case of modifying an existing instance.

A.3.2.1 Arguments

The autoconfig interface is called with a single argument –the name of the instance.

E.g., from the root directory execute:

`./autoconfig.pl test` (where test is the name of the instance)

A.3.2.2 Input

It then takes a complete instance description as input. This instance information should validate against the type description of that component. An example is shown below.

Table 1 An example instance description input is as follows:

```
<?xml version="1.0"?>
<odlsearch xmlns="http://oai.dlib.vt.edu/ODL/IRDB/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://oai.dlib.vt.edu/ODL/IRDB/config
                         http://oai.dlib.vt.edu/ODL/IRDB/config.xsd">
       <repositoryName>the Rep name</repositoryName>
       <adminEmail>root@cs.uct.ac.za</adminEmail>
       <database>DBI:mysql:Demo</database>
       <dbusername>root</dbusername>
       <table>stuff</table>
       <archive>
              <identifier>mytest</identifier>
              <url>http://talc.cs.uct.ac.za/cgi-bin/OAI-
XMLFile/XMLFile/mytest/oai.pl</url>
              <metadataPrefix>oai_dc</metadataPrefix>
              <interval>86400</interval>
              <interrequestgap>10</interrequestgap>
              <overlap>1</overlap>
              <granularity>second</granularity>
       </archive>
</odlsearch>
```

A.3.2.3 Output

The output of autoconfig takes one of the following two forms depending on whether errors occurred or not. The errors are outputted as an XML document and should validate against the following XML schema:

Table 2 Schema that the autoconfig output should validate against.

```
<schema targetNamespace="http:/simba.cs.uct.ac.za/Autoconfig"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:odl="http:/simba.cs.uct.ac.za/Autoconfig" elementFormDefault="qualified"
attributeFormDefault="unqualified">
       <element name="autoconfig">
              <annotation>
                     <appinfo>The output of the autoconfig interface</appinfo>
              </annotation>
              <complexType>
                     <sequence>
                            <element name="configured">
                                   <simpleType>
                                          <restriction base="string">
                                                 <enumeration value="yes"/>
                                                 <enumeration value="no"/>
                                          </restriction>
                                   </simpleType>
                            </element>
                            <element name="error" type="string" minOccurs="0"
                                     maxOccurs="unbounded"/>
                     </sequence>
                </complexType>
         </element>
</schema>
```

If an error occurred, the output should be an XML document similar to the one below:

Table 3 Example of the XML output of autoconfig indicating at least one error occurred:

```
<?xml version="1.0"?>
<IRDBautoconfig xmlns="http://simba.cs.uct.ac.za/xmlfile"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://simba.cs.uct.ac.za/xml/xmlfile
http://simba.cs.uct.ac.za/xml/xmlfile.xsd">
       <configured>no</configured>
       <errors>
              <call>Missing configuration Name</call>
              <environment>DBI module not installed</environment>
              <instance>
                     <table>Missing Databse Table Prefix name</table>
                     <adminEmail>Incorrect format for Admin email</adminEmail>
                     <archive>
                            <url>Missing base url</url>
                            <identifier>Missing arhive identifier</identifier>
                     </archive>
              </instance>
       </errors>
</IRDBautoconfig>
```

If no errors occurred, the output of autoconfig should be similar to the following XML document.

Table 4 Example of the XML output indicating no errors occurred:

```
<?xml version="1.0"?>
<IRDBautoconfig                        xmlns="http://simba.cs.uct.ac.za/xmlfile"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://simba.cs.uct.ac.za/xml/xmlfile
http://simba.cs.uct.ac.za/xml/xmlfile.xsd">
       <configured>yes</configured>
       <errors></errors>
</IRDBautoconfig>
```

A.3.3 getType

The getType interface returns to standard output an XML schema with the type description for a particular component against which the instance information must validate.

A.3.3.1 Arguments

This interface takes no arguments.

E.g., from the root directory execute:

`./getType.pl` (no arguments)

A.3.3.2 Output

A typical output is displayed below.

Table 5 An example type description output is as follows:

```
<schema targetNamespace="http://oai.dlib.vt.edu/ODL/IRDB/config"
xmlns:odl="http://oai.dlib.vt.edu/ODL/IRDB/config"
xmlns="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
       <annotation>
              <documentation>
         Configuration for ODL-Search IRDB implementation
      </documentation>
       </annotation>
       <element name="odlsearch" type="odl:odlsearchType"/>
       <complexType name="odlsearchType">
              <sequence>
                     <element name="repositoryName" type="string"/>
                     <element name="adminEmail" type="string"/>
                     <element name="database" type="string"/>
                     <element name="dbusername" type="string" minOccurs="0"/>
                     <element name="dbpassword" type="string" minOccurs="0"/>
                     <element name="table" type="string"/>
                     <element name="archive" type="odl:archiveType"/>
              </sequence>
       </complexType>
       <complexType name="archiveType">
              <sequence>
                     <element name="identifier" type="string"/>
                     <element name="url" type="anyURI"/>
                     <element name="metadataPrefix" type="string"
maxOccurs="unbounded"/>
                     <element name="interval" type="integer"/>
                     <element name="interrequestgap" type="integer"
minOccurs="0"/>
```

```
                        <element name="set" type="string" minOccurs="0"/>
                        <element name="overlap" type="integer" minOccurs="0"/>
                        <element name="granularity" type="odl:granularityType" 
minOccurs="0"/>
                </sequence>
        </complexType>
        <simpleType name="granularityType">
                <restriction base="string">
                        <enumeration value="day"/>
                        <enumeration value="second"/>
                </restriction>
        </simpleType>
</schema>
```

A.3.4 listInstances

A call to the listInstances interface returns a list of all the instances of that component. This is particularly useful for modifying a specific instance when the name is unknown or, for example, discovering available archives in a data provider component to which an ODL-search component can connect.

A.3.4.1 Arguments

This interface takes no arguments.
E.g., from the root directory execute:
`./listInstances.pl` (no arguments)

A.3.4.2 Output

The output of listInstances is simply list a of all the instance descriptions of that component on the server. The instances are displayed in alphabetical order, with the name of the instance preceding each instance description. The output of listInstances validates against the following schema:

Table 6 The schema for the listInstances interface

```
<schema targetNamespace="http://simba.cs.uct.ac.za/listInstances" 
xmlns:odl="http://simba.cs.uct.ac.za/listInstances" 
xmlns="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" 
attributeFormDefault="unqualified">
        <element name="listInstances">
                <annotation>
                        <documentation>Output of the listInstances 
interface</documentation>
                </annotation>
                <complexType>
                        <sequence minOccurs="0" maxOccurs="unbounded">
                                <element name="name" type="string"/>
                                <element name="instanceDescription">
                                        <complexType>
                                                <sequence>
                                                        <any namespace="##any" 
processContents="strict"/>
                                                </sequence>
                                        </complexType>
                                </element>
                        </sequence>
                </complexType>
        </element>
</schema>
```

Below is an example of the XML output from listInstances:

Table 7 An example output of the listInstances interface is as follows:

```
<?xml version="1.0"?>
<listInstances xmlns="http://simba.cs.uct.ac.za/instance"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://simba.cs.uct.ac.za/instance
                         http://simba.cs.uct.ac.za/instance.xsd">
<name>TEST 1 </name>
<instance>
<xmlfile xmlns="http://oai.dlib.vt.edu/xmlfile"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://oai.dlib.vt.edu/xmlfile
                         http://oai.dlib.vt.edu/xmlfile.xsd">
        <repositoryName>test1 OAI Archive</repositoryName>
        <adminEmail>oai@oai.dlib.vt.edu</adminEmail>
        <archiveId>test1</archiveId>
        <datadir>data</datadir>
        <filematch>[^\.]+\.[xX][mM][lL]$</filematch>
        <metadata>
                <prefix>oai_dc</prefix>
                <namespace>http://www.openarchives.org/OAI/2.0/oai_dc/</namespace>
                <schema>http://www.openarchives.org/OAI/2.0/oai_dc.xsd</schema>
                <root>dc</root>
                <transform>/usr/local/bin/xsltproc dc.xsl - |</transform>
        </metadata>
        <metadata>
                <prefix>oai_vracore</prefix>

        <namespace>http://www.gsd.harvard.edu/~staffaw3/vra/vracore3.htm</namespace>
                <schema>http://oai.dlib.vt.edu/OAI/1.1/oai_vracore.xsd</schema>
                <root>vra</root>
                <transform>/usr/local/bin/xsltproc vra.xsl - |</transform>
        </metadata>
</xmlfile>
</instance>

<name>TEST2</name>
<instance>
<xmlfile xmlns="http://oai.dlib.vt.edu/xmlfile"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://oai.dlib.vt.edu/xmlfile
                         http://oai.dlib.vt.edu/xmlfile.xsd">
        <repositoryName>test2 OAI Archive</repositoryName>
        <adminEmail>oai@oai.dlib.vt.edu</adminEmail>
        <archiveId>test2.oai.dlib.vt.edu</archiveId>
        <recordlimit>1</recordlimit>
        <datadir>data</datadir>
        <longids>yes</longids>
        <filematch>[^\.]+\.txt$</filematch>
        <metadata>
                <prefix>oai_dc</prefix>
                <namespace>http://www.openarchives.org/OAI/2.0/oai_dc/</namespace>
                <schema>http://www.openarchives.org/OAI/2.0/oai_dc.xsd</schema>
                <transform>/usr/local/bin/xsltproc dc.xsl - |</transform>
        </metadata>
</xmlfile>
</instance>

<name>TEST3</name>
<instance>
<xmlfile xmlns="http://oai.dlib.vt.edu/xmlfile"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://oai.dlib.vt.edu/xmlfile
                         http://oai.dlib.vt.edu/xmlfile.xsd">
```

```
       <repositoryName>test3 OAI Archive</repositoryName>
       <adminEmail>oai@oai.dlib.vt.edu</adminEmail>
       <archiveId>test3</archiveId>
       <recordlimit>500</recordlimit>
       <datadir>data</datadir>
       <longids>no</longids>
       <filematch>[^\.]+\.[xX][mM][lL]$</filematch>
       <metadata>
              <prefix>oai_dc</prefix>
              <namespace>http://www.openarchives.org/OAI/2.0/oai_dc/</namespace>
              <schema>http://www.openarchives.org/OAI/2.0/oai_dc.xsd</schema>
       </metadata>
</xmlfile>
</instance>
<listInstances>
```

A.3.5 getInstance

The getInstance interface retrieves a specific instance of a particular component and sends it to standard output.

A.3.5.1 Arguments

The getInstance interface is called with a single argument –the name of the instance.
E.g., from the root directory execute:
`./getInstance.pl test` (where test is the name of the instance)

A.3.5.2 Output

The output from getInstance (see below), the instance description, should validate against the type description of that component.

Table 8 An example instance description output is as follows:

```
<?xml version="1.0"?>
<odlsearch xmlns="http://oai.dlib.vt.edu/ODL/IRDB/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://oai.dlib.vt.edu/ODL/IRDB/config
                         http://oai.dlib.vt.edu/ODL/IRDB/config.xsd">
       <repositoryName>the Rep name</repositoryName>
       <adminEmail>root@cs.uct.ac.za</adminEmail>
       <database>DBI:mysql:Demo</database>
       <dbusername>root</dbusername>
       <table>stuff</table>
       <archive>
              <identifier>mytest</identifier>
              <url>http://talc.cs.uct.ac.za/cgi-bin/OAI-
XMLFile/XMLFile/mytest/oai.pl</url>
              <metadataPrefix>oai_dc</metadataPrefix>
              <interval>86400</interval>
              <interrequestgap>10</interrequestgap>
              <overlap>1</overlap>
              <granularity>second</granularity>
       </archive>
</odlsearch>
```

A.3.6 removeInstance

Given the name of an instance, this interface deletes that specific instance.

A.3.6.1 Arguments

The removeInstance interface is called with a single argument –the name of the instance.
E.g., from the root directory execute:
`./removeInstance test` (where test is the name of the instance)

A.3.6.2 Output

The output of this interface is simply a 1 for success or a 0 for unsuccessful deletion.

Appendix B

Evaluation Methodology

B.1 Pre-Test Questionnaire

Welcome to the Blox system test!
Please start by answering the following background questions:

Program and current year of study (e.g. BA 2rd year, Msc):

Major (e.g. Computer Science):

Have you ever used modeling software or component-based development system such as RationalRose, Microsoft Visual Studio, Delphi or Visual Basic? (yes/no)

Have you ever used a Unix/Linux operating system before?

Have you installed a CGI web-server script/application before (e.g., website guestbook, website counter)?

Do you know what a digital library is?

Have you ever installed a digital library before?

How would you rate your understanding of the functioning of web services? (Involving UDDI, WSDL etc. Circle appropriate answer)

Excellent Good Adequate Poor Terrible

Appendix B – Evaluation Methodology

How would you rate your proficiency with ODL Components?
Excellent Good Adequate Poor Never heard of them

Rate your familiarity with the OAI Protocol for Metadata Harvesting
Excellent Good Adequate Poor Never Heard of it

B.2 Pilot Study Background Information Handout

This handout introduces you to a few basic concepts required to successfully create a digital library.

What is a digital library?

A digital library is an electronic system that allows one to retrieve resources such as documents and multimedia and may contain services such as the ability to browse through the resources, rate them or annotate them with extra information.

These resources are typically stored as metadata in the archive. Each metadata item is encoded using one or more standard formats, the most common one is Dublin Core, represented as "oai_dc".

Resources may be associated with one or more sets. Sets enable you to group your resources in the archive. E.g. your theses on "Education in IT" may be associated with the set "compsci and the set "edu".

What is the OAI and OAI-PMH?

The Open Archives Initiative (OAI) is a consortium created to address issues of interoperability amongst digital libraries. The OAI came up with a protocol digital libraries must conform to in order for them to exchange metadata. This protocol is known as the Open Archives Initiative's Protocol for Metadata Harvesting (OAI-PMH).

The OAI-PMH consists of 6 requests:

- Identify (What kind of archive are you?)
- ListSets (What sets do you have?)
- ListIdentifiers (Give me the identifiers of all your records)
- ListMetadataFormats (What metadata prefixes do you support? E.g. oai_dc)
- ListRecords (Give me all your records)
- GetRecord (Give me a specific record)

This protocol is used in order for one archive to obtain specific data from another archive in a format that is understood by both parties.

What are ODL components?

Open Digital Library (ODL) components are digital library components that utilise the concepts of the OAI-PMH. The idea behind it is, if two archives can communicate using the OAI-PMH, why can’t for instance, a search engine, use the same protocol to communicate with a user interface? So the ODL was developed to implement this idea by creating components which implement the OAI-PMH’s philosophy.

Components used in this exercise

In this experiment, we will use the following components:

Box	**A lightweight database driven archive**
XMLFile	A data provider module that creates an OAI-compliant repository out of a set of XML files that contain the metadata.
OADP	The Open Archives Data Provider allows you to use other archives available on the WWW with the Graphical User Interface.
IRDB	A simple search engine
Swish-E	A non-ODL search engine interfaced to mimic IRDB.
DBBrowse	A browsing engine
UI	A simple user interface
PhpBB	A popular PHP-driven bulletin board
DBRate	A component that enables you to rate resources.

What is a Type?

A Type in the context of this research is a skeleton for a component. It tells you what information is needed for the component to work.

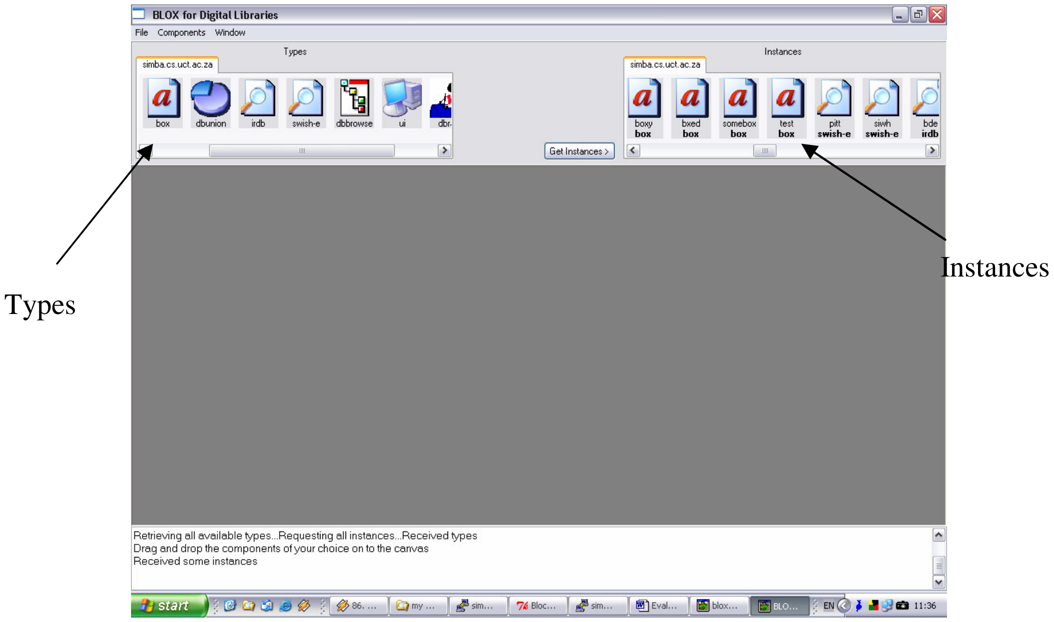
What is an Instance?

An Instance is the result of configuring a component. I.e., you have a type, you fill in the information requested and you get an instance.

The Blox+ System

The Blox+ system is a GUI that enables you to get components hosted on several servers and use these components to create a DL. The instances in the Blox+ system have the word **created** on the title bar.

In the diagram of the Blox+ System below, the Types are on the left and the Instances are on the right.

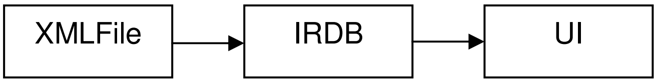


Retrieving all available types...Requesting all instances...Received types
Drag and drop the components of your choice on to the canvas
Received some instances

B.3 Command-Line (Manual) Configuration

This is an exercise to demonstrate the installation, use, and composition of Open Archives and Open Digital Library components. The exercise comprises the following steps:
Configuring an XMLFile Open Archive component
Connecting an Open Archive (Box) to a search engine component (IRDB)
Adding a simple WWW interface to the search engine component

The goal of the exercise is to create a simple digital library system out of components, with the following system architecture:



The components have already been installed for you, so just follow the steps outlined below in order for you to create a digital library.

Comments about some of the components and tools are in italics. These are purely informational and not part of the sequence of steps for the exercise.

1. Configuration of a simple Open Archive (XMLFile)

XML-File is a data provider module that creates an OAI-compliant repository out of a set of XML files that contain the metadata. Using this requires minimal effort while retaining all the flexibility of the OAI protocol.

1.1 Configuration

Change to the XMLFile component directory located at /home/Blox/public_html/cgi-bin/OAI-XMLFile/XMLFile/

You should be in the cgi-bin/ directory so do:

```
cd OAI-XMLFile/XMLFile
```

Notice there are a couple of folders such as test1 and test2. These are previously configured XMLFile instances. If we had to create a new instance, to use it in any meaningful way we would first need to put data into our archive in the form of XML files. So we will just use one of the existing instances which already contain xml files with the data.

1.2 Testing

Let's test the XMLFile instance named test2 to see if it works as it should. There are two ways of doing this:

1.2.1 Direct execution

First we can test by directly invoking the script to see if the script executes without any errors. Change to the "test2" directory

```
cd test2
```

and run the following command:

```
QUERY_STRING='verb=Identify' ./oai.pl
```

You should see the XML response to the Identify OAI-PMH service request.

1.2.2 Internet Explorer

Run Internet Explorer and type in the following URL (the baseURL of your component as well as your OAI-PMH service request):

```
http://simba.cs.uct.ac.za/~Blox/cgi-bin/OAI-
XMLFile/XMLFile/test2/oai.pl?verb=Identify
```

You should get the same response as before.
(This also works in Netscape 6 but you have to "View Source" to see the output nicely formatted.)

2. Connecting an Open Archive to a search engine component (IRDB)

IRDB is a small-scale search engine that harvests data from an Open Archive and has a simple machine interface to issue queries and return results.

For this part of the exercise we will use a MySQL database.

2.1 Configuration

Change to “ODL-IRDB-1.3/IRDB” directory (you may have to go up to the cgi-bin directory first)

```
cd ../..
cd ODL-IRDB-1.3/IRDB
```

Run

`./configure.pl myirdb` (or any name of your choice)

Answer the questions asked by the configuration script as listed below:

Database Driver : mysql
Database : blox
Username : Blox
Password (leave blank):
Database Table: <AnythingYouWant>
Archive Identifier: test

Repository Name, Admin Email: leave at defaults

Archive URL: enter the baseURL for your XMLFile archive:
http://simba.cs.uct.ac.za/~Blox/cgi-bin/OAI-XMLFile/XMLFile/test2/oai.pl

metadataPrefix: oai_dc

Use defaults for everything else.

2.2 Testing

Populate with data from the XMLFile archive by running:

```
myirdb/harvest.pl
```

Run a test query from the command-line:

```
myirdb/testsearch.pl “someid”
```

Now run the same test query using the machine (ODL) interface by issuing:

```
QUERY_STRING='verb=ListRecords&metadataPrefix=oai_dc&set=odlsearc
h1/someid/1/10' myirdb/search.pl
```

3. Adding a simple WWW interface to the search engine component

A search engine is not very useful without a user interface. UI is a simple user interface that can be connected to your IRDB search engine.

3.1 Configuration

By now you should have gotten the hang of this:

Change to the BRSUI/BRSUI directory

```
cd BRSUI/BRSUI
```

Run

`./configure.pl myui` (or any name of your choice)

Follow the on-screen instructions entering the baseURL of your search engine (http://simba.cs.uct.ac.za/~Blox/cgi-bin/ODL-IRDB-1.3/IRDB/myirdb/search.pl -- once again replacing myirdb with whatever you named your IRDB component)

Leave the prompts for other baseURLs such as browse and rate blank.

3.2 Testing the interface

Enter the UI's baseURL into your web browser as:

```
http://simba.cs.uct.ac.za/~Blox/cgi-bin/BRSUI/BRSUI/myui/index.pl
```

Try entering the query "`someid`" in the input box. You will get a list of titles of records that match the search query.

You have just built a simple digital library out of components to correspond to the architecture shown on the first page

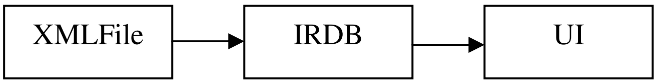
That's it!

B.4 GUI-based configuration

This section will guide you through the process of using the Graphical User Interface to create a digital library from distributed components.

The GUI enables a user to connect to one or more servers hosting digital library components and create new instances of those components or use existing instances in order to create a digital library.

In this exercise we are going to recreate the 3-component digital library depicted in the following diagram, using the GUI:



Italics are used to explain what you are about to do and are not actual tasks that need to be performed.

<u>TASKS</u>

1. Start-up the GUI by going to Start->Programs->Blox ->BloxClient

2. *The first thing you need to do is specify the server(s) on which the components you wish to use are located:*

   On the menu select File->Server Manager

3. *For this exercise all the components are located on one server, and that server is on the "saved list".*

   Double-click "simba.cs.uct.ac.za" server to load it and then click "OK" at the bottom of the dialog.

   All the component types registered on that server will now be retrieved.
   We will now create the digital library depicted above.

4. Drag and drop the **Box**, **IRDB** and **UI** components onto the canvas.

5. *Notice the three tabs: Details, Configuration and Connections on each component.*

   For all three components:
   Name the components by entering any name of your choice in the input field "name" in the **Details tab** using a different name for each of the components.

6. *In the **Configuration tab**, all the fields have been populated with defaults so they should work without further modification.*

Modify some of the information in the **configuration tab** of the **UI** component, clicking on the help button for more information on the purpose of the field.

7. Now click on the **connection tab** of all three components.

8. Connect the search component (IRDB) to the archive (Box) by clicking and dragging the archivebaseurl link into the connection tab window of the Box component.

9. Connect the UI component to the search component (IRDB) by clicking and dragging the serarchbaseurl link into the connection tab window of the IRDB component.

10. Now to publish your digital library, select Components->Publish from the menu.

11. If there were no errors during the process, you will be asked if you want to view your digital library. Click “Yes”.

12. Click on the “Copy” button, open your browser and paste the link in the address textbox, then press enter. Your DL user interface will now be loaded.

13. Test your DL by entering “test” into the input box. Records that match your search criteria will now be retrieved.

That’s it!

B.5 Final Evaluation

Thank you for taking the time to complete the exercises. Please answer the following questions by indicating (circling the closest choice) to what degree you agree or disagree with the statements as related to the experiment on configuring OAI/ODL Components using Blox.

General Questions

I understand how the Blox system functions.
Strongly Agree Agree Neutral Disagree Strongly Disagree

I understand the concept of types and instances.
Strongly Agree Agree Neutral Disagree Strongly Disagree

I think it is a good idea to be able to graphically assemble systems from distributed components.
Strongly Agree Agree Neutral Disagree Strongly Disagree

I found the interface consistent with my previous experiences with Visual IDEs.
Strongly Agree Agree Neutral Disagree Strongly Disagree

I would consider using ODL and OAI components if I need to build a system with requirements similar to the exercise.
Strongly Agree Agree Neutral Disagree Strongly Disagree

I would use a system such as Blox to create digital libraries if I had the need to create one.
Strongly Agree Agree Neutral Disagree Strongly Disagree

I would use a system such as Blox to connect and configure other types of components (e.g. other Web Services, or COM or CORBA objects) if it had support for them.
Strongly Agree Agree Neutral Disagree Strongly Disagree

Usability Questions

When I created a digital library using Blox, it produced the digital library I expected it to produce.
Strongly Agree Agree Neutral Disagree Strongly Disagree

The interface responded as I expected it to.
Strongly Agree Agree Neutral Disagree Strongly Disagree

I found it easy to use Blox to accomplish the tasks.
Strongly Agree Agree Neutral Disagree Strongly Disagree

I found it easy to apply my previous GUI experiences with Blox.

Strongly Agree Agree Neutral Disagree Strongly Disagree

I like the idea of representing a component as a window.
Strongly Agree Agree Neutral Disagree Strongly Disagree

Comparison Questions

I found using Blox to create a digital library easier than manually using OAI/ODL Components.
Strongly Agree Agree Neutral Disagree Strongly Disagree

I found using Blox to create a digital library faster than manually using OAI/ODL Components.
Strongly Agree Agree Neutral Disagree Strongly Disagree

What was the most difficult part of the whole exercise?

What in the entire exercise did you least understand?

State anything you would add, subtract or otherwise do differently to the system.

Enter any additional comments:

THE END

Thank you for your time

.

Appendix C

Test Users' Comments

Note: Similar comments are generally not repeated

C.1 What was the most difficult part of the whole exercise?

- None. All straightforward
- Nothing really, just type and voila!
- Creating the library using the command line
- Creating a digital library without the GUI
- You had to enter many characters exactly as they were written
- The command prompt section – having to re-enter long addresses when all I got wrong was one letter!
- Manually creating component, it is very case sensitive and that is a problem.
- Understanding the objective of the experiment, initially
- Trying to understand what was happening in the text window.
- I think if I understood a bit about computer science, then it would have been very easy. But it was not a difficult task.
- Trying to understand how to create the search engine.
- Typing in the long links for the search baseURL. Is it possible to create a command line within the program similar to Unix which keeps record of commands you have typed? So if a mistake is made you can just press the up arrow instead of retyping the whole command segment.
- Finding out what each component means and the purpose of each field.

- Reading the instructions. I'm dyslexic.

C.2 What in the entire exercise did you least understand?

- The DL terminology
- The technical computer language
- Connecting the types with XMLFile using the command line
- What Blox is, how it works, what makes Blox different from all the things I don't know.
- All the setting up in the prompt version (perhaps I didn't really take enough time to read it) but that's what Blox is there to "cover".
- Everything was pretty simple; I just had to follow instructions.
- I understood everything because I got some assistance.
- The file names and changing into directories
- Digital library concept. More research on the subject would help improve understanding
- ODL – Still not sure about it.
- Everything!
- The stuff with cd and ls. Difficult!
- What the XMLFile does. I just tested it but didn't know what it was supposed to do.
- Wasn't quite sure why the rate had to be connected to the XMLFile as well.
- What is Blox is really about? And what are we searching? Web pages, files, or just a word in a database?
- The graphical part of it is clear, though difficult.
- The XML that appeared when in the command line setup.
- Initially what each of the components did, however this became clear later.

C.3 State anything you would add, subtract or otherwise do differently to the Blox system.

- Don't make the arrows change colour
- Have an item appear in the linked-to component that is connected to the original item
- Deletable arrows
- When you want to view the library that you have created, you need to click OK, and it should take you directly to the library
- I'm afraid I can can't add to something I don't understand
- It is good as it is
- People should be told to create ui->irdb->xmlfile – i.e. work from left to right (it's more intuitive).

- I think the arrows should be more noticeable, as it's a new way of representing things (window-to-window connections).
- A built-in preview of the digital library as it appears in the browser
- The way items appear when you had to drag. I think they should be a bit bold
- Don't know how, but it is a little slow.
- I would start the Blox system with a help window containing a crash course in digital libraries and an explanation of where the Blox system fits in.
- No. I think its good, but I'm no expert on the subject.
- Add a help button/icon at the top, next to File, Components, Window.
- Its perfect.
- A help button for troubleshooting, guidelines and stuff.
- I think the system is cool and easy to understand considering that it was my first time using it.
- The arrows should be straighter.
- Automatically open the default web browser instead of copy/paste, and more clearly defined links between components in the GUI.
- Types and Instance "blocks" on top of GUI could be better spread and thus easier to view.

C.4 Enter any additional comments

- This is the new system to use in future.
- A more detailed explanation on how things work before beginning the exercise would have been useful
- A very easy-to-use and visually pleasing interface!
- The experiment was very easy to follow
- Get Instances took a while, but I didn't see and indication that it was doing something.
- Blox makes designing the digital library much easier
- The connecting of GUI is somehow confusing at first, although it is really interesting
- Blox is very easy to use, the instructions you gave were helpful.
- Pictures on the digital library User Interface.
- I think that the Blox program is very good, personally I knew nothing about components and stuff before but with this program it has made me understand.
- Manual part is simple for someone who knows how to use Unix/Linux but not every computer users e.g. users using windows.
- I'm very fascinated by the subject, and the tutor was very helpful.
- Pretty schnazzy to create own search engine.

- I think the instructions were not too clear, because I kept asking what I had to do next.
- My first programming experience was fantastic.
- I don't know what other systems are like, but I would advice people to use Blox.
- Can I get the copy of this program, because in second year we are doing garbage stuff which is not interesting, please!
- Opening the digital library to modify was a bit slow.
- Using the windows and the drag-and-drop was MUCH simpler and user-friendly than the manual part.
- Satisfied customer!
- Brilliant idea!
- The IDE could be easier to use.
- Good layout of Blox GUI shows how the components fit together in an easy to understand layout. Dragging and dropping of fields between components is a good idea, saves unnecessary typing.