Flexible Packaging Methodologies for Rapid Deployment of Customisable Component-based Digital Libraries

A dissertation submitted to the Department of Computer Science,
Faculty of Science at the University of Cape Town
in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in
Computer Science

— Siyabonga Mhlongo —

Supervisor
Dr. Hussein Suleman

UNIVERSITY OF CAPE TOWN

June, 2006

ABSTRACT

Software engineering is a discipline concerned with manufacturing or developing software. Software plays a pivotal role in everyday life, an absence of which will be devastating to a number of governmental, recreational and financial activities, amongst many others. One of the latest branches of software engineering, component-based software engineering, is concerned with the development of software systems using already existing components which speculatively will ensure rapid and inexpensive software development processes.

Parallel with the advances in software engineering, the field of digital libraries — a field dealing with Web-based access to and management of structured digital content — has adopted this development model from software engineering to shift focus from developing and using traditionally monolithic software systems to developing and using more flexible component-oriented software systems.

Since componentised development approaches are relatively recent, other areas such as packaging and managing component-based software systems still remain unattended to. This dissertation presents research on techniques and methodologies for packaging customisable component-based digital libraries such that deployment is rapid and flexibility is not compromised. Although the reference point of this research was that of component-based digital library systems, it is believed that this research can be generalised across the family of Web-based component-based software systems.

An outcome of this research was a prototype packaging system consisting of a pair of tools: a package builder tool and a package installer tool. This packaging system was developed to model the ideas and methodologies that were identified as important to the processes of packaging and installing component-based digital library systems. These tools consequently underwent a user evaluation study whereby they were evaluated for understandability, usability and usefulness to the processes of packaging and installing component-based digital libraries.

A key contribution of this research was identifying requirements for a generic component packaging framework. For a component to be seen as ”fit-to-package”, it must posses the following at the very least: the component must be configurable automatically; the component must have a formal description of its dependency software; there must be formal descriptions that describe individual components as well as systems composed of components; and there must be a way whereby installation questions are formally encoded such that components are able to correctly receive configuration information.

In totality, this research has shown that component-oriented software development approaches can benefit from an infrastructure which allows for component-based software systems to be composed, distributed and installed effortlessly.

ACKNOWLEDGEMENTS

Thanks are due
to my supervisor, for his patience, understanding and
impeccable guidance throughout this research,
to my family, for their continuous support and encouragement and
to my friends.

Thanks are given to the National Research Foundation (NRF) and Telkom SA Limited Center of Excellence (CoE) for the funding they provided for this research.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The Web, a global collection of resources accessible via the Internet, is one of the most useful inventions to come about in the field of computing yet. It has informed the current state of affairs across myriad societies and disciplines since its inception almost two decades ago. A consequent of the Web is the Web-based services paradigm of computing. With this paradigm, corporate and academic institutions alike have had much success in terms of economic growth, structural reformation and deliverance, mostly through an adoption of some or other kind of Web-based solutions [21, 25, 36]. With current and predicted future Web-based applications, the Web will remain integral to daily life, making it more comfortable, more endurable and more affordable.

To take full advantage of Web-based services and what they have to offer, software engineers need to equip themselves with necessary tools and strategies to deal with the dynamics of this computing paradigm. Inevitably for software developers, developing Web-based applications will become a natural choice. This will be driven by an expectation of simpler, better and faster methodologies with which to configure, deploy and manage applications from the general public. Dividing a software system into smaller parts (or components) has become common practice in software development in general and is encouraged by the software engineering field [14, 15]. This already is a remarkable step towards simplifying software development processes which offers a multitude of benefits such as component reuse, which are otherwise unattainable. However, as good a practice as this might be, there are challenges associated with this approach. One of these challenges is the effort required to develop, maintain and provide support for these components. This research has focused on investigating the effects of introducing a packaging solution for Web-based component-based software systems, paying particular attention to the processes of embodying a Web-based software system as a single installable package, configuring and deploying the resultant package as well as maintaining the deployed package throughout its lifetime.

This chapter introduces the bigger project which this research is part of. The research problem is briefly introduced in Section 1.2. Section 1.3 presents the domain and scope within which this research is confined while Section 1.4 gives a detailed breakdown of the structure of this dissertation thenceforth.

1.1 FLEXIBLE DIGITAL LIBRARIES

The Flexible Digital Libraries (FDL) project is a research initiative which aims to investigate certain issues surrounding component-based approaches to building and maintaining Web-based component-based software systems in the context of component-based digital library systems. Suleman *et al.* break down the aims of the FDL project into the following subgoals [48]:

Visual Component Composition — Many software components provide a manner with which they can be instantiated and be connected with other software components or complete software systems. Usually, this is done through interfaces that are command-line-based and difficult to work with. Eyambe [19, 20] has shown that a component-based digital library can be constructed visually using the BLOX system [37].

Interface Customisation — This is work in progress which deals with the customisation of digital library interfaces to accommodate varying system configurations. The general idea is that a digital library user can design, from within a Web-based interface, a front-end interface by selecting elements (including but not limited to text boxes, images and page types) to form part of the interface, identifying digital library services (such as content searching, browsing and rating) that are to be accessible over this interface and defining the workflow to which the selected elements and services must adhere. Preliminary results obtained through a series of participatory design sessions have shown that this approach is preferable over traditional approaches seen in many digital library systems where interfaces are either hard-coded or require programming knowledge for their customisation.

Flexible Component Packaging — In order to show that there is no loss of generality in employing component-based approaches over traditionally monolithic systems, components must be packaged into a single distributable package together with their formal descriptions, connectivity specifications as well as any other entities necessary for installation. This package must appear as a single entity and should maintain flexibility while promoting rapid deployment. Research presented in this dissertation attends to this subgoal. Section 1.2 elaborates further on this topic.

Scalability of Component-based Systems — This part of the FDL project is concerned with identifying techniques and methodologies whereby any relatively large component-based digital library system can be distributed over a cluster of computers. Initial research has demonstrated that it is possible for a simple digital library to run on a distributed set of machines. Future work will be looking at issues surrounding migration and replication of component instances on a computing cluster.

The visual component composition and flexible component packaging goals are closely related. Putting this relation into perspective, a digital library user can visually compose a component-based digital library – an outcome of which can be a package representation of this visual composure and possibly a live demonstration of the composed digital library. The BLOX system already provides functionality to instantiate and test a fully functional digital library live on a server. Later chapters discuss how a digital library package can be created as a consequence of this visual composition. Figure 1.1 demonstrates these two outcomes.

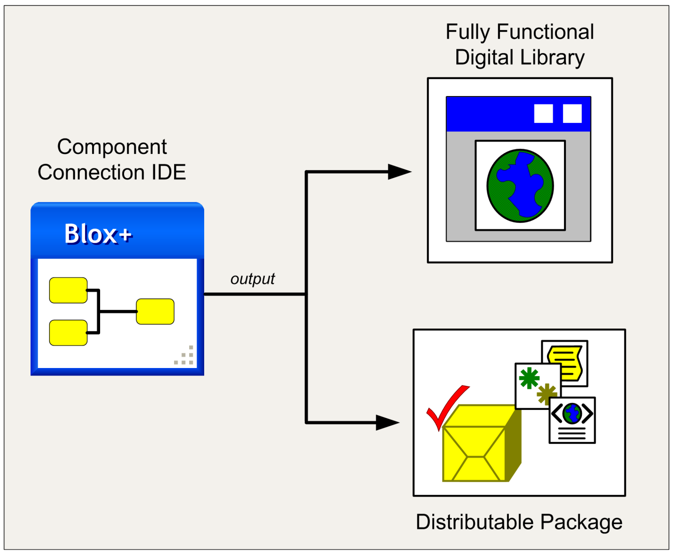


Figure 1.1: Visual component composition and flexible component packaging relationship

1.2 Flexible Component Packaging

Recent efforts in the field of digital libraries have resulted in a component-based approach to composing digital libraries, yielding numerous complexities despite significant advantages inherent in the approach. One such complexity is that of component management. Component management encompasses the processes of building a software package for a digital library from a resource pool of components, deploying this software package on a target system and ensuring proper functioning of the package while in operation on a target system.

Digital libraries need to be simple enough to be understood and utilised by everyone. Unfortunately this is far from reality. Instead, digital libraries are perceived by many as systems that only a select few can utilise. Componentising digital libraries is a remarkable step towards achieving this simplicity. However, it is important that any undesired effects, such as managing components, which result from this componentisation are adequately handled.

The main aim of this research was to investigate requirements for a generic component packaging framework by first assessing available resources and subsequently producing a packaging system to model some of these requirements. Strong emphasis was put towards developing techniques and methodologies that can be employed in packaging component-based digital libraries for distribution, deployment as well as managing components belonging to installed component-based digital library packages on a particular system. Another important aspect of this research was to ensure that the flexibility provided by individual components is not compromised in the context of packaging while ensuring a rapid deployment process.

1.3 Research Domain and Scope

This research utilises principles and procedures from established branches of computing and as such, it is logical that terms and concepts that are pivotal throughout this dissertation are clearly introduced before delving into more elaborate detail. Although some definitions are given as the dissertation unfolds, the following definitions present the fields of computing which this research spans.

1.3.1 **Web Service**

A Web service is an application or an integrated set of applications that appear as one and which can be invoked through the Internet. Web services are usually accessed over the Hyper Text Transfer Protocol (HTTP) through messages that are encoded using Web-related standards such as Web Services Description Language (WSDL), Simple Object Access Protocol (SOAP) and eXtensible Markup Language (XML) [7].

Web services offer an array of benefits over using other kinds of Internet protocol-based services. Classically, they provide very loose coupling between applications that use these Web services and the actual Web services themselves. This promotes flexibility and allows for either of these entities to be altered without negatively affecting the other. Other advantages of utilising Web services lie in their interoperability, usability, re-usability, reliability and deployability.

1.3.2 **Information Management**

Historically, information management has been perceived as the process of administering the access to, usage of, dissemination of and life-cycle of traditionally filed, mostly paper-documented information. However, with the proliferation of the Information and Communication Technology (ICT) era, information management has grown to mean a lot more [33]. In addition to the above definition, information management is concerned with administering knowledge that is acquired through some study or experience or both, by ensuring that the tasks of creating, capturing, registering, classifying, indexing, storing, retrieving, modifying and disposing of this knowledge, in digital format or otherwise, are clearly accounted for. Some distinguished branches of information management include imaging, records management, document management as well as knowledge management systems.

1.3.3 **Digital Library**

A digital library is an organised integrated set of services for capturing, cataloguing, storing, searching, protecting and retrieving information all of which is accessible over the Internet. In other words, a digital library is a type of Web service that deals with information and the management thereof through a series of identified services.

There is an array of other definitions, each with its own historical connotations, which have prevailed in this field such as those given by Gladney *et al.* in [24] and Suleman in [47], however the definition that has been provided above will suffice throughout this dissertation. Chapter 2 dwells more on the history of research in the digital libraries field.

1.3.4 **Software Component**

A software component is a piece of software that provides functionality, usually a single function, service or commodity, in a black-box manner through a well defined interface to human agents or other software artifacts. A good component, at least in the context of this research, can be identified as one which can stand alone, is reusable, adheres to some specification and can be easily integrated with other components with relatively minimal effort.

The history of software components goes as far back as in the late 1960s, but it has only been in the last decade or so that the computing industry has taken much interest in the principles of software components. The field of software engineering is especially active when it comes to component research and this has resulted in a number of novel solutions and practices that govern the development of software in a componentwise environment. Chapter 2 gives a more detailed discussion on this matter.

1.3.5 **Software Installation Authoring**

Software installation authoring is a process that deals with drafting the process of installing some software. A common and somewhat traditional manner with which to achieve software installation authoring is to compile a set of instructions, usually as a *readme* file, that users need to read and follow so to accomplish the task of installing some software. Over the years, this practise has proven to be cumbersome. This can be attributed to the facts that: increasingly many derived software solutions tend to require much more attention at installation time; and documentation that accompanies software that is meant to assist users during its installation process is inadequately compiled.

Current software installation authoring practices are aligned more towards automation, which will ultimately make things easier. Chapter 2 elaborates more on a kind of software installation authoring commonly known as software packaging.

1.4 **DISSERTATION OUTLINE**

The remainder of the dissertation is arranged as follows:

Chapter 2: This chapter presents relevant researched information pertaining to this research that was available at the time of writing this dissertation. The chapter first introduces software engineering and one of its branches that deals with component-based software. Subsequent sections in the chapter report on available tools and resources relating to this research project as well as report on the history of digital libraries. The main purpose of the chapter is to draw attention to the convergence of research in the software engineering field towards a component-based platform, while analogously linking these proceedings to developments in the field of digital libraries and Web-based services in general.

Chapter 3: The aim of this chapter is to introduce the problem statement of this research project. This problem statement is then broken down into a series of smaller manageable tasks, which are further structurally outlined. The remainder of the chapter is concerned with the procedures and methodologies that were adopted in addressing each of the problem statement tasks.

Chapter 4: The outcome of Chapter 3 was that a component management platform, a set of tools, to deal with the issues raised by the problem statement was to be developed. This chapter therefore introduces the design and implementation of this component management platform. There is a clear distinction between specifications and tools that formed part of the design and implementation process, supported by adequate illustrations and examples.

Chapter 5: Following the design and implementation of the tools described in Chapter 4 was an appropriate evaluation process in order to assess the acceptability of this component management platform across different criteria. This was in a form of a user study where 25 users participated in a questionnaire-based experimental study. The balance of the chapter presents a thorough statistical analysis of the obtained results and correlates this analysis to the problem statement outlined in Chapter 3.

Chapter 6: This chapter draws conclusions to this research based on the results and the analysis thereof portrayed in Chapter 5. These conclusions show that the approach that has been taken in addressing the problem statement of Chapter 3 is favourable and overwhelmingly preferred.

Chapter 7: In this chapter, other approaches that would speculatively have improved the results of this research are presented. Also included in this chapter are means with which the current component management platform can be improved in order to attend to those needs where minimal effort has been put or to those that are currently unattended.

CHAPTER 2

BACKGROUND

Software engineering, although the very nature of its title warrants it to be an *engineering*, has been caught in a war of decidability of whether it should belong to one field or the other. Contributors of high stature in software engineering have argued on different grounds, some saying that it should be a branch of mathematics, an argument raised by Edsger Dijkstra [16], while others believe that it is somewhat artistic and therefore should be an art, as Donald Knuth famously argued in his seven-volume series entitled The Art of Computer Programming. The reality of this matter is that although software engineering reflects traces of other renowned fields, it is too big a discipline to perfectly fit as a branch of any other. It goes without saying that software engineering is understood differently by different communities and such is even reflected by well established bodies of standards that govern the specifics of technology that are failing to agree on a unified definition of what software engineering is or should be. Perhaps it should be viewed as a field in its own right, which is the view that will persist for the remainder of this dissertation.

Since its inception not five decades ago, the field of software engineering has grown to become one of the pioneering disciplines at the forefront of this revolutionary ICT era. Software engineering is concerned with manufacturing or developing software. Undeniably, software plays a pivotal role in everyday life and as emphasised by the Software Engineering 2004 Volume, government, recreational and financial activities amongst other things will be hindered by the absence of software [41].

Software engineering is a relatively new field and as such: it is vastly dynamic; other advantageous practices that are evident in other fields, such as componentisation[1] and automation, are still being experimented upon; and the credibility of design methodologies, tools and entities is still questionable. This can arguably explain why software development projects often run behind schedule and exhaust their allocated budgets. Wang believes that software engineering still has a long way to go and further postulates that software engineering has not benefitted from automation to the same extent as other disciplines [52].

[1]Henceforth, componentisation should be understood as *i)* an approach of utilising parts (or components) to form, build or compose a whole entity; or *ii)* a decomposition of traditionally monolithic entities into parts (or components).

Software is developed with an eventual intent that it will be utilised by users, hence the many user-aware software development models and methods where users give constant input, especially feature-based input, during software development. Unfortunately, users (and consequently, software developers) often tend to concentrate more on software features and in the process, overlook equally important areas such as the ease of installing the software on target systems as well as dealing with post-installation (or maintenance) issues that may arise and complicate their lives. Figure 2.1 shows a software publishing process, roughly illustrating where users and developers fit in this process.

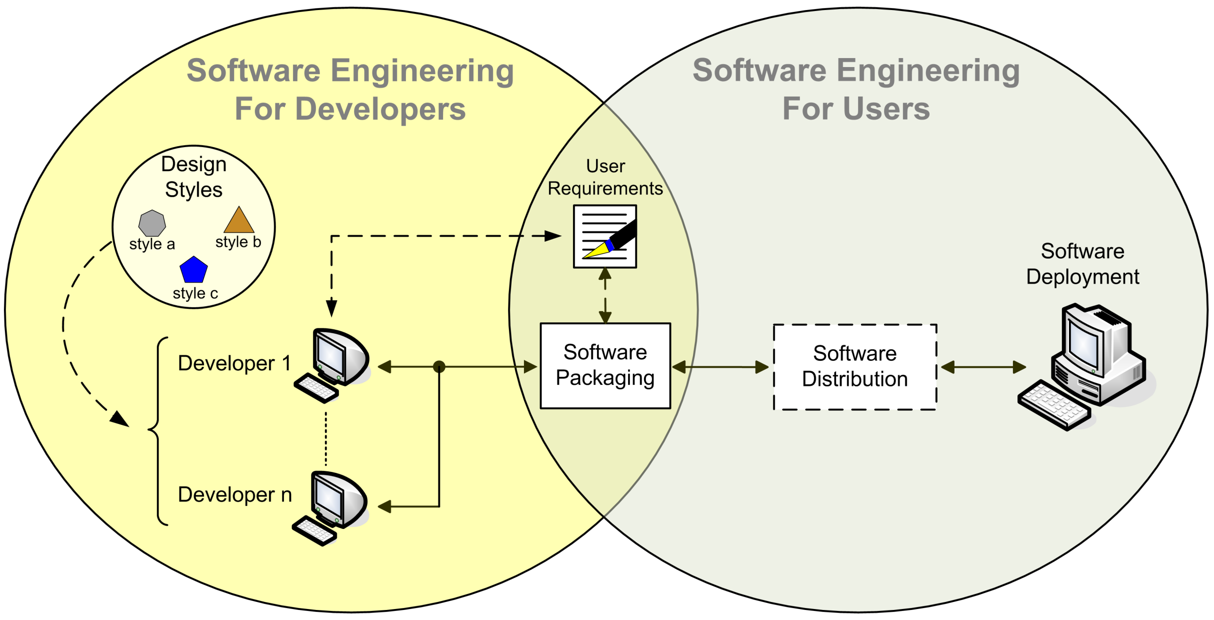


Figure 2.1: A software engineering aware process of publishing software

Figure 2.1 emphasises that software engineering should devote as much attention to users as it does to developers. The remainder of this chapter is centered around this figure. Sections 2.1 and 2.2 respectively give an analysis of current provisions of software engineering for developers and users as well as an account of the direction that this field might be heading towards in the near future. Section 2.3 looks into the field of digital libraries relative to the current progress in the field of software engineering and reflects on how successful software engineering has been with respect to Web-based systems.

2.1 Software Engineering for Developers

Software developers are now able to draw from a variety of available resources when developing software systems. There are hundreds of programming languages and development methods that are at the disposal of developers, each of which is favourable more than others with respect to the tasks at hand and of course the preferences of the developers. Since the days of the Fortran and Cobol programming languages, there has been much progress in the types and paradigms of programming languages and in ways that software systems are developed in general. The following section looks at how programming languages have evolved relative to periodic improvements in computer hardware as well as resultant changing needs in various societies. Section 2.1.2 then discusses one of the latest branches of software engineering — component based software engineering (CBSE).

2.1.1 **Why did Programming Languages Evolve?**

There are mainly four programming paradigms into which programming languages can be classified: procedural, logical, functional and object-oriented. Object-oriented programming, the examples of which include C++ and Java^{TM}, is regarded as the latest of these paradigms and has been appraised for attending to areas of development where other paradigms were still struggling. Prior to object-orientation, it was considered normal to write a computer program line-by-line, which consequently led to delayed project deadlines and other significant disadvantages. With the advent of the object-oriented paradigm, software development became more bearable and more practical as this paradigm gained support from diverse communities. It seemed probable that object-orientation was the solution to reusability of software [18].

Object-oriented programming is structured around objects. Objects are better understood as distinct individual elements (blocks of code) that are sometimes able to receive messages, process these messages and produce some outcome. Furthermore, object-oriented programming is founded upon three concepts that govern how these objects are manipulated: encapsulation, polymorphism and inheritance. Encapsulation ensures that objects are prohibited from altering, in any way, the internal structure of other objects. This ultimately implies that an object can be altered only by itself. Polymorphism allows for different objects to respond to a single method call. The concepts of encapsulation and polymorphism can then be enhanced by inheritance. Inheritance allows for objects to be defined as specialised types of other objects that already exist. These three concepts ultimately reduce the number of lines of code in a program, a result that would otherwise not be possible with earlier programming paradigms.

Wang in [52] argues that although object-oriented programming results in shortened solutions to many programming problems, it still is just programming but based on a totally different way of thinking. This way of thinking is mostly unique to individual developers and can be difficult to learn and apply in practice. Emmerich takes the argument further and discredits object-orientation saying that software reuse in a broader view has never been achieved by object-oriented development [18]. Emmerich further emphasises that the reuse of objects in object-oriented development is hampered due to a large number of fine-grained objects that are developed in accordance with some design but are then buried deep within other aggregations and generalisations, which consequently makes it hard for them to be extracted and used in other contexts.

From what has been stated above, it can be affirmed that one of the reasons why programming languages evolved (but more importantly, why the object-oriented paradigm of programming emerged) was the need to urgently meet the changing needs of society by offering better solutions whereby software systems can be developed at much faster rates hence lowering costs. This of course has been facilitated by accommodating improvements in hardware systems. What has been changing over the years is the manner with which to reason about, and ultimately formulate an informed solution to, particular programming problems. Writing code line-by-line has persisted. Wang insists that line-by-line coding should be replaced by a more coarse-grained process of developing systems, a process that will require minimal programming knowledge from a developer in order to develop some software system. This process can theoretically continue from where the efforts of the object-oriented paradigm ended [52]. Inevitably, code must be written, but there can be solutions in place that will shift the focus of producing software systems from writing code to other higher levels of abstraction.

The Software Engineering Institute (SEI) at Carnegie Mellon University has speculated that one of the latest branches of software engineering, CBSE, promises to deliver exactly what has been missing in the field of software engineering to date [2]. The following section introduces CBSE and discusses its advantages and disadvantages as an approach to composing software systems.

2.1.2 **Component-based Software Engineering**

CBSE is concerned with the development of software systems using already existing components. According to Crnkovic, CBSE has been put in place to attend to the development of systems as an assembly of parts (components) [11]. Crnkovic further elaborates saying that CBSE also encompasses the development of parts as reusable entities and qualifies the maintenance and upgrading of systems by customising and replacing such parts as an elementary part of CBSE.

There are component-oriented models and technologies that directly or indirectly form a subdomain of CBSE. These include Sun Microsystems' JavaBeans^{TM} and Microsoft's Component Object Model (COM+). Commercial off-the-shelf (COTS) systems are a variation of componentised approaches. These systems are commercially focused and in most cases, are more coarse-grained than other component-oriented models. Developing software involves a number of necessary steps, which can be a variation of requirements analysis, specification, design and architecture, coding, testing, documentation and maintenance. This process will soon be altered by the prevalence of these component-oriented approaches and these steps will become more applicable to developing components rather than to developing complete software systems. Figure 2.1 has illustrated that developers are able to choose from various development styles when developing software and componentisation can/should/will now be one of those styles to choose from. Putting this into perspective, from user requirements, the developer can infer whether to develop a system from its roots or gather together some components and produce the required system or an amalgam of both. In whichever case, the requirements will inform the developer of an approach to adopt and determine what is ultimately packaged.

CBSE promises to offer a variety of advantages such as a decrease in development cycles of projects, an increase in the usability and reusability of developed software entities as well as a decrease in production costs. It is undeniable however that as CBSE is a recent sub-discipline of software engineering, it is subject to an array of disadvantages which the proponents of this field still need to attend to. Crnkovic *et al.* believe that there soon will be what they term ”automated component-based software engineering” which will primarily deal with the intricacies of software deployment in an automated manner [12]. Other areas that are still lacking attention include the availability, testing, adaptability, security, performance, versioning, maintenance and credibility of these components [5, 9, 11, 12, 38]. On the area of testing for instance, once these components have been made available, whose responsibility is it to test them and against whose specifications should these tests be conducted? How trustworthy are these components and what should the key distinguishing factors be between components aligned with carrying out identical tasks?

In general, CBSE has many obvious benefits but other areas in this sub-discipline still need thorough attention in order to not jeopardise the future success of CBSE. Software packaging (see Figure 2.1) is a gateway to delivering software systems to users and it is arguably through this packaging process that most or all of these problematic areas of CBSE can be addressed. The following section discusses software engineering for users with a specific interest in users' involvement in the the processes of software packaging and deployment.

2.2 SOFTWARE ENGINEERING FOR USERS

User requirements and software packaging are an intersection of developers' and users' participation in a software engineering aware process of publishing software portrayed in Figure 2.1. Ideally, users should be able to create their own software systems — this, with the proliferation of CBSE and similar component-oriented approaches, without relaying their requirements to developers. However, this is still work in progress. The process of packaging software strongly relies on the type of software that must be packaged, the operational platform that this package must be created for and the packaging tool that is used in creating such a software package. It is disappointing to reveal that packaging tools are not intuitive enough to be utilised by typical users and furthermore, they are still playing catchup to recent software development solutions such as CBSE. In the following section, the design and functionality of various prominent packaging tools is looked at with an aim of highlighting most of the prohibiting factors that make these tools difficult to handle. The final section looks at how these tools facilitate the processes of installing software as well as managing installed software.

2.2.1 Software Packaging

Software packaging is a process concerned with creating a single bundle of software, in most cases executable, for distribution and installation on a target computer. Installing software from source is regarded as a painstaking and error-prone practice that is seldom admired by the majority of software users [40]. It is therefore imperative that solutions which attempt to simplify the process of installing software be effective enough to make the software installation process minimally frustrating to its intended audience. More often than not, software developers derive brilliant solutions to various problems but the brilliancy of those solutions never quite makes it through to consumers, thanks to unnecessarily complicated installation instructions and procedures.

Before elaborating further on the process of and tools for software packaging, it is best to first define formally what a software package is and what constitutes the process of building a software package.

Building a Software Package

A software package is a bundle of software that projects itself as a coherent software system once installed on a target system. Just like a physical package, it has a descriptor that describes the contents, contents that make up a specific object as well as instructions on how to assemble and use the contents.

When building a software package, the type and size of the anticipated package informs the building steps that need to be followed. Generally, the package building process encompasses the preparation of all the elements needed for successfully creating an installation package. Staelin breaks this process down into five steps explained below [46]:

Create Package Manifest — A manifest is a list of files that the package will install on the target machine when the package is deployed. It is critical that all files be included in the package, otherwise the package might not behave as it is expected to behave. For some packages, creating a manifest is a relatively straight-forward task that can be accomplished easily by visually inspecting the software at hand. For other packages however, this process is inundated by hundreds or even thousands of files in which case it becomes a bit more difficult to produce an accurate and complete package manifest in an ad hoc manner. The two most common errors that arise when creating a package manifest are including unrelated files and excluding necessary files.

Determine Package Dependencies — It is often the case that a package depends on other software packages in order for it to function as desired. It is therefore important that when building a package the package composer is presented with the ability of specifying dependency software for the package. Some dependencies are able to be bundled with other package files and be distributed while others have intellectual property rights (IPRs) restrictions imposed on them. Depending on the dependency type, the onus is on either the package composer or the package user to ensure the availability of dependency software when it is needed.

Develop Scripts — The specifications of a candidate system that is likely to have the package being constructed installed on are usually unknown when the package is being built and as such, it is desired that the package being built be 'smart' enough to be able to determine most, if not all, the characteristics of a target system that it will need in order for it to be installed successfully. A multitude of tasks such as determining the operating system that a target system runs, the version thereof and other system dependent variables amongst other things may need to be known to the package before, during or even after package deployment. This requires that there are scripts, whether pre-built or built on-the-fly, which are able to attend to the above mentioned tasks and which also control the installation process as a whole.

Gather Package Contents — The gathering of package contents involves collecting all the files that have been identified as belonging to the package (i.e., all the files that are included in the package manifest), all the packagable dependencies as well as all the control scripts that have been developed. All these entities are then usually placed in some provisional location in preparation for the remaining step of assembling the package.

Assemble Package — At this stage of the package building process, most of the work is already done. Assembling the package simply means bundling everything found in the provisional location mentioned in the previous step into a single package file. This package file can be as simple as a `ZIP` file or a `tarball`, or as complex and custom as the packager may wish it to be.

Most tools that are (or can be) used to create software packages adhere to the steps that are listed above. The following section looks at some of the prominent package managers and analyses the stages and procedures that these tools follow when creating software packages.

Software Packagers

Software packagers, which are alternatively known as package builders or package managers, are tools which are in place to ease the process of building (and sometimes installing) a software package by introducing simpler and sometimes automated solutions to achieving the five steps that have been mentioned in the previous section. The work of most package managers is twofold: building a software package; and managing the package throughout its lifetime once deployed on a particular system. Figure 2.2 show the structure of a typical package manager as well as other entities that it interacts with.

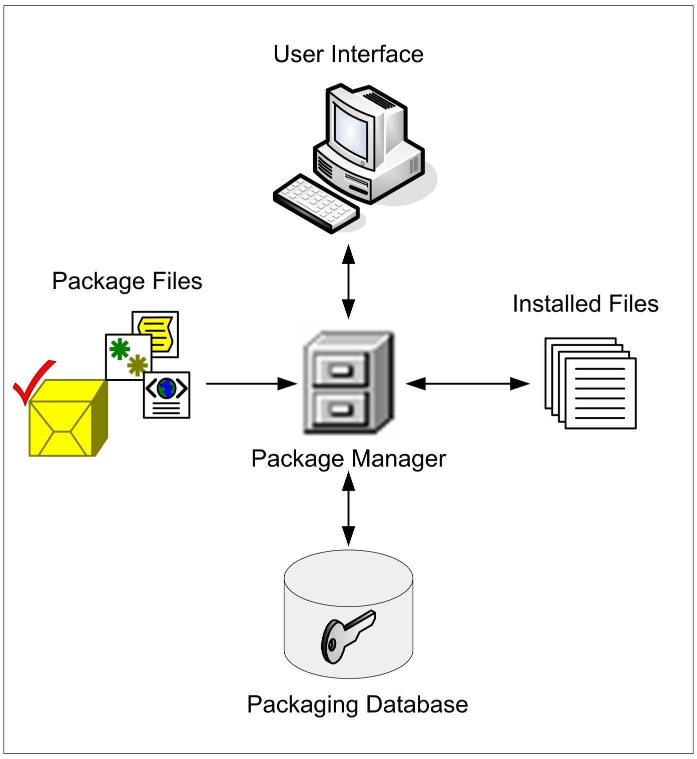


Figure 2.2: Interactions of a typical package manager

Each of these entities play a special role in the way the package manager functions. These roles are described briefly below:

User Interface — The user interface provides a platform through which a user can communicate and interact with the package manager. Most package managers that are available for open source operating systems employ a command-line-based user interface with a select few providing a graphical user interface. Where graphical user interfaces are provided, they often operate as a layer above the text interface (e.g., rpmdrake, a graphical-based tool for Red Hat Package Manager (RPM) on Mandriva GNU/Linux distributions).

Package Files — A package file, as earlier defined, is that which includes all the parts (program, data, documentation and configuration files) necessary to carry out an installation.

Installed Files — These are files that have been installed on a particular system as a result of a package installation.

Package Database — This database describes every installed package. For each installed package, it stores its descriptor as well as information pertaining to files that the package owns. For instance, given an arbitrary file, it should be possible to associate that file with a package to which it belongs, provided of course that the files belongs to a package, and not the base operating system.

The balance of this section looks at a few package building tools with respect to the manner in which they handle the package building process. It is worth noting that in order to build a software package in general, one needs a thorough understanding of how the software works and how it is installed manually.

Red Hat Package Manager

RPM is the most widely used package manager that most commercial GNU/Linux distributions are based on [3, 23]. An entry point into building an RPM package is creating a formal recipe that RPM will understand and convert into a package according to the recipe's specifications. This recipe, or better known as the RPM specification file, has sections that address specific aspects of the package creation process as well as characteristics of the distribution package to be built. Figure 2.3 shows an example of an RPM specification file showing some of the sections that are important when creating an RPM package. These sections are analysed in greater detail next.

`preamble` — The preamble is an unlabeled section that constitutes everything appearing before the `%prep` section in an RPM specification file. The preamble presents a human readable descriptor of the package to be built by RPM, which is returned when a user requests information about the package. When building a package, details such as `Name`, `Version` and `Release` are used to ensure uniqueness of the package amongst other similar ones. Although some of the fields are not mandatory in the preamble section of an RPM specification, it is usually best practice to include as much information as possible. The field names in an RPM specification are relatively straight forward save for the `Source` and `Requires` fields, which are explained further below:

`Source` — This field tells RPM where to find the source files of the package being described by the specification. If this field contains a Uniform Resource Locator (URL), RPM will attain these files automatically from the enlisted URL, otherwise RPM will look for the source file locally in the RPM SOURCES directory (see Figure 2.4). In the case where there are more than one source files, the listing convention becomes `Source0, Source1, Source2` and so on until all the source files can be listed.

`Requires` — To specify dependencies manually, a package composer can list such dependencies under a `Requires` field. Dependency specification can be as simple as just stating that the package requires a particular dependency package — as is the case with

```
Summary:       Version 2 of the spreadsheet formulae package
Name:          spreadsheet-formulae
Version:       2
Release:       11
Copyright:     GPL
Group:         Applications/Office
Source:        http://www.spreadsheet.org.uk/downloads/spreadsheet-formulae-2.11.tar.gz
URL:           http://www.spreadsheet.org.uk/documents/
Packager:      Matthew Stone
ExclusiveArch: i386
Requires:      spreadsheet >= 2.1
Requires:      perl
Requires:      postgresql

%description
This is Version 2 of the spreadsheet formulae package distribution. This version includes
support for most Fourier Analysis and Mechanical Engineering formulae as well as a rich
database of common equations and examples.

%prep
rm -rf $RPM_BUILD_DIR/spreadsheet-formulae-2.11
zcat $RPM_SOURCE_DIR/spreadsheet-formulae-2.11.tar.gz | tar -zvf -

%build
./getsystemdeps.sh
./createconfig.sh
./configure.sh
make

%install
make install

%files
%doc README
```

Figure 2.3: Example of an RPM specification file

`perl` and `postgresql` in the illustration. It is possible also to specify dependencies in a more thorough fashion. This is seen with the `spreadsheet` dependency in the illustration where the least version of the dependency is specified (at least version 2.1). Other version range combinations can be stated in this manner. RPM also claims that shared library dependencies are automatically determined and satisfied by RPM [3].

`%prep` — The `%prep` section is where preparation for installing the package is done. Like any of the remaining sections, any shell constructs can be included in this section. As illustrated in Figure 2.3, RPM is told to remove any old builds of the same package (for precautionary measures) and extract the contents of the source tarball into the build directory. For packages that require patching and other similar tasks, such tasks can be done or prepared for in this section.

`%build` — This section contains the exact instructions that would need to be executed in the case of a manual build process. In the case of the example in Figure 2.3, to build the package being described, a series of shell scripts will need to be run before running the `make` utility command-line tool.

`%install` — The `%install` section, just like the `%build` section, contains the exact instructions that would be executed for a manual installation process. The illustration shows that installation instructions are contained in a makefile, however, other custom scripts can be included in this section.

`%files` — One of the five steps for creating a package mentioned earlier was concerned with creating a package manifest. Together with the preamble section of an RPM specification file, this section completes the task of creating a manifest for the package being described. In this section, all files that cannot be specified anywhere else in a specification file are specified. These are usually documentation files, such as in the example, which RPM will consequently place in a documentation directory on a target system once the package is deployed. A typical directory for this example would be `/usr/doc/spreadsheet-formulae-2.11/`.

Once an RPM specification file has been constructed, it can then be placed in the SPECS directory. Figure 2.4 shows how all the RPM build directories are structured.

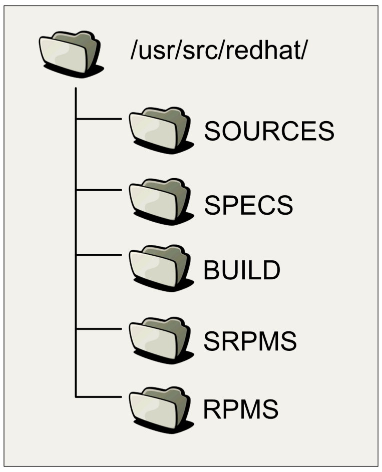


Figure 2.4: Layout of RPM build directories

RPM can then be told to create the package using the `rpm` command. The command `rpm -ba spreadsheet-formulae-2.11.spec` executed from within the SPECS directory will tell RPM to build all (`-ba`) packages (i.e., the binary and source packages) by iterating through `%prep`, `%build` and `%install` sections of the RPM specification file shown above in the above example. The output packages are placed in their respective locations (SRPMS for source packages and RPMS for binary packages). When the `-b` flag is used alone, only the binary package will be built. However, there are no specific entries in the specification file to differentiate between source and binary packages. If there are any errors with the specification file, RPM will report these to the package composer — otherwise the package(s) will be available and will be ready to be tested and distributed.

RPM is known to have inconsistencies when it comes to package names, package contents and dependency handling, but perhaps the most noticeable disadvantage is that only privileged users are able to build and install RPM based packages, since the RPM database (as shown in Figure 2.2) can be accessed only by privileged users. Another negative with RPM is that it is command-line-based and as such, offers a steep learning curve for those interested in learning how it works. There are other systems that are used in open source operating systems, which are somewhat similar to RPM. These are briefly looked at in the next section.

Other Software Packagers for GNU/Linux-based Operating Systems

Debian and Gentoo are two types of GNU/Linux distributions that each employ a different package manager tool, just as RPM is to Red Hat and its derivative distributions. This section briefly looks at two tools that these systems as well as their derivatives employ for their package management needs.

Debian GNU/Linux Package Manager — This is a suite of programs for creating, installing and removing package files, which was initially targeted at Debian and Debian-based GNU/Linux distributions but may now work on or be ported to other GNU/Linux systems [28]. More commonly known as 'dpkg', it is functionally similar to RPM, but differs in its architecture in that it employs other tools such as the Advanced Packaging Tool (APT) to sit at a higher level of abstraction and provide a simplified interface for accessing the functionalities it provides.

The process of building a Debian package, commonly known as a 'deb', begins with creating a specification file, called a control file, which holds all the information about the deb to be built. This file is strikingly similar to that employed by RPM. Once this control file has been constructed, its corresponding package can then be built using a combination of the `dpkg-deb`, `dpkg-source` and `dpkg-buildpackage` tools. A big advantage in using Debian-based GNU/Linux distributions is that there are thousands of software packages which have been created and are ready for download (whether as stand-alone applications or as specified dependencies of other packages) using the APT `apt-get` command. This provides a smooth procedure when creating and deploying debs. For RPM, resources such as *http://rpmseek.com/* and *http://rpmfind.net/* allow users to search for and download any packages that they may be requiring.

Notable disadvantages with dpkg are that, just like RPM, it maintains a database for accounting for all the packages that it installs on a target system. This of course means that only privileged users are able to find complete satisfaction when working with debs. Another point worth noting is that the modularised structure of the dpkg tool may intimidate novice users since there are many tools to get around.

Portage — The Gentoo GNU/Linux distribution employs Portage as its package management system. Unlike RPM and dpkg, Portage does not support the notion of packages in a traditional sense, but prefers that software be acquired as source code, and be compiled and installed on demand [10].

The common element that Portage has with RPM and dpkg is that it also has build recipes, called ebuilds, which describe candidate packages. Essentially, each ebuild describes the package's metadata (which includes amongst other things, source and dependency information) and contains instructions on how the package can be compiled, installed and configured on a target system. In general, creating ebuilds is a lot more complicated than creating RPM's specification or dpkg's control files. This can be attributed to the fact that ebuilds are shell-based scripts with various subroutines for controlling each step of the deployment process and as such, implementing these subroutines can quickly get confusing and complicated.

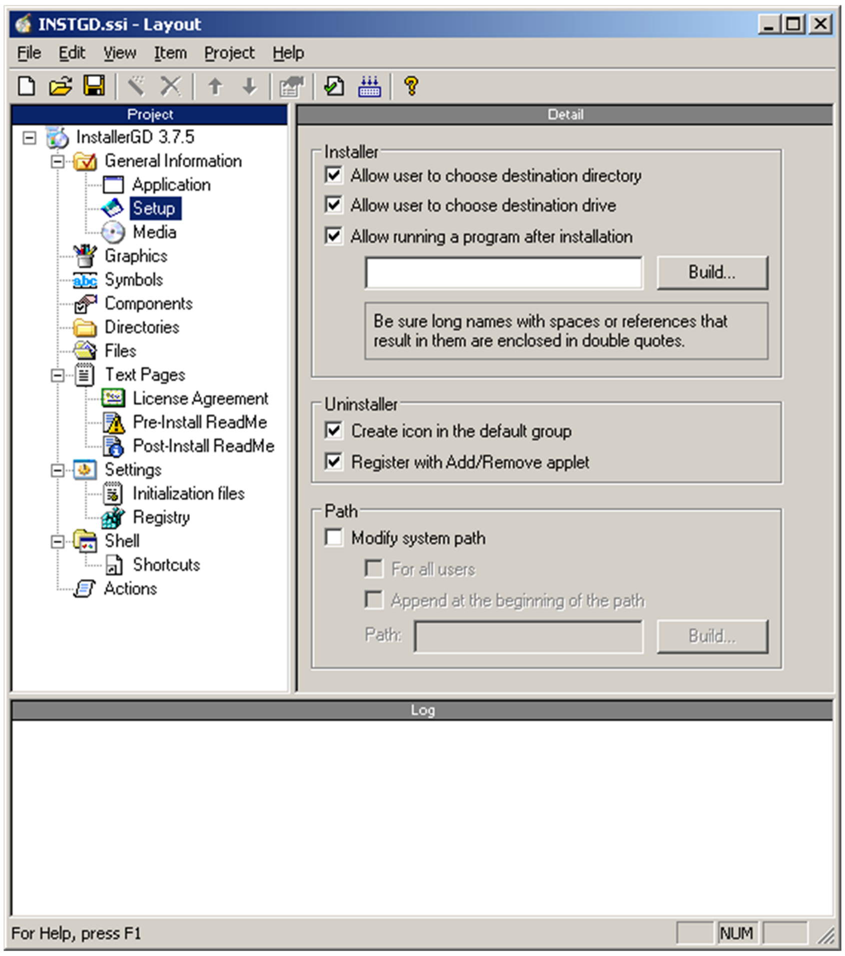
On the front-end of the Portage package management system is the `emerge` utility command (or tool) that users use to access the features of Portage. The `emerge` utility tool has many useful (and sometimes elaborate) commands but a simple command such as `emerge mozilla-firefox` will tell the `emerge` tool to download the Mozilla Firefox Web browser as source, compile all the downloaded source files and install the Web browser in a sandbox environment. This sandbox environment is one of the prime attractions of how Portage manages packages. It ensures that the rest of the system is relatively safe from whatever harm may be inflicted by an installation. Portage is also a global package management system and as such, the `emerge` utility can be used only by users with specific privileges.

Software Packaging for the Microsoft Windows Operating System

The Microsoft Windows operating system is different from any GNU/Linux-based operating system and as such, package management follows a slightly different approach to those discussed in earlier sections. At the heart of the Microsoft Windows operating system is the Windows Installer tool. Rather than an installation program or package manager, Windows Installer is the base installation tool with which applications are installed on Microsoft Windows systems. Many package management applications such as Altiris's Wise [1], Macrovision's InstallShield [32], Proggle's Installer GD [42] and Caphyon's Advanced Installer [4], to name but a few applications, use the Windows Installer engine when installing software in order to maintain consistency in the internal database that Windows Installer maintains. This ensures reliable operation of important installation features such as rollback and software versioning. Another powerful feature inherent in using Windows Installer is the automatic generation of the uninstallation sequence for a particular application.

Characteristic of the Microsoft Windows operating system, all of the package management tools mentioned above employ a graphical user interface, which makes for a much favourable experience if compared with package managers of GNU/Linux-based operating systems. Figure 2.5 shows a screen-shot of a general setup interface of the Installer GD software packager tool. The interface in other tools is typically similar to the one shown in this figure.

The Windows Installer tool does not provide for any dependency handling methods. It is therefore up to package managers to deal with dependency issues. Some of these tools, such as the Advanced Installer tool, go a certain distance towards entertaining dependency specific issues while other tools do not even look into such matters. Ideally, all tools should attend to dependency issues. It is interesting to note that the latest version of InstallShield (version 11) now has support for RPM and a few other packaging platforms as well as support for various other operating systems beyond Microsoft Windows. Perhaps in the not so distant future, such practices will become a norm in package management software, which will be a welcomed improvement in current software publishing processes.



Installer
- [✓] Allow user to choose destination directory
- [✓] Allow user to choose destination drive
- [✓] Allow running a program after installation

Be sure long names with spaces or references that result in them are enclosed in double quotes.

Uninstaller
- [✓] Create icon in the default group
- [✓] Register with Add/Remove applet

Path
- [ ] Modify system path
- [ ] For all users
- [ ] Append at the beginning of the path

Path:

For Help, press F1

Figure 2.5: Screen-shot of Installer GD general setup interface

2.2.2 **Software Deployment**

Software deployment is a process whereby a software system is introduced and monitored throughout its lifetime on a target system. Carzaniga *et al.* define software deployment as an array of activities, including but not limited to the release, activation, deactivation, update and removal, applicable to software systems on consumer computers [5]. Arguably, software packaging and software distribution (as depicted in Figure 2.1) form part of software deployment [9].

With the recent advances in software design such as CBSE, it is beyond the duties of software producers to distribute complete software systems and moreover, the growing complexity of software systems prompts that software deployment activities be given special attention. Coupaye *et al.* advocate that the process of application deployment persists in an ad hoc and very poorly automated manner, possibly because it has previously been considered as unimportant or too complex or hardly feasible with respect to available technologies [9]. Recently though, a fair number of technologies have begun to emerge (some of which are mentioned in Section 2.2.1) to attend to the software deployment problem. However, these technologies still present a staggered view of how the issue of software deployment should be tackled.

The remainder of this section explains some of the activities that constitute the process of software deployment before looking at a number of issues that complicate the process of deploying software.

Software Deployment Activities

For the purposes of this research, the process of software deployment was regarded as that which could be arranged into four distinct groups as described below. These groups of activities differ slightly to those mentioned in the literature but in general, present similar ideas. The main cause of this difference is that other software deployment activities are not applicable to certain types of software systems.

Packaging and Distribution — The process of packaging software has already been introduced in the previous section. This process is undergoing a transformation due to various recent models for developing and delivering software to consumers. User requirements now play an increasingly important role in informing software vendors and/or developers of what to package and distribute. Ideally (presumably in the near future should the promises of CBSE materialise), users should be able to craft and package their own software systems with minimal effort. Software packaging can be viewed as an entry point to the process of software deployment, following which is the distribution of a packaged software solution.

With the advent of the Web, it has become intuitive for software vendors and/or developers to make software available over the Internet. This allows users to easily search and download software at their will, provided of course that the luxury that is the Internet is at their disposal. Other methods of distributing software include DVD-ROMs, CD-ROMs and floppy disks and these methods are preferable for distributing either relatively large software systems (such as comprehensive word processors, accounting packages and operating systems) or software (such as software drivers) that accompanies new hardware. It is equally important, especially if the software is not tailored according to a particular consumer, that software be advertised appropriately to ensure that interested parties are made aware of the characteristics of the system [5].

Installation — Software installation is introducing a software system (software package) to a consumer's machine. From this process on, the user interacts directly with the software system as it was developed and packaged. At this stage of the deployment process, the software system is configured and prepared to interact with the user's system. This usually includes evaluating the package's software and possibly hardware prerequisites as well as gathering other required input from the user. The installation process culminates in a coherent software system at a consumer's site, which can then be launched and utilised on demand. Using the installed software differs with respect to the type of software in question. For instance, some software systems are launched through some sort of a clickable icon, others are Web-based and require some URL to be accessed while others are command-line-based daemons.

Updating and Upgrading — Updating and upgrading a software system are two amongst many post-installation activities. Updating encompasses modifying the current system to a logically next version, usually referred to as a minor version. This can be achieved through applying patches, which themselves can be perceived as stand-alone, deployable packages [13]. Other means of updating software include automated procedures such as those employed by Microsoft Windows XP and Mozilla Firefox but these rely entirely on the availability of the Internet. The frequency of updates vary from weeks to months according to a number of factors such as the system's stability in the conditions it is exposed to and bugs that have been identified since its distribution.

Upgrading a software system is slightly more complex than updating it. This usually requires that the current version of software be completely removed from the consumer's system so that the logically next version (usually referred to as a major version) can be installed. This is evident mainly in operating system software where is it common to upgrade, say, from Microsoft Windows 98 SE to Microsoft Windows XP or from Mandrake 9.1 to Mandriva 11.0. The granularity of upgrading software also varies according to certain conditions but it is commonly in a range of months to years.

Uninstallation — When a software system is no longer needed by a consumer it, in most cases with its prerequisite software packages, can be totally removed from the consumer's system. For some software systems, this process can be devastating. Consider removing a software system that shares dependency software with one or more other systems. If removing a software system implies removing its dependency software, other software systems that rely on the dependency may not function appropriately at its removal should the software being removed not be aware of these other software systems. This has been the motivation behind keeping a database of all installed software and its metadata at the disposal of package management tools as discussed in Section 2.2.1. In ideal situations, the stability of a consumer's system should not be compromised by the removal of some or other software system.

Software Deployment Issues

The process of deploying software systems, whether it is identical software packages or not, is almost always unique to the characteristics of the consumer's site. Nevertheless, there are some aspects in which all the instances of deploying software are comparable. This section lists some of the common complications that surface at software deployment.

Heterogeneous Operational Platforms — Developing software for use in more than one operating system has always been a challenge. Different consumers are comfortable with different platforms and as such, it is required of the software developers that the software they develop be able to work on various operational platforms with little or no extra effort required from consumers. Carzaniga *et al.* qualify that the coexistence and the interoperability of heterogeneous platforms pose new challenges for software deployment and furthermore, the platform type becomes a new variable that has to be taken into account when dealing with configuration and dependencies [5]. Coupaye *et al.* emphasise the importance of ensuring uniformity across all computers at an organisation, even if the computers are running different platforms [9].

If developers are using cross-platform tools such as Java^{TM} for development, then the issue of developing for different operational platforms becomes a step closer to being solved, but even with such practices, software may still be interacting with platform-specific libraries, which is still undesirable. Other means of dealing with this issue is to have different ports[2] of the software for as many operating systems as is possibly required.

Resolving Dependencies — There are various types of dependencies that software packages have to resolve during the deployment process. Some of these are hardware types, for instance, a specific type of a graphics card may be required for an image editing/processing software package. The majority of these dependencies however, are software-based. Section 2.2.1 has introduced how some package managers handle dependency specification and satisfaction. During the process of specifying dependencies, it sometimes may become tricky to specify those dependencies that are known by more than one name such as the Apache Web server, which can either be *apache* or *httpd* depending on the server's configuration and platform it is configured on. For these types of dependency software packages, some packagers do make provision for an *or* (as in *apache or httpd*) during specification, but for RPM, *or* is not defined. Another overhead at dependency specification is that package composers need to know the exact versions of dependency software packages that the package needs, which may take some effort to discover.

When a software package has been distributed to a consumer's site, it needs to resolve (if possible) dependencies upon installation. Some of these dependencies are required to drive the installation process. For example, the software package require the Perl interpreter to execute configuration scripts or an archiving tool for those packages that are distributed as compressed archives. Other types of software dependencies that are required for the functioning of the software can be checked for on a user's system — otherwise some software packagers allow for these dependencies to be automatically downloaded and then installed. The download process alone can prove to be a complication, but another is when installing the downloaded dependency at a consumer's site. Some dependency packages are accompanied by licence enforcements where users are usually required to accept or agree to the terms stated in the licences before the installation can continue. Licence restrictions may also prohibit dependency software from being bundled and distributed with software packages or even from being automatically downloaded, in which case it is up to the consumer to resolve any dependencies manually.

Interoperability — Software systems usually have features that are absent in other similar software systems. In this case, consumers may want to access the same piece of work using different software systems or access data belonging to other applications from within another application without any complications. An example of this is using Microsoft Paint for basic image editing and Adobe Photoshop for advanced image editing.

Data Migration — It is sometimes desirable that data be migrated from system to system. For example, if upgrading a database management system (DBMS), it is often the case that the user might still need the data (both content- and configuration-type data) from the old DMBS to be available for the latest system. Many software systems provide for such deployment issues, but in most of these systems, this is poorly done and not as effective as it ought to be.

[2]Modified versions of software for use on different machines or platforms.

An alternative view to data migration is seen in content management systems, where it makes sense that content management software systems be distributed with some data, sufficient to demonstrate the functionality of the system and also be readily usable. This however depends on users' preferences since different users may require different content distributed with their packages, which may impose a lot more work on current packaging processes.

Updates and Upgrades — These two processes are multifaceted. Generally, they refer to any change in the environment in which a software system is deployed, which can consequently affect the functioning of the software system. On the hardware side, computer systems can be upgraded by adding better hardware components such as network adapters and other peripherals, which in turn will require respective changes in software (e.g., newer software drivers) that could then affect the functioning of the installed software.

Another instance of this problem is when dependency software is updated either by other systems or voluntarily, which can cause the software system to not function with the new update of the dependency software.

A less common problem, although equally problematic, can occur when a software system composed of different components is updated/upgraded componentwise, that is, some components get updated and/or upgraded while others remain unaltered. This may result in a mis-communication between system components where components are failing to communicate or are communicating incorrectly with others.

Internet Solutions — The Internet has significantly improved the software deployment process by providing a common interface through which software developers communicate with consumers. However, the Internet has also fallen victim to criticism when considering certain aspects of security relating to the deployment process. Consumers need procedures to ensure the integrity of the software (and software-related information) that is distributed over the Internet. It can be devastating for a consumer to install an application acquired over the Internet only to find that it violates the coherency of that consumer's system. For those types of applications requiring constant contact with online servers for updates or whatever other reasons, authentication and privacy should be guaranteed for the consumer. No unauthorised party should have access to the data being transmitted by an application at a consumer's site.

Besides security aspects, not having access to the Internet can prove to be unsettling, especially when looking at applications that have some interaction with it, since the consumer should probably now have to derive other means of mimicking what the Internet would otherwise have been responsible for.

So what this means is that developers should be extra cautious when it comes to developing applications that might utilise the Internet in some way. They must address both security aspects in the case where the Internet is available to consumers and the possibility that the Internet might not be present at a consumer's site.

The issues that have been presented above are some of many that cause complexities during the software deployment process. The stem of many of these issues is at the development phase and it is at this stage that counter-measures can be introduced. In most instances, it is difficult for system developers to infer counter-characteristics to the above issues from user requirements. It is therefore at their discretion that these issues can be implemented for. In an ideal scenario, there should be regulations or policies in place that govern the process of developing software so to ensure that certain software deployment requirements are met before advertising and distributing software packages to consumers.

2.3 DIGITAL LIBRARY SOFTWARE SYSTEMS

A digital library, as initially defined in Section 1.3.3, is an electronic platform that provides an organised integrated set of services for capturing, cataloguing, storing, searching, protecting and retrieving information, all of which may be accessible over the Internet. Digital library software systems are a specialisation of what is commonly known as information management systems since they possess most, if not all, characteristics of information management systems. This section covers most of the recent developments in the field of digital libraries with an intention of highlighting the effects of software engineering in Web-based software systems.

In the late 1990s, the need to efficiently and effectively disseminate scholarly content without any prohibiting factors arose [26]. The Open Archives Initiative (OAI) stepped into the challenge and provided a Protocol for Metadata Harvesting (OAI-PMH) which has since been a key player in promoting interoperability amongst digital library systems through a simplified approach, addressing the issue of connecting multiple digital library systems in a distributed environment [30]. The OAI-PMH has since been implemented and supported by various projects worldwide, some of which include Kepler [34, 35] and NCSTRL [50]. These digital library software systems were initially aimed at disseminating scholarly content. Other digital library systems such as Greenstone[3] [43], DSpace [31, 45] and EPrints [39] mimic, as closely as possible, traditional libraries in that they were designed to house and manage any type of digital content beyond documents.

Initially, the systems that have been mentioned above were relatively monolithic by the nature of their design. This presented a problem when it came to the portability of as well as user requirements for each system amongst other things. For instance, since these systems were inflexible, users would often be overwhelmed by features that they do not require, which would in turn overshadow the basic features that each of the systems claim to provide. As research progressed, modularisation or componentisation approaches begun to emerge in the field of digital libraries, which ultimately saw prominent digital library systems adopt modular-based designs. This process has already been witnessed in other branches of computing such as in programming languages. Figure 2.6 depicts the structure of a typical component-based digital library composed of two components: the *Search* and *Browse* components. These components work on the digital library data *Collection* and their services are made available to users through the *User Interface*.

According to Witten *et al.*, digital libraries need to be dynamic [17]. They (Witten *et al.*) support this by emphasising the need for administrators to routinely add new collections or new user interfaces or completely new kinds of services to a digital library at runtime, that

[3]Greenstone is not a digital library system per se, but rather, a software suite for building and distributing digital library collections [43].

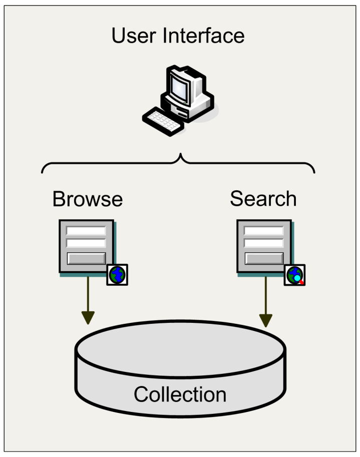


Figure 2.6: A typical component-based digital library system

is, without bringing it to a complete halt. These requirements are also applicable to other software systems that have adopted a component-based approach in their design [13]. This has seen the realisation of the recent Greenstone 3 project, now an agent-based (component-based) digital library tool, something that its predecessors were not. Other digital library systems are following in Greenstone's footsteps.

The transition from the mentioned monolithic systems to more flexible approaches was initiated by research efforts like the OpenDLib [6] and Open Digital Libraries (ODL) [47, 49] projects. The primary aim of the OpenDLib project was to create a system that manages digital library services by providing an infrastructure with which a digital library can be customised on-the-fly, hence making the digital library expandable. The outcome of the ODL project was an array of lightweight components that can be connected and can communicate with one another through a well defined set of protocols influenced by the OAI-PMH where each component corresponds to a typical digital library service such as searching or browsing. These efforts have partly been driven by the growing need for simple digital libraries in varied communities. Another influencing factor was that breaking down a complex Web-based system into smaller manageable pieces can be motivated as being a good strategy for taking maximum advantage of the distributed nature of networks within which the resultant component-based system will be housed. A more classical factor however is that it is regarded as good practice in the field of software engineering to break down a complex system into smaller manageable pieces.

It is characteristic of most digital library systems to utilise some sort of DBMS for data storage or perhaps some Web-server for publishing purposes. In general, prerequisite software for most digital library systems is similar. Rhyno suggests that using open source systems for digital libraries will facilitate the development and functioning of digital libraries in today's communities [44].

From the above discussion, it can be ascertained that the design trends in the field of digital libraries are similar to those in the software engineering discipline — a shift from inflexible designs to much more flexible component-based approaches. This means that the same problems such as component testing, integrity, maintenance and so forth, as well as all the advantages of component-based design that are associated with CBSE, are equally applicable to the digital libraries field. If considering the deployment process of current digital library software systems, including the processes of packaging and distributing a digital library system, there are varying degrees of similarities in the way these processes are handled. With EPrints for instance, although it only functions in Unix-like operating systems, it does not employ any of the package management solutions as it is distributed as a tarball and installed and configured by manually running corresponding scripts from command-line. DSpace operates in a similar fashion. This is possibly the case to avoid any limitations to the operational platforms since there is no generic package management tool to span across most Unix-like systems. Greenstone 3 on the other hand boasts a Java^{TM} design, which means that it is flexible across most platforms. Although there are different distributions for various operating systems (Microsoft Windows, Unix and Macintosh OS X) due to different platform-specific dependencies, the underlying features are the same and each distribution package, when executed, presents a graphical user interface where the installation and configuration processes are carried through.

2.4 Summary of Key Points

Software engineering as a field in its own right has seen much progress in the past decade, with a shift of focus from developing complete systems to developing components that can ultimately be glued together on demand to create flexible systems according to consumer's needs. This approach is also witnessed in other fields concerned with the development of software such as in Web-based service-oriented fields (of which digital libraries is an example) where the design of software systems in such fields is adapting to meet the changing needs in their respective societies. There are still some inconsistencies with component-based approaches despite all the colourful advantages and it will be with a firm addressing of these inconsistencies that the success of modularised approaches will be determined.

CHAPTER 3

RESEARCH OVERVIEW

Previous chapters have been employing a relatively general tone and have not been thoroughly specific with respect to defining the problem statement that this research has been guided by. The main aim of this chapter is to introduce the problem statement to the reader and put into context some of the theories, concepts and practices that were discussed in Chapter 2. Section 3.1 gives a brief discussion on some of the points that motivated this research. The problem statement is explained in detail in Section 3.2.

3.1 MOTIVATION

Any field that falls under the computing industry, or any industry for that matter, is subject to growth as time progresses. Notwithstanding this unavoidable growth, it is equally important that the stakeholders associated with any field that is subject to this growth are well prepared in order to avoid any possible shortcomings. Following are some anchoring points that have motivated the proceedings of this research:

- **Digital libraries are evolving:** This can be attributed to the change in needs in the audiences and communities that these systems are applicable to. Suleman *et al.* in [48] reaffirm that traditional digital library platforms, often portrayed as immutable and monolithic, are now facing extinction and are making way for much tidier, simpler and flexible compositional platforms to challenge the future of the digital libraries field. However, there are still some concerns due to various inadequacies that are inherent in component-based approaches as discussed in detail in Chapter 2.

- **Current packaging practices are delaying to adjust accordingly:** The design of current package management tools is unable to accommodate software systems composed from a pool of components. This is so because, traditionally, these tools were not developed for the possibility that they would one day need to align their functionality and goals with those of technology trends such as component-based solutions.

- **There is a need to complement and support the growth in the field:** This should be in a timely manner so to promote continuity and encourage future growth. Successfully attending to the inadequacies of component-based approaches will demonstrate that these approaches are sound and will shift the focus to possibly exploring other avenues where component-based approaches can prove to be equally rewarding.

3.2 Problem Statement

This research is part of a bigger project whose aim was concerned with investigating techniques, models and tools for constructing flexible digital libraries by addressing the question of how to effectively and efficiently build digital library systems based on simple components arranged into a network of services. Specifically, this research tackled the component management issue as thoroughly manifested in Chapter 2 and in the previous section by formulating the following problem statement and employing it as a starting point.

To investigate techniques and methodologies for packaging component-based digital library systems such that deployment is rapid and flexibility is not compromised.

This problem statement was further broken down into a series of distinct smaller tasks in order to promote manageability. The remainder of this section presents each of these tasks. For each of the identified tasks, a motivational description is given as well as methodologies and strategies that were followed in addressing that particular task.

3.2.1 Packaging of Heterogeneous Components for Heterogeneous Operational Platforms

Description

Recent research has shown the idea to compose a component-based digital library system from components initially developed under different environments and attending to different needs to be possible [19, 20]. This given, the first part of this task was to achieve whether or not the process of moving from a specification of such a heterogeneous composure of a component-based digital library system to a readily installable package is possible, and if it is, within which scope can this process be useful. The second part of this task is dependent on the outcome of the first part. It seeks to determine if this installable package can be built irrespective of operating systems' barriers.

Methodology

Fundamentally, this task was the cornerstone of this research, the outcome of which influenced that of the other three. It was not arguable that some sort of a packaging and management platform was to be employed, whether developed from scratch or adopted from existing solutions, en route to completing this task. The main issue of concern was that of heterogeneous packaging, however attained. The following are steps that were followed in completing this task.

- **Launched a study into existing packaging tools:** This study commenced by looking into popular package management tools (e.g., RPM, InstallShield, etc. as documented in Chapter 2) with a goal of establishing whether any of them are suitable to serve as a basis of this research.

- **Employed the most flexible of these tools:** RPM proved to be the viable choice due to its functionality and overwhelming popularity as a package management tool amongst the general public [3, 23].

- **Evaluated the chosen tool against the intentions of this task:** A component-based digital library was built into an RPM package and these were the main observations:

  - RPM packages are only compatible with Red Hat and Red Hat-based Linux distributions, thus they do not offer a great degree of heterogeneity.
  - To install an RPM package, one requires special administrative rights, otherwise the installation will not succeed. This goes against the *rapid* and *flexible* goals of this research.
  - Finally, RPM does not allow for an interactive installation process, forcing for pre- and post-installation user interaction, which is less than desirable for component-based digital libraries and once again is in conflict with the *rapid* and *flexible* goals of this research.

Taking the observations noted on the final step above into consideration, it was then decided that a new packaging system would be designed specifically to be as portable and minimalistic as possible, with an emphasis on the ability to compose and configure individual components during installation. The design and implementation of this system is presented in Chapter 4.

3.2.2 Comparatively Assessing the Effort in Installing Individual Components Making up a Bigger System and Installing the Same System but as Packaged Components

Description

Background research revealed the advantages and disadvantages associated with package management software. Consolidating all the advantages and putting them into practice minimises the amount of work that users have to go through to successfully install software. However, as involved as the individual component installation might be, many users still prefer it over packaged solutions, claiming that individual component installation offers more freedom. Putting this into perspective, suppose a simple digital library system is available as both a package and as loose components (typically, ODL components). Some users might not be interested in how the components are connected and will settle for the default settings that the package provides while other users may want to manually configure each component according to their desires. This task has been set out to uncover the preferences of the majority of the users.

Methodology

To comparatively assess these two processes, a comparative user experiment was constructed and incorporated into the final evaluation process. In this experiment, users were requested to perform both processes, namely, installing a component-based digital library system by configuring each component separately and through a package, and give feedback on their experiences. Chapter 5 gives a detailed breakdown of the results obtained from this user evaluation process.

3.2.3 Efficiently and Effectively Dealing with Package Dependencies and their Intricacies

Description

Software installations are hampered due to license restrictions which forbid bundling of all necessary dependency packages in a single distribution [14]. More often than not when installing a software package, it is the case that the package may depend on other packages in order for it to function as desired. There are a number of employable solutions, but mostly outdated, that can address this matter. One of these, and a trivial one at that, is to shift the dependency burden to be dealt with by users whereby they have to manually satisfy dependency requirements prior to installing software packages. This is only reliable pending both the availability of these dependency packages from their nominated locations and the clarity in the communication between the package and the user in terms of requirements and so forth. This solution is regarded as unacceptable and painstaking by users. Another of these solutions, which seems to be more profitable, is that of automating this dependency requirements satisfaction process. Chapter 2 has presented a detailed discussion on this matter and from that it is evident that automation will yield desired results but not before founding a normalised, uniform and flexible manner to create packages and specify dependencies.

Methodology

Most package management tools have various means of dealing with dependencies. At first glance, these seem uniform but differ drastically at ground level. A step taken in dealing with this task was to learn how prominent package managers dealt with dependencies, all of which is shown in Chapter 2. A satisfactory level of automation is achieved by these package managers but there are still areas that lack attention. For instance, when specifying dependencies in the `Preamble` section of an RPM specification, one does so under the `Requires` tag where one specifies these dependencies by name. In this case, a problem is encountered when a dependency goes by more than one name such as the Apache Web Server which some systems understand as *apache* and others as *httpd*. It then becomes more complicated to specify these types of dependencies.

A more serious problem, however, is that of attending to dependencies that have strict license restrictions. These are more complicated to provide automation for and it seems that package managers have turned a blind eye to these. Chapter 4 presents an approach which has been formulated and structured to deal with dependencies within the scope of this research project.

3.2.4 Identifying and Addressing Users' Post-deployment Needs

Description

Software is developed with an eventual goal of making the lives of its end users simpler. It is therefore imperative that users' input and experiences are incorporated at software development level. This becomes especially essential once the software has been installed on a user's machine after which point it will require constant maintenance to keep it going. Depending on the type of software system installed, there are different maintenance processes but the most common include updates, upgrades and uninstallations. Completing this task meant identifying pressing post-deployment user needs in the field of digital libraries and consequently devising means with which to address the most pressing of these user needs.

Methodology

Digital library communities are dispersed globally and, as such, it can prove to be ineffective, if not redundant, attempting to determine digital libraries users' post-deployment interests by simply looking at a local user sample representing such large communities. This motivated the idea of methodically scanning mailing lists of prominent digital library systems, specifically, those of DSpace and EPrints, with a hope that this will offer maximum insight into users' post-deployment needs on a global front as communities worldwide contribute to these mailing lists.

Fourteen months (to the end of February 2005) worth of mailing list entries were scanned for possible post-deployment needs and the following list presents the most popular of these needs. Section 5.1 shows the complete list of possible post-deployment user needs from scanned entries.

- Upgrading to newer software versions
- Migrating data from system to system, and
- Clarifying installation instructions and requirements

Some of these needs have been incorporated into the design of the packaging system and Chapter 4 elaborates more on this while a more thorough analysis on relevant scanned entries is done in Chapter 5.

3.3 Summary of Key Points

The problem statement — conveniently broken down into a series of smaller, manageable tasks, commanded initial research on a number of topics that this research touched on. The outcome of this preliminary research showed that current technologies are inappropriate in attending to the aims presented in the problem statement. This resulted in a decision that a new packaging system be built with an emphasis on the minimalistic and portable characteristics that this system must possess.

CHAPTER 4

DESIGN AND IMPLEMENTATION

Chapter 3 has presented the overview of this research giving the problem statement broken into simpler goals. The outcome of the initial research showed that in order to properly attend to the problem as stated, a packaging system was to be developed accordingly. This chapter therefore discusses the design and implementation strategies that were adopted in the development of this packaging system. The immediate section looks into the specifications that played a major role in the design stages of this packaging system. Subsequent sections describe the system architecture in depth.

4.1 SYSTEM ARCHITECTURE: SPECIFICATIONS

The design and implementation of the packaging system was enhanced by formalisms, some of which have been developed from scratch and others adopted from elsewhere. In this section, these specifications are introduced and discussed, aided by illustrations and examples where appropriate.

4.1.1 Component Connection Language

The Component Connection Language (CCL) stems from the design of the BLOX component composition Integrated Development Environment (IDE). Eyambe describes the CCL as a simple and effective XML-based manner with which to formalise the connectivity of components within the BLOX framework [19]. This formal specification has since been adopted and is now a vital entity in the package building and installation processes described in later sections. The next two sections will show and describe the original and modified CCL specifications.

Original CCL Specification

The CCL specification contains instance descriptions of all connected components. Figure 4.1 shows a shortened example of the original CCL specification. In this case, all the elements have been collapsed but briefly explained below since the details contained within are not a very important part of this research. For a full example, see Appendix A (A.1).

Within the `instance` elements are all the details pertaining to a particular component including configuration details, server details as well as details applicable only to the BLOX IDE.

```
    <?xml version="1.0" encoding="UTF-8"?>
[-] <ln:CCL xmlns:ln="http://nala.cs.uct.ac.za/ccl">
[+]     <instance>
[+]     <instance>
        .
        .
        .
[+]     <instance>
[+]     <connection>
        .
        .
        .
[+]     <connection>
    </ln:CCL>
```

Figure 4.1: Abridged original CCL specification example

Parts of the `instance` element are analysed further in later sections. Connectivity information describing the connectivity of each of the components described within the `instance` elements is contained within the `connection` elements. One of the important advantages of the CCL specification is that it facilitates automatic configuration of the specified components. This is achieved by extracting a particular instance's configuration and its associated connectivity information off the CCL specification and feeding that information to the component's automatic configuration script. Much of this process is described in Section 4.3.2.

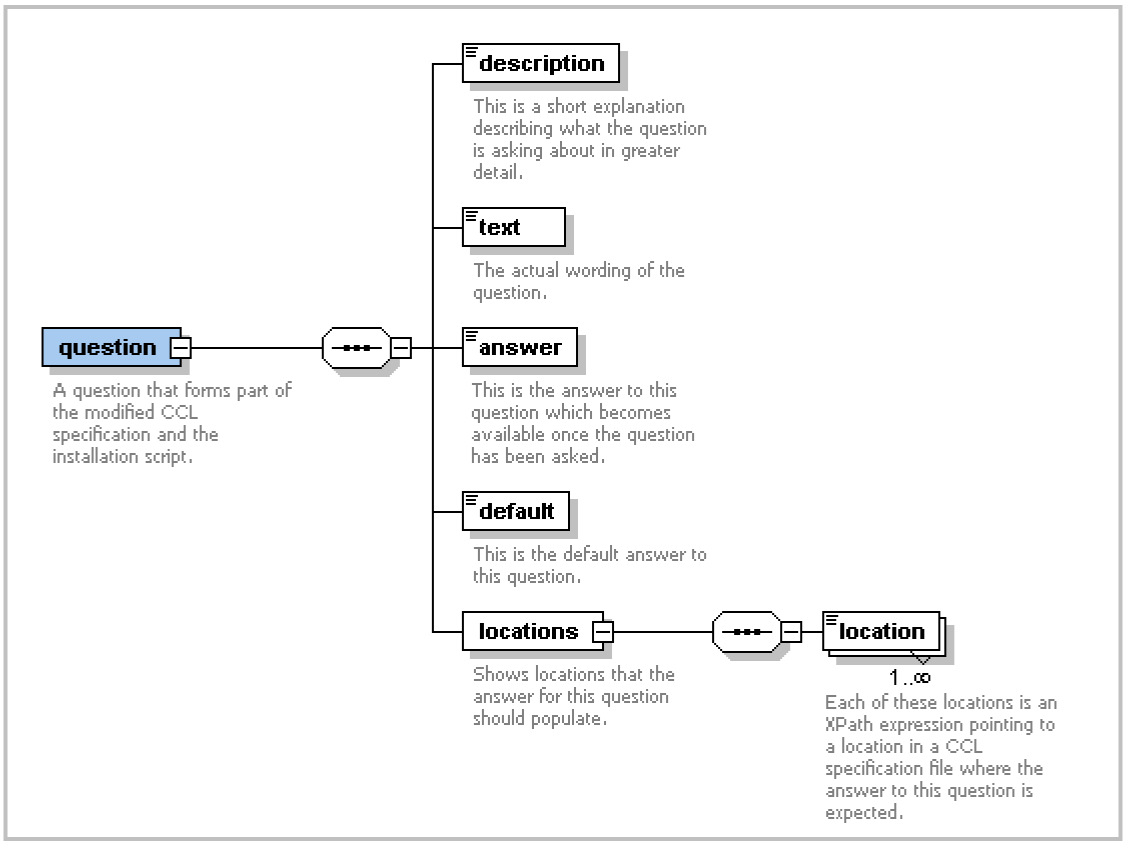
Modified CCL Specification

When configuring any ODL component manually, there is a series of questions aimed at gathering configuration information, which are asked by the component's configuration script. If installing a digital library composed of many ODL components, all the questions for each of the components (some of which are repeated for most if not for all components) need to be attended. This clearly shows that the amount of effort required to install such a component-based digital library is multiplied. In the BLOX IDE, the notion of a question has been abstracted to an interface where relevant values are inserted by the digital library composer per component. However, this manner of abstraction cannot be directly inherited by the implementation of the packaging system.

This motivated the design of the `questions` element, which forms part of the modified CCL specification. This element encloses an unlimited number of `question` elements each of which formally describes a single question. Figure 4.2 shows a skeleton example of a modified CCL specification, a full version of which can be seen in Appendix A (A.1). Figure 4.3 shows the structure of each of the `question` elements with annotations. The purpose for including questions in the CCL specification is twofold: at the very least, to ensure that the most mandatory of these questions (i.e., those questions that ask for the important configuration information without which the components will not function) are listed to be utilised by the package builder tool; and to reduce the effort required from the digital library composer in specifying these installation questions.

```
    <?xml version="1.0" encoding="UTF-8"?>
[-] <ln:CCL xmlns:ln="http://nala.cs.uct.ac.za/ccl">
[+]     <instance>
[+]     <instance>
        .
        .
        .
[+]     <instance>
[+]     <connection>
        .
        .
        .
[+]     <connection>
[-]     <questions>
[+]         <question>
[+]         <question>
            .
            .
            .
[+]         <question>
        </questions>
    </ln:CCL>
```

Figure 4.2: Abridged modified CCL specification example



A question that forms part of the modified CCL specification and the installation script.

This is a short explanation describing what the question is asking about in greater detail.

The actual wording of the question.

This is the answer to this question which becomes available once the question has been asked.

This is the default answer to this question.

Shows locations that the answer for this question should populate.

Each of these locations is an XPath expression pointing to a location in a CCL specification file where the answer to this question is expected.

Figure 4.3: Structure of the `question` element of the modified CCL specification

```
.
.
.
<question>
    <description>
        This is the username that will be used to connect to
        the database that has been specified.
    </description>
    <text>Please Input the Database Username</text>
    <answer></answer>
    <default>username</default>
    <locations>
        <location>/CCL/instance[1]/.../irdb/dbusername</location>
        <location>/CCL/instance[2]/.../dbunion/dbusername</location>
    </locations>
</question>
.
.
.
```

Figure 4.4: Example of the `question` element

In Figure 4.4 is a populated example of the `question` element. This example gives the actual wording of the question as well as a brief description of what the question asks about. In addition, two XPath locations into which the answer to this question should go are shown.

4.1.2 **Component Name Mappings**

The packaging and installation processes, as will be shown in Section 4.3, need to be able to map from any component's name that can be contained within a CCL specification onto an actual file name associated with that particular component. There is no defined relationship between component file names and those component names that can be contained within a CCL specification. This prompted that a formal method of describing these mappings be devised.

Figure 4.6 shows an annotated structure of the component name mappings specification. An elaborate mappings example can be found in Appendix A (A.2). There are no limitations to the number of component mapping descriptions that can be contained in a mappings specification as illustrated in Figure 4.6.

An example showing mapping information of an ODL component, `DBUnion`, is shown in Figure 4.5. It gives a short description of the component and shows that the component has three versions. Each `version` element contains text corresponding to a filename of a component being described from which an actual version number can be extracted. These versions are listed chronologically, with the first version listed first and the latest version listed last.

```
    <?xml version="1.0" encoding="UTF-8"?>
[-] <components>
[+]     <component>
[+]     <component>
        .
        .
        .
[-]     <component>
            <name>dbunion</name>
            <description>
                An ODL union archive component. Harvests several open archives
                and roles as a service provider for other ODL components.
            </description>
            <versions>
                <version>DBUnion-1.0.tar.gz</version>
                <version>DBUnion-1.1.tar.gz</version>
                <version>DBUnion-1.2.tar.gz</version>
            </versions>
        </component>
        .
        .
        .
[+]     <component>
    </components>
```

Figure 4.5: Example of the `component` element

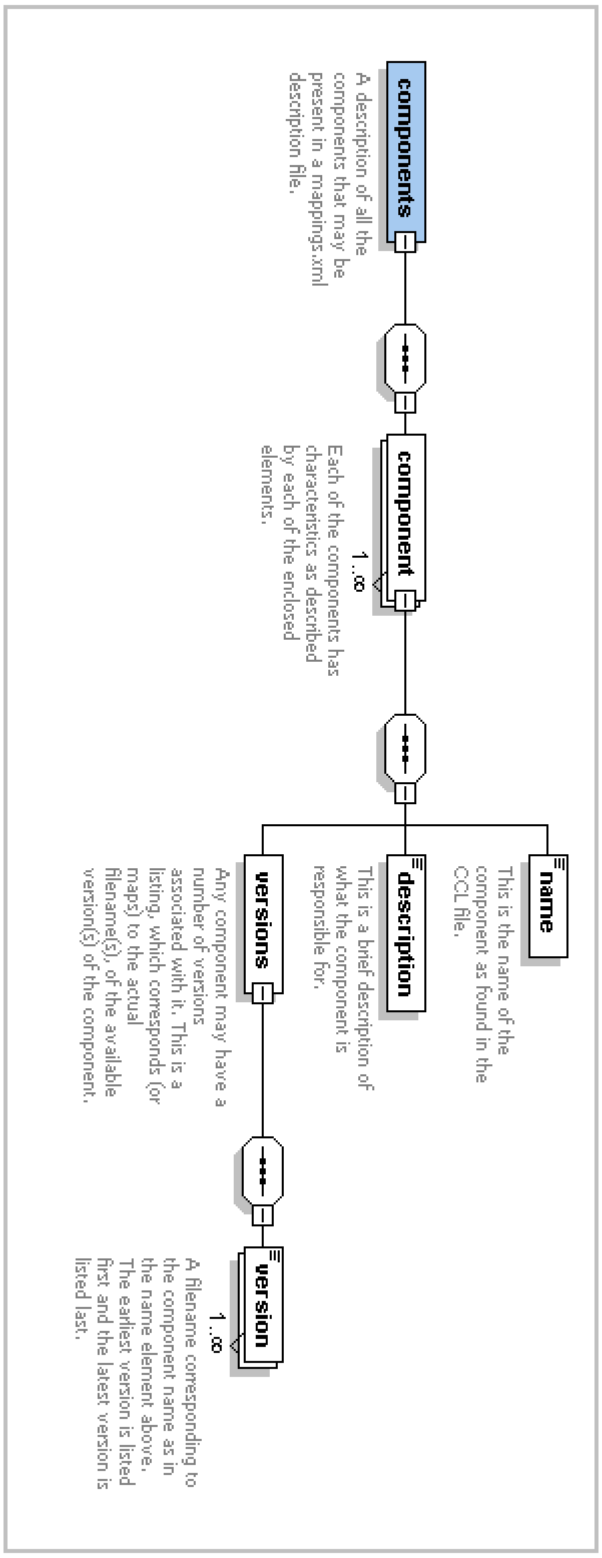

components
A description of all the components that may be present in a mappings.xml description file.
component
1..∞
Each of the components has characteristics as described by each of the enclosed elements.
name
This is the name of the component as found in the CCL file.
description
This is a brief description of what the component is responsible for.
versions
Any component may have a number of versions associated with it. This is a listing, which corresponds (or maps) to the actual filename(s), of the available version(s) of the component.
version
1..∞
A filename corresponding to the component name as in the name element above. The earliest version is listed first and the latest version is listed last.

Figure 4.6: Structure of the component name mappings specification

4.1.3 Dependency Specification

The primary aim for developing a dependency specification was so that components are able to formally describe software packages that they require in order for them to function appropriately. Each of the components describes its own set of dependencies together with sufficient information per dependency which can lead to various automated solutions at different stages of building and installing a package. Appendix A (A.3) shows a detailed example of the dependency specification.

If there are no dependencies associated with a component, the dependency specification can be without any `dependency` elements. Section 4.2 shows how dependency specification fits into the file structure of a component. Figure 4.7 shows the structure of the dependency specification. From this figure, it is evident that even a dependency can have its own set of dependencies which can be recursively specified. The dependency specification also allows for operating system (interchangeably used with ’platform’) related information to be encoded for each of the specified dependencies. Figure 4.8 structurally shows how this can be achieved.

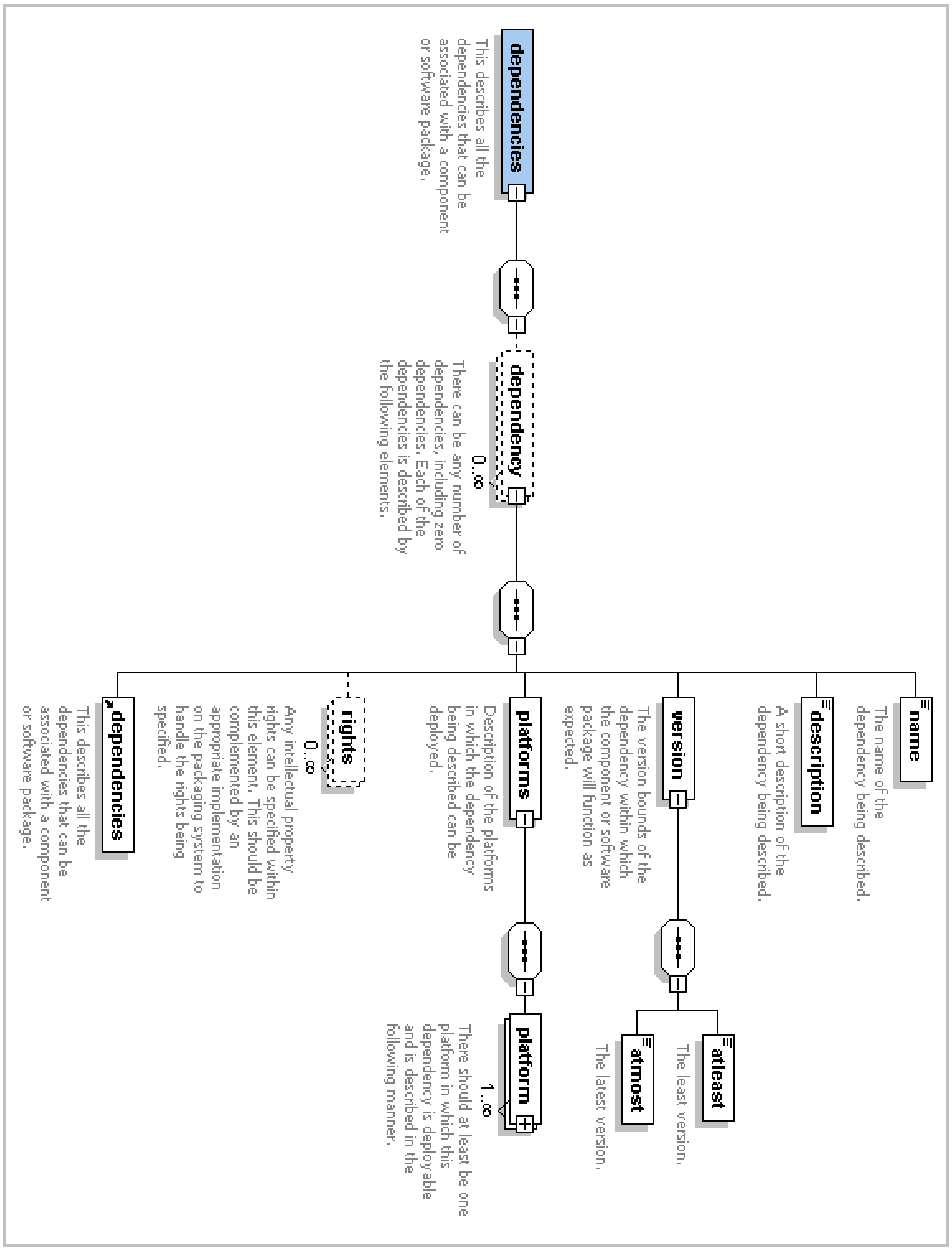

dependencies
This describes all the dependencies that can be associated with a component or software package.
dependency
0..∞
There can be any number of dependencies, including zero dependencies. Each of the dependencies is described by the following elements.
name
The name of the dependency being described.
description
A short description of the dependency being described.
version
The version bounds of the dependency within which the component or software package will function as expected.
atleast
The least version.
atmost
The latest version.
platforms
Description of the platforms in which the dependency being described can be deployed.
platform
1..∞
There should at least be one platform in which this dependency is deployable and is described in the following manner.
rights
0..∞
Any intellectual property rights can be specified within this element. This should be complemented by an appropriate implementation on the packaging system to handle the rights being specified.
dependencies
This describes all the dependencies that can be associated with a component or software package.

Figure 4.7: Structure of the dependency specification

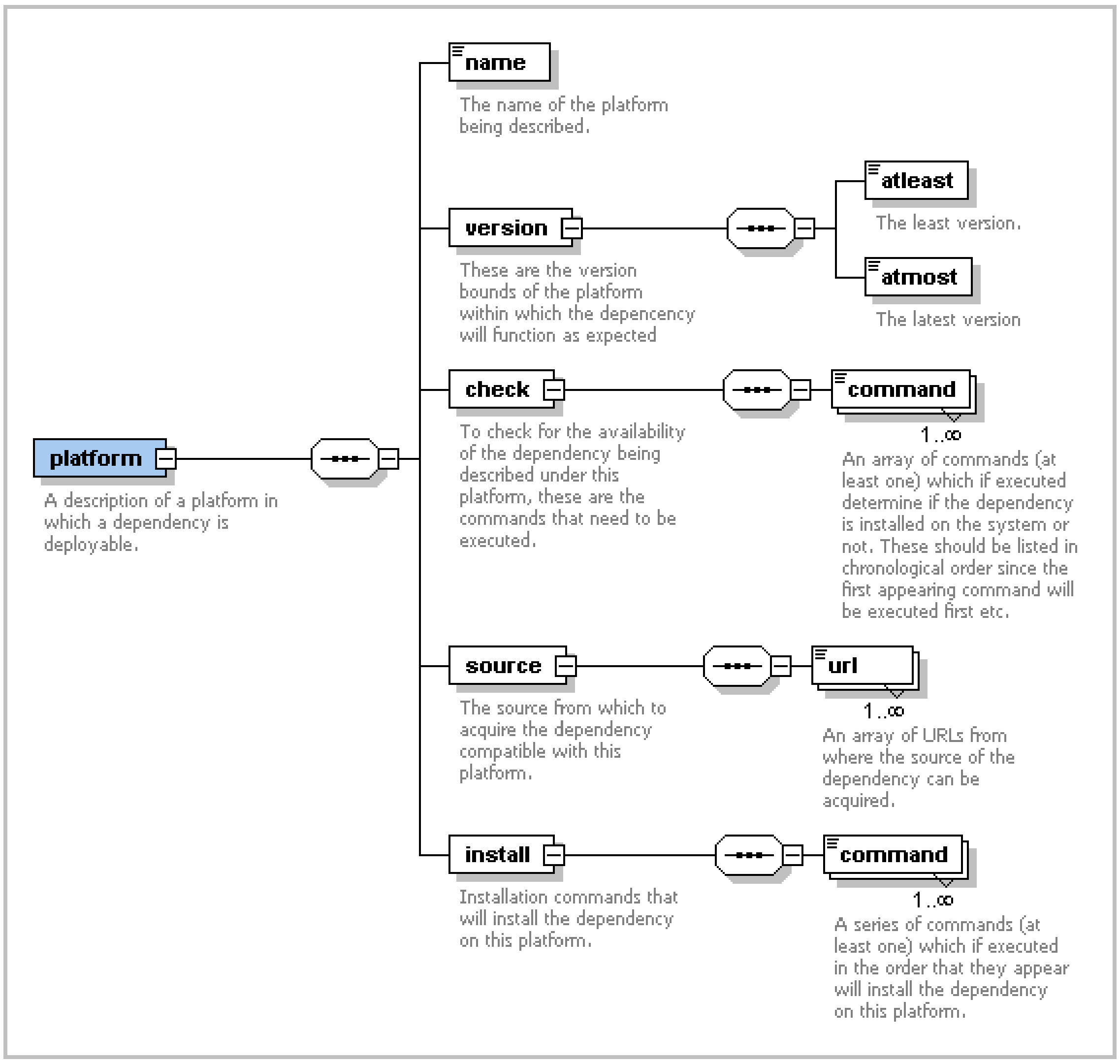

name
The name of the platform being described.
version
These are the version bounds of the platform within which the depencency will function as expected
atleast
The least version.
atmost
The latest version
check
To check for the availability of the dependency being described under this platform, these are the commands that need to be executed.
command
1..∞
An array of commands (at least one) which if executed determine if the dependency is installed on the system or not. These should be listed in chronological order since the first appearing command will be executed first etc.
platform
A description of a platform in which a dependency is deployable.
source
The source from which to acquire the dependency compatible with this platform.
url
1..∞
An array of URLs from where the source of the dependency can be acquired.
install
Installation commands that will install the dependency on this platform.
command
1..∞
A series of commands (at least one) which if executed in the order that they appear will install the dependency on this platform.

Figure 4.8: Structure of the `platform` element of the dependency specification

Figure 4.8 is an extension of Figure 4.7 and shows how platform information can be encoded within the dependency specification. Many software systems have varying versions for different operating systems, which is why it was essential to enable the encoding of information for each of the software system versions. This information can then be used in various ways as will be shown in Section 4.3.

An example of a `dependency` element is shown in Figure 4.9. It describes information of the `mysql` dependency, giving information of two platforms in which this dependency can be deployed. Appendix A (A.3) shows a complete example of specifying dependencies.

```
    <?xml version="1.0" encoding="UTF-8"?>
[-] <dependencies>
[+]     <dependency>
[+]     <dependency>
        .
        .
        .
[-]     <dependency>
            <name>mysql</name>
            <description>
                An Open Source Software relational Database Management System (DBMS)
                which uses a subset of ANSI SQL (Structured Query Language). This
                DBMS is be used by various components for data storage and retrieval
                purposes.
            </description>
            <version>
                <atleast>4.1.1</atleast>
                <atmost>5.0.15</atmost>
            </version>
            <platforms>
                <platform>
                    <name>GNU/Linux</name>
                    <version>
                        <atleast>2.2.13</atleast>
                        <atmost>2.6.15</atmost>
                    </version>
                    <check>
                        <command>java checkmysql</command>
                    </check>
                    <source>
                        <url>http://www.mysql.com/downloads/mysql-5.0.15.tar.gz/</url>
                    </source>
                    <install>
                        <command>gzip -cd | tar -xf - mysql-5.0.15.tar.gz</command>
                        <command>mysql-5.0.15/install</command>
                    </install>
                </platform>
                <platform>
                    <name>Windows</name>
                    <version>
                        <atleast>98</atleast>
                        <atmost>XP</atmost>
                    </version>
                    <check>
                        <command>java checkmysql</command>
                    </check>
                    <source>
                        <url>http://www.mysql.com/downloads/mysql-5.0.15.exe/</url>
                    </source>
                    <install>
                        <command>mysql-5.0.15.exe</command>
                    </install>
                </platform>
            </platforms>
            <rights/>
            <dependencies/>
        </dependency>
        .
        .
        .
[+]     <dependency>
    <dependencies>
```

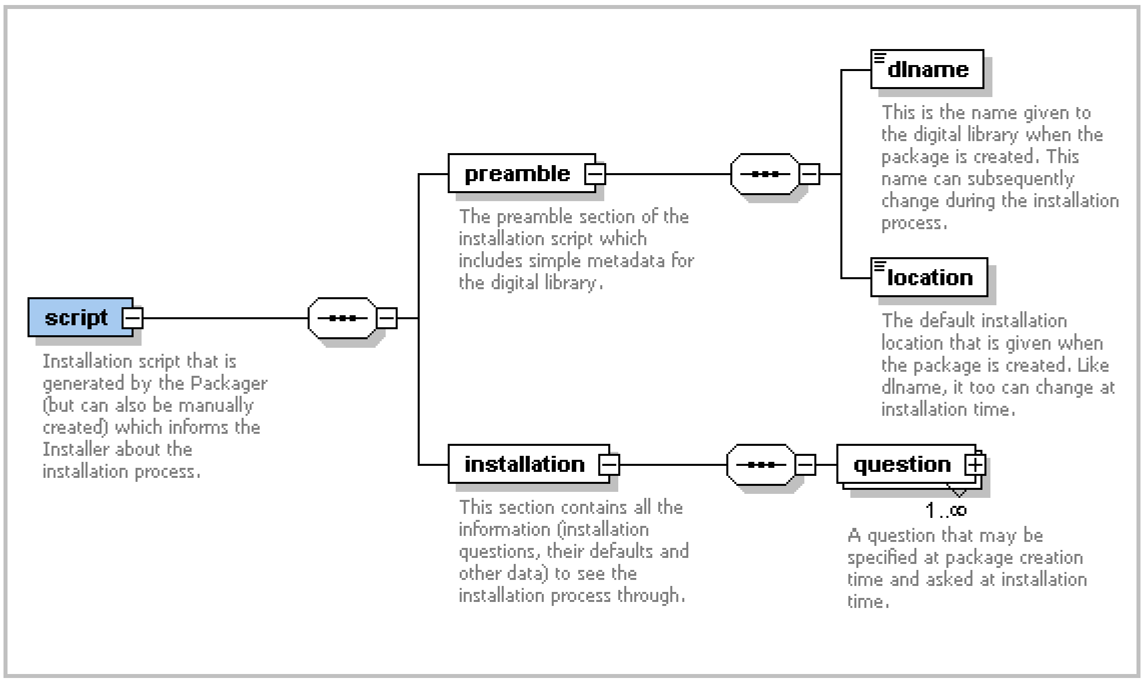
Figure 4.9: Example of the `dependency` element

4.1.4 Installation Script

The installation process is controlled by the installation script which is created by the packager tool at package creation time. The installation script is divided into two main sections, the `preamble` and the `installation` sections, as seen in Figures 4.10 and 4.11.

```
    <?xml version="1.0" encoding="UTF-8"?>
[-] <script>
[-]     <preamble>
            <dlname/>
            <location/>
        </preamble>
[-]     <installation>
[+]         <question>
[+]         <question>
            .
            .
            .
[+]         <question>
        </installation>
    </script>
```

Figure 4.10: Abridged installation script example



Installation script that is generated by the Packager (but can also be manually created) which informs the Installer about the installation process.

The preamble section of the installation script which includes simple metadata for the digital library.

This is the name given to the digital library when the package is created. This name can subsequently change during the installation process.

The default installation location that is given when the package is created. Like dlname, it too can change at installation time.

This section contains all the information (installation questions, their defaults and other data) to see the installation process through.

A question that may be specified at package creation time and asked at installation time.

Figure 4.11: Structure of the installation script

The `preamble` section contains simple metadata for the digital library. The name and the installation location of the digital library package are respectively encoded under the `dlname` and `location` elements. The `installation` element contains installation questions that are included at packaging time and are asked at installation time. These follow the exact structure as that which is depicted in Figures 4.3 and 4.4. A full example of the installation script is seen in Appendix A (A.4)

4.2 System Architecture: File Structures

The core of the design and implementation process of this research was to devise a methodology whereby an installable package can be built from loose components. These components, ODL components, have been introduced in earlier chapters but not much detail has been provided about their internal file structures. In this section, the internal file structure of these components is highlighted together with that of packages that are created with the package builder tool discussed in Section 4.3.1. This has been done so that the interface through which the packaging system as a whole interacts with components is clearly defined before delving into how the tools carry out their respective processes.

4.2.1 ODL Components' File Structure

The file structure of ODL components is similar and as such it is sufficient to describe it using any one component as an example. Figure 4.12 partially shows how files are organised in ODL components relative to the root location using the `DBUnion` component as an example. For clarity purposes, files that are not directly related to this research have been omitted in the figure.

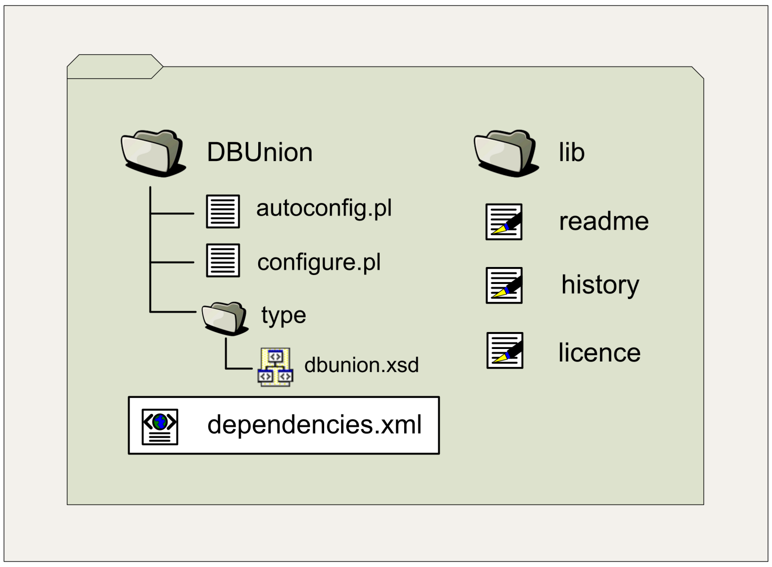


Figure 4.12: File structural organisation of ODL components

The `DBUnion` directory contains two important files: `configure.pl` and `autoconfig.pl`. These files are responsible for manually and automatically configuring an instance of the `DBUnion` component respectively upon request. The `type` directory, also contained within the `DBUnion` directory, holds the XML schema definition of the `DBUnion` component. Libraries that the component relies on are contained in the `lib` directory. The `readme`, `history` and `licence` files are self-explanatory.

An important addition to the file structure of the components is that of the dependency specification file, `dependencies.xml` — highlighted with a white background in Figure 4.12, describing all the dependencies that the component requires in order to function appropriately. The structure and format of this file is as documented in Section 4.1.3.

4.2.2 Package Format

The output of the package creation process is a package. This package is in the form of a compressed `ZIP` file which contains all the files necessary for an installation to carry through successfully. Figure 4.13 shows the structure and organisation of this package.

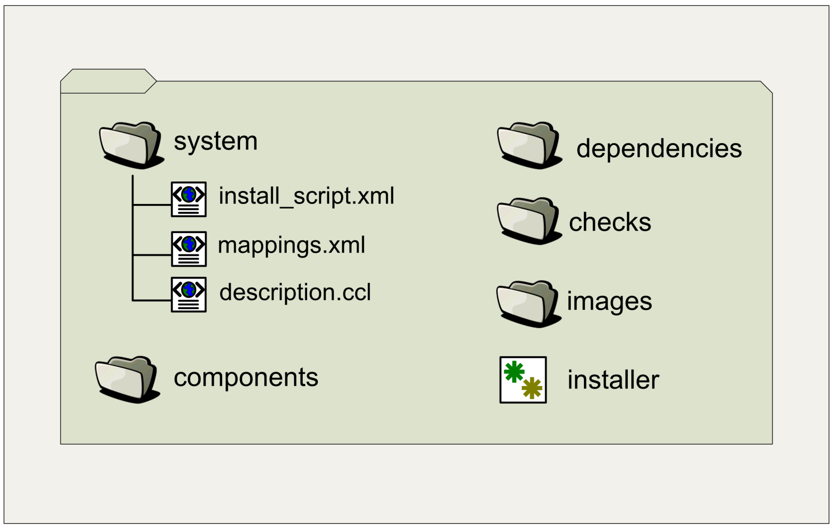


Figure 4.13: Package structure and contents

The `system` directory contains all the important scripts and specifications that are required at installation time. To drive the installation process is the installation script `install_script.xml`. The component name mappings specifications, `mappings.xml`, and the CCL specification, `description.ccl`, also form part of the `system` directory. These have all been introduced and described in Section 4.1.

All the components that form part of the package (described in the CCL specification) are contained in the `components` directory. The file structure of components has been described in the previous section. If any of these components have any dependency requirements, scripts and/or programs that are responsible for checking for the availability of these dependencies can be found in the `checks` directory and if these dependencies are allowed to be bundled and distributed with any software package, these dependencies can then be found in the `dependencies` directory.

To install the package, the user needs to run the `installer` program. This is an executable Java^{TM} Archive (`JAR`) file which contains the compiled source of the package installer tool. The `images` directory contains images that assist in rendering the interface of the package installer tool. Section 4.3.2 further discusses the package installer tool.

4.3 SYSTEM ARCHITECTURE: TOOLS

The packaging system is comprised of a pair of tools that complete the processes of constructing and ultimately installing a component-based digital library package. One of the subtasks of the problem statement presented in Chapter 3 was to design for heterogeneous operating systems. Employing the cross-platform Java^{TM} programming language in conjunction with XML for data presentation, as manifested in the design of specifications in Section 4.1, was a design strategy adopted to deal with the heterogeneity of operational platforms.

The remainder of this section looks into the design and implementation of the pair of tools that are key to the packaging system. This will put into context the specifications and other information from previous sections of this chapter.

4.3.1 Package Builder

Building a package essentially means structuring the package suitably by including in the package all files necessary, excluding all files unnecessary and resolving most possible complications that may arise at package installation time, all in a consolidated effort to improve and make the process of installing software more bearable. Figure 4.14 summarily shows all the steps of the package building process.

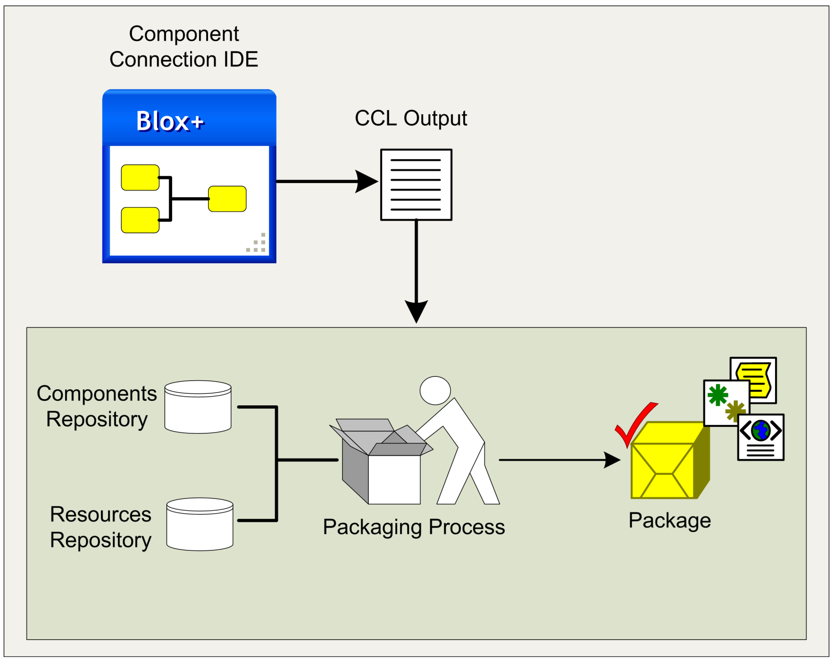


Figure 4.14: Package building process

This package building process can be better understood as that comprising of two distinct stages: gathering and analysing input; and structuring the output package. The following two sections look into these two steps in further detail.

Input and Input Analysis

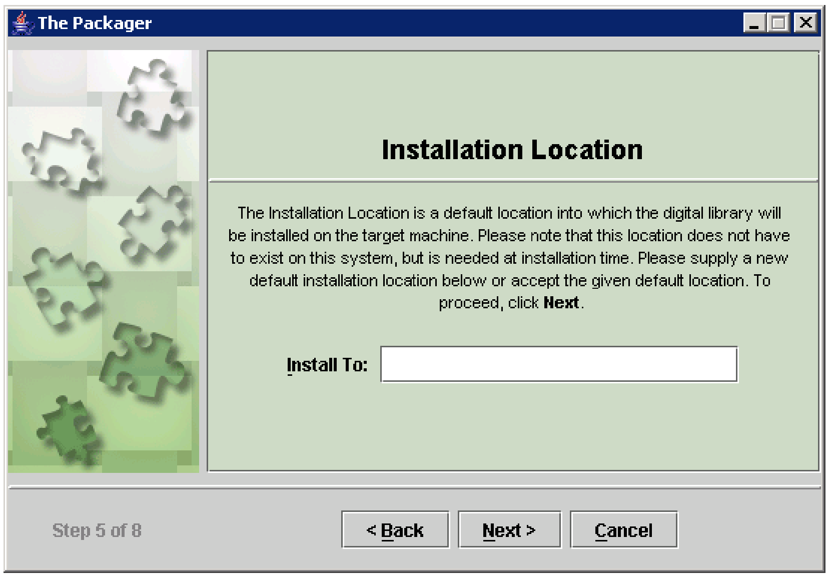
The aim of the package builder is to build a digital library package according to an input specification. The input to the packaging process is a CCL specification file as shown in Figure 4.14. This specification file is produced primarily by the BLOX system and is slightly modified as detailed in Section 4.1.1 to serve as input to the package creation process. Once the specification has been supplied to the package builder tool, the components that make up the digital library are systematically identified and consequently, other information related to the components, such as dependency information, is identified. At this point, the component name mappings specification, `mappings.xml`[1], is used to find the exact filenames of components in preparation for the final step of structuring the package.

Structuring the Package

Structuring the package is concerned with gathering the package's metadata as well as other information and entities that are inferred from the input specification to ultimately produce a single installable package. This process can therefore be segmented into three sections which are elaborated upon in the following sections.

Static Questions

Static questions are designed to gather information about the package (metadata) that is used at installation time. These are static in that they are built into the packaging tool and need to be answered irrespective of the package being built. Figure 4.15 shows an example of one such static question.



The Installation Location is a default location into which the digital library will be installed on the target machine. Please note that this location does not have to exist on this system, but is needed at installation time. Please supply a new default installation location below or accept the given default location. To proceed, click **Next**.

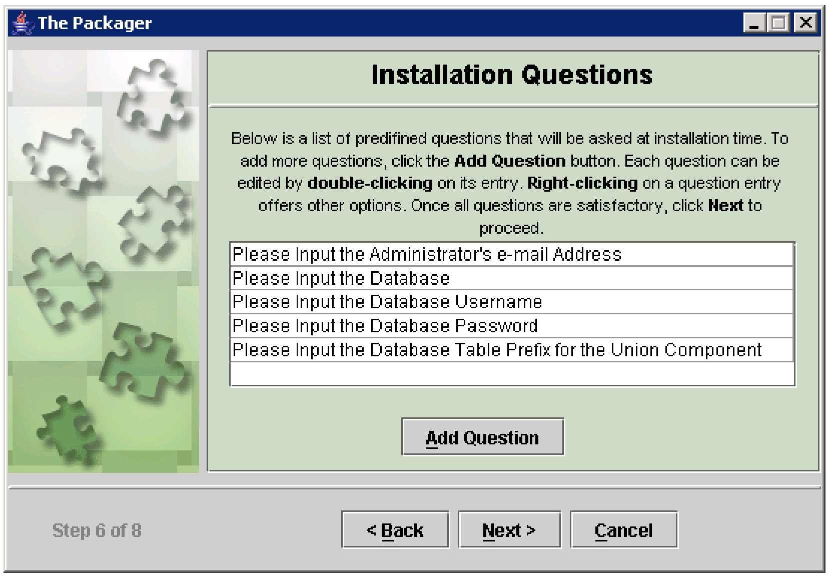
Figure 4.15: Static question: Default digital library installation location

[1]The `mappings.xml` specification can be created manually as new versions of components become available. Ideally, a simple application that handles component additions can be more useful.

This question asks for the default installation location whereas the other static question asks for the default digital library name. These static questions are given default values when the package is being built and these values are presented at installation time.

Dynamic Questions

Dynamic questions are those that are constructed by the package designer when the package is being built. To reduce the effort required from the package designer, the essential of these questions are already encoded within the input CCL specification, hence the modified CCL specification in Section 4.1.1. These questions are specific to the configuration of the components that the digital library is composed of. Figure 4.16 shows an interface through which these questions are accessible. Each of these questions can be edited by the package designer if there is a need to do so. Additional questions can also be constructed.



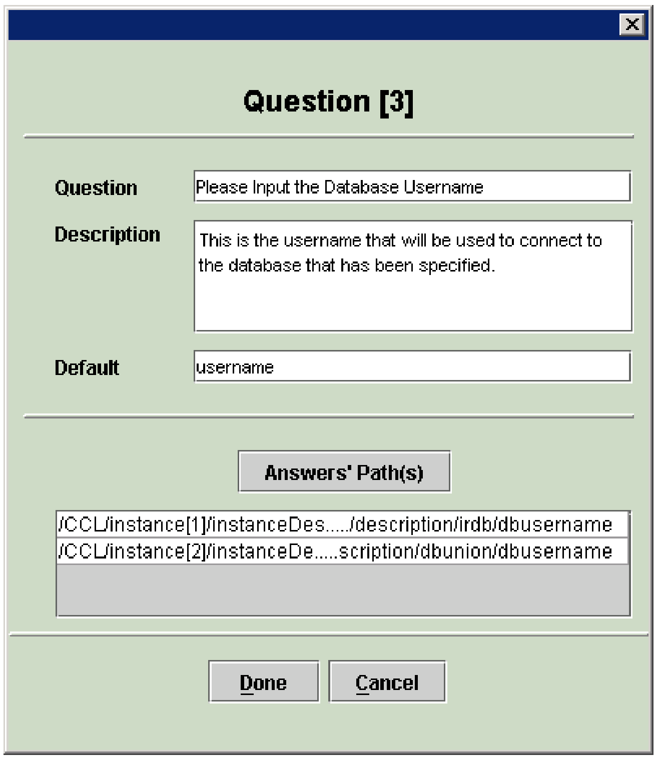
Below is a list of predifined questions that will be asked at installation time. To add more questions, click the **Add Question** button. Each question can be edited by **double-clicking** on its entry. **Right-clicking** on a question entry offers other options. Once all questions are satisfactory, click **Next** to proceed.

Please Input the Administrator's e-mail Address
Please Input the Database
Please Input the Database Username
Please Input the Database Password
Please Input the Database Table Prefix for the Union Component

Figure 4.16: Dynamic questions: Constructing digital library installation questions

When installing components individually, each component has its set of questions that it asks of a user. For ODL components, these questions have a varying degree of similarity across most of them. It is therefore necessary to ensure that during package installation, the user is presented with a seemingly uniform installation process while each of the components' configurations take place transparent to the user. This was one of the problems that the packager tool dealt with. Figure 4.17 shows an example question which asks for a database username of a database shared between two components.

What Figure 4.17 also shows is a question editing/adding interface. Three of the fields in this interface, the `question`, `description` and `default` fields are relatively self-explanatory. The `Answers' Path(s)` deserves further explanation. The package designer needs to specify components which expect an answer to this question. This is done through the The `Answers' Path(s)` button, which presents the package composer with an interface containing a graphical presentation of the input CCL specification. Through this interface, the package composer can select a path to which the answer to the question being asked must go, once available at installation time. Figure 4.18 shows this interface.



Question [3]

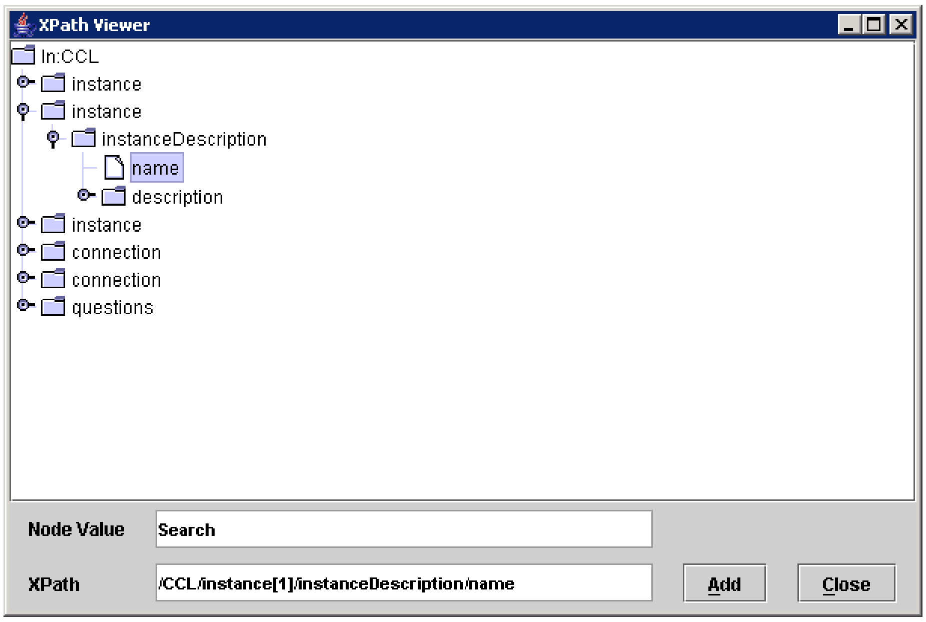
Question: Please Input the Database Username

Description: This is the username that will be used to connect to the database that has been specified.

Default: username

/CCL/instance[1]/instanceDes...../description/irdb/dbusername
/CCL/instance[2]/instanceDe.....scription/dbunion/dbusername

Figure 4.17: Dynamic question: Question constructing interface



Node Value: Search

XPath: /CCL/instance[1]/instanceDescription/name

Figure 4.18: Specifying answer location from CCL specification

A package composer is able to select the answers' path guided by the `Node Value` and `XPath` fields. These fields show the node value and the XPath expression of the selected node.

```
.
.
.
<instance>
    <instanceDescription>
        <name>Search</name>
        <description>
            <blox:irdb>
                <repositoryName>ODL Search Engine</repositoryName>
                .
                .
                .
                <archive>
                    <identifier>HUSPICS</identifier>
                    <url>
                        <!--URL_TO_CGI_LOCATION-->/ODL-DBUnion-1.2/
                        DBUnion/evaluation/union.pl
                    </url>
                    .
                    .
                    .
                </archive>
            </blox:irdb>
        </description>
    </instanceDescription>
</instance>
.
.
.
```

Figure 4.19: Special token for path to CGI location

There is another question type which the package builder tool deals with internally. This can be thought of as a static dynamic question type — a static question which gets dynamically added if needed. To put this into perspective, consider a case of the installation location of an ODL-component-based digital library. ODL components need to be instantiated in a location where CGI/Perl scripts are executable. For these components to interconnect, each component requires URLs (baseURLs) of all the components that it connects to. However, for other types of components, this may not necessarily be the case. So, the package builder tool needs to dynamically determine whether to include a question, such as asking for a URL to a CGI location, for the installation process or not. To achieve this, special tokens are embedded in the CCL specification file, which consequently inform the packager tool of these static dynamic questions to include when finalising the package. Figure 4.19 shows an excerpt from a CCL specification file with a special token `<!--URL_TO_CGI_LOCATION-->`.

Special tokens such as these correspond to predefined questions both (the special token and its corresponding predefined question) of which can be specified in the source code of the package builder tool. The package builder tool systematically scans the CCL specification file for any special tokens and include any corresponding questions when finalising the package. The special token illustrated in Figure 4.19 tells the package builder tool to include its corresponding question shown Figure 4.20.

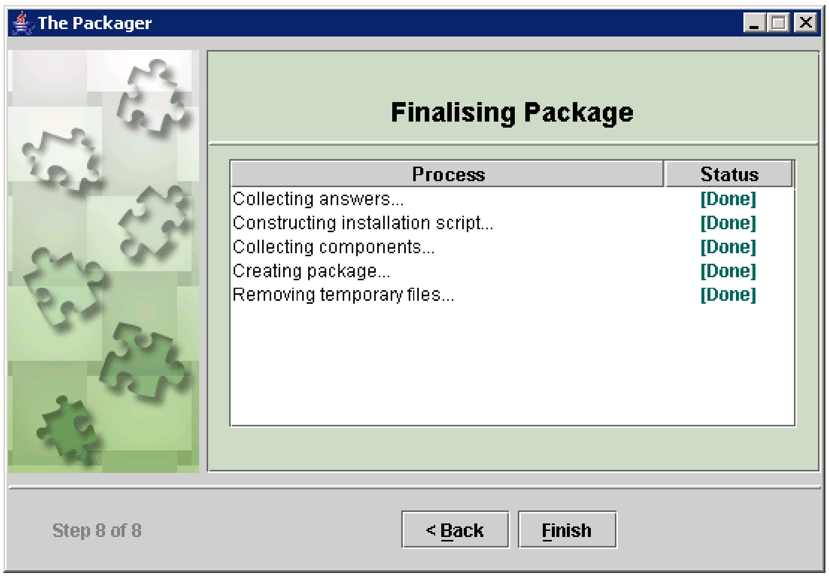
```
.
.
.
<question>
    <description>
        The URL to CGI location is that which grants access to the
        CGI location where this Digital Library will be installed.
    </description>
    <text>Please Input URL to CGI Location</text>
    <answer> </answer>
    <default>http://localhost/cgi-bin/</default>
    <locations>
        <location> <!--URL_TO_CGI_LOCATION--> </location>
    </locations>
</question>
.
.
.
```

Figure 4.20: Static dynamic question: Question for path to CGI location

Once the answer to this question is available at installation time, it replaces the special token found in the CCL specification file, otherwise the default value replaces the special token.

Finalising the Output Package

The final stage of the package creation process follows this sequence: collect all questions and their associated data; create an installation script; collect components and other associated entities (from the components and resources repositories as in Figure 4.14); create the actual package; and dispose of unwanted files that were utilised by the package builder tool during the package creation process. Figure 4.21 shows the interface that gives feedback on these sequential processes.



Process	Status
Collecting answers...	[Done]
Constructing installation script...	[Done]
Collecting components...	[Done]
Creating package...	[Done]
Removing temporary files...	[Done]

Figure 4.21: Package finalising process

The installation script is made up of all the questions that are collected in the first step. The `preamble` section of the installation script (recall from Section 4.1.4) is made up of all the information gathered by static questions. The remainder of the questions (the dynamic and static dynamic questions) form the rest of the installation script and are encoded within the `installation` element of the installation script.

From the input stage of the package creation process, components and other associated entities are known. These are then collected and structured accordingly, conforming to the package format described in Section 4.2.2, before being bundled and compressed as a `ZIP` file ready for distribution.

4.3.2 Package Installer

The process of installing a digital library package is relatively simpler if compared to that of building a digital library package and really depends on how the package was structured at creation time. Figure 4.22 shows a summary of steps that are taken when installing a package.

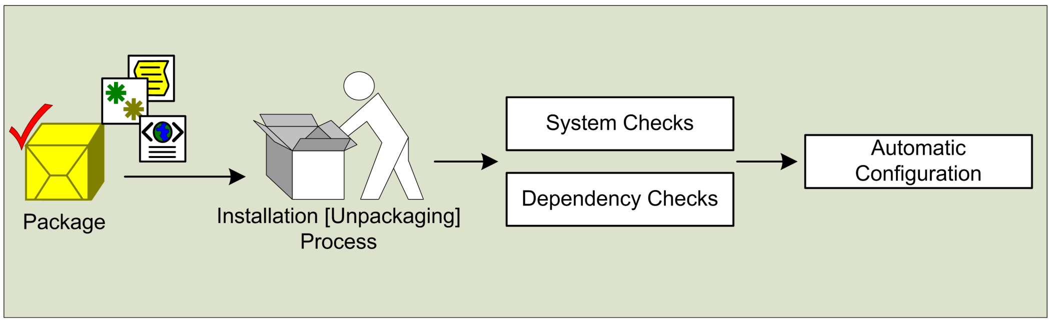


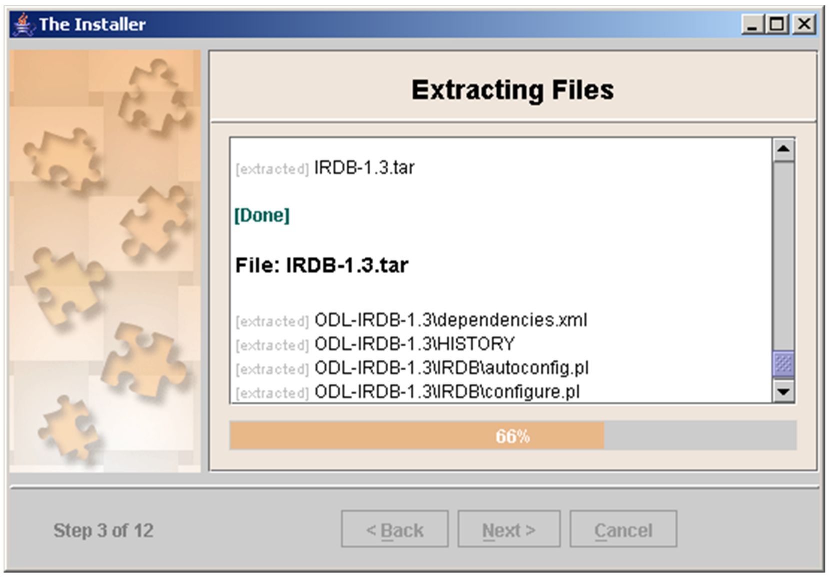
Figure 4.22: Package installation process

There are four steps that this process follows. These are: package contents extraction; system checks; questioning; and finalising the installation process. The following sections elaborate more on these steps.

Package Contents Extraction

The extraction of the contents of the package is twofold: it is the extraction of the package itself as it is a compressed `ZIP` file; and it is the extraction of the components into their nominated installation location. The former extraction process is intuitive and results in a root directory adhering to the package format described in Section 4.2.2. The latter process happens during the actual installation of the package once the `installer` program has been launched.

The installer tool prompts the user to supply an installation location into which the digital library should be installed. The components are then extracted into this location after which they are ready to be instantiated. Figure 4.23 shows the extraction of all the component files into a target installation location.

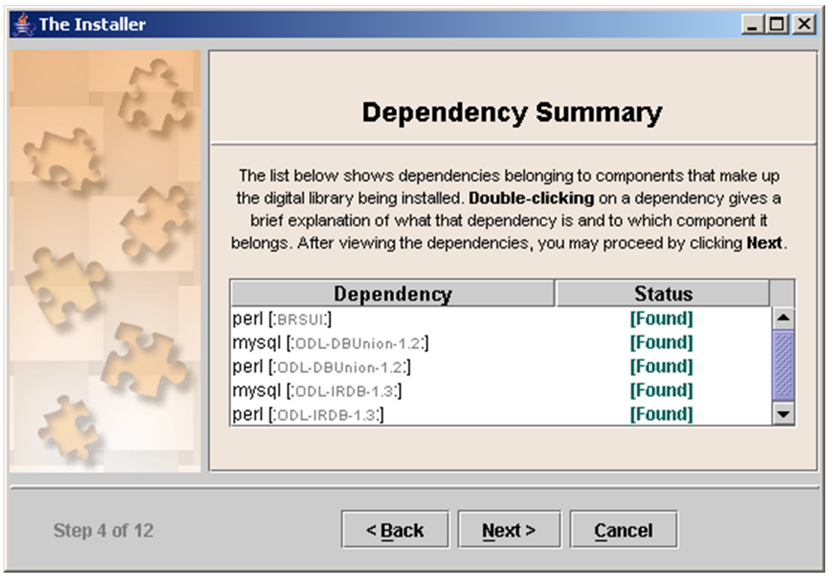


```
[extracted] IRDB-1.3.tar

[Done]

File: IRDB-1.3.tar

[extracted] ODL-IRDB-1.3\dependencies.xml
[extracted] ODL-IRDB-1.3\HISTORY
[extracted] ODL-IRDB-1.3\IRDB\autoconfig.pl
[extracted] ODL-IRDB-1.3\IRDB\configure.pl
```

Figure 4.23: Extracting component files into an installation location

System Checks

Once the components have been extracted into an installation location, the package installer tool then systematically scans through all the components' root directories for their associated dependencies specified in `dependencies.xml` files, the structure of which is introduced in Section 4.2.1. Once all the dependency information has been collected, the package installer is then able to perform some checks based on the dependency specifications. Figure 4.24 shows a typical result set after performing some of these checks.



The list below shows dependencies belonging to components that make up the digital library being installed. **Double-clicking** on a dependency gives a brief explanation of what that dependency is and to which component it belongs. After viewing the dependencies, you may proceed by clicking **Next**.

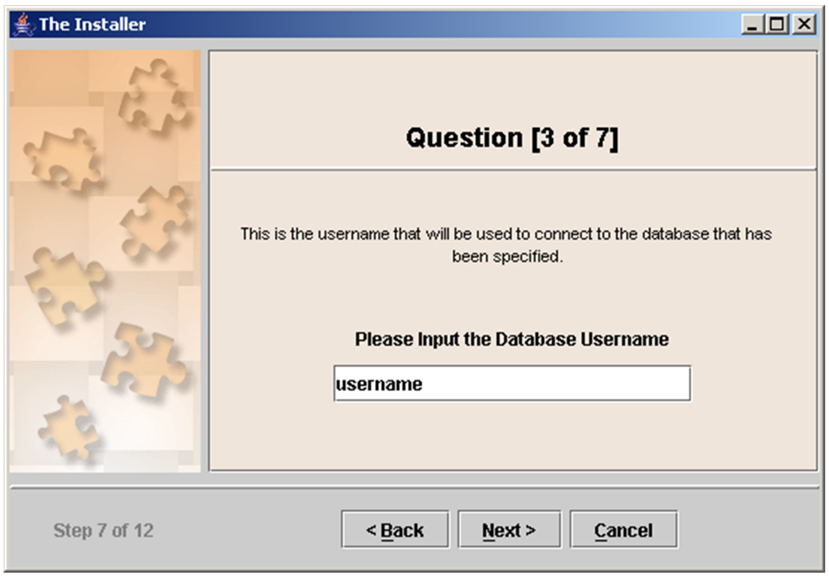
Dependency	Status
perl [:BRSUI:]	[Found]
mysql [:ODL-DBUnion-1.2:]	[Found]
perl [:ODL-DBUnion-1.2:]	[Found]
mysql [:ODL-IRDB-1.3:]	[Found]
perl [:ODL-IRDB-1.3:]	[Found]

Figure 4.24: Dependency check summary

The dependency summary shows the dependencies as well as their associated components. For example, `perl [:BRSUI:]` means that `perl` is a dependency associated with the `BRSUI` component. The status column shows the outcome status after the dependency checks have been performed. The checking process depends on the availability of checking scripts and/or programs found in the `checks` directory (recall from Section 4.2.2) which should correspond to the dependencies specified by components. Should these checks return unfavourable results, it is likely that the installation process will not succeed.

Questioning

The questioning process is the most important part of the installation process. This is where the installer tool gets to ask the dynamic and static dynamic questions that have been included at package creation time. Figure 4.25 shows an interface through which a question is asked at installation time.



This is the username that will be used to connect to the database that has been specified.

Please Input the Database Username

username

Figure 4.25: Asking a question at installation time

This question asks for a database username of a database to be used by the components being installed. At this point, as emphasised in Section 4.3.1, the user is unaware that this question is in fact linked with two components. Once all the questions have been answered, the next step is to finalise the installation process.

Finalising Installation Process

This process is sequentially similar to the final step of the package creation process. Figure 4.26 shows how this process is carried through in three simple steps. In the initial step of this process, all the answers to the questions that have been asked are collected. These answers are then stored in the installation script, under corresponding `answer` elements, for future references and are also distributed to all the XPath locations, found under the `locations` element of the `question` element as seen in Section 4.1.1, in the CCL specification file. The following two steps build towards instantiating/configuring each of the components to produce a fully fledged digital library.

Recall from the structure of ODL components in Section 4.2.1 that there are two scripts,

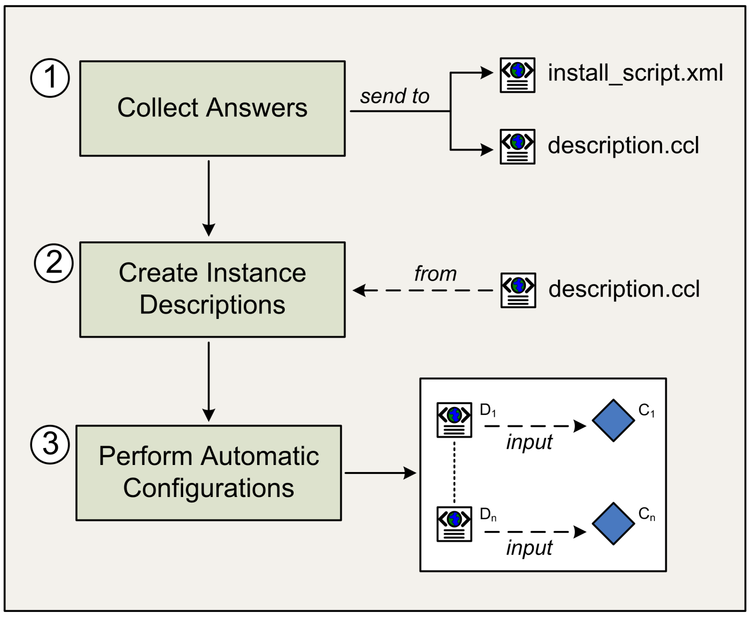
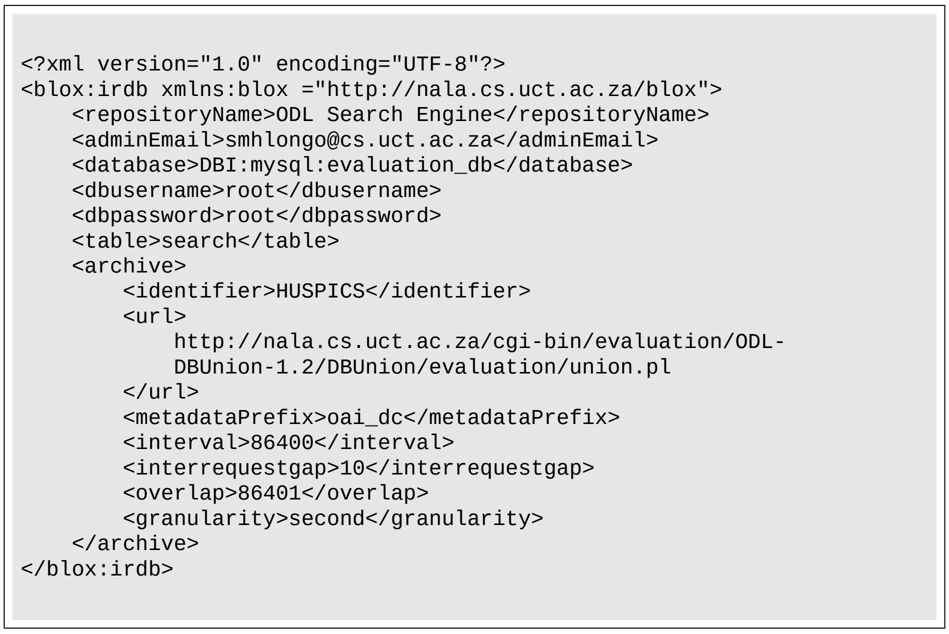


Figure 4.26: Finalising installation process

`configure.pl` and `autoconfig.pl`, that deal with the configuration of the component. For automatic configuration, the `autoconfig.pl` script is used. This script is a byproduct of the work done by Eyambe on a component connection assembly of digital libraries [19]. This script takes as input an instance name and an instance description encoded in an XML file named `config.xml`. Figure 4.27 shows an example of how this instance description file is set up for the `IRDB` ODL component.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<blox:irdb xmlns:blox ="http://nala.cs.uct.ac.za/blox">
    <repositoryName>ODL Search Engine</repositoryName>
    <adminEmail>smhlongo@cs.uct.ac.za</adminEmail>
    <database>DBI:mysql:evaluation_db</database>
    <dbusername>root</dbusername>
    <dbpassword>root</dbpassword>
    <table>search</table>
    <archive>
        <identifier>HUSPICS</identifier>
        <url>
            http://nala.cs.uct.ac.za/cgi-bin/evaluation/ODL-
            DBUnion-1.2/DBUnion/evaluation/union.pl
        </url>
        <metadataPrefix>oai_dc</metadataPrefix>
        <interval>86400</interval>
        <interrequestgap>10</interrequestgap>
        <overlap>86401</overlap>
        <granularity>second</granularity>
    </archive>
</blox:irdb>
```

Figure 4.27: Example of an ODL component instance configuration information

This instance description can in fact be derived from each of the `instance` elements that are part of the CCL specification. So, after the answers to all the questions have been made available to the components that need them, the package installer tool is then able to extract instance descriptions (D_is) from the CCL specification and use them as input in the automatic configuration of each of the components (C_is) making up the digital library. This is illustrated by steps 2 and 3 of Figure 4.26. The order of the configuration is determined by the connection information, also encoded in the CCL specification under `connection` elements. Besides instantiating a component, it may be required that it harvests (collects) data from other components or Open Archives[2] that it connects to before other components that connect to it can be fully functional. This task also forms part of the installation finalisation process.

4.4 SUMMARY OF KEY POINTS

The design and implementation phase of this research was concerned with identifying development strategies aligned with meeting the requirements presented in Chapter 3. A clearly defined set of specifications was developed to be ultimately used in the development of the packaging system composed of two tools: the package builder and package installer. The specifications were presented in XML format while the tools were a Java^{TM} implementation. This ensured the heterogeneity of the packaging system, which was one of the aims presented by the problem statement.

[2]An Open Archive is a data store which is compliant with the OAI.

CHAPTER 5

EVALUATION AND RESULT ANALYSIS

"... evaluation is what it takes in order to know if all your hard work is achieving the results you seek! " — http://www.benton.org/ —

Chapter 3 introduced the problem statement of this research. This problem statement was subsequently broken down into smaller tasks and the methodologies that were employed in completing these tasks were then outlined. The key outcome was that a packaging system for packaging and managing component-based digital libraries was to be implemented in meeting some of these tasks while some research was concerned with discovering user requirements in the field of digital libraries, some of which would later be incorporated into the design of the packaging system. In this chapter, an analysis of these user needs is presented in the immediate section. Section 5.2 presents and discusses results that were obtained from the controlled user study that was conducted in order to evaluate the packaging system across different criteria.

5.1 DIGITAL LIBRARY USERS' NEEDS

Identifying digital library users' post-deployment needs was one of the key tasks related to the problem statement. In Chapter 2, an outline of activities constituting a software deployment process as well as issues that complicate this process was presented. Although these activities and issues are applicable to the deployment process of digital library systems, it was necessary to uncover those which were (or more correctly, are) most important to digital library communities. This was achieved by periodically scanning the mailing lists of the DSpace and EPrints digital library systems, as these systems are two of the highly regarded in the field. Fourteen months (to the end of February 2005) worth of mailing list entries were manually scanned for possible deployment needs. The most relevant of the scanned entries are shown in Figures 5.1 and 5.2. Table 5.1 shows how each of these entries can be distributed across 6 different categories and the analysis of these categories is presented in the following sections.

```
1  License agreement for each item [2003-12-30 15:31]
2  DSpace running problems: am I missing a package of some sort [2004-05-11 12:25]
3  Batch import tools [2004-06-23 12:05]
4  Importing items [2004-08-18 13:42]
5  Moving items from one collection to another [2004-09-01 09:48]
6  Migrating data between servers [2004-09-07 08:59]
7  Java component license [2004-09-10 08:37]
8  Moving items from one collection [2004-09-29 01:34]
9  Moving a collection [2004-10-18 05:39]
10 Upgrade 1.1.1 to 1.2, now select e-person doesn’t pop up [2005-02-11 12:33]
11 Need information on possible DSpace security issue [2005-02-16 06:59]
12 Import problem [2005-02-24 08:23]
13 FW: DSpace upgrade - please help!!! [2005-02-28 00:36]
```

Figure 5.1: Scanned entries from the DSpace technical mailing list

```
1  Mass import of files [2005-01-26]
2  Export citation function[2004-11-30]
3  Moving an archive from one table to another [2004-11-01]
4  EPrints 3 update [2004-09-10]
5  Moving databases [2004-09-24]
6  Loss of eprints in database when moving to new server [2004-08-17]
7  When are upgrades appropriate [2004-08-26]
8  Problems with eprints installation [2004-07-15]
9  New installation: Configuration files [2004-07-27]
10 Installation problems [2004-07-29]
11 Importing external data [2004-07-29]
12 New install [2004-05-28]
13 Import EPrints [2004-05-31]
14 Moving EPrints to a new server [2004-03-15]
15 Migrating EPrints to new server [2004-02-16]
16 Also stuck while migrating/upgrading [2004-02-21]
17 Perl modules bundled with EPrints [2004-02-26]
18 Security & memory [2004-02-29]
19 Importing data into EPrints [2004-03-03]
20 Problems with apache.conf in EPrints 2.3.3 [2004-03-08]
21 Moving EPrints to another server [2004-01-20]
22 Moving EPrints to a new server [2004-02-10]
23 Upgrade blues [2004-02-09]
```

Figure 5.2: Scanned entries from the EPrints technical mailing list

5.1.1 Post-deployment User Needs

Data and System Migration — Out of all the entries that are shown in Figures 5.1 and 5.2, more than half are concerned with moving data into or out of systems and moving complete systems from one server to another. Data migration is mostly a result of a fresh installation of a similar digital library system being installed elsewhere, whereby data needs to be moved out of an existing installed system to a newly installed one. Another factor leading to data migration is system backups (especially upgrade-induced). It is safe that when upgrading a digital library system, data be somehow backed up to be restored once the upgrade has succeeded so that at no point during the upgrade process is the data compromised (see Figure 5.2 entry 6). A simple example of data migration is with

Category	DSpace Entries	EPrints Entries	Total
Data System Migration	3, 4, 5, 6, 8, 9, 12	1, 2, 3, 5, 6, 11, 13, 14, 15, 16, 19, 21, 22	20
System Installation	2	8, 9, 10, 12, 20	6
System Updates and Upgrades	10, 13	4, 7, 23	5
Entity and Prerequisite Software Licences	1, 7	-	2
Security Measures	11	18	2
Dependency Packaging	-	17	1

Table 5.1: Classification of user needs into categories

MySQL's `mysqldump` command, where data stored in some database can be 'dumped' into a file to be restored at a later point. This and other similar commands on DBMSs can play a pivotal role during data migration in systems that utilise a DBMS for data storage.

Moving an entire system to a newer server is a result of a server software/hardware upgrade, where the new system has better performance than the older one. This is a complex process since there may be some mismatches between the digital library system and the new software/hardware which will need to be attended to before the digital library system can again function as expected.

System Updates and Upgrades — The main concern is the frequency with which updates and upgrades should be carried through. Digital library system administrators are concerned with updating and upgrading the systems they maintain, saying that updating or upgrading whenever a newer version is released is not worth the effort since, especially with updates, new versions are released quite often. A discussion in the thread of entry 7 in Figure 5.2 affirms that this is not worth the effort because there is a lot that is bound to go wrong during these processes such as other features of the digital library system failing to function as they ought to (see Figure 5.1 entry 10), which will consequently increase the digital library system's downtime.

5.1.2 Additional User Needs

System Installation — Issues surrounding system installation largely stem from the documentation that has been provided to facilitate the installation process. Sometimes this documentation is too summarised and does not highlight other important procedures for the installation process. Provided documentation usually lists prerequisite software, installation steps and troubleshooting information for certain common problems. If the documentation is relatively acceptable, installation problems can arise from the heterogeneity of consumers' systems, where maybe some system libraries are missing or system configuration files, such as Web server configuration files, are problematic and so on. Sometimes it is possible that the software be installed without any notable complications but gives unexpected performance at runtime as a result of missing/incorrect configuration information.

Entity and Prerequisite Software Licences — The DSpace, EPrints and possibly other digital library systems hold data that may have intellectual property restrictions enforced over it. These restrictions apply to, amongst other things, modifying and distributing these data entities. Some entity licences, such as the GNU Public Licence (GPL), are flexible enough to facilitate the functionality of digital library systems while others are not as accommodating. Dependency software licences also present some or other difficulty to digital library systems' administrators.

Security Measures — System administrators are increasingly getting concerned with security issues that are associated with digital library systems they manage. Although this is not as urgent as other needs, it is still important that there are counter measures in place, should the digital library system be subject to any external threat. Most of these systems employ different levels of access to users which, to a certain extent, ensure that certain features of the system are accessed only by relevant entities.

Dependency Packaging — The final need prevalent in the scanned entries is that of packaging some of the important modules (dependency software) with the digital library system distribution. This is possible if there are no restrictions on bundling of dependency material. The thread of entry 17 in Figure 5.2, entitled `'Perl modules bundled with EPrints'`, discussed this further and it is evident from this discussion that in general, this may prove to be unsuccessful since bundled versions of dependency software may conflict with other versions already installed on a consumer's system. However, this can be useful for consumers in the case where the dependency does not exist on their systems.

There is a correlation between the above needs (issues) with those mentioned in Section 2.2.2. A distinguishing factor with software deployment issues is the type of software system in question which, together with users' experiences, determines the most frequent of these issues. Ultimately, these issues should be taken into account during the design stages of software packaging systems.

5.2 PACKAGING SYSTEM EVALUATION

The main aim of the packaging system evaluation exercise was to comparatively assess the effort in installing individual components making up a bigger system and installing the same system but as packaged components. This assessment was done across different aspects, including understandability and usability associated with both installation methods. The packaging system was evaluated in a controlled exercise session in which each of the participants had to perform a series of exercises and provide feedback on a questionnaire. The structure of this questionnaire as well as the results obtained in the exercises are all discussed in subsequent sections.

5.2.1 About the Questionnaire

The questionnaire that was used for the evaluation exercise was divided into 5 different sections: introduction; background; digital library componentwise installation; digital library package building and installation; and an overall survey. Appendix B shows the questionnaire. A brief explanation of each of the sections is given below:

Introduction — The purpose of the introduction section was to introduce some background information relevant to the exercise, introduce the tools that the participants were to interact with as well as define some terminology that was used throughout the evaluation exercise.

Background Information — The background information section was aimed at gathering participants' background knowledge relevant to the evaluation exercise. This section also collected qualification details from the participants.

Installing a Digital Library Componentwise — This was an exercise in which participants were requested to install a component-based digital library by configuring each of the components individually. To ease this process, one of the three components had already been configured on behalf of the participants.

Building and Installing a Digital Library Package — In this exercise, participants were requested to build and install a component-based digital library package using the packaging system that had been designed and described in Chapter 4.

Survey — Feedback on each of the two exercises described above was gathered directly after each exercise. This section therefore collected overall feedback of the participants' experiences during these two exercises.

There were no guarantees that the participants will be random. Furthermore, it was possible that the participants' feedback on the first of the two exercises could influence their feedback on the second exercise. To counter both these concerns, the two exercises were swopped around in about half of the questionnaires.

The majority of the questions that were asked during feedback stages throughout the questionnaire were in a form of 5-point Likert-type questions. The rest of the questions were open-ended questions where participants were able to respond with open and constructive comments. Before presenting the results and the analysis thereof, the next section introduces Likert-type questions, highlighting some of the problems associated with gathering data using these types of questions, and presents a statistical analysis method that has been used in analysing Likert-type data collected during the evaluation exercise.

5.2.2 Methods for Collecting and Analysing Data

Likert-type Questions

These types of questions are a simple way in which to gather feedback especially if there is a lot of feedback required. For these to be effective, it is best to present them in a balanced form (i.e., with two or an odd number of choices). An option of 'no response' can also be provided over and above specified choices and this can prove to be useful if the meaning of this choice is properly introduced.

Problems that arise with Likert-type questions are due to the interpretation of the centre choice, usually denoted as 'neutral', and that of the 'no response' or 'not applicable' choices if such choices are part of the question being presented. A summary online discussion conducted by the Overseas Chinese Association for Institutional Research (OCAIR) presented a discussion on ”Neutral and Not Applicable on the Likert Scale. [22]” In this discussion, amongst other points, it was said that some users choose the centre choice as a way of implying ”I do not know” while others treat it as ” I do not want to take a stand on either side.” Further to these, some users even leave the entire question unanswered, which subsequently complicates the data analysis process.

The χ^2 Analysis Test

Data collected from Likert-type questions is categorical and as such, it does not make much sense to statistically reason about characteristics like the standard deviation and mean in the context of this data and in general, well known statistical analysis methods, such as Pearson Correlation and *t*-test which are primarily targeted at non-categorical data, do not readily apply [8, 27, 29]. This leaves only a handful of analysis methods of which the χ^2 (chi-square) test is the most prominent. Howell defines the χ^2 test as a statistical test often used for analysing categorical data and further discusses this statistical analysis test in Chapter 19 (pages 371 – 392) of his book [27].

There are two types of χ^2 tests that have been used in the result analysis stages. These are the goodness-of-fit test and the test of independence, and are both types of hypothesis testing. The χ^2 goodness-of-fit test is used to compare recorded results with some theoretical expected distribution as an indicative measure of how these results might vary for an arbitrary group of participants. The test of independence on the other hand is used to determine if two or more variables are independent of one another. The first step in employing any of these tests is to formulate a hypothesis to test, often called a null hypothesis or hypothesis of no difference, which is usually denoted by H_0 and from that, apply the χ^2 formula to determine if the formulated hypothesis does or does not hold. For the two types of tests mentioned above, the applicable formula follows:

$$\chi^2 = \sum \left[\frac{(E-O)^2}{E}\right]$$

In this formula, E is the expected frequency while O is the observed frequency per category. The 5-point Likert-type questions used throughout the exercises each contain 5 categories, hence 5-point. Table 5.2 defines other symbols that are used during the analysis.

To determine if the null hypothesis should or should not be rejected, the significance level (α) or rejection level is considered. This is the criterion that is used to reject the null hypothesis. Traditionally, most research cases use the $\alpha = 0.05$ (or 5%) level of rejection. There is no rule regarding this choice but in general, the lower the significance level, the more the data must converge from the null hypothesis so as to be statistically significant. If the calculated $P\text{-}Value$ is lower than the significance level, then the null hypothesis is rejected. Alternatively, if the calculated χ^2 value is greater than or equal to $\chi^2_\alpha(DF)$ for a particular value of DF and α, then the null hypothesis is rejected.

The χ^2 test has a few disadvantages/constraints associated with it [29]. A few of these are shown in the list below:

- The representative sample must be random.

Symbol	Definition
N	Number of categories
DF	Degrees of freedom computed as $N - 1$
$P\text{-}Value$	Probability that the obtained results occurred by chance
$\chi^2_\alpha(DF)$	The critical value of χ^2 value given DF and significance level α
θ_E	A set of N expected frequencies

Table 5.2: Chi-square analysis symbol table

- The acquired data must be categorical.
- Individual observations must be independent of each other.
- Although not obligatory, the sum of observed frequencies (O) must equal that of expected frequencies (E) especially for smaller sample sizes.
- The size of the representative sample must be adequate. In a 2×2 table (test for independence) χ^2 should not be used if the number of participants falls below 20. In a larger table, no expected value should be less than 1 and not more than 20% of the variables can have expected values of less than 5.

These constraints are addressable with relatively minimum effort save for the last one. It specifies some constraints regarding the allowed values of expected frequencies. Hidden within this constraint is the fact that the more categories the data may fall into, the larger the random sample must be so that this constraint is met. The minimum number of participants to fit five categories (as in the 5-point Likert-type questions in the evaluation exercise) is thus 21 (typically, with θ_E being a permutation of $\{5, 5, 5, 5, 1\}$) such that not more than 20% of the expected values are less than 5 (i.e., only one category can have an expected value of less than 5 and that value must at least be 1).

One of the positives with employing the χ^2 test for evaluation data analysis is that it does not require a normally distributed population, which means that the manner with which the sample was acquired, as will be explained in later sections, is adequate enough to analyse the resultant data using the χ^2 test.

5.2.3 **Participants' Background Analysis**

This section presents the background information that was gathered from the participants. This information is structured into four sections as in the questionnaire and is presented below.

Qualification Details

In the advertisement for participating in the evaluation exercise, it was mentioned that participants needed to be at least in their second year of study and be in either of the computer science or information systems fields. The reason behind this was to ensure that participants focus their attention only to the evaluation exercise rather than on other factors, such as computer affordances etc., which could have unnecessarily prolonged the evaluation exercise. Table 5.3 shows a breakdown of the participants' qualification/study details.

Question	**Responses**	
Total Number of Participants	25	
	Students:	24
	Professionals:	1
Majors/Fields	Computer Science:	24
	Information Systems:	1
Study Levels	B.Sc. 2nd Year:	6
	B.Sc. 3rd Year:	13
	B.Comm. 3rd Year:	1
	B.Sc. (Hons):	2
	M.Sc.:	2
	Ph.D.:	1

Table 5.3: Qualification details of the evaluation exercise participants

Software Installations

The software installations part of the background section uncovered participants' installation patterns on the Microsoft Windows and GNU/Linux operating systems. Participants were asked to indicate if they had installed any software packages on these two platforms. The results are presented in Table 5.4.

Operating System	**Responses**		
	Yes	**No**	*Blank*
Microsoft Windows	25	0	0
GNU/Linux	8	13	4

Table 5.4: Responses on the participants' software installation background

In the case where participants responded with a **Yes**, they were asked to mention a few examples of the software packages they had installed. These user examples included installing *Microsoft Office* and *Winamp* for the Microsoft Windows operating system and *Open Office* and *DSpace* for GNU/Linux-based operating systems.

Software Management Tools

In this section, participants were asked to rate their ability to install, configure, manage and uninstall software packages created with various packaging tools. Table 5.5 shows the participants' responses.

Tools	**Responses**						
	Very Good	**Good**	**Neutral**	**Poor**	**Very Poor**	**No Response**	*Blank*
InstallShield	6	13	1	0	0	3	2
Wise/UnWise	4	6	4	0	0	7	4
RPM	1	8	2	0	0	10	4

Table 5.5: Responses on the participants' familiarity with various packaging tools

This question was one of many that followed to be presented as a 5-point Likert-type question. In addition, it had an option that participants respond with a **No Response** choice, an option which had initially been introduced in the introduction section of the questionnaire and essentially meaning that, in this case, the participant *did not know or had not used the tool in question*. Users were requested to comment on any other packaging tools that they had encountered before. The Portage package management system was one of the tools mentioned.

Software Systems

The final part of the background section gathered users' understandings of types of software systems from user and programmer perspectives. These types of software systems were: component-based; Web-based; and online information management. Table 5.6 shows the participants' responses from a user perspective.

Software Systems	**Responses**					
	Very Good	**Good**	**Neutral**	**Poor**	**Very Poor**	*Blank*
Component-based	7	5	8	4	0	1
Web-based	5	8	9	2	1	0
Information Management	4	8	9	3	1	0

Table 5.6: Participants' user perspectives on certain types of software systems

Participants were further questioned if they are (or have ever been) computer programmers. Those participants that fell in this category had to give their responses pertaining to their understanding of these software systems from a programmer's point of view. Table 5.7 shows responses from 22 programmers.

Software Systems	**Responses**					
	Very Good	**Good**	**Neutral**	**Poor**	**Very Poor**	*Blank*
Component-based	5	10	4	2	0	1
Web-based	5	8	5	1	2	1
Information Management	2	10	6	2	1	1

Table 5.7: Participants' programmer perspectives on certain types of software systems

The final question in this section asked participants if, prior to the evaluation exercise, they knew what digital libraries were. 21 of the 25 participants responded positively (i.e., with a **Yes**) to this question and Table 5.8 shows how these participants were familiar with digital libraries.

Installed	Used	Heard About
3	11	7

Table 5.8: Participants' familiarity with digital libraries

All this gathered background information played an important part in analysing and understanding the results that were obtained from the remaining sections of the questionnaire. The next section provides a detailed analysis of these results.

5.2.4 Results Analysis

Sections 3 through 5 (Sections B.2 through B.4 in Appendix B) of the questionnaire presented exercises and surveys which participants undertook. This section presents an analysis of the restricted-type questions' responses to these sections. The χ^2 goodness-of-fit test as well as the χ^2 test of independence are partly employed in this analysis. Section 5.2.5 consolidates this analysis with the qualitative responses gathered from the open-ended questions as well as with participants' background analysis.

Building and Installing a Digital Library Package

Building and installing a digital library package was one of the exercises that participants had to carry out. This was split into two sub-exercises: building a digital library package and installing the package once it had been built. Figure 5.3 shows this digital library composed of 3 ODL components: `DBUnion` — which harvests (collects) data from one or more Open Archives and keeps it in a local archive; `IRDB` — a search engine which searches specified Open Archives and returns results; and `BRSUI` — a Web-based user interface through which various services, including those provided by `IRDB`, can be accessed and utilised.

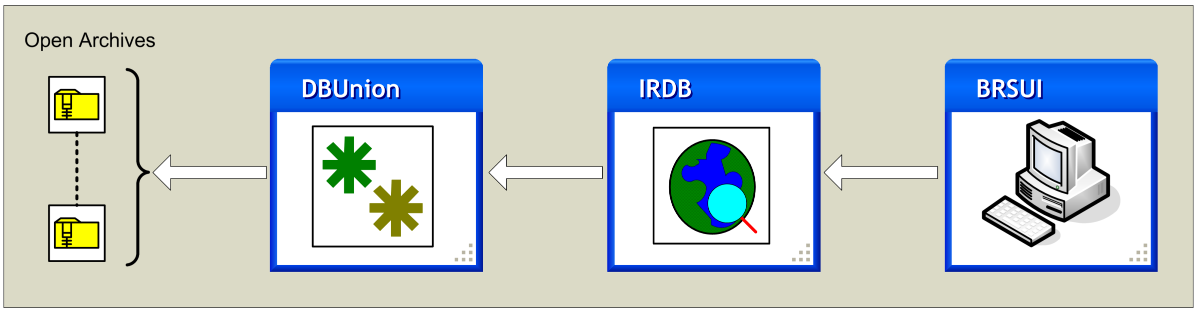


Figure 5.3: The component-based digital library used for the evaluation exercise

All participants managed to build the digital library package without any complications. Feedback questions on the exercise of building a digital library package requested users' understanding of the package building process as well as their perspective with respect to the usability associated with building a component-based digital library package. Table 5.9 shows these results.

Aspect	**Responses**					
	Very Good	**Good**	**Neutral**	**Poor**	**Very Poor**	*Blank*
Understanding	9	11	4	1	0	0
Usability	10	14	1	0	0	0

Table 5.9: Understanding and usability aspects of the package building process

Once the participants had successfully created the digital library package, the second part of the exercise was to then install the package that they had built. The `DBUnion` component was configured to harvest a specific set of data from *Hussein's Photo Album* with baseURL *http://www.husseinsspace.com/cgi-bin/VTOAI/hspics/hspics/oai.pl*. 24 out of the 25 participants successfully installed the digital library. From passive observation, the only participant who could not install the digital library successfully failed to configure the database as per the instructions. Feedback questions on this exercise requested users' understanding of the package installation process as well as their perspective on the usability associated with installing the component-based digital library as a package. Table 5.10 shows the participants' responses.

Aspect	**Responses**					
	Very Good	**Good**	**Neutral**	**Poor**	**Very Poor**	*Blank*
Understanding	7	12	5	1	0	0
Usability	6	16	2	1	0	0

Table 5.10: Understanding and usability aspects of the package installation process

Besides knowing for instance that a majority of the users (20[1] out of the 25 users) at least significantly understood[2] the process of building a component-based digital library package, there are other meaningful conclusions that can be drawn from the results in Tables 5.9 and 5.10. The following section looks at the above results and analyses them using the χ^2 goodness-of-fit test.

[1]This is the sum of the responses from the **Very Good** and **Good** categories in Table 5.9 since the **Very Good** category subsumes the **Good** category.

[2]As defined in the questionnaire, category **Good** meant: *Significantly agree with, significantly acquainted with, or significantly accept the idea, concept or subject presented by the question at hand.* See introduction section of the questionnaire in Appendix B

Statistics on the Understanding and Usability Results

By observing the results presented in Tables 5.9 and 5.10, it is clear that these results are anything but uniformly distributed. However, as a way of introducing the χ^2 goodness-of-fit test, consider the following null hypothesis:

H_{0_a} = *There is an equal chance that participants are* **evenly** *spread across all categories in the context of understanding the process of building a component-based digital library package.*

This means that the expected frequency (E) across all categories is uniform when considering the understanding aspect of building a component-based digital library package. Table 5.11 tabulates this comparison and shows calculations towards a respective χ^2 value.

	Expected (E)	**Observed (O)**	$(E-O)^2$	$\frac{(E-O)^2}{E}$
Very Good	5	9	16	3.2
Good	5	11	36	7.2
Neutral	5	4	1	0.2
Poor	5	1	16	3.2
Very Poor	5	0	25	5

Table 5.11: Chi-Square goodness-of-fit test for package building understanding

From Table 5.11, the following can be calculated:

$$\chi^2 = 18.8$$

In this case, $DF = N - 1 = 5 - 1 = 4$, and $\chi^2_{0.05}(4) = 9.49$. Because the obtained value is 18.8 and is clearly greater than 9.49, null hypothesis H_{0_a} can be rejected. This means that the results are statistically significant and it can be concluded that the participants are **unevenly** distributed across categories of understanding the component-based digital library package building process.

This confirms that there is no uniform distribution of the observations across categories as assumed above. Now consider the following null hypothesis:

H_{0_b} = *There is a normal distribution in the participants' understanding of the the process of building a component-based digital library package with a mean centred on* **Good**.

To theoretically assume expected frequencies to fit this null hypothesis, consider Figure 5.4.

This figure shows a bar chart of the observed frequencies for the aspect of understanding the process of building a component-based digital library package. Superimposed over this bar chart is a normal distribution curve with a mean centred on **Good**. From the curve, the following set of expected frequencies can be estimated:

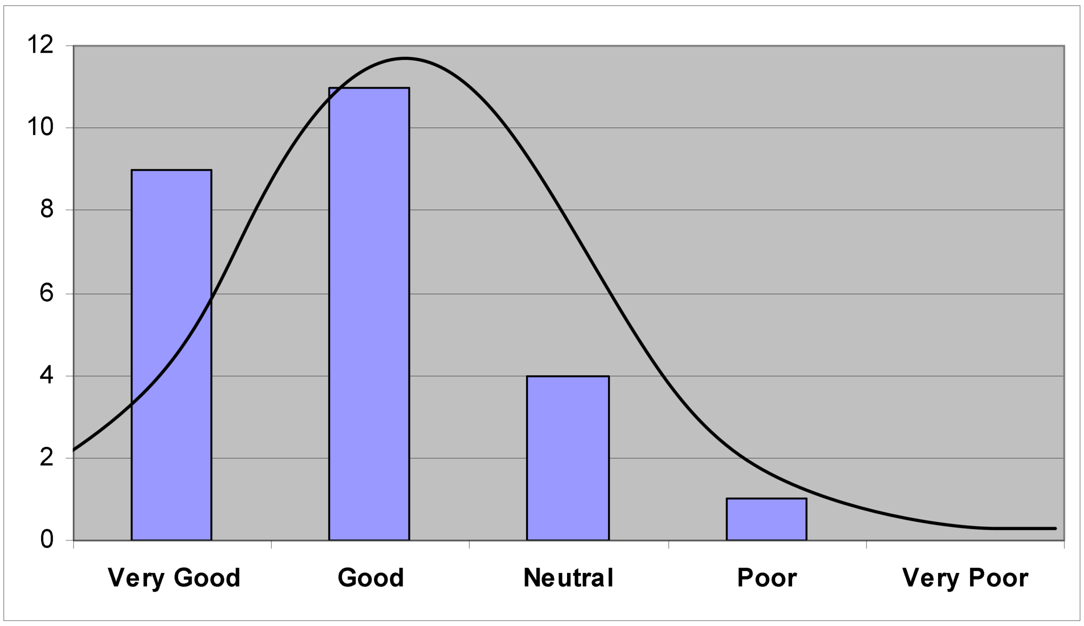


Figure 5.4: A theoretical assumption of expected frequencies for package building understanding

$$\theta_{E_1} = \{6, 12, 5, 1, 1\}$$

Using set θ_{E_1} yields the following results:

$$\chi^2 = 2.78$$

However, the set of expected frequencies, θ_{E_1}, does not meet the requirements to be used in a χ^2 test as explained earlier in Section 5.2.2. The following set of expected frequencies is the closest χ^2-safe expected frequencies set that can be used for this analysis:

$$\theta_{E_2} = \{6, 8, 5, 5, 1\}$$

With this set of expected frequencies, calculating the χ^2 value gives:

$$\chi^2 = 7.025$$

This means that null hypothesis H_{0_b} cannot be rejected since the obtained χ^2 value (in both cases) is smaller than the critical value (9.49) with $DF = 4$ and $\alpha = 0.05$. Therefore, it is true that there is a normal distribution in the participants' understanding of the process of building a digital library package. Furthermore, the participants' mean response is **Good**.

For the understanding of the package installation process, consider the following null hypothesis:

H_{0_c} = *There is a normal distribution in the participants' understanding of the process of installing a component-based digital library package with a mean centred on* **Good**.

To theoretically assume a set of expected frequencies to fit this null hypothesis, consider Figure 5.5.

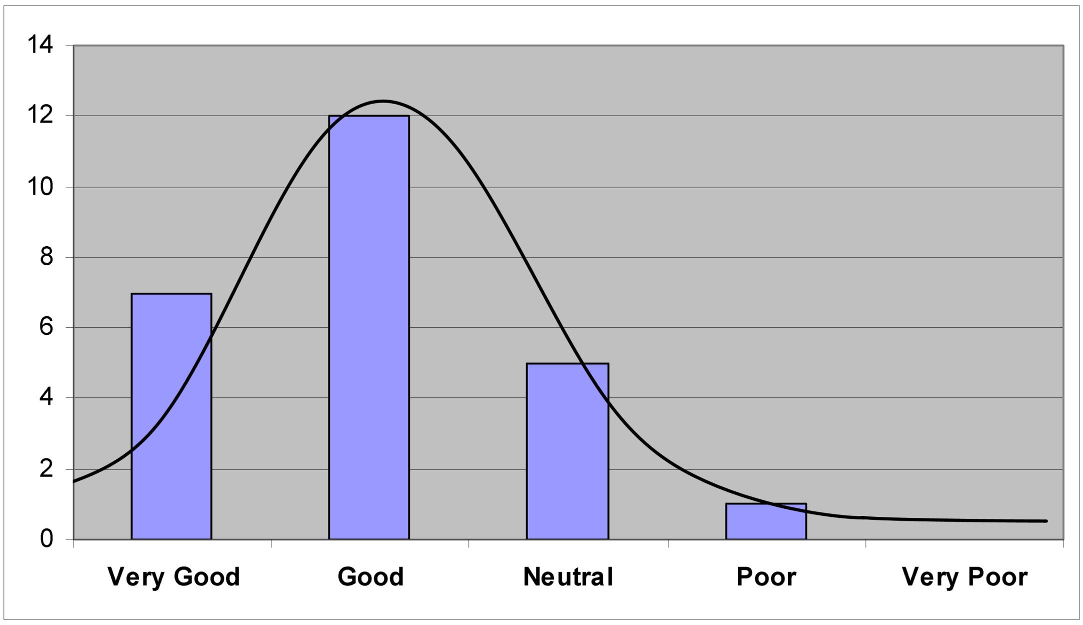


Figure 5.5: A theoretical assumption of expected frequencies for package installation understanding

This figure shows a bar chart of the observed frequencies for the aspect of understanding the process of installing a component-based digital library package. Superimposed over this bar chart is a normal distribution curve with a mean centred on **Good**. From the curve, the following set of expected frequencies can be estimated:

$$\theta_{E_3} = \{6, 12, 5, 1, 1\}$$

Using set θ_{E_3} yields the following results:

$$\chi^2 = 1.17$$

With the same argument as above, the following set of expected frequencies is the closest χ^2-safe expected frequencies set that can be used for this analysis:

$$\theta_{E_4} = \{6, 8, 5, 5, 1\}$$

With this set of expected frequencies, the following χ^2 value can be obtained:

$$\chi^2 = 6.37$$

This means that null hypothesis H_{0_c} cannot be rejected since the obtained χ^2 value (in both cases) is smaller than the critical value (9.49) with $DF = 4$ and $\alpha = 0.05$. Therefore, it is true that there is a normal distribution in the participants' understanding of the process of installing a component-based digital library package. Furthermore, the participants' mean response is **Good**.

In the analysis of the usability results, a similar approach to the above was taken. The following null hypotheses were tested:

H_{0_e} = *There is a normal distribution in the participants' usability ratings of the process of building a component-based digital library package with a mean centred between* **Very Good** *and* **Good**.

H_{0_f} = *There is a normal distribution in the participants' usability ratings of the process of installing a component-based digital library package with a mean centred between* **Very Good** *and* **Good**.

Table 5.12 summarises the results from these null hypotheses. Shown in this table are theoretical estimations (from a normal distribution curve superimposed over each set of results and from creating a closest χ^2-safe set to the one obtained from the normal distribution curve) of the expected frequencies sets as well as their respective χ^2 value calculations.

Aspect	**Process**	**Normal Distribution Estimation**	**χ^2-safe Estimation**
Usability	Building	$\theta_E = \{9, 12, 2, 1, 1\}$ $\chi^2 = 2.95$	$\theta_E = \{6, 8, 5, 5, 1\}$ $\chi^2 = 16.37$
	Installing	$\theta_E = \{6, 14, 3, 1, 1\}$ $\chi^2 = 1.62$	$\theta_E = \{5, 9, 5, 5, 1\}$ $\chi^2 = 11.65$

Table 5.12: Chi-square analysis results of the usability aspect of the package building and installation processes

From Table 5.12, if considering the χ^2 values obtained from the normal distribution curve estimation, null hypotheses H_{0_e} and H_{0_f} can be accepted since the obtained χ^2 values are both smaller the than the critical value of χ^2 with $DF = 4$ and $\alpha = 0.05$ (recall $\chi^2_{0.05}(4) = 9.49$).

If looking at the χ^2-safe estimations however, it is clear that both the calculated χ^2 values are greater than the critical value, which means that null hypotheses H_{0_e} and H_{0_f} can be rejected on these values. This means one of two possibilities:

- There is no normal distribution in the way that the participants rated the usability associated with building and installing a component-based digital library package. This further means that there may be another type of distribution curve, such as a log-normal distribution curve or a positively skewed distribution curve (since at least 88% of the observed frequencies in both usability cases are clustered on the **Good** and **Very Good** categories)[3], which can be superimposed on each set of the observed usability results.

[3]For package building: $14+10 = 24$ out of 25 participants (96%) occupy the **Good** and **Very Good** categories. For package installation: $16 + 6 = 22$ out of 25 participants (88%) occupy the **Good** and **Very Good** categories.

- The size of the sample from which the results were obtained was not suitable for the analysis test that was used for testing hypotheses H_{0_e} and H_{0_f} and as such, incorrect results may have been obtained (due to the limitations of the χ^2 test). However, it is interesting to note that if the significance level is reduced to (from 0.05 to 0.001) both null hypotheses can be accepted since $\chi^2_{0.001}(4) = 18.47$ and the calculated χ^2 values are both smaller than this critical value.

In whichever case, more results per category are required in order to formulate more accurate estimations. This ultimately means that a larger group of participants could be required (assuming that this group of participants will distribute, in the same manner as in the observed results, their responses across all categories).

Testing for Prior Understanding Effects

To establish if any two or more given variables depend on one another, the χ^2 test of independence can be used. The same formula as previously introduced applies for the test of independence but there are slight variations to some variables. The degrees of freedom are now computed as:

$$DF = (R - 1) \times (C - 1)$$

Where R is the number of rows and C is the number of columns of the computation table. The expected frequencies are computed as:

$$E_{ij} = \frac{T_i \times T_j}{T}$$

Where E_{ij} is the expected value for the entry at row i and column j, T_i is the total of all entries of row i, T_j is the total of all entries in column j and T is the total number of participants in the computation table.

Now consider the following null hypothesis:

H_{0_g} = *The participants' understanding of the component-based digital library package installation process is independent of their previous understanding of component-based software systems from a user's perspective.*

Table 5.13 shows the computation table which tests this null hypothesis.

The expected frequencies have been calculated using the formula described above and the values are shown in parentheses. In this table, the final category (**Very Poor**) has been omitted. This is because there are no responses on this category in both sets of observed values. Calculating the χ^2 value from this table gives the following result:

$$\chi^2 = 11.86$$

		Prior Understanding				
		Very Good	Good	Neutral	Poor	*TOTAL*
Installation Understanding	Very Good	3 (2.04)	2 (1.46)	2 (2.33)	0 (1.17)	7
	Good	1 (3.21)	3 (2.29)	3 (3.67)	4 (1.83)	11
	Neutral	2 (1.46)	0 (1.04)	3 (1.67)	0 (0.83)	5
	Poor	1 (0.29)	0 (0.21)	0 (0.33)	0 (0.17)	1
	TOTAL	7	5	8	4	24

Table 5.13: Chi-square test of independence for prior understanding of component-based software systems and component-based digital library package installation process

In this case, $DF = (R-1) \times (C-1) = (4-1) \times (4-1) = 9$, and $\chi^2_{0.05}(9) = 16.92$. Because the obtained value of 11.86 is smaller than this critical value of χ^2, null hypothesis H_{0_g} cannot be rejected. It can thus be concluded that the participants' understanding of the component-based digital library package installation process does not necessarily depend on their prior understanding of component-based software systems from a user's perspective.

Now consider the following null hypothesis:

H_{0_h} = *The participants' understanding of the component-based digital library package installation process is independent of their previous understanding of component-based software systems from a programmers's perspective.*

In testing this null hypothesis, a similar test as detailed above results is a χ^2 value of 11.83. Since this value is smaller than the critical value of χ^2 with $DF = 9$ and $\alpha = 0.05$, this means that null hypothesis H_{0_h} can be accepted, implying that the participants' understanding of the process of installing a component-based digital library package does not necessarily depend on their prior understanding of component-based digital library systems from a programmer's perspective.

Further tests of independence were conducted to test for the independence of the package building process. The following null hypotheses were tested and the obtained results are summarised in Table 5.14.

H_{0_i}	=	*The participants' understanding of the component-based digital library package building process is independent of their previous understanding of component-based software systems from a user's perspective.*
H_{0_j}	=	*The participants' understanding of the component-based digital library package building process is independent of their previous understanding of component-based software systems from a programmer's perspective.*

Aspect	**Perspective**	χ^2 **Value**
Building Process Understanding	User	6.79
	Programmer	6.14

Table 5.14: Chi-square test of independence analysis results of the understanding aspect of the package building processes

Both χ^2 values shown in Table 5.14 are less than the critical value of χ^2 with $DF = 9$ and $\alpha = 0.05$. Therefore, null hypotheses H_{0_i} and H_{0_j} can be accepted. This means that the participants' understanding of the process of creating a component-based digital library package does not necessarily depend on their prior understanding of component-based software systems as either a user or a programmer.

To summarise the results from these tests, it can be concluded that, since the participants' prior understanding (from user and programmer perspectives) of component-based software systems is independent of the participants' understanding of the component-based digital library package building and installation processes, the packaging system as a whole is simple enough to be understood and utilised satisfactorily without any prior knowledge or understanding of the types of software systems that can be modeled with the packaging system.

Componentwise Digital Library Installation

This was another of the exercises that the participants had to carry out in which they had to install a component-based digital library one component at a time. This was the same digital library as depicted in Figure 5.3. To reduce the amount of effort and time required from the users, the `DBUnion` component had already been configured on their behalf. 24 out of the 25 participants managed to successfully install the digital library componentwise. The only participant who could not install the digital library successfully failed to configure the database as per the provided instructions. However, this was a different participant to the one who could not install the component-based digital library package. In a similar fashion as with the package installation exercise, feedback questions on this exercise requested participants' understanding of the componentwise installation process as well as their perspective with respect to the usability associated with installing the digital library componentwise. Table 5.15 shows the users' responses to the feedback questions.

Section 5.2.5 comments further on the results presented in Table 5.15.

Aspect	**Responses**					
	Very Good	**Good**	**Neutral**	**Poor**	**Very Poor**	*Blank*
Understanding	3	16	3	3	0	0
Usability	4	6	11	3	0	1

Table 5.15: Understanding and usability aspects of a componentwise installation

Survey

The survey section of the questionnaire gathered overall feedback on all the exercises that users performed. This section was comprised of four questions, three of which were open-ended. The only restricted-type question gathered the users' views on the aesthetic design of the packaging system tools (i.e., the package builder and installer tools). Table 5.16 shows the responses to this question.

Very Good	**Good**	**Neutral**	**Poor**	**Very Poor**
6	17	2	0	0

Table 5.16: Responses on the aesthetic design of the packaging system

These responses show that a majority of the participants rate the packaging system as either **Good** or **Very Good** in terms of its aesthetic design. The other three questions are discussed in Section 5.2.5 below.

5.2.5 **Discussion**

This section finalises the packaging system evaluation section by consolidating the responses from Sections 2 through 5 (Sections B.1 through B.4 in Appendix B) of the questionnaire. The analysis that has already been presented on Likert-type questions, the open-ended-type questions as well as the users' background details are taken into account in this discussion which is presented in the following subsections.

Understandability

On an overall basis, the participants showed a significant level of understanding of all the concepts and methods that were presented in each of the exercises that they had to carry out. This can partly be attributed to their relative backgrounds in fields of study since these concepts and methods are not totally foreign within these fields and the participants may have, directly or indirectly, encountered similar ideas. The statistical analysis tests affirmed the significance of the results which further shows that any random sample of users from a similar population will have an either **Good** or **Very Good** understanding of all the ideas presented in the evaluation exercise.

Usability

It was not an unexpected outcome that users rated the componentwise installation process as less usable than the package installation process (considering that a majority of the participants (22) were concentrated on the **Good** and **Very Good** categories for the digital library package installation process (see Table 5.10), while a majority of the participants (17) were concentrated on the **Good** and **Neutral** categories for the digital library componentwise installation process (see Table 5.15)). The main visible difference with these processes was that one was command-line-based while the other was presented as a graphical user interface. Table 5.4 shows that all users have, at some point, installed software on the Microsoft Windows operating system while less than half of the participants previously installed software on the GNU/Linux operating system. This said, it was not surprising that users preferred a graphically presented installation process over the command-line-based. Software users in general tend to avoid cases where they have to interact with systems at command-line level and this was evident from the users' responses on the usability aspect. The test for independence also showed that there is a possible link in the manner that users view these two processes.

Overall User Preference

From the survey section of the questionnaire, it can be inferred that an overwhelming percentage (92% – 23 out of 25) of participants preferred the package installation process over the componentwise installation. Below is a summary of some of the comments that the participants provided to motivate their selection:

- "The package installation process is a lot faster and it is less likely that mistakes are made during this process whereas with a componentwise installation process, one needs to know a lot more about the components."
- "The interface presented by package installation process is a lot easier to use and is more familiar while the componentwise approach presents a command-line-based installation process which is more error-prone and confusing."
- "The package had a better looking interface and was easier to follow while the componentwise approach did not have a friendly interface and required lots of unnecessary user input."

Other user comments were some or other variation of these. As for the feedback from the remaining participants, 1 of them preferred the componentwise installation process while the other was indifferent. Their motivating points were as follows:

- "The componentwise approach is easier to work with and provides for a flexible configuration process whereas with the package approach, most processes are abstracted."
- "For power users, componentwise is the best option and for novice users, the package is a lot simpler."

Common with these two participants was the fact that they were both in their final year of study (3^{rd} year), which means that they had a higher level of experience to reason about these two processes adequately.

In general, all users believed the packaging tools to be fairly similar to orthodox installers that they have, at some point, come across. Most users offered suggestions as to how they could go about improving on these tools. Some of these suggestions are incorporated in Chapter 7.

5.3 SUMMARY OF KEY POINTS

Digital library users' needs were uncovered by periodically scanning the mailing lists of the DSpace and EPrints digital library software systems and ranked according to popularity. Some of these needs were addressed in the design of the packaging system that was described in Chapter 4. This packaging system was subsequently evaluated across different criteria, including understandability and usability of the packaging system tools. The evaluation analysis process employed the χ^2 goodness-of-fit and independence tests to statistically reason about the results that were obtained since most of the collected data was categorical. The overall result analysis process shows that there is an exceptional level of acceptance of the packaging systems amongst most participants.

CHAPTER 6

CONCLUSION

This dissertation presented work towards simplifying the process of software deployment mainly concentrating on packaging software systems for a better installation process. Software systems now come in various flavours thanks to advances in relevant fields, mainly software engineering, responsible for software delivery. With current component-oriented developments and Web-based solutions, it has become increasingly important that current software deployment procedures be substituted by simpler, better and faster methods that deal with software deployment needs in the context of all the latest advances.

Although the reference point to this research was component-based digital library systems, the findings of this research can apply to Web-based component-based software systems and to component-based software systems at large. The remainder of this chapter presents some concluding remarks with respect to the initial goals of this research.

Building Heterogeneously for Heterogeneous Platforms — It is possible to build a component-based digital library package composed of different components. The designed packaging system boasts a Java^{TM} implementation and a strong utilisation of XML for data presentation and interoperability, all of which promote heterogeneity.

Effort in Installation Processes — Installing a digital library componentwise is more effort than installing the same component-based digital library using the developed packaging system. The evaluation study uncovered that in the context of understandability and usability, a majority of the employed participants preferred the functionalities presented by the packaging system over the componentwise installation. These findings can be summarised as meaning that the packaging system simplifies the process of installing component-based digital libraries.

Dealing with Dependencies — Software dependencies are a 'necessary evil' when it comes to software development — software is hardly usable if its dependency requirements are not met but attending to these requirements can prove to be a rather daunting task. The packaging system presented a manner with which to deal with managing dependencies in its context. For this approach to be generalised across software systems, more factors may need to be taken into consideration implying that more work may be required. In an ideal situation, a dependency specification standard which software developers can adhere to can dismiss all the complications currently presented by software dependencies.

User's Post-deployment Needs — In general, software systems have similar user needs. However, the priority of these needs is informed by the type of software system in question. With digital library communities, it is more urgent to be able to migrate data or an entire digital system or both while the need for a tightly secure digital library system is somewhat not as urgent. A common set of user needs in general covers: simple installation procedures; addressing of heterogeneous operational platforms; adequate upgrade/update methodologies; interoperability at various levels; software system security; and software uninstallation.

An outcome of this research was identifying requirements for a generic component packaging framework. There are four aspects that components or a component packaging framework must adhere to. These are: components must be configurable automatically; each component must possess a formal description of its dependency software; there must be formal descriptions that describe individual components as well as systems that are composed of components; and there must be a way whereby installation questions are formally encoded such that components are able to correctly receive configuration information. These four aspects are the least that are expected of a component packaging framework.

A consolidated outcome of this research was to show that flexible packaging methodologies for Web-based component-based applications in general facilitate the management of these applications. It is hoped that with better management in place, it can be more likely that end users will adopt Web tools based on components and that developers will create more Web-based software components to be used in various contexts of composing software systems. Ultimately, this will increase the quantity and quality of applications and services delivered over the Web.

CHAPTER 7

FUTURE WORK

This chapter presents alternative approaches and outstanding issues, an addressing of which would speculatively have improved the results of this research.

A Web-based Packaging System

As it stands, the packaging system is a pair of desktop applications which is controlled by a user and adheres to the environment in which this pair of tools exist. The packaging system could have been implemented as a hybrid of a Web-based solution and a desktop application. Although the main aim of this design strategy was that users gain full control of the package building and package installing processes, the package building process is more of an administrative process than the package installation process. If the packaging process could be shifted to be accessible over the Web, the following would be attained:

- The CCL specification could still be an input to the packaging process, but a Web interface could be easily suited for building a digital library package from bottom up (i.e., without a CCL specification file as input). This can be done in a similar manner as that presented by von Thile *et al.* on Web-based IDEs [51].
- With a Web-based packaging process, all the components (together with all their available versions and, if possible, all the necessary dependencies) could be located in a central location. This can inform the manner in which users can issue update/upgrade requests from their installed packages.

The output of this Web-based packaging process could still be a package as in the current process. One of the disadvantages with this solution is that it requires the Internet to function adequately. Furthermore, constant maintenance of this central location may be required to ensure that all the available resources are up to date.

Specifying Dependencies

The main challenge during the design and implementation process was dealing with dependencies. Admittedly, dependency related matters are often too complex to attend to since they too are dependent on other factors. A serious question that can be raised in relation to the manner in which dependencies were handled in the context of this research is: between the package composer and component creator who should be liable for building the dependency checking procedures and scripts as well as for compiling a dependency specification for each component? Ideally, this should be done by a person well acquainted with the component in order to ensure correctness. In addition, there should be a common and well defined set of specifications that all the dependency checking entities and dependency specifications adhere to. This set of specifications should address issues such as dependencies going by multiple names amongst other things. In this manner, this set of specifications can facilitate the way in which package management software functions while improving the quality of component-oriented approaches such as CBSE amongst other things.

Packaging and Installing the DSpace Digital Library

By the nature of the design of the packaging system, it may be possible that other types of software can be packaged and installed using the packaging system provided that such a software system can be described using the CCL specification. In the case of the DSpace digital library system software, a CCL specification describing this system would consist of a single component. This component would be represented by the DSpace main program, which could subsequently describe as its dependencies, all the prerequisite software packages.

The following processes would need to be clarified before the DSpace system can be packaged for a successful installation process: encoding configuration questions; relaying configuration information from the packaging system interface at installation time to the DSpace system; and automatically (silently) configuring and installing the DSpace system.

Addressing User Deployment Needs

From this research, it was evident that in certain types of software systems, certain user deployment needs are more urgent than in other types of software systems. This makes it more challenging to formulate a generalised approach for addressing users' deployment needs. However, the process of packaging software was discovered to be the entry point to the process of software deployment and as such, it is at this level that most (if not all) user needs can be attended to. This means that current packaging solutions need to flex and generalise their functionalities such that most software systems that employ them for software distribution are able to address as much of their associated user needs as possible.

Evaluation Methodology

The evaluation exercise was composed of three different exercises: building a digital library package; installing a digital library package; and installing a digital library componentwise. The most important of these three processes were the last two exercises. A better approach to the evaluation process would be to separate the whole process into two parts, part I concerned with evaluating only the package building process and part II concerned with evaluating the two installation processes, with both parts having two separate and different groups of participants. A bottleneck to this approach would be the availability of participants to participate in this two-part evaluation process. However, the participants would be focusing on either the building or the installation process and not both (as was done in the evaluation exercise whereby the participants' responses may have been influenced by the other process).

Another improvement to the evaluation process could be in the gathering of results. The evaluation questionnaire could be structured in such a way that the data collectible from the participants accommodates more than one statistical analysis method. This would provide alternative angles from which to view and analyse results.

BIBLIOGRAPHY

[1] Altiris. Wise Solutions, Inc. Website. Available: http://www.wise.com/ Last Accessed: October, 2005.

[2] Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition. Technical Report CMU/SEI-2000-TR-008 ESC-TR-2000-007, Carnegie Mellon Software Engineering Institute, Pittsburgh, PA 15213-3890, May 2000.

[3] Edward C. Bailey. *Maximum RPM*. Red Hat, Inc., 2000. Available: http://www.rpm.org/max-rpm/.

[4] Caphyon. Advanced Installer. Website. Available: http://www.advancedinstaller.com/ Last Accessed: October, 2005.

[5] Antonio Carzaniga, Alfonso Fuggetta, Richard S. Hall, Dennis Heimbigner, Andre van der Hoek, and Alexander Wolf. A Characterisation Framework for Software Deployment Technologies. Technical Report CU-CS-857-98, University of Colorado, Colorado, USA, April 1998.

[6] Donatella Castelli and Pasquale Pagano. A System for Building Expandable Digital Libraries. In *Proceedings of the Third ACM/IEEE-CS Joint Conference on Digital Libraries*, pages 335–345, Houston, Texas, USA, May 2003.

[7] Kamalsinh F. Chavda. Anatomy of a Web Service. *Journal of Computing Sciences in Colleges*, 19(3):124–134, January 2004.

[8] Dennis L. Clason and Thomas J. Dormody. Analyzing Data Measured by Individual Likert-Type Items. *Journal of Agricultural Education*, 35(4):31–35, 1994.

[9] Thierry Coupaye and Jacky Estublier. Foundations of Enterprise Software Deployment. In *Proceedings of the Conference on Software Maintenance and Reengineering*, pages 65–73, Zurich, Switzerland, February 2000. IEEE Computer Society.

[10] Andrew Cowie. Gentoo for all the Unusual Reasons. *Linux Journal*, 2005(130):8, February 2005.

[11] Ivica Crnkovic. Component-based Software Engineering - New Challenges in Software Development. *Software Focus*, 2(4):127–133, November 2001.

[12] Ivica Crnkovic, Heinz W. Schmidt, Judith Stafford, and Kurt Wallnau. Automated Component-based Software Engineering. *The Journal of Systems and Software*, 74(1):1–3, January 2005.

[13] Joseph Dadzie. Understanding Software Patching. *ACM Queue*, 3(2):24–30, March 2005.

[14] Merijn de Jonge. Bundling Free Software Components. Extended Abstract, Available: http://www.cwi.nl/m̃dejonge/papers/BundlingFreeSoftwareComponents.pdf, November 2000.

[15] Merijn de Jonge. Source Tree Composition. In *ICSR-7: Proceedings of the 7th International Conference on Software Reuse*, pages 17–32, London, UK, 2002. Springer-Verlag.

[16] Edsger W. Dijkstra. How do we Tell Truths that Might Hurt? *ACM SIGPLAN Notices*, 17(5):13–15, May 1982.

[17] Katherine J. Don, David Bainbridge, and Ian H. Witten. The Design of Greenstone 3: An Agent Based Dynamic Digital Library. Unpublished, Available: http://www.sadl.uleth.ca/greenstone3/gs3design.pdf, August 2003.

[18] Wolfgang Emmerich. Distributed Component Technologies and their Software Engineering Impications. In *International Conference on Software Engineering*, pages 537–546, Orlando, Florida, USA, May 2002. Association for Computing Machinery.

[19] Linda Eyambe. A Digital Library Component Assembly Environment. Master's thesis, University of Cape Town, Cape Town, Western Cape, RSA, January 2005.

[20] Linda Eyambe and Hussein Suleman. A Digital Library Component Assembly Environment. In *SAICSIT '04: Proceedings of the 2004 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries*, pages 15–22, Stellenbosch, Western Cape, South Africa, 2004. South African Institute for Computer Scientists and Information Technologists.

[21] Richard Foard, Tom Daly, Art Jahnke, and Sonny Willams. Deliverance. *CIO Web Business Magazine*, July 1999. Available: http://www.cio.com/archive/webbusiness/070199_ups.html Last Accessed: November, 2005.

[22] Overseas Chinese Association for Institutional Research (OCAIR). Summaries of On-line Discussion - Statistical Analysis: Neutral and Not Applicable on the Likert Scale. Website, April 2003. Available: http://www.ocair.org/files/KnowledgeBase/Statistics/TheOptionOfNeutral.asp Last Accesed: December, 2005.

[23] Marcel Gagne. Cooking with Linux: Rapid Program-Delivery Morsels, RPM. *Linux Journal*, 2000(73es):25, May 2000.

[24] Henry M. Gladney, Nicholas J. Belkin, Zahid Ahmed, Edward A. Fox, Ron Ashany, and Maria Zemankova. Digital Library: Gross Structure and Requirement (Report from a Workshop). In *Workshop on On-line Access to Digital Libraries*, May 1994.

[25] Ernest Goss. The Internet's Contribution to U.S. Productivity Growth: Putting Some Rigor Into the Estimates - Statistical Data Included. *The National Association of Business Economists*, October 2001. Available: http://www.findarticles.com/p/articles/mi_m1094/is_4_36/ai_80924110 Last Accessed: December, 2005.

[26] Stevan Harnad. Free at Last: The Future of Peer-Reviewed Journals. *D-Lib Magazine*, 5(12), December 1999.

[27] David C. Howell. *Fundamental Statistics for the Behavioral Sciences*. Duxbury Press, 4th edition, 1999.

[28] Ian Jackson, David A. Morris, Christian Schwarz, Manoj Srivastava, and Julian Gilbey. *Debian Packaging Manual*. The Debian Policy Group, 3.2.1.0 edition, August 2000.

[29] James P. Key. Research Design in Occupational Education: Module S7 - Chi-square. Website, 1997. Available: http://www.okstate.edu/ag/agedcm4h/academic/aged5980a/5980/newpage28.htm Last Accessed: December, 2005.

[30] Carl Lagoze and Herbert Van de Sompel. The Open Archives Initiative: Building a Low-barrier Interoperability Framework. In *ACM/IEEE Joint Conference on Digital Libraries*, pages 54–62, June 2001.

[31] MIT Libraries and Hewlett-Packard Company. DSpace Federation. Website. Available: http://www.dspace.org/ Last Accessed: October, 2005.

[32] Macrovision Europe Ltd. InstallShield: The Industry's Leading Installation-Authoring Solution. Website. Available: http://www.installshield.com/products/installshield/ Last Accessed: October, 2005.

[33] Elena Macevičiūtė and T.D. Wilson. The Development of the Information Management Research Area. *Information Research*, 7(3), April 2002. Available: http://InformationR.net/ir/7-3/paper133.html Last Accessed: December, 2005.

[34] Kurt Maly, Mohammad Zubair, and Xiaoming Liu. Kepler - A Digital Library For Building Communities. Website. Available: http://kepler.cs.odu.edu/ Last Accessed: October, 2005.

[35] Kurt Maly, Mohammad Zubair, and Xiaoming Liu. Kepler - An OAI Data/Service Provider for the Individual. *D-Lib Magazine*, 7(4), April 2001.

[36] Michael J. Mandel and Irene M. Kunii. The Internet Economy: the World's Next Growth Engine. *Bussiness Week Online*, October 1999. Available: http://www.businessweek.com/1999/99_40/b3649004.htm Last Accessed: December, 2005.

[37] David Moore, Stephen Emslie, and Hussein Suleman. BLOX: Visual Digital Library Building. Technical Report CS03-20-00, Department of Computer Science, University of Cape Town., October 2003. Available: http://pubs.cs.uct.ac.za/archive/00000075/.

[38] Leonardo Murta, Hamilton Oliveira, Cristine Dantas, Luiz Gustavo Lopez, and Claudia Werner. Towards Component-based Software Maintenance via Software Configuration Management Techniques. In *XVIII Brazilian Symposium on Software Engineering , Workshop on Modern Software Maintenance*, Brazil, October 2004.

[39] University of Southampton. EPrints.org. Website. Available: http://www.eprints.org/ Last Accessed: October, 2005.

[40] Ben Okopnik. Installing Software from Source -or- What Do I Do with this file.tar.gz Thing? *Linux Gazette*, (74), January 2002. Available: http://linuxgazette.net/issue74/okopnik.html Last Accessed: November, 2005.

[41] The Joint Task Force on Computing Curricula: IEEE Computer Society and Association for Computing Machinery. *Software Engineering 2004 - Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering - A Volume of the Computing Curricula Series*. The National Science Foundation, Association for Computing Machinery and IEEE Computer Society, August 2004.

[42] Proggle. Installer/GD - A Better Windows Installer. Website. Available: http://www.proggle.com/installer/ Last Accessed: October, 2005.

[43] New Zealand Digital Library Project, UNESCO, and Human Info. Greenstone Digital Library Software. Website. Available: http://www.greenstone.org/ Last Accessed: October, 2005.

[44] Art Rhyno. *Using Open Source Systems for Digital Libraries*. Libraries Unlimited, 1st edition, 2004.

[45] MacKenzie Smith, Mary Burton, Mick Bass, Margret Branschofsky, Greg McClellan, Dave Stuve, Robert Tansley, and Julie Harford Walker. DSpace - An Open Source Dynamic Digital Repository. *D-Lib Magazine*, 9(1), January 2003.

[46] Carl Staelin. mkpkg A Software Packaging Tool. In *Proceedings of the Twelfth Systems Administration Conference (LISA 98)*, Boston, Massachusetts, USA, December 1998.

[47] Hussein Suleman. *Open Digital Libraries*. PhD thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, USA, November 2002.

[48] Hussein Suleman, Kevin Feng, Siyabonga Mhlongo, and Muammar Omar. Flexing Digital Library Systems. In *The 8th International Conference on Asian Digital Libraries*, Bangkok, Thailand, December 2005.

[49] Hussein Suleman and Edward A. Fox. A Framework for Building Open Digital Libraries. *D-Lib Magazine*, 7(12), December 2001.

[50] Old Dominion University. Networked Computer Science Technical Reference Library. Website. Available: http://www.ncstrl.org/ Last Accessed: October, 2005.

[51] Alexander Hilliger von Thile, Ingo Melzer, and Hans-Peter Steiert. Managers Don't Code: Making Web Services Middleware Applicable for End-Users. In Liang-Jie (LJ) Zhang and Mario Jeckle, editor, *Web Services: European Conference, ECOWS 2004 Proceedings*, volume 3250, Erfurt, Germany, September 2004. Springer-Verlag GmbH.

[52] Ju An Wang. Towards Component-based Software Engineering. In *Proceedings of the Eigth Annual Consortium on Computing in Small Colleges Rocky Mountain Conference*, pages 177–189, November 2000.

APPENDIX A

SPECIFICATIONS

A.1 COMPONENT CONNECTION LANGUAGE

```
<?xml version="1.0" encoding="UTF-8"?>
<CCL xmlns:blox="http://nala.cs.uct.ac.za/blox">
    <instance>
        <instanceDescription>
            <name>UI</name>
            <description>
                <blox:ui>
                    <uiname>ODL User Interface</uiname>
                    <adminEmail>smhlongo@cs.uct.ac.za</adminEmail>
                    <searchbaseurl>
                        <!--URL_TO_CGI_LOCATION-->/ODL-IRDB-1.3/IRDB/evaluation/search.pl
                    </searchbaseurl>
                    <titlebar>Digital Library User Interface</titlebar>
                    <bodytitle>Experimental Digital Library User Interface</bodytitle>
                    <message>
                        This is an experimental user interface that allows you to search,
                        browse and rate resources depending on the services and components
                        that have been installed and configured.
                    </message>
                    <footer>If there are queries, send them to the administrator.</footer>
                    <webservices>
                        <name/>
                    </webservices>
                </blox:ui>
            </description>
        </instanceDescription>
    </instance>
    <instance>
        <instanceDescription>
            <name>Search</name>
            <description>
                <blox:irdb>
                    <repositoryName>ODL Search Engine</repositoryName>
                    <adminEmail>smhlongo@cs.uct.ac.za</adminEmail>
                    <database>DBI:mysql:test</database>
                    <dbusername>root</dbusername>
                    <dbpassword>root</dbpassword>
                    <table>search</table>
                    <archive>
                        <identifier>HUSPICS</identifier>
                        <url>
                            <!--URL_TO_CGI_LOCATION-->/ODL-DBUnion-1.2/DBUnion/evaluation/union.pl
                        </url>
                        <metadataPrefix>oai_dc</metadataPrefix>
                        <interval>86400</interval>
                        <interrequestgap>10</interrequestgap>
                        <overlap>86401</overlap>
                        <granularity>second</granularity>
                    </archive>
                </blox:irdb>
            </description>
        </instanceDescription>
    </instance>
    <instance>
        <instanceDescription>
            <name>Union</name>
            <description>
                <blox:dbunion>
                    <repositoryName>ODL Union Engine</repositoryName>
                    <adminEmail>smhlongo@cs.uct.ac.za</adminEmail>
                    <database>DBI:mysql:test</database>
                    <dbusername>root</dbusername>
                    <dbpassword>root</dbpassword>
                    <table>union</table>
                    <recordlimit>200</recordlimit>
                    <archive>
                        <identifier>HUSPICS</identifier>
                        <url>
                            http://www.husseinsspace.com/cgi-bin/VTOAI/hspics/hspics/oai.pl
                        </url>
                        <metadataPrefix>oai_dc</metadataPrefix>
                        <interval>86400</interval>
                        <interrequestgap>10</interrequestgap>
                        <set>200301ctcs</set>
                        <overlap>86400</overlap>
                        <granularity>day</granularity>
                    </archive>
                </blox:dbunion>
            </description>
        </instanceDescription>
    </instance>
    <connection>
        <from>UI</from>
        <to>Search</to>
    </connection>
    <connection>
        <from>Search</from>
        <to>Union</to>
    </connection>
    <questions>
        <question>
            <description>
                The administrator’s e-mail is that to which all correspondence about the
                digital library will be directed.
            </description>
            <text>Please Input the Administrator’s e-mail Address</text>
            <answer> </answer>
            <default>administrator@domain.suffix</default>
            <locations>
                <location>/CCL/instance[0]/.../description/ui/adminEmail</location>
                <location>/CCL/instance[1]/.../description/dbbrowse/adminEmail</location>
                <location>/CCL/instance[2]/.../description/irdb/adminEmail</location>
                <location>/CCL/instance[3]/.../description/dbunion/adminEmail</location>
            </locations>
        </question>
        <question>
            <description>
                Some of the components need to use a database to store their internal data.
                This database should already be created. If not, please create it before
                continuing.
            </description>
            <text>Please Input the Database</text>
            <answer> </answer>
            <default>etds_db</default>
```

```
            <locations>
                <location>/CCL/instance[1]/.../description/dbbrowse/database</location>
                <location>/CCL/instance[2]/.../description/irdb/database</location>
                <location>/CCL/instance[3]/.../description/dbunion/database</location>
            </locations>
        </question>
        <question>
            <description>
                This is the username that will be used to connect to the database that has
                been specified.
            </description>
            <text>Please Input the Database Username</text>
            <answer> </answer>
            <default>username</default>
            <locations>
                <location>/CCL/instance[1]/.../description/dbbrowse/dbusername</location>
                <location>/CCL/instance[2]/.../description/irdb/dbusername</location>
                <location>/CCL/instance[3]/.../description/dbunion/dbusername</location>
            </locations>
        </question>
        <question>
            <description>
                This is the password that will be used in conjunction with the username that
                has been specified to connect to the database.
            </description>
            <text>Please Input the Database Password</text>
            <answer> </answer>
            <default>password</default>
            <locations>
                <location>/CCL/instance[1]/.../description/dbbrowse/dbpassword</location>
                <location>/CCL/instance[2]/.../description/irdb/dbpassword</location>
                <location>/CCL/instance[3]/.../description/dbunion/dbpassword</location>
            </locations>
        </question>
        <question>
            <description>
                This is the database table prefix that will be used within the database in
                order to differentiate data that belongs to the Union [dbunion] component.
            </description>
            <text>Please Input the Database Table Prefix for the Union Component</text>
            <answer> </answer>
            <default>union_archive</default>
            <locations>
                <location>/CCL/instance[3]/instanceDescription/description/dbunion/table</location>
            </locations>
        </question>
    </questions>
</CCL>
```

A.2 Component Name Mappings

```
<?xml version="1.0" encoding="UTF-8"?>
<components >
    <component>
        <name>irdb</name>
        <description>An ODL resource searching component.</description>
        <versions>
            <version>IRDB-1.0.tar.gz</version>
            <version>IRDB-1.1.tar.gz</version>
            <version>IRDB-1.2.tar.gz</version>
            <version>IRDB-1.3.tar.gz</version>
        </versions>
    </component>
    <component>
        <name>dbbrowse</name>
        <description>An ODL resource browsing component.</description>
        <versions>
            <version>DBBrowse-1.0.tar.gz</version>
            <version>DBBrowse-1.1.tar.gz</version>
            <version>DBBrowse-1.2.tar.gz</version>
        </versions>
    </component>
    <component>
        <name>dbrate</name>
        <description>An ODL resource rating component.</description>
        <versions>
            <version>DBRate-1.0.tar.gz</version>
        </versions>
    </component>
    <component>
        <name>ui</name>
        <description>
            A user interface that is designed to interact with, and deliver services offered
            by ODL components.
        </description>
        <versions>
            <version>BRSUI-1.0.tar.gz</version>
        </versions>
    </component>
    <component>
        <name>dbunion</name>
        <description>
            An ODL union archive component. Harvests several open archives and roles as a
            service provider for other ODL components.
        </description>
        <versions>
            <version>DBUnion-1.0.tar.gz</version>
            <version>DBUnion-1.1.tar.gz</version>
            <version>DBUnion-1.2.tar.gz</version>
        </versions>
    </component>
</components>
```

A.3 DEPENDENCIES

```
<?xml version="1.0" encoding="UTF-8"?>
<dependencies>
    <dependency>
        <name>mysql</name>
        <description>
            An Open Source Software relational Database Management System (DBMS)
            which uses a subset of ANSI SQL (Structured Query Language). This
            DBMS is be used by various components for data storage and retrieval
            purposes.
        </description>
        <version>
            <atleast>4.1.1</atleast>
            <atmost>5.0.15</atmost>
        </version>
        <platforms>
            <platform>
                <name>GNU/Linux</name>
                <version>
                    <atleast>2.2.13</atleast>
                    <atmost>2.6.15</atmost>
                </version>
                <check>
                    <command>java checkmysql</command>
                </check>
                <source>
                    <url>http://www.mysql.com/downloads/mysql-5.0.15.tar.gz/</url>
                </source>
                <install>
                    <command>gzip -cd | tar -xf - mysql-5.0.15.tar.gz</command>
                    <command>mysql-5.0.15/install</command>
                </install>
            </platform>
            <platform>
                <name>Windows</name>
                <version>
                    <atleast>98</atleast>
                    <atmost>XP</atmost>
                </version>
                <check>
                    <command>java checkmysql</command>
                </check>
                <source>
                    <url>http://www.mysql.com/downloads/mysql-5.0.15.exe/</url>
                </source>
                <install>
                    <command>mysql-5.0.15.exe</command>
                </install>
            </platform>
        </platforms>
        <rights/>
        <dependencies/>
    </dependency>
</dependencies>
```

A.4 Installation Script

```
<?xml version="1.0" encoding="UTF-8"?>
<script>
    <preamble>
        <dlname>My First Digital Library</dlname>
        <location>/installations/dls/</location>
    </preamble>
    <installation>
        <question>
            <description>
                The administrator’s e-mail is that to which all correspondence about the
                digital library will be directed.
            </description>
            <text>Please Input the Administrator’s e-mail Address</text>
            <answer> </answer>
            <default>administrator@domain.suffix</default>
            <locations>
                <location>/CCL/instance[0]/.../description/ui/adminEmail</location>
                <location>/CCL/instance[1]/.../description/dbbrowse/adminEmail</location>
                <location>/CCL/instance[2]/.../description/irdb/adminEmail</location>
                <location>/CCL/instance[3]/.../description/dbunion/adminEmail</location>
            </locations>
        </question>
        <question>
            <description>
                Some of the components need to use a database to store their internal data.
                This database should already be created. If not, please create it before
                continuing.
            </description>
            <text>Please Input the Database</text>
            <answer> </answer>
            <default>etds_db</default>
            <locations>
                <location>/CCL/instance[1]/.../description/dbbrowse/database</location>
                <location>/CCL/instance[2]/.../description/irdb/database</location>
                <location>/CCL/instance[3]/.../description/dbunion/database</location>
            </locations>
        </question>
        <question>
            <description>
                This is the username that will be used to connect to the database that has
                been specified.
            </description>
            <text>Please Input the Database Username</text>
            <answer> </answer>
            <default>username</default>
            <locations>
                <location>/CCL/instance[1]/.../description/dbbrowse/dbusername</location>
                <location>/CCL/instance[2]/.../description/irdb/dbusername</location>
                <location>/CCL/instance[3]/.../description/dbunion/dbusername</location>
            </locations>
        </question>
        <question>
            <description>
                This is the password that will be used in conjunction with the username that
                has been specified to connect to the database.
            </description>
            <text>Please Input the Database Password</text>
            <answer> </answer>
            <default>password</default>
            <locations>
                <location>/CCL/instance[1]/.../description/dbbrowse/dbpassword</location>
                <location>/CCL/instance[2]/.../description/irdb/dbpassword</location>
                <location>/CCL/instance[3]/.../description/dbunion/dbpassword</location>
            </locations>
        </question>
        <question>
            <description>
```

```
                This is the database table prefix that will be used within the database in
                order to differentiate data that belongs to the Union [dbunion] component.
            </description>
            <text>Please Input the Database Table Prefix for the Union Component</text>
            <answer> </answer>
            <default>union_archive</default>
            <locations>
                <location>/CCL/instance[3]/instanceDescription/description/dbunion/table</location>
            </locations>
        </question>
        <question>
            <description>
                This is the database table prefix that will be used within the database in
                order to differentiate data that belongs to the Search [irdb] component.
            </description>
            <text>Please Input the Database Table Prefix for the Search Component</text>
            <answer> </answer>
            <default>search</default>
            <locations>
                <location>/CCL/instance[2]/instanceDescription/description/irdb/table</location>
            </locations>
        </question>
        <question>
            <description>
                The URL to CGI location is that which grants access to the CGI location where this
                Digital Library will be installed.
            </description>
            <text>Please Input URL to CGI Location</text>
            <answer> </answer>
            <default>http://localhost/cgi-bin/</default>
            <locations>
                <location> <!--URL_TO_CGI_LOCATION--> </location>
            </locations>
        </question>
    </installation>
</script>
```

APPENDIX B

EVALUATION EXERCISE

FLEXIBLE PACKAGING METHODOLOGIES FOR RAPID DEPLOYMENT OF CUSTOMISABLE COMPONENT-BASED DIGITAL LIBRARIES

EVALUATION EXERCISE

CONTENTS

Approximate Duration Time: 30 Minutes

INTRODUCTION

Before commencing with the evaluation exercise, it is essential to paint a clear picture of what it is all about, as well as introduce some unfamiliar terms that are used throughout this evaluation exercise. This section introduces some background information relevant to this study, the tools that will be used in various sections and used terminology, all of which will help in successfully completing the evaluation exercise.

Digital Library

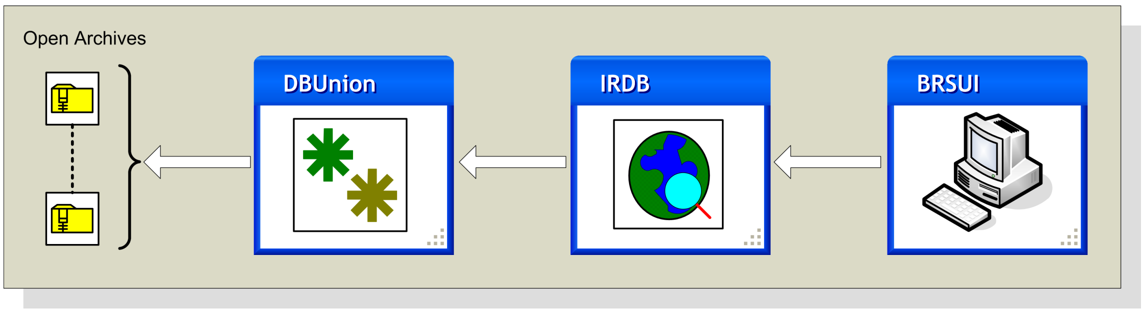
Simply put, a digital library is a managed and accessible electronic information management system. This information can be any of, but not limited to, documents or multimedia items. Each information item is associated with metadata — information about an information item, encoded in one or more metadata encoding standards, like the Dublin Core Metadata Standard, and is used for describing that particular information item. There are several services that a digital library can provide to access and sometime alter its contents. These include the ability to search through an archive, browse through an archive as well as record other high level annotations to each information item. There are various means that can be used by digital libraries in sharing and passing information amongst themselves, the most prominent being the Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH).

Open Digital Libraries

The Open Digital Libraries (ODL) project was concerned with building digital libraries out of components. The outcome of the ODL project was a suite of lightweight components, ODL components, which can be connected together and can communicate with each other and with other OAI compliant digital libraries using an extension of the OAI-PMH. The table below presents some ODL components that will be used throughout this evaluation exercise, with their associated functions.

Component	Function
DBUnion	Harvests (collects) data from one or more Open Archives — OAI compliant archives and keeps it in a local archive
IRDB	A search engine which searches a specified Open Archive and returns results
BRSUI	A user interface through which various services, including those provided by IRDB, can be utilised

The diagram below shows a component-based digital library composed of ODL components that have been described above. For the remainder of this exercise, this system will be referred to as "**the digital library.**"



baseURL

A baseURL is a Uniform Resource Locator (URL) that can be used in accessing an ODL component's information or accessing any other OAI compliant digital library's information over the Web. Considering the digital library shown above, the BRSUI component connects to the IRDB component using IRDB's baseURL, the IRDB component connects to the DBUnion component using DBUnion's baseURL and so on.

Questions

Most of the questions that are asked throughout the exercise, are scaled questions. The table below explains the vocabulary that is used in the scaled questions.

Phrase	Explanation
Very Good	Exceptionally agree with, exceptionally acquainted with, or completely accept the idea, concept or subject presented by the question at hand
Good	Significantly agree with, significantly acquainted with, or significantly accept the idea, concept or subject presented by the question at hand
Neutral	Moderately agree with, moderately acquainted with, or moderately accept the idea, concept or subject presented by the question at hand
Poor	Somewhat agree with, somewhat acquainted with, or somewhat accept the idea, concept or subject presented by the question at hand
Very Poor	Insufficiently agree with, insufficiently acquainted with, or insufficiently accept the idea, concept or subject presented by the question at hand
No Response	Do not know or are not is a position to give a meaningful or appropriate answer to the question at hand

For questions that present a choice box (☐), use either a tick ($\surd$) or a cross ($\times$) to indicate your preferred answer. Other questions are open-ended and require written feedback. These will offer an opportunity to express your views in an unconstrained manner.

Tools and Exercises

The system that will be evaluated is composed of two tools, the *Packager* and the *Installer* tools. The first exercise will involve using the *Packager* tool to build the digital library package. The second exercise is divided into two sections; the first involving installing the digital library from the package created in the first exercise and the second involving installing the digital library from loosely placed and partially configured components. At the end of each exercise as well as in the final section, there are questions aimed at gathering feedback.

PLEASE NOTE that you are taking part in this exercise on a voluntary basis, purely for academic purposes, and the observations that will be recorded by means of pencil/pen and paper as well as the results thereof, will be confidential. This analysis is strictly based on the tools provided, and at no point will the focus of the analysis be on you or your computer literacy. Feel free to ask questions at any time while performing the tasks and you are not obliged to complete any or all of the tasks, should there be a need for you not to.

B.1 Background Information

In this section, you are requested to provide some background details relating to your *Qualifications, Software Installations, Software Management Tools* as well as some aspects of *Software Systems.*

B.1.1 Qualification Details

Please fill in appropriate fields below. If you are a student, fill in under *Student* otherwise fill in under *Professional*:

Student		*Professional*	
Programme	:	Profession:	. . . .
Year of Study	:	Qualification:	. . . .
Major(s)	:		

B.1.2 Software Installations

Have you installed any software packages on the following platforms? If **Yes** is selected, please supply at least one example.

Windows:	Yes ☐	No ☐	Example(s) : . . . .
Linux:	Yes ☐	No ☐	Example(s) : . . . .

B.1.3 Software Management Tools

How would you rate your ability to install, configure, manage and uninstall packages created with the following tools? *Recall:* **No Response = Do not know or have not used the tool in question.**

InstallShield. Very Good ☐ Good ☐ Neutral ☐ Poor ☐ Very Poor ☐ No Response ☐
Wise/UnWise. Very Good ☐ Good ☐ Neutral ☐ Poor ☐ Very Poor ☐ No Response ☐
RedHat Package Manager (RPM). Very Good ☐ Good ☐ Neutral ☐ Poor ☐ Very Poor ☐ No Response ☐
Other: .... Very Good ☐ Good ☐ Neutral ☐ Poor ☐ Very Poor ☐
Other: .... Very Good ☐ Good ☐ Neutral ☐ Poor ☐ Very Poor ☐

B.1.4 Software Systems

As a user of software systems, how would you rate your **understanding** of the following?

Component-based software system (E.g. JavaBeans, C++ Builder, ActiveX) Very Good ☐ Good ☐ Neutral ☐ Poor ☐ Very Poor ☐
Web-based software systems (E.g. Online Guestbook, Online Photo Album, Online Store) Very Good ☐ Good ☐ Neutral ☐ Poor ☐ Very Poor ☐
Online information management systems (E.g. Britannica Online, Wikipedia, ACM Digital Library) Very Good ☐ Good ☐ Neutral ☐ Poor ☐ Very Poor ☐

Are you, or have you ever been a computer programmer?

Yes ☐ No ☐

If **Yes**, how would you rate your **understanding** of the following from the perspective of using the tools as a programmer?

Component-based software system (E.g. JavaBeans, C++ Builder, ActiveX)				
Very Good ☐	Good ☐	Neutral ☐	Poor ☐	Very Poor ☐
Web-based software systems (E.g. Online Guestbook, Online Photo Album, Online Store)				
Very Good ☐	Good ☐	Neutral ☐	Poor ☐	Very Poor ☐
Online information management systems (E.g. Britannica Online, Wikipedia, ACM Digital Library)				
Very Good ☐	Good ☐	Neutral ☐	Poor ☐	Very Poor ☐

Prior to this exercise, did you know what Digital Libraries are?

Yes ☐	No ☐

If **Yes**, how would you rate your knowledge about them?

Installed ☐	Used ☐	Heard About ☐

B.2 DIGITAL LIBRARY COMPONENTWISE INSTALLATION

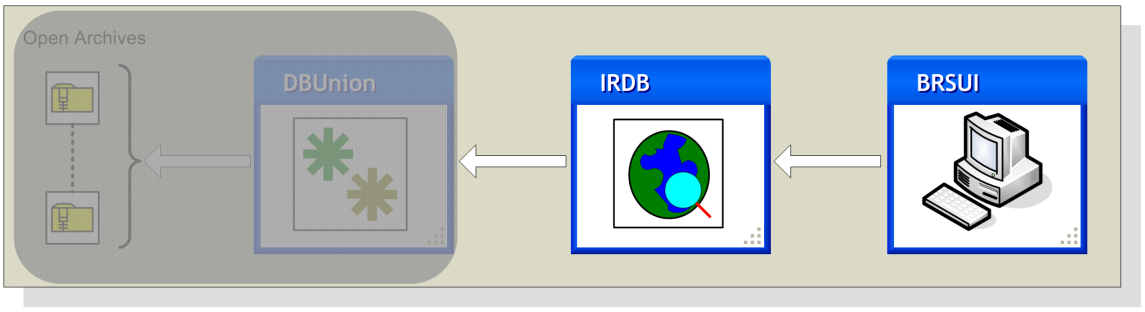
In this exercise, you will be installing the digital library one component at a time. During the installation, there are some cases where there will already be default values for some fields. It is therefore advisable to read the question thoroughly, in order to decide whether to keep the default value or specify a more meaningful answer. Guidelines on answers are given where relevant.

Aim

To install the digital library componentwise.

Method

The diagram below shows part of the installation process (the shaded area) which has already been performed on your behalf. What this entailed was configuring the DBUnion component to gather data from *Husseins' Photo Album* with the baseURL *http://www.husseinsspace.com/cgi-bin/VTOAI/hspics/hspics/oai.pl*. This data was then stored in a mySQL database *evaluation_db*. For this exercise, you are requested to configure the remaining components — IRDB and BRSUI, using the methods and parameters given below.



Configuring the IRDB Component

A *terminal* through which you will be able to configure the IRDB component has been opened for you. To run the configuration script, type the following command: *perl configure.pl evaluation* followed by <ENTER>. You will then be able to answer the questions that follow, with the following guidelines.

- Database Connection
  - Driver — *mysql*
  - Name — *evaluation_db*
  - Username — *root*
  - Password — *root*
  - Table — *irdb_2*
- Repository Name — [*accept default*]
- Administrator E-Mail — *fcp@cs.uct.ac.za*
- Archive
  - Identifier — *Local_DBUnion*

- URL — *http://nala.cs.uct.ac.za/cgi-bin/evaluation2/ODL-DBUnion-1.2/DBUnion/evaluation2/union.pl*
- Harvesting Interval — [*accept default*]
- Harvesting Overlap — [*accept default*]
- Harvesting Granularity — [*accept default*]
- Metadata Prefix — *oai_dc*
- Sets — [*accept default*]

To verify if the configuration was successful, type the following commands in the terminal:

cd evaluation <ENTER>	*switching to the configured instance*
perl harvest.pl <ENTER>	*gathering data from the DBUnion archive*
perl testsearch.pl ’uct’ <ENTER>	*issuing a test query to verify that IRDB has been properly configured*

Configuring the BRSUI Component

A *terminal* through which you will be able to configure the BRSUI component has been opened for you. To run the configuration script, type the following command: *perl configure.pl evaluation* followed by <ENTER>. You will then be able to answer the questions that follow, with the following guidelines.

- User Interface Name — *Evaluation Exercise User Interface*
- Administrator E-Mail — *fcp@cs.uct.ac.za*
- Title Bar — *BRSUI User Interface*
- Body Title — [*accept default*]
- Message — [*accept default*]
- Footer — [*accept default*]
- Search baseURL — *http://nala.cs.uct.ac.za/cgi-bin/evaluation2/ODL-IRDB-1.3/IRDB/evaluation/search.pl*
- Browse baseURL — [*leave blank*]
- Rate baseURL — [*leave blank*]

To verify if the configuration was successful, point a browser at the the following URL: *http://nala.cs.uct.ac.za/cgi-bin/evaluation2/BRSUI/BRSUI/evaluation/index.pl* and issue a test search query ’uct’

Feedback

Have you successfully installed the digital library?

Yes ☐	No ☐

How would you rate your **understanding** of the installation process?

Very Good ☐	Good ☐	Neutral ☐	Poor ☐	Very Poor ☐

How would you rate the overall **usability** associated with installing the digital library componentwise?

Very Good ☐	Good ☐	Neutral ☐	Poor ☐	Very Poor ☐

Comment on any other aspect of the **componentwise installation** process.

B.3 DIGITAL LIBRARY PACKAGE BUILDING AND INSTALLATION

This exercise is divided into two sections. The first section will be to build the digital library package while the second section will be installing the digital package that will be built in the first section.

B.3.1 Building the Digital Library Package

The process of building the digital library package is concerned with specifying default values like *name* and *installation location* which will be used when the package is installed at a later point. Furthermore, when building a package, additional questions to be asked at installation time which do not form part of the initial digital library specification can be specified and any of the questions edited or removed. This exercise will cover most of the aspects of creating a digital library package.

Aim

To build the digital library package described by the CCL file: $evaluation.ccl$

Method

1. Launch the *Packager* tool to begin building the package
2. Follow on-screen instructions to progress through the following stages of building a package
   - **Welcome:**
     Gives an overview of the package building process
   - **Load CCL File:**
     Load a specification file describing the digital library. Find the CCL file at this location: $Desktop$
   - **Digital Library Components:**
     Shows all the components that make up the digital library
   - **Digital Library Name:**
     Specify the default name with which the digital library will be known. This can be any name you wish it to be. For example: *My First Digital Library, mylib* or *Library*
   - **Installation Location:**
     Specify the default installation location for the digital library. For example: */dls/installations/*
   - **Installation Questions:**
     Add, edit or remove questions to be asked during installation. Please perform the following tasks:
     – Edit the ’**Please Input the Database**’ question by changing the default value to ’**DBI:mysql:evaluation_db**’
     – Add a new question with the following fields:

Question:	Please Input the Database Table Prefix for the Search Component
Description:	This is the database table prefix that will be used within the database in order to differentiate data that belongs to the Search [irdb] component
Default:	irdb
Path:	/CCL/instance[1]/instanceDescription/description/irdb/table

- **Save Package:**
  Specify the name of the package as well as the location where it should be saved. Use the $Desktop$ location
- **Finalising Package:**
  Creates the package and removes unwanted data
- Verify by checking in $Desktop$ if the package has been created and stored

Feedback

Have you successfully created the digital library package?

Yes ☐	No ☐

How would you rate your **understanding** of the package building process?

Very Good ☐	Good ☐	Neutral ☐	Poor ☐	Very Poor ☐

How would you rate the overall **usability** of the *Package Builder* tool?

Very Good ☐	Good ☐	Neutral ☐	Poor ☐	Very Poor ☐

Comment on any other aspect of the **package building** process.

B.3.2 Installing the Digital Library Package

The process of installing the digital library package has a similar feeling to that of building the digital library package as in Section B.3.1. During the installation, there are some cases where there will already be default values for some fields. It is therefore advisable to read the question thoroughly, in order to decide whether to keep the default value or specify a more meaningful answer. Guidelines on answers are given where relevant.

Aim

To install the digital library package from the previous exercise, Section B.3.1.

Method

1. Unzip the package created in Section B.3.1 and launch the *Installer* tool.
2. Follow on-screen instructions to progress through the following stages of installing a package.
    - **Welcome:**
      Gives an overview of the package installation process
    - **Installation Location:**
      Specify a location to which the digital library will be installed. Specify the package installation location as: *F:\Perl and Apache2\Apache2\cgi-bin\evaluation*
    - **Extracting Files:**
      Extracts all the necessary files for installing the digital library
    - **Dependency Summary:**
      Shows a summary of components and their associated dependencies
    - **Questions:**
      All the questions that have been specified at build time will be asked here. Give appropriate answers or accept the default values. For the question: **Please Input URL to CGI Location** use the value: **http://nala.cs.uct.ac.za/cgi-bin/evaluation**
    - **Finalising Installation:**
      Completes the installation process by running automatic configuration scripts and removing unwanted data
3. Launch the installed digital library on a browser. Use the URL given at **Finalising Installation**
4. Uninstall the digital library

Feedback

Have you successfully installed the digital library package?

Yes ☐ No ☐

How would you rate your **understanding** of the package installation process?

Very Good ☐	Good ☐	Neutral ☐	Poor ☐	Very Poor ☐

How would you rate the overall **usability** of the *Package Installer* tool?

Very Good ☐	Good ☐	Neutral ☐	Poor ☐	Very Poor ☐

Comment on any other aspect of the **package installation** process.

B.4 Survey

How would you rate the overall **aesthetic design** of both the *Package Builder* and *Package Installer* tools?

Very Good ☐ Good ☐ Neutral ☐ Poor ☐ Very Poor ☐

What would you omit, add or otherwise do if you were to design the *Package Builder* and *Package Installer* tools.

Did the *Package Builder* and *Package Installer* tools strike you as something completely different to what you are already acquainted with? Explain.

Which installation process, according to your experience, do you regard as better? Explain your choice **comparatively** i.e., by listing the strengths and weaknesses of both processes.

☐ Package or
☐ Componentwise