A Java-based Smart Object Model for use in Digital Learning Environments

Vara Prashanth Pushpagiri

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

Dr. Saifur Rahman, Chair

Dr. Ing-Ray Chen

Dr. Yao Liang

Date: 06/13/03

Alexandria Research Institute
Alexandria, Virginia

Keywords: Smart Objects, Digital Libraries, E-learning techniques, Intelligent Learning Objects

© Copyright 2003, Vara Prashanth Pushpagiri

A Java-based Smart Object Model for use in Digital Learning Environments

Vara Prashanth Pushpagiri

Abstract

The last decade has seen the scope of digital library usage extend from data warehousing and other common library services to building quality collections of electronic resources and providing web-based information retrieval mechanisms for distributed learning. This is clear from the number of ongoing research initiatives aiming to provide dynamic learning environments.

A major task in providing learning environments is to define a resource model (learning object). The flexibility of the learning object model determines the quality of the learning environment. Further, dynamic environments can be realized by changing the contents and structure of the learning object, i.e. make it mutable. Most existing models are immutable after creation and require the library to support operations that help in creating these environments. This leaves the learning object at the mercy of the parent library's functionality. This thesis work is an extension of an existing model and allows a learning object to function independent of the operational constraints of a digital library by equipping learning objects with software components called methods that influence their operation and structure even after being deployed. It provides a reference implementation of an aggregate, intelligent, self-sufficient, object-oriented, platform-independent learning object model, which is conformant to popular digital library standards.

It also presents a Java-based development tool for creating and modifying smart objects. It is capable of performing content aggregation, metadata harvesting and user repository maintenance operations, in addition to supporting the addition/removal of methods to a smart object. The current smart object implementation and the development tool have been deployed successfully on two platforms (Windows and Linux) where their operation was found to be satisfactory.

Acknowledgements

I would like to thank Professor Saifur Rahman, my academic and research advisor for his guidance and support for the duration of my program. Without his patience and advice, this work would not have been possible. I would also like to thank Dr. Ing-Ray Chen and Dr. Yao Liang for reviewing this work and providing valuable suggestions and comments.

I also wish to express my gratitude to the National Science Foundation's National Science Digital Library (NSDL) initiative, whose financial support for the Digital Library Network for Engineering and Technology (DLNET) helped in modeling these Java-based Smart Objects for digital learning.

Special thanks go out to Dr. Michael Nelson at the Old Dominion University for his guidance and suggestions throughout this work. My sincerest thanks go out to Dr. Thomas D. Wason, for his suggestions and initial interactions that helped identify this research. Also, I would like to thank Yonael Teklu, research associate and Ph.D. student at the Alexandria Research Institute, for his suggestions and views on this work, for his patience during times when I bugged him with technical and non-technical questions and most of all for being a great friend.

I would like to thank everyone at the Alexandria Research Institute, George Hagerman, Kaustubh Phanse, Latricia Nell, Manisa Pipattanasomporn, Sai Kroviddi, Vikram Dham, Vivek Srivastava, and others whose names I may have missed. Their friendship and moral support helped greatly in the completion of my program. Finally, I owe it to my parents and family members for their love, support and open-mindedness, for standing by me in every step/decision I made over the years. I truly would not have come so far without them.

Table of Contents

Abstract ii
Acknowledgements iii
1.0 Introduction 1
  1.1 Overview 1
  1.2 Motivation and Problem Statement 2
  1.3 Approach 3
  1.4 Thesis Contribution 4
2.0 Background Readings 5
  2.1 Overview 5
  2.2 Kahn-Wilensky Digital Objects 5
  2.3 Microsoft LRN 7
  2.4 DLNET Learning Object 8
  2.5 Sharable Content Object Reference Model (SCORM) 10
  2.6 Smart Object Dumb Archive Model (SODA) 12
3.0 Metadata 16
  3.1 Overview 16
  3.2 Metadata as XML 17
  3.3 Dublin Core Metadata 18
    3.3.1 Dublin Core with DC-Ed Extensions 20
  3.4 IMS Specifications 21
    3.4.1 IMS/IEEE Learning Object Metadata Specification 21
    3.4.2 IMS Content Packaging Specification 23
4.0 Smart Objects 26
  4.1 Overview 26
  4.2 Definition 26
  4.3 Characteristics and Requirements 27
    4.3.1 Aggregation 27
    4.3.2 Intelligence 27
    4.3.3 Object-oriented 29
    4.3.4 Platform Independent 29
    4.3.5 Ease of adoption and conformance 29
  4.4 Smart Object Architecture 30
    4.4.1 File Structure 30
    4.4.2 Metadata Storage 34
    4.4.3 Smart Object Configuration 35
    4.4.4 User Repository and Permissions 37
  4.5 Smart Object Methods 38
    4.5.1. Display 39
    4.5.2. Add Package 40
    4.5.3. Add Element 41
    4.5.4. Delete Package 41
    4.5.5. Delete Element 42
    4.5.6. List Methods 42
    4.5.7. Metadata 43
    4.5.8. List Source 44

4.5.9. Version 44
4.5.10. Set Version 44
4.5.11. Add Terms and Conditions(TC) 44
4.5.12. Delete Terms and Conditions 45
4.5.13. List Logs 46
4.5.14. Delete Logs 46
4.5.15. ID 46
4.5.16. Pack 47
4.5.17. Add Metadata 47
4.5.18. Delete Metadata 47
4.5.19. Add Principal 48
4.5.20. Delete Principal 48
4.6 How are Smart Objects different? 49
4.6.1 DLNET LO vs. Java Smart Objects 49
4.6.2 SCORM vs. Java Smart Objects 49
4.6.3 Buckets vs. Java Smart Objects 50
5.0 Smart Object Development Tool 51
5.1 Overview 51
5.2 Tool Requirements 51
5.2.1 Content Aggregation 51
5.2.2 Metadata Extraction 52
5.2.3 Support for Methods 52
5.2.4 User repository and Access Control 52
5.2.5 Content Packaging 53
5.2.6 Platform Independence 53
5.3 Tool Architecture and Layout 53
5.3.1 Content Aggregation 56
5.3.2 Metadata Extraction 57
5.3.3 Supported Methods 58
5.3.4 User repository and Access Control 59
5.3.5 Content Packaging 60
5.3.6 Platform Independence 60
5.4 Creating and deploying Smart Objects - An example 61
6.0 Conclusions and Future Work 67
6.1 Summary 67
6.2 Conclusions 68
6.3 Future Work 68
6.3.1 Additional Functionality 68
6.3.2 Centralized Policy Distribution 69
6.3.3 Increased Security 69
6.3.4 SCORM conformance 70
Appendix – Smart Object API 71
References 91
Vitae 93

Table of Figures

Figure 2.1. Kahn Wilensky Digital Object 6
Figure 2.2. Microsoft's LRN Toolpad 8
Figure 2.3. DLNET Learning Object Representation 10
Figure 2.4. SCORM Content Aggregation Model 12
Figure 2.5. Bucket representation 14
Figure 3.1. IMS Learning Object Metadata Model 22
Figure 3.2. IMS Content Package* 24
Figure 3.3. IMS Content Packaging XML Binding 25
Figure 4.1. Aggregation Example 27
Figure 4.2. Learning Object with intelligence 28
Figure 4.3. File System representation of a smart object 31
Figure 4.4. Sample manifest instance 33
Figure 4.5. A sample metadata instance 34
Figure 4.6. A sample configuration instance 36
Figure 4.7. Sample user repository instance 38
Figure 4.8. Element permissions example 38
Figure 4.9. Display method (method=display&package=pkg1) 40
Figure 4.10. Display method (method=display&package=pkg1&element=elem1&start=true) .. 40
Figure 4.11. Output from the metadata method 43
Figure 5.1. Three-tier architecture of the Smart Object Development Tool 54
Figure 5.2. Graphical User Interface of the Smart Object development tool 55
Figure 5.3. Content Aggregation interface 56
Figure 5.4. Metadata harvesting interface 57
Figure 5.5. Supported methods panel 58
Figure 5.6. Sample user repository interface 59
Figure 5.7. Smart Object example: Root metadata 62
Figure 5.8. Smart Object example: User Access Panel (Add User) 63
Figure 5.9. Smart Object example: Add Package 64
Figure 5.10. Smart Object example: Package/Element structuring information 65
Figure 5.11. Sample Apache Tomcat Configuration instance 66

Table of Tables

Table 3.1. Dublin Core Metadata Elements 19
Table 4.1. Comparison of DLNET LO, SCORM, Buckets and Java Smart Objects 50

Chapter One

Introduction

1.1 Overview

Digital Libraries are ever growing entities in the current era. The nineties have seen digital library research get a shot in the arm thanks to a substantial number of initiatives [44]. The scope of digital library usage grew from electronic preservation of rare artifacts/documents to the digitization of common library services so as to make possible the acquisition of paper-based resources in an electronic format [44, 25]. Current digital library prototypes are mostly aimed at building quality collections of electronic resources and providing web-based information retrieval mechanisms for distributed learning. Research in this field has given birth to numerous models and methodologies for aggregating and representing electronic content as well as archival and discovery strategies. Digital library concepts and methodologies have justified their existence/usage in an age mostly dominated by web crawlers and search engines [25]. Although both seemingly provide similar services to an end-user, i.e. help in information discovery and retrieval, they differ vastly in the concepts and levels of applicability to various learning contexts. While World Wide Web (WWW) search engines intend to expose any and all web-based content to a user, Digital Libraries are more selective and provide a higher level of granularity in the search services they provide. Digital libraries typically host and manage electronic content in a more learner-oriented context. They serve as warehouses for electronic resources with learning value and can be used in continuous education environments. This thesis is an offspring of one such initiative aimed at promoting life-long learning opportunities of practicing engineers, technical professionals, students and faculty of the engineering community. Entitled “Digital Library Network for Engineering and Technology” (DLNET) [7], the project is part of a national initiative to provide online education from K-12 and beyond named the National Science Digital Library Initiative (NSDL) [31]. This thesis presents a scalable learning resource model that is applicable to any digital library infrastructure due to its inherent ability to be self-sufficient, yet manageable and highly structured, yet easy to use with high adaptability to changing learning environments, one that is truly in pace with the current well-known digital library and learning resource models.

1.2 Motivation and Problem Statement

Digital Library research has taken the frontiers of learning beyond the classroom to a new level. Only considered a pipe-dream a couple of decades ago, the internet today serves as an inexhaustible resource for communication via the digital realm. This growth gave way to novel ideas leading to the possibility of knowledge dissemination via the internet. Digital Library research sprouted as a result giving way to numerous initiatives aimed at delivering knowledge digitally. One idea that differentiated one library from the other was a model for digital representation and dissemination. Numerous models for realizing such an electronic system have been proposed and implemented with various levels of success in digital libraries. Some of these models are based on generic standards of digital learning and can further be extended to suit individual needs of a library. More often however, these models are tailor-made to suit the requirements of a digital library architecture and blend into the library's operational framework. Localization is a problem that has been long neglected. Localized models usually tend to be driven by repository architectures. Such models may not guarantee uniform behavior in heterogeneous platforms and hence are dependent on environment of deployment and operation. In other words customized designs may not guarantee portability of electronic resources.

Another issue currently in the limelight is that of independence. Current digital library architectures are such that the behavioral characteristics of a Learning Object are highly dependent on the repository's functionality. In other words, a learning object is *dumb* in that it has no knowledge or control over the operations performed over it. All the functionality or *intelligence* is possessed by the repository. For instance an important functionality that next generation digital libraries seek is a mechanism for metadata editing and cataloguing. A repository holding a variety of learning resources would want to at some point be involved in metadata editing activities either to address changes in the structure of the resource or to address version control. Such an act requires a centralized authority (the repository) to perform all necessary operations. Repositories not aware of such methods struggle to perform these functions or may need to call for radical changes in repository architecture. Current architectures require an operation like require version control to be implemented via manual intervention of an editor/administrator or by representing the content as an independent learning object.

To address these two problems, a common scalable model for learning resource representation needs to be defined that can address portability as well as internal restructuring issues. Standards like SCORM[37] that aim at providing a unifying approach to information representation tend to solve the first problem of portability, but fail to succeed in addressing internal changes to the resource. A model proposed by Nelson at the Old Dominion University addressed these issues in a model they named Smart Object Model or the Bucket Model. The model however has not caught up in the digital library world due to the lack of conformance to any well known standard currently in use. This thesis intends to exemplify the use of such a model with well known standards while addressing some shortcomings and future work the model proposed. Noticeable changes to the existing model [chapter 2] include the adoption of a SCORM-compliant XML standard for representing learning object metadata and the use of a more web-versatile implementation language (Java) as opposed to Perl, proposal of an n-tier architecture for the learning resource model (referred to as Smart Object Model), additional functionality (intelligence) imparted to these Smart Objects and a management tool to create and control these objects. Such a model would decouple the existing link between object representation and repository architecture thus allowing for independent growth and development of the two with the knowledge that seamless integration of one into the other's framework is no trouble. The introduction of IMS content package and metadata specification into the smart object model would allow for seamless integration into any well-known digital library infrastructure.

1.3 Approach

We have discussed the need for decoupling the development of digital object development from repository architecture and the merits of doing so in the previous section. The Sharable Content Object Reference Model (SCORM), is an excellent reference model for digital objects currently in development as part of the Advanced Distributed Learning (ADL)[2] initiative by with contributions from leading industrial organizations. The Smart Object Model [26] proposed as part of a NASA research in digital libraries on the other hand provides an object-oriented approach to resolving the issues we discussed. The disadvantages of the model having a low acceptance rate due to antique standards make it an improbable candidate for seamless integration into current repository architectures. Hence our proposal to modify the existing object model and retrofit it with the current buzzing technologies makes it possible to address

integration and portability issues. It is to be noted here that the changes we make as part of this thesis are in the best interest of the object model's conformity to a well-known and accepted standard such as SCORM and by no means do we propose this as the only way to achieving the objective.

This is how the rest of the thesis is organized: Chapter 2 delves into related and base models for this research and gives a comparative study of how one model is different from its nearest conforming model, Chapter 3 details metadata and current standards of metadata represented as XML and how and why the choice of IMS metadata would solve scalability and integration problems, Chapter 4 discusses in depth about the current implementation of the Smart Object Model and compares the features of this model with other existing implementations, Chapter 5 discusses the design aspects of the Smart Object Creation and Management Tool used for creating these smart objects, Chapter 6 summarizes the current work and highlights future research in this area.

1.4 Thesis Contribution

Learning objects are key to providing dynamic learning environments. The robustness, scalability and interoperability of a learning object model determines the operational effectiveness of a digital library. To this end, this thesis contributes towards the realization of a novel learning object model which is scalable, self-sufficient, interoperable, intelligent and specification-conformant and is an alternate implementation and extension of the Bucket model [26]. It discusses some known learning object models like Buckets, SCORM and DLNET's LO model and a variety of specifications currently available to represent learning object metadata. Based on these discussions, the work proposes a suitable architecture for developing learning objects with intelligence and self-sufficiency while resolving scalability and metadata issues that other models suffer from. In addition, this work equips smart objects with a schema-based lightweight XML user repository, configuration instance and software modules (implemented in Java) that provide intelligence and self-sufficiency to the smart object. Finally, the work involved developing a Java-based tool for creating and managing smart objects and discusses deployment issues.

Chapter Two

Background Readings

2.1 Overview

The choice of providing a scalable learning object model that can be self-sufficient, manageable, object-oriented in nature requires a clear understanding of the existing models. This chapter serves as a review of the existing learning object models that can be closely associated with the objective we have proposed in the previous chapter. Here we provide a review of four existing learning object models, all of them bearing similarities to one another while composing how the extension of the Smart Object Model lead to SCORM conformance.

2.2 Kahn-Wilensky Digital Objects

The Kahn-Wilensky architecture[34] was one of the first concrete models for representing digital entities or objects with an expandable framework and seamless integration into repositories. Most modern digital learning object models are developed with this architecture as the genesis of their representations. The architecture was proposed in 1995 as part of an ARPA effort to promote digital learning by Robert Kahn of the Corporation for National Research Initiatives and Robert Wilensky of the University of California at Berkeley. Two key components were highlighted in the proposal. The first was an unambiguous definition for digital representations of resources, termed *digital objects* and the second was an insight into the Repository Access Protocol (RAP) used to manage digital objects. The Kahn-Wilensky architecture defined basic elements that are fundamental aspects of an open, extensible infrastructure/repository [34]. The model defined a digital object as an aggregation of digital data elements (file system components) and key metadata providing information about the elements. A digital object is identified via a handle for universal identification and is represented as part of the key-metadata. Handles are typically issued by handle servers which also maintain mapping information of currently registered handles. Key-metadata would comprise of the handle along with other metadata. Modern digital libraries propose definitions for customized digital objects and are more than often an extension of the Kahn-Wilensky proposal. A rudimentary representation of the digital object they proposed is represented in figure 2.1 below.

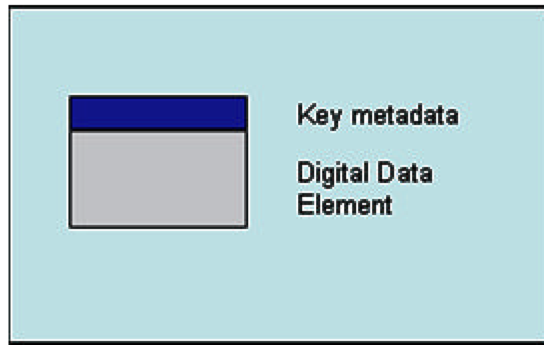

Figure 2.1. Kahn Wilensky Digital Object

The Repository Access Protocol proposed by Kahn and Wilensky was intended to provide a framework for integrating digital objects into repository architectures. It defined the objectives of the RAP as being the nucleus of communication with a digital object. It enables deposition and access mechanisms of digital objects in a repository and a mandatory component of any digital library implementation. Basic proposals of RAP included methods such as ACCESS_DO for accessing the digital objects and metadata via the handle, DEPOSIT_DO for depositing data, handle and/or metadata pertaining to the object and ACCESS_REF for accessing reference services such as metadata or the digital object itself via alternate schemes. While the model mandates the implementation of the RAP, it allows for alternate implementations for object handling by a repository.

This framework is the basis for many current implementations of digital objects. Over the years, the definition of a digital object has been extended beyond this proposal and yet been in conformance to the first definition. Currently, organizations such as the CNRI have developed a working model for the Digital Object Identifier (DOI) [39] scheme for aggregating digital objects. By using this interface, an originator [34] (an entity that authorizes each digital object and makes it available in a repository system) the RAP and handle concepts of however has seen diminishing popularity as repositories tend to desire custom implementations for repository access. This scheme has been adopted by as many as 200 organizations with an aggregate of over a million registered DOIs [40].

2.3 Microsoft LRN

Microsoft has been involved in the promotion and development of learning management systems for the past few years [43]. They have been active participants in the development of the IMS specifications for learning resource representations [13]. In view of the wide-spread adoption of the IMS Content Packaging [13] and metadata standards [14], Microsoft developed a windows-based software to create and modify what is called an LRN Package [27]. The software had the capability of creating learning resources according to the IMS content packaging specification. Due to the adoption of the IMS standard in SCORM [37], the software could be used to edit metadata for SCORM objects and by default any other digital object model based on the IMS. The specifics of the IMS standard are discussed separately in chapter 3 along with other standards for metadata representation and hence not probed into much detail in the current discussion.

The Microsoft LRN (pronounced 'Microsoft Learn') [27] software provides users with an opportunity to create an LRN package that can be used in a learning management environment. An LRN package is conformant to the IMS content packaging standard (ver 1.1) and IMS metadata specification (ver 1.2) and the SCORM (ver 1.2) at the time of documenting this thesis. An LRN package is comprised of a manifest file (an XML file conformant to IMS specifications) describing the contents of the package, the knowledge-providing resources themselves and wrappers for user interfaces. Key components of the Toolkit include the LRN Toolpad for creating LRN packages viewable via a browser, the LRN Manifest converter for updating older versions of the package, an LRN converter for creating packages from web pages created using a Microsoft Office product and an LRN editor to create and modify these packages. The LRN viewers (HTML application) created by this software are the primary knowledge providing interfaces for the resource. These constructs of this package creator are similar in some ways to that of the DLNET Learning Object package [23] we discuss in the next section. However, the model suffers from a potential disability. Being a predominantly browser-oriented education mechanism, the LRN viewer operates only on the latest version of browser provided by the organization (ver 6). This is primarily due to the usage of ActiveX components for XML access, a feature alien to most browsers today. Nevertheless, the LRN package is considered to be a valuable contribution to the learning community, one that justifies the usage of extended

metadata standards such as the IMS for learning. Figure 2 shows a pictographic representation of the software.

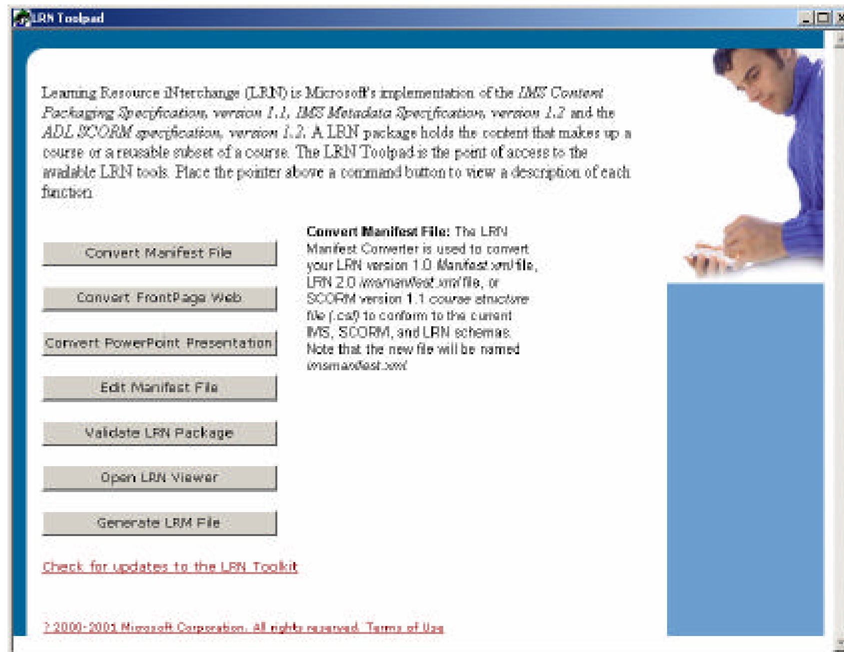

Figure 2.2. Microsoft's LRN Toolpad

2.4 DLNET Learning Object

The DLNET Learning Object [23] much like the other models we have discussed thus far came into being as a result of research aimed at providing lifelong education and learning opportunities to practicing engineers and professionals of the engineering community. The model was developed in view of the heterogeneous nature of learning resources available for distribution via the internet, ranging from hypertext content to multi-media resources. The intent of the learning object model was to enable standalone operation of high-quality knowledge providing material. The learning object defined as a structured electronic resource that encapsulates high-quality information in order to facilitate learning and pedagogy has a stated objective and a targeted audience.

The DLNET Learning Object Model derived its constructs from the Kahn-Wilensky model [34] discussed earlier. The model used an extended IMS standard for metadata representation and the IMS content packaging specification (ver 1.1.2) for package representation. It is similar in design to the LRN package we discussed earlier and is a browser-based information dissipation model. Unlike the LRN package however, it has fewer learner-oriented constraints. The DLNET

Learning Object does not necessitate the use of a particular browser or platform unless otherwise required by the resource itself, i.e., the learning object does not provide for any run-time needs of a resource. Part of this functionality comes due to a simplistic object preamble, essentially the face of the learning module containing information about the resource(s) and linkage to the resources themselves. The DLNET Learning Object package like its Microsoft counterpart comprises of a manifest instance containing the knowledge providing information, the knowledge providing material itself, and a wrapper to provide access to aforementioned components. Despite the extensions made to the IMS metadata standard, this model complies with the general IMS content packaging and metadata specifications and is expected to adapt to a changed domain or platform. The extensions this model provides, however, may be of little importance to a foreign domain hosting a DLNET Learning Object and be overridden by local specifications. A tool named the DLNET Learning Object Packaging tool distributed via the project's website [23] helps in creation of Learning resources. Figure 3 depicts a current representation of a DLNET Learning Object on Distributed Generation Technologies.

Despite the user-friendliness and simplicity of the learning object and its suitability to the DLNET framework, this model does have scope for improvements. The model supports the concept of package metadata, i.e., the metadata contained in the manifest instance is indicative of the package as a single entity. The information is expected to be representative of every component of the package, something not always true in a digital library context. A course module in Computer Network Security can be used to illustrate this deficiency. Such a course could span a semester's work and could be structured into numerous components. A lecture session may have slide shows, reading assignments/material, homework assignments, tool reviews etc. A course could typically comprise of many such lectures not to mention exams and other relevant material. Such a learning module is bound to be heterogeneous in nature with reading materials, slide shows, tools etc., all representative of specific tasks. A common metadata sample for such a resource package would fail to represent the diversity of each individual component of this package, needless to say confuse an end-user as to the exact nature of this package. It is in view of this that we experimented with the development of a dynamic, self-sufficient, manageable learning resource with an extensible structure and a satisfactory representation of learning resource metadata.

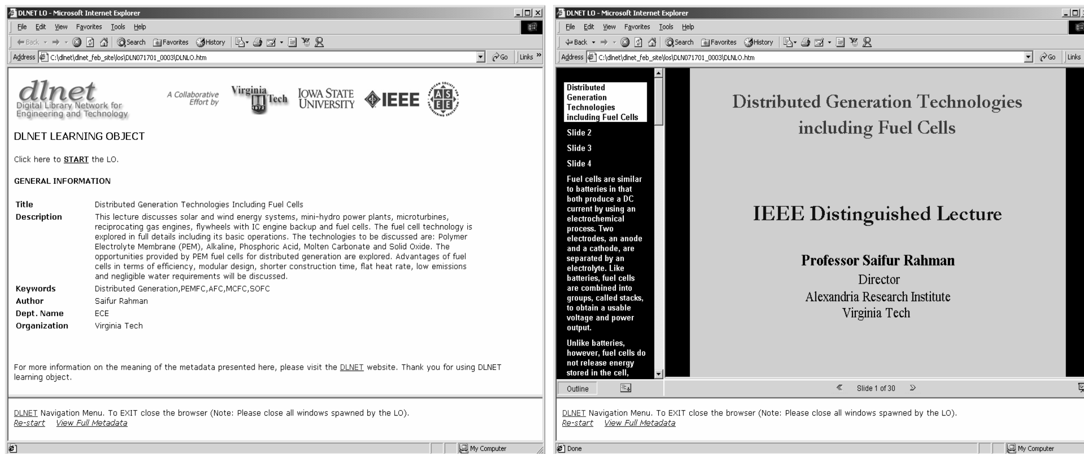

Figure 2.3. DLNET Learning Object Representation

2.5 Sharable Content Object Reference Model (SCORM)

The Sharable Content Object Reference Model (SCORM) is a part of the Advanced Distributed Learning Initiative (ADL), sponsored by the Department of Defense (DoD) since 1997 for the development of educational and learning information dissemination strategies to modernize education and e-learning standards. The initiative aspires to meet the high-level educational requirements of DoD by defining a compact model for enabling high-quality web-based education. The model defines a Content Aggregation Model (CAM) and a Run-time Environment for representing learning objects. The content aggregation model defines ways of identifying and aggregating resources into logically structured components of a learning object architecture. The run-time environment on the other hand proposes ways of communicating, launching and tracking content on the web. SCORM is a combined effort of institutions such as Advanced Distributed Learning (ADL), Alliance of Remote Instructional Authoring and Distribution Networks for Europe (ARIADNE), Aviation Industry CBT Committee (AICC), IEEE Learning Technology Standards Committee (LTSC), IMS Global Learning Consortium Inc among others.

The Content Aggregation Model [36] is composed of the three components – a Content Model-a nomenclature for identifying and defining the components of a learning entity, metadata-information to describe to the fullest extent possible the nature of the components of the resource and Content Packaging-A systematic approach to content representation and aggregation to

support mobility across domains. The Content Model for SCORM comprises of three sub-components/concepts named Assets, Sharable Content Objects and Content Aggregation. Entities defined as Assets are minuscule components such as web pages, documents with Asset metadata and can be involved in a knowledge providing experience independently. Sharable Content Objects (SCO) are collections of Assets that can be involved knowledge providing experiences upon launching a run-time environment supported by the Learning Management System (LMS). Such SCOs can then be mapped appropriately into an representation of learning resources (the process is called Content Aggregation). The later two (i.e. Metadata and Content Packaging are elaborated in chapter 3 dealing with XML, metadata and metadata standards). It is sufficient to realize for this discussion that metadata is information that better describes the digital entity in question and content packaging is a standard for aggregating sets of digital entities along with metadata into a package that is transferable across domains and that SCORM uses metadata and content packaging specifications proposed by IMS/IEEE-LTSC [13].

The Run-time Environment Model proposed by SCORM is based on the Data Model and the Communications API provided by the AICC. The model defines the need for common initiation techniques so learning resources can be platform independent and be able to communicate with an interoperable learning management system. The run-time environment model can be realized via the usage of browser initiated launch of communications between the learning object and the LMS. These communications result in data that is then launched onto the browser. The motive of this model is to make available durable, interoperable content which is easily searchable and re-usable. Durability is defined as the steadfastness and adaptability of a learning object to technological changes without the need for redesign/reconfiguration. Interoperability of content is defined as its ability to run on a wide-variety of operating platforms and browsers. Effective indexing techniques (made possible by the use of metadata) enable smart and quick searches while reusability is the ease of plucking components off an existing content structure and building a new structure with minimal data loss.

ADL's association with institutions such as the IEEE, IMS, ARIADNE, AICC and others helps it define a specification for web-based learning that can be ratified by the learning management industry. SCORM is an aggregation of the premier models and propositions of the partnering organizations. The content aggregation and packaging standard specified by IMS (and accepted widely by many learning organizations) is adopted by ADL to represent SCORM objects. This thesis will make an effort to map the Smart Object Concept (Bucket model) to gain partial conformance to the SCORM model we discussed here. On the other hand, the work reported in this thesis will help towards the possible induction of the Smart Object Model into the SCORM model in future releases.

Content Structure

Content Aggregation

Aggregation

Learning Resource

Aggregation

Learning Resource

Learning Resource

Learning Resource

Learning Resource

SCO

SCO

SCO

Asset

Asset

Asset

Asset

Asset

Asset

Asset

The SCORM Version 1.2 contains specialized instances of resources: Assets and SCOs.

Assets are learning content in its most basic form. Assets are electronic representations of media, text, images, sound, web pages, chat session, assessment objects or other pieces of data that can be delivered to a Web client.

SCOs represents a collection of one or more Assets and/or Sharable Resources that include a specific launchable asset that utilizes the SCORM Run-time Environment to communicate with LMSs. A SCO represents the lowest level of granularity of content that is able to be tracked by an LMS using the SCORM Run-time Environment.

Content Aggregation is the process of aggregating resources (SCOs/Assets) into a defined structure (content structure) to build a particular learning experience.

Figure 2.4. SCORM Content Aggregation Model

(Source: SCORM Content Aggregation Model, Oct 2001)

2.6 Smart Object Dumb Archive Model (SODA)

The Smart Object Dumb Archive model[26] resulted from the work done by Dr. Michael Nelson, Dr. Kurt Maly and co. at the Old Dominion University. The model resulted from the research carried out for NASA as part of the Networked Computer Science Technical Report Library Initiative (NCSTRL+). The work aimed at building a digital library for Scientific and Technical

Information (STI). The driving force for this research was the need for digital libraries to push the functionality provided at the repository level to the object level. This would create a user-centric environment for learning as opposed to a repository-centric architecture resulting in a richer learning experience for the user. The functionality embedded into the smart object would enable learning objects to act for themselves with little assistance from the repository. This de-coupling of digital objects from the repositories would diversify the scope for development in digital library services and object presentation and handling in an exclusive fashion.

The proposed model was an extension of the Kahn-Wilensky architecture we discussed at the beginning of this chapter since it still maintained the latter's definition of a digital object as containing data and metadata. Buckets (Smart Objects) are defined as self-contained, manageable, aggregate object-oriented container constructs containing logically grouped entities providing knowledge. Buckets have logically grouped components known as packages each of which contain logically mapped components known as elements. The learning experience provided by these entities is due to the presence of methods that handle functionality and presentation interfaces of a user. The proposed version of buckets implemented over 30 methods, essentially 30 ways of interacting with the learning object. Some of the functionality included in the buckets included methods to view the packages and elements of the learning object, methods to add or remove packages/elements to/from the learning object, methods for access control, methods to view/add metadata to the learning object among others. Methods and their specifics will be examined in Chapter 4 when we discuss further the smart object model and the extensions this thesis makes to it. Since buckets used http as the communication medium, communication and configuration is possible via the use of an internet browser. A schematic representation of a bucket is show in Figure 5 below. A bucket can comprise of files in heterogeneous formats, metadata represented as Bibliographical text, and access methods which are responsible for providing interactive user interfaces.

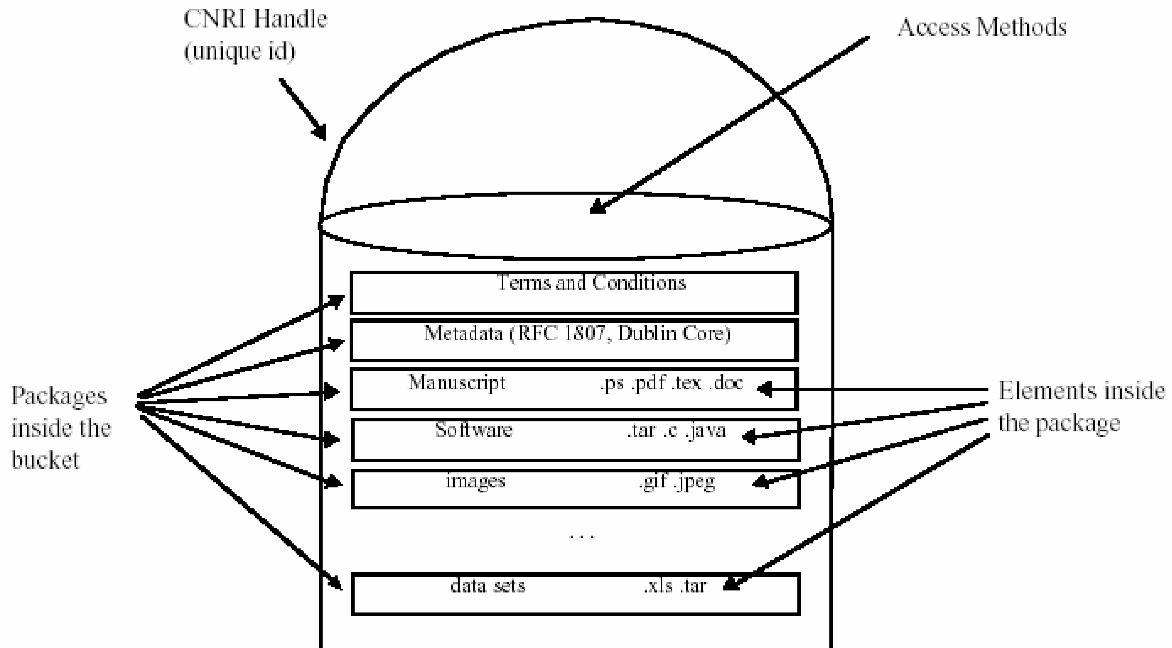

Figure 2.5. Bucket representation

(Source: Buckets: Smart Object for Digital Libraries, PhD dissertation by Dr. Michael Nelson)

Metadata for buckets was represented as RTF (Rich Text Format). The metadata representation was in a Biblio format [1]. Buckets by default were designed to be twin-layered, i.e. buckets supported a single level of packages and items within these packages. Buckets were implemented using Perl and tested on various platforms. Buckets also implement communication space. Bucket Communication Space (BCS) is the mechanism by which buckets communicate with each other via a BCS server. The model proposed a proof of concept implementation for file-conversion, metadata-conversion, bucket messaging and bucket matching.

The buckets described above had a huge scope for improvement. Buckets suffered from scalability issues due to the usage of text-based metadata [1]. An n-tier bucket would be difficult to realize due to the unstructured representation of metadata. Also, due to the heterogeneous nature of the learning resources (elements) in the bucket, the metadata would not be indicative of the true nature of the individual elements and diversity in metadata representation could not be realized. This thesis tries to address these issues by representing metadata in a popular metadata format, IMS [13, 14]. This will facilitate the realization of an n-tier smart object addressing the

scalability problem, the previous bucket versions suffered from. Also, as a result of adopting a standard as popular as IMS, we hope to create a model of the bucket that would be nearly compliant with SCORM specification gaining wide-spread approval in current and future implementations of digital libraries. In addition to exploit the full potential of the XML based metadata, we propose to support metadata representation for every package and/or element contained by the smart object. Also, we intend to provide an OAI provider interface to this smart object model for metadata harvesting by component digital libraries. Although such a feature is usually left to the repository implementation, the provision of an OAI interface would enable future versions of smart objects to communicate via the OAI interface resulting in a network of ambient-aware smart objects.

Chapter Three

Metadata

3.1 Overview

Metadata[24] is a reinvented buzzword in the Internet world. Deriving its origin from a Greek word "**µet?**" meaning alongside, metadata is a common term used in library management. Libraries for ages have been known to have cataloguing features to help users in identifying books and other library materials of interest. Cue cards have been known to contain author information, title, brief description, subject classification and even the floor/rack they can be found. The evolution of Internet saw the need for a digital cue card representation of descriptive information essential for cataloguing digital resources. Metadata[11] was then (re)defined in the digital paradigm. In its simplest form, metadata is defined as data about data.

To elaborate, metadata is an entity that provides descriptive information about an electronic resource. The nature of this descriptive information may vary with the context and environment of use. A library for instance would be interested in descriptive information like title, author information, keywords, abstract and classification information at the very least for a book or article while a GIS engineer may be interested in information regarding datasets, databases or geographic information system coverage associated with a Map[32]. Digital resources necessitate the use of various classes of metadata due to the variety in their very nature. Metadata can be descriptive, structural or administrative. Some simple assumptions regarding each of these requirements can help in determining the metadata layout for a resource. A media object (say an applet) demonstrating the operation of the FTP (File Transfer Protocol, RFC 959) will require metadata that reflects its richness and interactive educating approach (typically descriptive and administrative with a high degree of importance to descriptive information like interactivity type etc) as compared to a Request for Comments document detailing the operation of the FTP protocol which is more bound to be resource intensive, descriptive and structural in nature. Note that each of these metadata categories is granular so as to accommodate as many properties as possible, all of them either assisting directly in information discovery or in increasing educational value.

Metadata[6] creation and representation has been grouped into two categories, minimalist and structuralist. The minimalist version of metadata is a simple structured representation which is relatively easy to use even by an inexperienced person while the structuralist is a more complex, highly structured representation for use by trained professionals. This chapter discusses Dublin Core as an example of minimalist metadata representation and IMS as an example of structuralist representation. With all this in mind, it is important to remember that metadata is represented in a repository or a file system as a resource itself. Metadata for educational resources are represented either as flat formatted files or structured data sources adhering to well-known conventions and standards. Standards such as Dublin Core[38] and RDF which started off being represented as flat text files currently use the power of a structured markup language like XML.

3.2 Metadata as XML

Metadata has been around since the inception of libraries. Cue cards were used for cataloguing library resources (books, magazines etc.) to better assist in information discovery and retrieval. Note that the information discovery and retrieval in this context purports a user identifying a resource (s)he is interested in and retrieving it from the library's catalogues as compared to the system-aided electronic information discovery and retrieval mentioned elsewhere in this document. The growth of the Internet led to volumes of information made available across domains. Digital Libraries developed models for storage and dissemination of such large amounts of data across domains. These models made extensive use of metadata for representing descriptive information about resources that assisted directly in information discovery tasks. Such information encompassed a variety of traits and was initially stored in flat files.

The introduction of XML revolutionized the scope for metadata representation in digital environments. It provided an ideal means of data and information exchange across domains in a machine understandable fashion. It was so popular that efforts are being made to promote the use of XML in application messaging. Besides, XML being a more compatible markup language than HTML to SGML, it was readily accepted by publication societies using SGML for annotation purposes. This led to a wide acceptance rate for XML in learning environments and most metadata standards embraced XML. The more popular specifications like the Dublin Core Metadata Initiative (DCMI)[10] and IMS currently support a wide variety of metadata element-

sets and have recommended schema specifications to assist in automated machine readable, inter-domain, interoperable XML instances. So popular was XML that specifications for information interchange and metadata interchange across Learning Management Systems (LMS) received extremely good reaction and LMS' today are very much pushing for the acceptance of such open standards. This chapter will present three such XML-based specifications for metadata representation in Learning Environments. Later chapters will define their exact use in the Smart Object Model implementation presented by this work.

3.3 Dublin Core Metadata

The Dublin Core (DC) metadata standard was one of the first widely acceptable metadata standards that made possible a common, universally identifiable metadata representation for electronic resources. The groundwork for such a representation was initiated at an invitational Dublin Core Metadata Workshop, which gathered librarians, library researchers, mark-up experts, content experts and the like, a series which was started in 1995. The metadata specification was intended to serve as a means to ease information discovery and retrieval over the internet, providing information that can better enumerate the electronic resource in a learning context.

The Dublin Core metadata standard is one of the few that can now be used for resource description inside and outside of a resource. It can be embedded into a hypertext markup resource or be tagged along with a non-markup resource such as a portable document format (PDF) file[12]. Most search engines nowadays are tuned to recognize DC, RDF and generic markup metadata in web-based resources. The 15-element of set descriptors was intended to be conformant to commonly understood semantics, conformant to existing and upcoming technologies and standards for resource description, interoperable and extensible and most importantly simple to create and maintain. Simplicity is one feature that supports the wide-spread usage of DC metadata in some cases and opposes to its usage in some others. Since DC proposes a relatively small element set to describe a resource, it is very commonly used in describing electronic learning resources. The reason as we have already mentioned is simplicity. Descriptive information about a resource is easy to gather and much easier to represent. Also, the element set defined by DC is closer to natural language terminology used to provide descriptive information

about a resource which makes it a favorite of non-technical user groups. A brief summary of DC element set follows:

The 15-element set of descriptors proposed by the Dublin Core metadata initiative can be scoped into three groups to define the learning information of an electronic resource. These groups intend to expose (1) content related information that will assist in information discovery and retrieval, (2) intellectual and ownership related information to complement the resource and (3) generic information pertaining to initialization and current form of existence of the resource. Table 3.1 categorizes the Dublin Core element-set into these groups.

Content	Intellectual Property	Instantiation
Title Subject Description Type Source Relation Coverage	Creator Publisher Contributor Rights	Date Format Identifier Language

Table 3.1. Dublin Core Metadata Elements

As illustrated above, properties such as Title, Description, Subject of classification etc. which are directly reflective of the resource are classified as information relating to the Content, while properties such as Creator, Rights etc. reflective of intellectual information are classified as information relating to Intellectual Property while other information such as Identifier, Language etc. are classified as Instantiation information. It is also evident that these properties require little explanation as to what they represent and hence increase usage by non-technical community. The evolution of XML and other markup languages has led to widespread usage of metadata practices in digital libraries. DC metadata was one of the first to be adopted for use of representation using XML.

That said it is also widely accepted that DC has faced increased competition from emerging standards (the primary competitor being IMS) in aspects such as granularity and aggregation. The IMS specification discussed later in this chapter demonstrates a practical solution to

representing metadata with a high degree of granularity. While DC is comprised of a 15-element specification for learning resource description, the IMS currently has an 86-element specification. The DC specification also lacks a model for learning resource aggregation like IMS. Since most of these specifications are in an evolutionary stage, it leaves much scope for improvements and changes to the specification to make amends for such inadequacies while highlighting its strength – simplicity. Further information about the specification and/or definition of the metadata elements in Table 3.1 can be obtained from http://www.dublincore.org/.

In context with this work, the Dublin Core metadata standard will be used to render metadata via the OAI provider interface provided with the Smart Object proposed. The unqualified Dublin Core is one of the three metadata formats supported currently by this model.

3.3.1 Dublin Core with DC-Ed Extensions

The Dublin Core Educational workgroup was formed as a result of the Dublin Core Metadata Initiative's (DCMI)[10] intent to make Dublin Core adaptable to use in Educational Learning Environments. Towards this goal the workgroup reviewed existing digital libraries learning environments, metadata standards in existence and popular descriptive information types for educational resources.

The workgroup found the use of element sets in metadata standards like IMS to describe the targeted audience for a resource. The workgroup hence called for the induction of a new element called 'audience' with dc-ed namespace extension and another element 'mediator' to further granulize the 'audience' element. Another proposal made by the workgroup helped identify the standards referenced by the resources as part of the metadata. The DCMI usage board recommended the use of a qualifier 'conformsTo' for representing standards that may be referenced by the resource in question. A significant third recommendation that resulted from the DC Educational workgroup was the endorsement of three elements from IMS/IEEE LOM namespace. It endorsed the use of 'typicallearningtime', 'interactivitylevel' and 'interactivitytype' from the IEEE LOM in representing approximate learning time, level and type of interactivity of a resource in representing DC Educational metadata. These recommendations are part of this works OAI provider interface and are hence worth a mention.

3.4 IMS Specifications

The nineties saw many emerging metadata standards in view of the ever changing scope for learning technologies. The IMS Global Learning Consortium Inc. proposed one such metadata specification to explore a highly granular approach to descriptive information about learning resources (both electronic and non-electronic) and another for information exchange across domains popularly known as IMS Content Packaging Specification[13]. The IMS specification unlike the already existing metadata specifications provided a structured approach to metadata description. Two IMS specifications of interest are discussed in this section, the IMS/IEEE LOM Specification[15] for learning resources and the IMS Content Packaging Specification. Both specifications are key components in the design of Smart Object Model discussed in Chapter 4. A brief description of each specification follows:

3.4.1 IMS/IEEE Learning Object Metadata Specification

IMS proposed a metadata specification to develop and promote open specifications to aid distributed learning activities across domains including assisting in electronic information discovery and retrieval and interoperability with other known standards. The exhaustive solution was a direct result of an alliance with the IEEE Learning Technologies Standards Committee (IEEE-LTSC) Learning Object Metadata (LOM) workgroup. This LOM specification (now final draft standard 1484.12.25002) is currently used in the IMS Learning Object Metadata Specification ver.1.2.1 along with IMS approved modifications to the IEEE LOM. Hence, the two part specification covers the IEEE LOM in the first part whilst dealing with the IMS certified modifications to the LOM specification. Throughout this document, the term 'IMS metadata' in a static context and the term 'metadata' whilst describing the Smart Object Model is intended to refer to this specification unless specified otherwise.

IMS metadata[14] provides an extremely rich set of descriptors to capture learning resource information. It provides an 86-element descriptor set for representing learning object metadata. Although this document makes references to IMS metadata extensively over the next few chapters, it is beyond the scope of this document to discuss the complete specification. However, it will make an effort to highlight and enumerate the categories of metadata elements and some critical metadata elements.

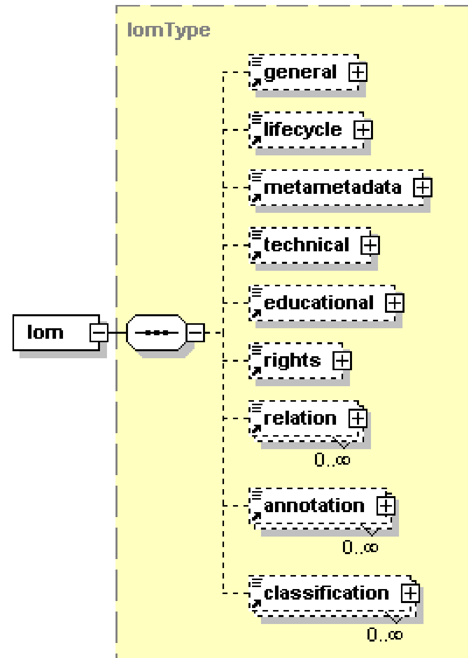

Figure 3.1. IMS Learning Object Metadata Model

IMS metadata for learning resource is designed to provide descriptive information about 9 categories (or properties) each sub-categorized into a number of properties themselves. Figure 3.1 depicts the XML bindings of these categories. A brief description of each category follows:

General: This category provides information that describes the resource(s) as a whole. Such information includes a generic title to represent the resource(s), descriptive information (abstract), keywords for identification, domain-specific identifiers to uniquely identify the resource(s) and others. A key component on representing learning object metadata, this category can be used in information retrieval and discovery mechanisms.

Lifecycle: This category provides information regarding domain-specific statuses and more importantly details of all persons/entities responsible for contributing to the resource. Such contributors include but are not limited to authors/creators of the resource(s) itself, metadata creators etc.

Metametadata: This category provides information regarding the metadata itself rather than the resource. This information can be used for differential treatment of metadata from various schemes.

Technical: The technical category enriches information relating to the format of a resource, disk size (if electronic), requirements for usage and any platform specific requirements to perceive the learning knowledge offered by the resource.

Educational: Educational category in IMS metadata specification is intended to represent pedagogical information regarding the resource. Such information would assist in educating the user of a resource's learning duration, intended end user, interactivity level etc. Although such information would seldom be used in resource discovery, it can provide valuable information resulting in better filtering by a user.

Rights: This category provides information about the copyrights issues attached to a resource's usage. Such properties include cost related information if a business model encompasses the resource.

Relation: This category intends to define a relation between a resource and other targeted resources.

Annotation: The category provides annotative information regarding the educational usage of a resource. It may also provide detailed information about the contributor of the annotation.

Classification: The classification category provides information that can be used for cataloguing purposes according to subject matter and area.

Given the comprehensiveness of this specification, it is believed that sufficient descriptive information regarding a resource can be gathered to adequately represent Smart Object metadata. This version of IMS metadata is used for the representation of learning object metadata at the object, package and element level. These components and their roles in the Smart Object Model are discussed in the following chapters.

3.4.2 IMS Content Packaging Specification

The IMS metadata specification[14] discussed above was effective to the extent that it highlighted a wide-range of characteristics/properties of a resource that would better assist in information discovery and in perceiving the learning value of a resource. Electronic resources presented with a new complexity in perceiving this definition. An electronic learning object unlike its library counterpart may comprise of multiple resources with dissimilar characteristics not to mention different file system representation and size. In such cases a metadata scheme to

accommodate Content Aggregation[13] is desired. Content Aggregation to enumerate further is a mechanism whereby sets of logically or physically related electronic resources are grouped together in some sensible fashion to create a larger resource providing greater learning value. Such an aggregation was also necessary for inter-LMS (inter-Learning Management Systems) distribution of electronic resources without a loss of accompanying metadata.

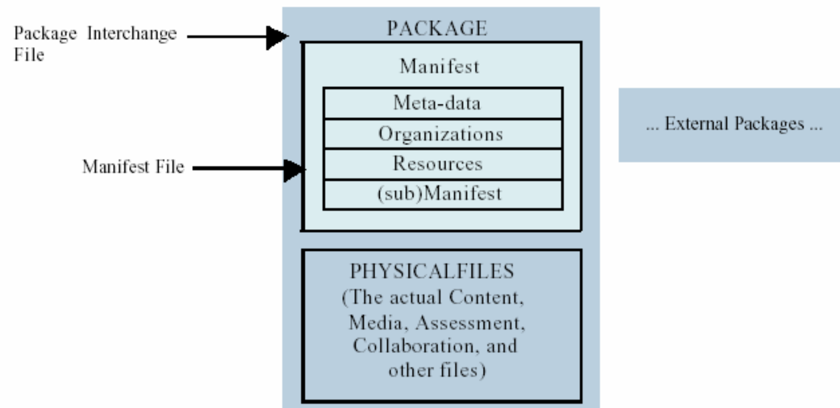

Figure 3.2. IMS Content Package*

* - Source: IMS Content Packaging Specification

The IMS Content Packaging Specification was proposed to provide efficient aggregation, distribution, management and deployment of electronic resource with embedded learning value. The standard like the metadata scheme was intended to be interoperable with other existing and emerging standards. The specification made use of the IMS/IEEE metadata schema for representing learning object and resource metadata. Resource metadata here means metadata pertaining to a resource while learning object metadata is aggregate pertaining to the resource in its entirety.

An IMS package as defined by the specification comprises of an XML file conformant to a particular schema and knowledge providing physical resource files. The instance of the XML file is a key component in the specification. This file (name *imsmanifest.xml*) provides information that would assist an LMS in gathering organizational information of the resource, metadata for each of the knowledge providing components and metadata for the package. An LMS can then use this information in providing a learner with an appropriate learning experience. The specification only provides a mechanism for aggregating logical grouped resources for use by an

LMS and not the actual mechanism or environment for providing learning experience. A manifest instance has three primary components that an LMS can exploit, a metadata section detailing package information, an organizations section providing structural information about the resource(s) and a resources section containing file system representation of the resources. A pictographic representation of the manifest instance is shown in figure 3.3.

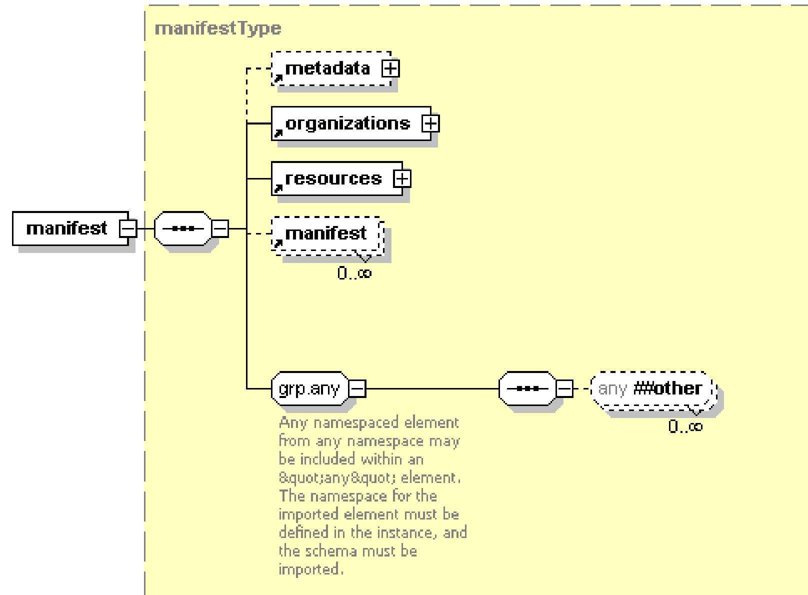

Figure 3.3. IMS Content Packaging XML Binding

A more detailed description of each component in provided in Appendix (will be appended later). This work uses a modified version of specification version 1.1.2, the latest in the series of packaging specifications and version 1.2.1 of metadata specifications to define package and element metadata.

Chapter Four

Smart Objects

4.1 Overview

A digital library's operational effectiveness is determined by its ability to formulate dynamic learning environments[29]. Formulating such learning environments is dependent on a number of factors including the flexibility of the library's architecture and the learning object model among others. Current digital library architectures struggle to provide such a dynamic environment mainly because of the static nature of learning content they serve. Even when learning objects are designed to contain interactive content, their effectiveness suffers with time. This disability cannot be overcome easily as it requires a library to support content enrichment operations (eg. version control). Also, the tightly bound relationship between a Learning Management System [4] and its content object model leave no room for independent development. The smart object model developed here (a direct extension of the bucket model proposed in the SODA model [26]) tries to overcome this difficulty by detaching the object model from its governing library's design and equipping the learning object with software modules that endow the object a degree of intelligence and self-sufficiency. The smart object modeled here is a platform-independent, scalable and specification-conformant implementation of learning object with intelligence. A detailed description of the model follows:

4.2 Definition

A Smart Object is a structured aggregation of learning resources and the associated metadata encapsulated by a set of methods that provide intelligence, self-sufficiency and platform independence while facilitating pedagogy in and outside the scope of operation of a digital library. It provides user-friendly interfaces to a user seeking a learning experience as well as to an administrative authority to manipulate its behavior remotely.

The above defined learning object model differs from the ones previously discussed (chapter 2) in numerous ways. Unlike them, it is an active object that is self-sufficient in that it can operate without any assistance from a digital library. In addition, it provides interfaces for serving itself to a learner seeking knowledge, as well as to an administrator performing managerial operations over it, all via a web interface. Action requests and responses are exchanged over HTTP.

4.3 Characteristics and Requirements

As defined above, smart objects are aggregated, intelligent, platform independent, object-oriented container constructs that can be used to disseminate knowledge in the presence/absence of a digital library. These characteristics together make smart objects self-sufficient and are discussed in further detail:

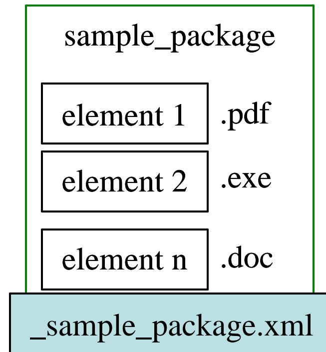

Figure 4.1. Aggregation Example

4.3.1 Aggregation

Typical learning objects are comprised of multiple sets of items that provide a user with an interactive learning experience. A course being modeled as a learning object for example may be comprised of discrete components with their own set of learning objectives and characteristics. It may have lecture slides as portable documents, audio/video clips demonstrating interesting examples, suggested readings as hypertext documents, examinations as word documents, interactive exercises as Java applets and so on. Based on the course structure, the components can be grouped logically into a set of modules. Such a grouping could involve aggregating items with similar characteristics (based on format and objective) or alternately based on time (weekly or monthly classification). The smart object model proposed here makes possible such a grouping of learning items. The smallest learning entities with a learning value are termed *elements* while groups of elements are aggregated to form *packages*. A smart learning object can have any number of elements and sub-packages contained in packages. A detailed file system representation of this notation is given in section 4.4.1.

4.3.2 Intelligence

A common characteristic of existing learning object models used in digital learning is the high degree of dependence of the object on the guardian library. The digital library is responsible for

any and all actions/features supported by the learning object, from its introduction into the library's content repository through the archival and post archival stages. The responsibility of the library is more significant during the post-archival stages though. Any changes to the learning object or its metadata intended to enrich the contents of the object itself or its metadata should be supported by the library. In most cases this may not be supported thus rendering the learning object immutable. Such learning objects would thereby remain passive entities with no possible way of modifying/enriching their contents over time rendering them ineffective in learning environments. To prevent such circumstances, smart objects have been designed to encapsulate learning items using software components (*methods*) that provide *intelligence* to the learning object. This intelligence at the very least equips the learning object with interfaces to serve itself to a learner. In addition, the functionality could encompass actions that include manipulating and changing the operation/contents of the learning object. To elaborate, interfaces allow users to experience a learning process, with the learning object presenting a dynamic learning path based on a user's interests. Metadata of elements and packages assist the user in this process. Further, the methods serve to manipulate the operation and structure of the learning object which is now independent of the functionality supported by a library. Such operations include but are not limited to adding/removing items from the learning object, manipulating metadata of elements and packages and maintaining access control lists. By providing the ability to structurally modify the learning object after creation/deployment, smart objects address the version control issue that is currently dependent on a library's functionality. The initial release of the smart object allows 20 actions to be performed on it (discussed in detail in section 4.4.5).

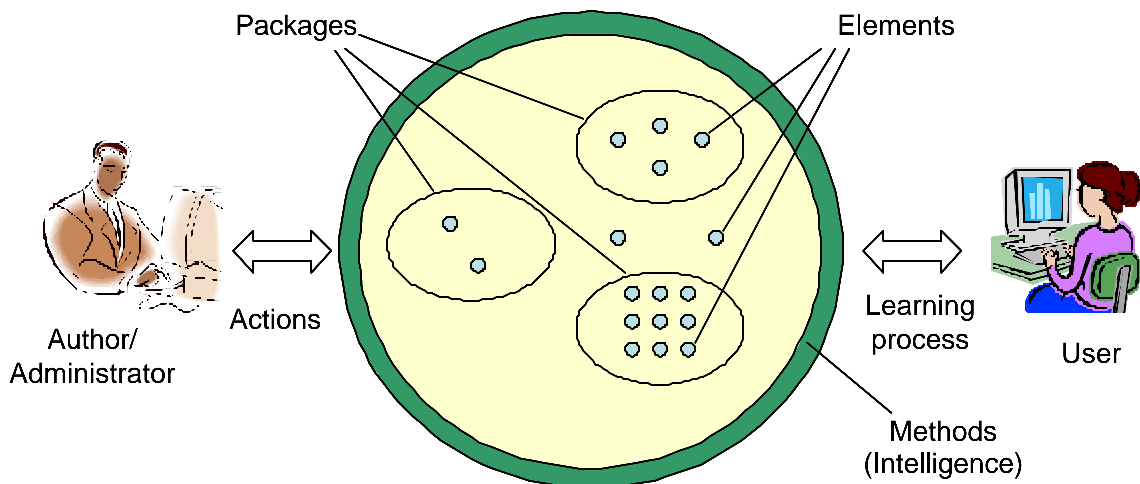

Figure 4.2. Learning Object with intelligence

4.3.3 Object-oriented

Every smart object is identified by an identifier, which may be a global value such as a CNRI handle[5] or a locally defined value. In addition to the identifier, a smart object contains information about learning items and packages within it. These two properties signify the state of the learning object. In addition, the smart object is equipped with methods that allow actions to be performed on it. These actions represent the behavioral characteristics of the smart object. State and behavior are two essential properties for an object. Since smart objects are equipped with both, they can be treated as object-oriented entities.

4.3.4 Platform Independent

Since traditional learning objects are passive, they do not interactively respond to user requests over the web. The guarding entity (digital library) is usually responsible for supporting all such activities. However, since smart objects are active entities independent of the functionalities provided by a library, they would require an operating environment that would allow them to process user requests. Since smart objects are expected to operate in a repository independent environment, design approaches require a platform independent container to hold them. To meet this requirement smart objects have been modeled using Sun's Java 2 Enterprise Edition (J2EE)[16] and Standard Edition (J2SE)[17] distributions. Since Java is an interpreted language, its operation is not constrained to the platform so long as an interpreter is available. In addition, smart objects require a web container that can host them and provide a computing environment. Details about deployment and building smart objects are dealt with in a later section.

4.3.5 Ease of adoption and conformance

Another key characteristic of a learning object's design that determines its acceptability for use in digital libraries is its conformance to currently existing specifications. The model should be such that an existing digital library should be able to adopt it without a major overhaul to its existing architecture. The smart object model developed here conforms to the IMS-IEEE Learning Object Metadata (LOM) v1.0 specification. The content packaging specification used here is an extension of the IMS Content Packaging Specification v1.1.2 while metadata scheme used is IMS-MD1.2.1.

4.4 Smart Object Architecture

The smart object model demonstrates an object-oriented approach to representing learning objects with intelligence. Smart objects allow for aggregation via the package/element concept. The model defines the smallest learning entity as an element, while groups of these elements can be ordered to form packages. Every smart object can have any number of elements and/or packages. The internal operation of the smart object has no bearing on the structure or method of aggregation. The smart object however operates under an internal assumption that the parent directory of the smart object is called the root package and contains certain restricted packages and elements. By default, sub-packages and elements are added to the root package if no package is specified. The root package is however not listed or available for manipulation. By design a package can have any number of sub-packages and elements within it. A major improvement of this model over the bucket model discussed in chapter 2 is scalability. The Smart Object Model proposed here is scalable in that an n-level grouping is allowed for while the bucket model could only support a two-level representation. In addition XML has been used to represent metadata and structuring information of the smart object. A discussion of smart object design follows:

4.4.1 File Structure

The smart object's file system representation is based on the package/element theory presented above. Every element of the smart object (excluding URLs) is mapped to a physical representation (a file) in the file system. An aggregation of elements representing a package is mapped to a directory. While there can be virtual elements (URLs or pointers to other smart objects), packages are not so.

Figure 4.3 shows the file system structure of a sample smart object. In addition to the custom made packages and elements, every smart object contains two reserved packages (*_sysFiles* and *WEB-INF*) and two reserved elements (*index.jsp* and *imsmanifest.xml*) necessary for its operation. A brief description of these components follows:

_sysFiles: This is the first of two mandatory packages required for smooth operation of a smart object. The contents of this package are a set of elements which present interactive interfaces to assist a user in the learning process or in modifying the structure of the learning object. To assist in the learning process, the reserved elements present interactive interfaces displaying pedagogical information and structuring information leading to a learning entity,

i.e use of metadata and structuring information to help identify a learning item accurately. For modifying the structure of the object, the elements present interfaces using which an administrator can change the structure of the resource, perform aggregative operations, add/remove components and metadata and perform monitorial functions. In addition, a sub-package (*_logs*) is used to collect visitor information which includes remote host information, request and response information. An additional sub-package (*_methods*) stores the source code for all the actions and core modules supported by the smart object.

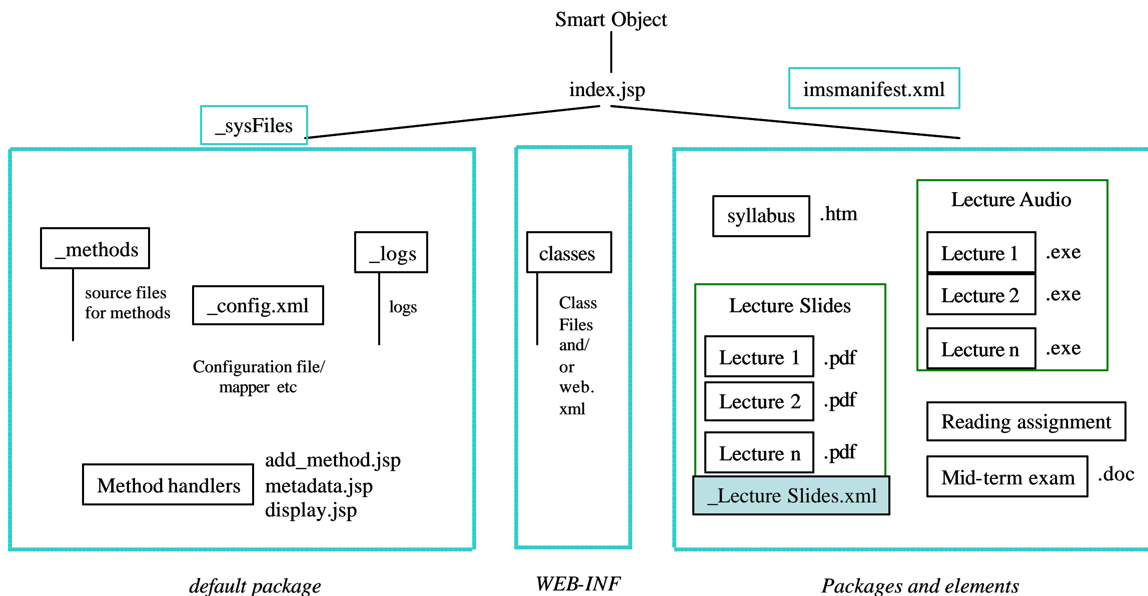

Figure 4.3. File System representation of a smart object

In addition to these reserved sub-packages, the _sysFiles package contains a number of reserved elements. Most of these are dynamically processed server pages (Java Server Pages, JSP)[21] and provide interactive interfaces for triggering appropriate modules in the *WEB-INF* package. The most important element of this package though is the *_config.xml* instance. This is responsible for maintaining the structure and aggregation details of the learning object in addition to detailed method mappings that are key to providing self-sufficiency to the object. All decisions (user and administrative) are made by the software modules based on the entries of this file. Further details of this element are provided in 4.4.3. The package also contains the user repository for the smart object (*_users.xml*) and is detailed in 4.4.4.

WEB-INF: This package holds the core modules that provide a variety of functionality to the learning object. Every action supported by the smart object maps to a java *class* file that is

responsible for handling the requests. In addition, the learning object is equipped with a set of core modules which allow the action modules to access components, their metadata and modify the structure of the object. These core modules include a highly flexible XML parser utility used for performing pattern matching and retrieval operations on XML instances, a package interface module capable of recording changes to structure and arrangement of components (adding/removing components and permissions), a manifest interface module capable of updating the manifest instance (records the structure and metadata details of learning components) and a metadata interface module capable of creating/modifying package/element metadata. In short, the WEB-INF contains the tools needed to add intelligence to a dumb learning object, a.k.a create a smart object.

index.jsp: This is the entry-point to the smart object and is responsible for forwarding HTTP requests to appropriate modules. This is the only element via which actions can be performed on the smart object. To elaborate, an HTTP request is constructed by the user and sent to this element to be forwarded to a reserved package that handles the request. The HTTP request indicates the action to be performed which in turn is mapped to the appropriate module that handles such requests. For example, a user wishing to view the contents of the smart object would query the smart object via a URL:

http://foobar.edu/some_url_path/index.jsp?method=display&package=

The above URL instructs the smart object to call the method *display* for a package '' (root package). Following the same pattern, the URL:

http://foobar.edu/some_url_path/index.jsp?
method=display&package=readings.book&element=chapter1

would display the contents of an element named *chapter1* in a package *book* which in turn is part of a package *readings*. Note that it is not required or expected for a user to have knowledge of the above URL. (S)he will be guided to this element via a set of user-friendly interfaces after accessing the root package. An example of such an interface is explained in section 4.5.1.

imsmanifest.xml: The manifest instance stores metadata related to every package and element of the smart object. In addition it contains a file system map of all the elements that are part of this smart object. The manifest instance is the mandatory component of an IMS content package and can be used by a Learning Management System (LMS)[21] to realize the structuring and aggregation details of a learning object. Since current LMS designs are

not aware of the smart model, organizational information is obtained via the manifest instance and not the configuration instance (4.4.3). A typical manifest instance would comprise of three parts, a metadata component to represent metadata of the entire learning object, an organizations component indicating structure and aggregation information and a resources component containing file system mapping to all items that make up the learning object. A sample manifest instance is shown in figure 4.4. The metadata for every component is conformant to the IMS metadata specification version 1.2.1 and is detailed in section 4.4.2.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- edited with XMLSPY v5 U (http://www.xmlspy.com) by Prashanth (ARI) -->
<manifest identifier="" xmlns="http://www.imsproject.org/xsd/imscp_rootv1p1p2"
xmlns:imsmd="http://www.imsglobal.org/xsd/imsmd_rootv1p2p1"
xmlns:vc="http://www.dlnet.vt.edu/xsd/vCardv1p0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.imsproject.org/xsd/imscp_rootv1p1p2 imscp_rootv1p1p2.xsd
http://www.dlnet.vt.edu/xsd/dlnmd_extv1p2p1  dlnmd_extv1p2p1.xsd
http://www.dlnet.vt.edu/xsd/vCardv1p0 vCardv1p0.xsd
http://www.imsglobal.org/xsd/imsmd_rootv1p2p1 imsmd_rootv1p2p1.xsd">
    <metadata>
    ............................................................
    </metadata>
    <organizations>
        <organization identifier="pkg1">
            <metadata>
            ..............................................
            </metadata>
            <title>Text Book</title>
            <item identifier="element" identifierref="pkg1.elem1">
                <metadata>
                ........................................
                </metadata>
            </item>
            <item identifier="element" identifierref="pkg1.elem2">
                <metadata>
                .......................................
                </metadata>
            </item>
        </organization>
    </organizations>
    <resources>
        <resource href="/book/chapter01.pdf" identifier="pkg1.elem1" type="webcontent">
            <file href="/book/chapter01.pdf"/>
        </resource>
        <resource href="/book/Chapter3.pdf" identifier="pkg1.elem2" type="webcontent">
            <file href="/book/Chapter3.pdf"/>
        </resource>
    </resources>
</manifest>
```

Figure 4.4. Sample manifest instance

```xml
<metadata>
    <schema>http://www.imsglobal.org/xsd/imsmd_rootv1p2p1</schema>
    <schemaversion>1.2:1.1 IMS:MD1.2</schemaversion>
    <imsmd:lom>
        <imsmd:general>
            <imsmd:identifier>Sample ID</imsmd:identifier>
            <imsmd:title>
                <imsmd:langstring xml:lang="en-US">ECE 5364:Electrical Energy &amp; Environmental Systems</imsmd:langstring>
            </imsmd:title>
            <imsmd:catalogentry>
                <imsmd:catalog>DLNET - Smart Object</imsmd:catalog>
                <imsmd:entry>
                    <imsmd:langstring>Sample ID</imsmd:langstring>
                </imsmd:entry>
            </imsmd:catalogentry>
            <imsmd:language>en-US</imsmd:language>
            <imsmd:description>
                <imsmd:langstring xml:lang="en-US">This course on Electrical Energy and Environmental Systems
                  discusses the impact of electricity generated from fossil and nuclear fuels, and renewable
                  resources. It discusses the impact of high voltage transmission lines and health effects of
                  electricity generation. It assesses cogeneration cycles and demand side management. It examines
                  emission control in the US electric utility industryand further evaluates uncertainties in
                  quantifying emissions impacts.</imsmd:langstring>
            </imsmd:description>
            <imsmd:keyword>
                <imsmd:langstring>Electrical Energy and Environmental Systems</imsmd:langstring>
            </imsmd:keyword>
            <imsmd:keyword>
                <imsmd:langstring> Demand Side Management</imsmd:langstring>
            </imsmd:keyword>
            <imsmd:structure>
                <imsmd:source>
                    <imsmd:langstring xml:lang="x-none">LOMv1.0</imsmd:langstring>
                </imsmd:source>
                <imsmd:value>
                    <imsmd:langstring xml:lang="x-none">Hierarchical</imsmd:langstring>
                </imsmd:value>
            </imsmd:structure>
            <imsmd:aggregationlevel>
                <imsmd:source>
                    <imsmd:langstring xml:lang="x-none">LOMv1.0</imsmd:langstring>
                </imsmd:source>
```


Figure 4.5. A sample metadata instance

4.4.2 Metadata Storage

To aid in information dissemination, search, retrieval and pedagogy learning resources are retrofitted with metadata. Metadata also serves to describe complicated data items which do not support text-based metadata harvesting. One example of such a scenario would be a streaming audio/video file with no parsable information that helps in pedagogy. Elements and packages of the smart object are fitted with metadata which provides upfront information about the component being accessed. It provides a user with sufficient information to assess its learning value without actually viewing it. This information typically comprises of but is not limited to the title, keywords, description, ID, author, contact information, format, typical learning time, intended user, subject area classification etc. To document this information with maximum

granularity, the IMS metadata specification v1.2.1 is used for smart objects. The current version of the model uses about half of the 84 element scheme is used to describe a resource with a limited sub-set (title, description, keywords, author information, format and classification) being defined as mandatory. Metadata for a package is representative of the aggregation of elements within it while that of an element is representative of itself. The metadata interface is responsible for retrieving structured textual metadata from the instance as well as generating the document object model (DOM) representation. Currently, this metadata is appended appropriately to the manifest instance (*imsmanifest.xml*) and can be extended to dissociate the metadata instance from it. A sample metadata instance showing snippets of metadata for a smart object is shown in figure 4.5.

Metadata for an element or an aggregated package can be set via the *add_metadata* method. The manipulated/new metadata obtained from a user replaces any existing metadata for that component. Existing metadata can be cleared via the *delete_metadata* method. Note that both these methods require modify permissions on the targeted component and hence are encapsulated by an authentication module.

4.4.3 Smart Object Configuration

The complex structure and aggregation information of the smart object is represented in a configuration instance located in the reserved package (*_sysFiles*) of the root directory. This instance contains information that is used only by the smart object to perform various internal operations. The entire operation of the smart object is governed by the structuring of this file. It contains information regarding the location of the user repository and the manifest instance (*ConfigFiles* tag in figure 4.6). In addition, the file also contains information about the content aggregation scheme and access control. As discussed before, a smart object can have zero or more standard packages each containing zero or more elements. Packages under the root package are documented under the xml tag *PackageDetails* while elements appear under *ElementDetails*. A sample configuration instance is shown in figure 4.6. Conforming to the definition, a package can comprise of sub-packages (that appear under the sub-tag *PackageDetails*) or elements (that appear under the sub-tag *ElementDetails*).

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<SmartObject>
    <Configuration>
        <ConfigFiles>
            <UserRepository>_users.xml</UserRepository>
            <Manifest>../imsmanifest.xml</Manifest>
        </ConfigFiles>
        <version>1.0.1.J</version>
        <WorkingDir>C:\dlnet\MyWork\Thesis\smart_objects\ece5364</WorkingDir>
        <RootAccess>
            <access>.......................</access>
        </RootAccess>
        <ElementDetails>
            <Element id="" name="elem1">
                <access>......................</access>
                <description>Chapter 1</description>
                <displayname>Chapter 1</displayname>
                <localname>drop_list.txt</localname>
            </Element>
        </ElementDetails>
        <PackageDetails>
            <Package id="" name="pkg1">
                <access>......................</access>
                <description>This folder contains an electronic copy of the book for the course</description>
                <displayname>Course Text</displayname>
                <localname>book</localname>
                <ElementDetails>.........................</ElementDetails>
                <PackageDetails/>
            </Package>
        </PackageDetails>
        <SmartObjectMap>
            <Methods>
                <Method name="add_package">
                    <forwardURI>SOsysFiles/addPackage.jsp</forwardURI>
                </Method>
                <Method name="add_method">
                    <forwardURI>SOsysFiles/addMethod.jsp</forwardURI>
                </Method>
                ................................
            </Methods>
        </SmartObjectMap>
    </Configuration>
</SmartObject>
```

Figure 4.6. A sample configuration instance

Every package or element also has three properties, a name, a localname and a diplayname. The name is an attribute to the package or element and is a unique identifier for that component throughout the smart object. For example, figure 4.6 shows one package with the name pkg1 and is unique with respect to the root package throughout the object. To exemplify further, pkg1 may contain another package by the name pkg1 and yet both of them are unique packages because the one in the root package is identified as pkg1 while the sub-package is identified as pkg1.pkg1. Packages are separated by using a package separator '.'. Hence, a package pkg1.pkg2.pkg3 would indicate that pkg1 has a sub-package by the name pkg2 which in turn has a sub-package

by the name pkg3. The other two properties *localname* and *displayname* represent the local file system mapping and a screen display name for a component. The package interface module is responsible for modifications to this xml file. Additionally, this instance also contains information to map supported methods to appropriate elements in this reserved package. For example, the figure shows two methods supported by the smart object, *add_package* and *add_method*. The reserved element *index.jsp* (discussed in 4.4.1) accesses the package interface module which informs it of two supported methods and the respective reserved elements that handle requests to these methods. If an add_package request was received by index.jsp, it forwards the request to _sysFiles/addPackage.jsp to be handled. This ensures dynamic configuration of the learning object as the configuration instance can be changed during the operational period of the object which would enable/disable a method without having to redevelop the learning object or its methods. The configuration instance also contains information which controls access to certain resources of the smart object to authorized/authenticated users only. The package interface has methods that support access control. A discussion of the user repository and access controls features supported by the object follows.

4.4.4 User Repository and Permissions

Every smart object is equipped with its own user repository containing information about users permitted to access the object. The user repository can be used to define an id-based access control scheme on the components of the smart object. It is an XML instance (*_users.xml*) containing a list of username-password combinations and user-group classifications. A sample instance is shown in Figure 4.4. Every user is identified via a username and a corresponding password which is usually represented as an *md5*[35] digest. The current implementation of the smart object supports three types of user-groups, '*Administrator*', '*User*' and '*Guest*'. Access to all restricted packages/elements and methods is limited to the Administrator group. Future versions on the smart object are expected to support access control based on user-groups and a wider set of user-groups.

Smart objects also support access control on every package and element via the *_config.xml* instance. Package and element permissions can be modified by a member of the 'Administrator' group. The current implementation supports three access control schemes and is based on the

userid scheme. A user can have READ, WRITE and/or MODIFY permissions on any given element or package. A user can be allowed or denied permission to perform any of these actions on every package or element. Figure 4.5 shows a sample permissions snippet for the element named *elem1* from the configuration instance discussed in 4.4.3. The example indicates that the element is accessible by two users in the user repository namely Administrator and guest. It also specifies that the user Administrator has read, write and modify permissions for this element while the user guest has read permissions for this element. Read permission grants a user access to the resource via the *display* method. Write permission allows a user to write to an element/package. In case of a package, write permission allows the user to add an element or sub-package to it. Finally, modify permission allows a user to make changes (delete) to a package or element. The security model for the smart object presented here is primitive and is expected to be much more comprehensive and robust in future releases.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<userconfiguration>
    <authentication>
        <user>
            <name>Administrator</name>
            <password digest="md5">7e9992fe0f629fa458a1b95f647992d9</password>
            <type>Administrator</type>
        </user>
        <user>
            <name>Prashanth</name>
            <password digest="md5">7e9992fe0f629fa458a1b95f647992d9</password>
            <type>Administrator</type>
        </user>
        <user>
            <name>guest</name>
            <password digest="MD5">b32d73e56ec99bc5ec8f83871cde708a</password>
            <type>Guest</type>
        </user>
        <user>
            <name>sampleuser</name>
            <password digest="MD5">5e8ff9bf55ba3508199d22e984129be6</password>
            <type>User</type>
        </user>
    </authentication>
</userconfiguration>
```

Figure 4.7. Sample user repository instance

```
<Element id="" name="elem1">
    <access>
        <username>Administrator</username>
        <permissions>
            <allow>
                <permission>READ</permission>
                <permission>WRITE</permission>
                <permission>MODIFY</permission>
            </allow>
            <deny />
        </permissions>
    </access>
    <access>
        <username>guest</username>
        <permissions>
            <allow>
                <permission>READ</permission>
            </allow>
            <deny>
                <permission>WRITE</permission>
                <permission>MODIFY</permission>
            </deny>
        </permissions>
    </access>
    <description>Chapter 1</description>
    <displayname>Chapter 1</displayname>
    <localname>drop_list.txt</localname>
</Element>
```

Figure 4.8. Element permissions example

4.5 Smart Object Methods

Smart objects bring to light features that most other learning objects do not support. Equipped with software modules, they display self-sufficiency features lacking in traditional learning object models. Self-sufficiency comes to them via methods. Smart object use HTTP to

communicate with users and provide interfaces and access control features to access and modify the learning object. Requests are piped through a single element (*index.jsp*) which maps them to existing method handling elements. This mapping is done based on the *method* parameter of the HTTP request. The current release of the java-based smart objects supports 20 methods. A brief discussion of these methods follows. The example listed below assumes that a smart object is currently deployed at http://foobar.edu/some_path/ and is being queried by a user/administrator.

4.5.1. Display

The *display* method is the most basic and perhaps the most important of all the methods. It is the default method of the learning object and is invoked in no method parameter is specified. The display method is responsible for providing a dynamically generated interface containing metadata information. In addition, the display method provides a listing of sub-packages/elements if a package is being displayed and alternately the URL to display the element is an element is being displayed. To illustrate further, the URL http://foobar.edu/some_path/ would be redirected to the display method (default method). The default method also takes as arguments *package* and *element* information. Since no other request parameters are available, the display method retrieves the root package information. A listing of metadata is followed by packages and elements in the root package. If a package named pkg1 is selected, the object is queried in the following manner.

http://foobar.edu/some_path/index.jsp?method=display&package=pkg1

The details of the package named pkg1 are retrieved and displayed (figure 4.9). If now, an element elem1 was accessed inside pkg1, the following query would be issued by the browser,

http://foobar.edu/some_path/index.jsp?method=display&package=pkg1& element=elem1

This requests the retrieval of element *elem1* in package *pkg1*. An additional parameter can be used with *element* parameter called *start*. Start is a boolean value indicating if an element should be displayed on screen (figure 4.10). Note that the configuration instance allows for specifying virtual names and display names for elements and packages. For example *pkg1* here may appear as '*Text Book*' in package listing and be mapped to a local directory names '*book*' while being referenced by the browser as *pkg1*. The display method like other methods is bounded by access control restrictions. The default settings award read permission to guests.

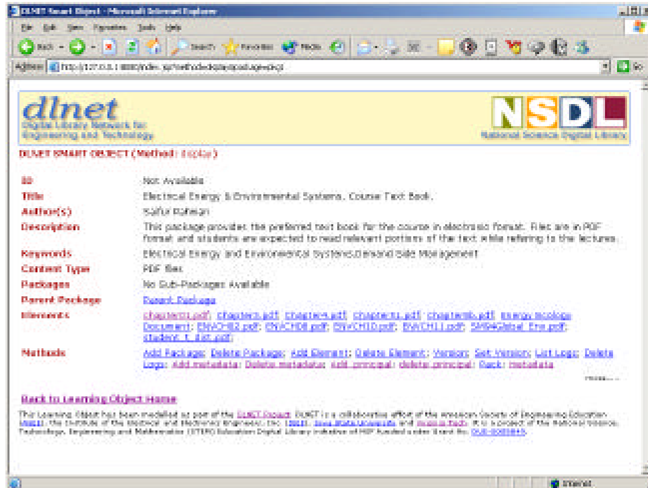

Figure 4.9. Display method (method=display&package=pkg1)

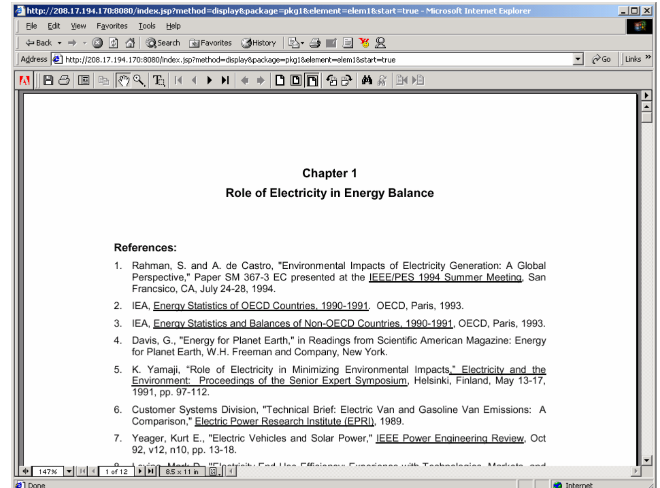

Figure 4.10. Display method (method=display&package=pkg1&element=elem1&start=true)

4.5.2. Add Package

Add/remove methods are key components to realizing a mutable smart object. The *add_package* method is used to add a package to the smart object. The user wishing to perform this action should either be an Administrator or have write access to the parent package. The action is initialized via a request of the type:

http://foobar.edu/some_path/index.jsp?method=add_package&package=pkg1

This returns with a listing of packages under package *pkg1*. Packages can be traversed by selecting the listings. On traversing to the package where the new package is to be added, the add_new_package form is filled out and submitted. The *add_package* method takes as input two mandatory parameters and two optional parameters. Parameter *package* indicates the package to which the new package is to be added and is mandatory along with parameter *new_package_name* which represents the reference name (used in identifying the package in a URL). The optional parameters include parameter *new_package_display_name* indicating the name to be used when displaying the package information to a user while the second optional parameter is *new_package_local_name* which gives the file system name for this package. Since a package is mapped to a directory, this would indicate the name of the directory to be created. By default, the optional parameters are set the new_package_name parameter. Add package data could be POSTED to the URL as:

http://foobar.edu/some_path/index.jsp?method=add_package&package=pkg1&new_package_name=sample_package& new_package_display_name='A Sample Package'

The above URL instructs the smart object to create a package identified as sample_package inside an existing package pkg1, having a display name 'A Sample Package' and a file system equivalent (directory) of the name sample_package. The information is appended to the manifest and configuration instances.

4.5.3. Add Element

The *add_element* method is used to add an element to the smart object. The user wishing to perform this action should either be an Administrator or have write access to the parent package. The action is initialized via a request of the type:

http://foobar.edu/some_path/index.jsp?method=add_element&package=pkg1

This returns with a listing of packages under package *pkg1*. Packages can be traversed by selecting the returned listings. On traversing to the package where the new element is to be added, the add_new_element form is filled out and submitted. The *add_element* method takes as input three mandatory parameters and one optional parameter. Parameter *package* indicates the package to which the new element is to be added and is mandatory along with parameter *new_element_name* which represents the reference name (used in identifying the element in a URL) and *element_file* which is a file object. The optional parameter is a parameter *new_element_display_name* indicating the name to be used when displaying the element information to a user. The localname of the element maps to the uploaded *element_file* name. The file upload is handled by the Multipart Request Handler module and is first saved to a temporary directory before being moved to the target package. Add element data could be POSTED to the URL as:

http://foobar.edu/some_path/index.jsp?method=add_element&package=pkg1&new_element_name=sample_element& new_element_display_name='A Sample Element'&element_file=<file>

The above URL instructs the smart object to create an element identified as sample_element inside an existing package pkg1, having a display name 'A Sample Element' and represented in the file system as element_file object name. The information is appended to the manifest and configuration instances.

4.5.4. Delete Package

The *delete_package* method allows aggregations (packages) to be deleted from a structured smart object. Deleting packages requires that a user have modify access on the super package.

Deleting a package involves deleting all sub-directories and files of the package, package mappings in the configuration instance and package mappings in the manifest instance. A request to delete a package is initiated by the default delete_package method being called on the package whose contents are to be deleted. This is done by issuing a request of the type:

http://foobar.edu/some_path/index.jsp?method=delete_package&package=pkg1

This lists the packages contained in package *pkg1*. The deleted package is then selected and a delete command issued.

http://foobar.edu/some_path/index.jsp?method=delete_package&
package=pkg1&delete_package_name=pkg3

This would delete the package *pkg3* (and its contents) inside *pkg1*.

4.5.5. Delete Element

The *delete_element* method allows for the deletion of an element in a given package. This element may be part of the root package or a sub-package. Deleting elements requires that a user have modify access on the super package which is obtained via the package interface (configuration instance). Deleting an element involves deleting the file system equivalent of the element (a file), element mappings in the configuration instance and item mappings in the manifest instance. A request to delete an element is initiated by the default delete_element method being called on the package whose contents contain the element to be deleted. This is done by issuing a request of the type:

http://foobar.edu/some_path/index.jsp?method=delete_element&package=pkg1

This lists the packages and elements contained in package *pkg1*. The to-be-deleted element is then selected and a delete command issued.

http://foobar.edu/some_path/index.jsp?method=delete_element&
package=pkg1&delete_element_name=elem1

This would delete the element *elem1* inside *pkg1*.

4.5.6. List Methods

The *list_methods* method enumerates the currently implemented/supported methods contained by the smart object. The listing is obtained from the _config.xml instance which is responsible for maintaining the up-to-date information about the methods supported. An optional request parameter xml with a corresponding boolean value of true generates an XML response. A query of the format: *http://foobar.edu/some_path/index.jsp?method=list_methods&xml=true* would generate a response of the following format.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<SmartObject>
        <method>add_package</method >
        <method>add_element </method >
        <method>delete_element </method >
        <method>display</method >
        .....................................................
</SmartObject>
```

4.5.7. Metadata

The *metadata* method is responsible for disseminating component metadata according to select schemes. The metadata method can be used to obtain metadata in XML in a format requested/supported by the smart object. The current implementation can disseminate metadata in IMS[14] and Dublin Core[38] format. The request parameter *format* indicates the metadata format requested. A typical request for the metadata interface would be of the type:

http://foobar.edu/some_path/index.jsp?method=metadata&format=dc

would yield metadata conformant to the Dublin core specification (figure 4.11).

```
<?xml version="1.0" encoding="UTF-8"?>
<metadata xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xsi:schemaLocation="http://purl.org/dc/elements/1.1/
  http://dublincore.org/schemas/xmls/simpledc20021212.xsd">
    <dc:title>Introduction to nanoscience</dc:title>
    <dc:creator>Vicki Colvin, Ph.D.</dc:creator>
    <dc:subject>Electrical Engineering</dc:subject>
    <dc:subject>Electronics</dc:subject>
    <dc:subject>Nanotechnology, Opto-electronics</dc:subject>
    <dc:description>This colorful and concise powerpoint presentation on nanoscience gives
                    specifics about "nano" engineering and how it is useful in technology
                    development. Perfect presentation for middle school through high school
                    science class. Contains hotlinks and references.</dc:description>
    <dc:publisher>Digital Library Network for Engineering and Technology</dc:publisher>
    <dc:date>2001-06-10</dc:date>
    <dc:format>PDF</dc:format>
    <dc:identifier>http://foobar.edu/some_path/index.jsp?
              method=display&package=pkg1</dc:identifier>
    <dc:language>en-US</dc:language>
    <dc:rights>This resource has copyright restrictions;
          reuse of this resource for standard educational purposes
          is subject to the author's consent.</dc:rights>
</metadata>
```

Figure 4.11. Output from the metadata method

4.5.8. List Source

The *list_source* method retrieves the source code responsible for processing and generating output for a particular action. For example, a list source on add package would list the source of the addPackage.jsp file (while triggers the add package module), the session management module which verifies whether the specified user can add a sub-package to the current package, the package interface manager (adds the new package to the _config.xml instance) and manifest interface manager (adds the new package to the imsmanifest.xml instance). The request URI for the list_source method would be of the type:

http://foobar.edu/some_path/index.jsp?method=list_source&action=add_package

The response would include source code of all dependencies of the add_package method.

4.5.9. Version

The *version* method is used to obtain the current version of the smart object. The version number is usually indicative of the release of the smart objects methods and is not changed unless a change to the software modules has been made. A typical version request would be of the type

http://foobar.edu/some_path/index.jsp?method=version

Version information is stored in the _config.xml instance and can be modified via the package interface manager. An optional parameter xml with a boolean value can be used to obtain version information as XML as shown below.

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<SmartObject>
        <version>1.0.1.J</version>
</SmartObject>
```

4.5.10. Set Version

The *set_version* method is used by administrators/developers to modify the version number of a smart object. The method takes one text parameter *version* as input. A typical request URI would be of the type: *http://foobar.edu/some_path/index.jsp?method=set_version&version=1.0.0.J.B*

4.5.11. Add Terms and Conditions(TC)

The *add_tc* method can be used to add additional access control restrictions on the smart object and its components. The current implementation supports addition of two types of terms and conditions. The first is a blocked userid list. Every bucket maintains a list of blocked users from its user repository that are not allowed to access the smart object. For example, to prevent remote

administrative login, a TC is set to disable login of a user from across the network. The query for such a rule is

http://foobar.edu/some_path/index.jsp?method=add_tc&blockedid=Administrator

This would prevent future administrative logins to the smart object.

A second set of TC allows access control to a particular component of the smart object. For example, a TC rule can be used to set the permissions to a package or element in the smart object. A rule can be set that allows read permissions while denying write and modify permissions to a particular element or package. The HTTP query for such a rule would look like:

http://foobar.edu/some_path/index.jsp?method=add_tc&
package=pkg1.pkg2&adduserid=sample&perm=100

This rule specifies that the userid sample should be added to the access control list of package pkg2 in pkg1 with permissions to read and not write or modify. The perm parameter takes a binary string of three characters length as input, the first of these representing read permissions, the second specifying write permissions and the third specifying modify permissions. A value of '1' for any one of these permissions indicates an accept while a value of '0' indicates a deny. Hence, in this example the user sample can read the contents of the package pkg1.pkg2 but cannot write or modify its contents or properties.

4.5.12. Delete Terms and Conditions

The *delete_tc* method is used to reverse any previously set terms and conditions. The method can be used to unblock a particular userid and allow remote execution of methods. As a continuation of the example from the Add TC method, a blocked user can be unblocked using the HTTP query:

http://foobar.edu/some_path/index.jsp?method=delete_tc&blockedid=Administrator

Also, the delete_tc method can be used to reset permissions for any previously set access control permissions.

http://foobar.edu/some_path/index.jsp?method=delete_tc&
package=pkg1.pkg2&deleteuserid=sample

The above query would result in resetting the permissions of the userid sample on package pkg1.pkg2. This would result in user sample not being able to access the package until permissions are added via the add_tc method.

4.5.13. List Logs

The *list_logs* method allows an administrator to monitor the logs generated by the smart object. Logs are generated as a result of interactions between the users and the smart object. Log files are generated based on the date and contain information such as remote host information, time of access, request details and response details. The list of logs stored in the _logs package of the reserved package _sysFiles can be obtained by querying the object via the list_logs method. A typical query to the smart object would be of the type:

http://foobar.edu/some_path/index.jsp?method=list_logs

This would result in a listing of the log files recorded over time. Further, a parameter *date* is used to view log files created on specific dates. A typical request would be of the type:

http://foobar.edu/some_path/index.jsp?method=list_logs&date=2003-04-04

A listing of the logs for may 15th 2003 is generated:

```
2003-04-04 01:31:36 208.17.194.170 guest123 GET
index.jsp?method=display&package=pkg1 - 200
2003-04-04 01:31:49 208.17.194.170 guest1 GET
index.jsp?method=display&package=pkg1&element=elem1 - 200
2003-04-04 01:31:51 208.17.194.170 guest 223 GET
index.jsp?method=display&package=pkg1&element=elem1&start=true - 200
2003-04-04 01:32:06 208.17.194.170 guest    GET
index.jsp?method=add_package&package=pkg1 - 200
2003-04-04 01:32:24 208.17.194.170 guest  GET index.jsp?method=version - 200
```

4.5.14. Delete Logs

The *delete_logs* method allows an administrator to delete logs of user requests and responses. Access logs are usually prefixed with bucket_access_log and suffixed by .txt. The log to be deleted is identified by the *date* parameter passed in the request. The *date* parameter is of yyyy-mm-dd format. A sample query to delete a log would be of the type:

http://foobar.edu/some_path/index.jsp?method=delete_logs&date=2003-03-04

Again, this method can be performed only by an Administrator and package permissions as such do not affect the decision.

4.5.15. ID

The *id* method retrieves the unique identifier of the smart object. The ID is a unique identifier to represent the content universally or within the scope of a digital library. The id method takes an optional parameter package which retrieves the ID of a particular package in the smart object.

Since smart objects are scalable object-oriented entities, one smart object may contain another smart object usually identified by a different ID. An id request would of the type:

http://foobar.edu/some_path/index.jsp?method=id&package=pkg10.pkg20

The above query retrieves the ID of package *pkg20* a sub-package of *pkg10*.

4.5.16. Pack

The *pack* method helps in moving/replicating the smart object to another location. The method generates a compressed stream of the entire smart object including the elements, directory structure, configuration and database instances. A typical pack query would be of the type:

http://foobar.edu/some_path/index.jsp?method=pack

Future releases of pack would include downloading individual packages or elements and their associated metadata also.

4.5.17. Add Metadata

The smart object allows for addition and/or modifying of metadata to every component within it via the *add_metadata* method. The current implementation allows for adding/modifying metadata to any component of the smart object by a user who has modify permission over a component. The add_metadata method supports different functionality for the HTTP GET and POST methods. When metadata for a particular component is to be retrieved, the GET method is used while the POST method is used when metadata is being added. For example, to add metadata to a package pkg1.pkg2, the request query is formulated as:

http://foobar.edu/some_path/index.jsp?method=add_metadata&package=pkg1.pkg2

This would retrieve the metadata of the pkg2 package and present it to the user. When applying metadata changes to the component, the data is POSTed to the same URI. The method makes use of the metadata interface module which generates/modifies metadata relating to any component and is conformant to the IMS metadata specification v1.2.1. A sample of the metadata generated is shown in figure 4.5.

4.5.18. Delete Metadata

The *delete_metadata* method is responsible for handling requests to delete metadata of existing components of the smart object. The method takes two parameters as input (*package* and/or *element*). The user performing this action on a package or element is expected to have modify permissions on it as the component in question. Delete metadata clears the metadata node from

the imsmanifest.xml instance corresponding to the package or element selected. For example the request URL:

http://foobar.edu/some_path/index.jsp?method=delete_metadata &package=pkg1.pkg11&element=elem1

This instructs the object to delete the metadata associated with *elem1* in package *pkg11* in package *pkg1*.

4.5.19. Add Principal

The smart object allows for creation and maintenance of a user repository, a list of recognized userids that can access the smart object features and functionality. Users can be added to such a repository via the add_principal method. Since this is an administrative action, only users with administrative privileges can access this method. The current implementation allows for adding/removing users to the repository via the manage users interface. The HTTP GET or POST methods can be used to add a new user to the repository. The interface can be used to specify the *principal* (userid), *passwd* (password) and *UserType* (Administrator/User) of a new user. This information is then stored in the user repository and can be used to provide access to the smart object or components of the smart object. For example, to add a user to the smart object user repository, the request query is formulated as:

http://foobar.edu/some_path/index.jsp?method=add_principal&principal=sample_id& passwd=passworD&UserType=User

This would add a user with ID sample_id with password passworD into the user repository. The passwords are stored in the repository as MD5 hashes or SHA-1 hashes. This information can be used to allow/deny users access to the smart object and its components for learning purposes.

4.5.20. Delete Principal

The *delete_principal* method handles removal requests from the user repository. The methid takes a principal parameter as input and removes the userid from the repository. The user performing this action is expected to have Administrative privileges. The method takes the *principal* as an input parameter and removes all occurrences of the principal in the user repository and the configuration instance. For example the request URL:

http://foobar.edu/some_path/index.jsp?method=delete_principal &principal=sample_user

instructs the object to delete the user sample_user from the user repository and all occurrences of the userid in the smart object configuration instance.

4.6 How are Smart Objects different?

The Java-based smart object model discussed above is an alternate implementation of the Bucket model[26] presented in chapter 2 and addresses the shortcomings of this model. This new model has close relations to three other models discussed in chapter. It is a direct extension of the Bucket model[26], is a suggested replacement for the DLNET Learning Object model[23] and hopes to achieve SCORM-compliance[37] in future versions. A brief comparison of each of these three models with the Java-based model follows:

4.6.1 DLNET LO vs. Java Smart Objects

The DLNET Learning Object model[23] is a simple organized representation of learning resources and metadata conformant to the IMS content packaging specification. The Java smart object on the other hand is a superior, active learning object with a well-defined content model. It allows for content aggregation (with metadata conformant to the IMS metadata specification[14]) and is equipped with a set of methods that make it self-sufficient. The DLNET LO being a passive learning entity does not have this 'intelligence' and is dependent on the library's ability to support metadata and content enrichment operations necessary to create a dynamic learning environment. The smart object's methods assist it in being self-sufficient and independent of a library's operational limitations allowing its structure and learning value to be modified thus contributing to the development of a dynamic learning environment.

4.6.2 SCORM vs. Java Smart Objects

The SCORM[37] is a reference model for most modern digital libraries. Like the Java smart object model, it allows for content aggregation and uses IMS compliant metadata and content packaging schemes. However, SCORM and Java Smart Objects have one unique characteristic missing in the other. In addition to a Content Aggregation Model[36], the SCORM is equipped with an runtime model using which a learning object can convey information about the learning process back to the digital library. This information can be used by the library to adjust the learning value associated with the object for future uses. On the other hand, the Java smart object is equipped with methods that provide it with self-sufficiency and intelligence.

4.6.3 Buckets vs. Java Smart Objects

Buckets come closest to Java Smart Objects in terms of design objectives and ideology. There are significant differences in the methodology used to realize these objectives though. The bucket model adopted bibliographic metadata scheme (and text-based configuration instances), which is less acceptable in modern digital libraries. In addition the use of text files led to scalability issues. The Java smart object model on the other hand uses XML for metadata (and configuration storage) in a format that can readily be inducted into current digital library frameworks with relative ease. In addition, the Java smart object has been equipped with an XML user repository which better equips the object to enforce access control restrictions on smart object components.

A summary of these comparisons is shown in table 4.1

	DLNET LO	SCORM	Buckets	Java Smart Objects
Content Model	Not really	Yes	Yes	Yes
Content Aggregation	Yes	Yes	Yes	Yes
Run-time model	No	Yes	No	No
Metadata	IMS	IMS	RFC-1807	IMS
XML	Yes	Yes	No	Yes
Content Packaging	Yes	Yes	No	Yes
Independence/Self-sufficiency	No	No	Yes	Yes
Intelligence	No	No	Yes	Yes
Reusability	Yes	Yes	No?	Yes
Scalability	Yes	Yes	No	Yes
Towards standardization	No	Yes	?	Adopt SCORM approach

Table 4.1. Comparison of DLNET LO, SCORM, Buckets and Java Smart Objects

Chapter Five

Smart Object Development Tool

5.1 Overview

To summarize our discussion from the previous chapter, smart objects are intelligent, aggregate, object-oriented, platform independent representations of learning resources for use in digital learning environments. Such objects can be created and modified using a prototype standalone development tool modeled as part of this thesis work. This chapter discusses the design requirements, architecture and layout issues of such a development tool that can be used to create a smart object package. The package created by the tool uses a modified version of the IMS Content Packaging specification[13] with metadata conformant to the IMS/IEEE LOM v1.0[15,14] standard for individual and aggregated learning items. A packaged smart object is then ready to be deployed on a container capable of hosting it. An example of creating a smart object and deployment instructions are discussed toward the end of the chapter.

5.2 Tool Requirements

Smart objects are aggregate, intelligent and platform independent object-oriented container constructs for use in digital learning environments. The tool used to develop such objects is required to support the creation and manipulation of these properties. Toward this objective, a set of requirements have been identified as being necessary and are discussed below.

5.2.1 Content Aggregation

Smart Objects support content aggregation. Content Aggregation is the process of physically and/or logically grouping together learning items based on physical or learning characteristics. It is therefore required for the tool used to develop smart objects to support content aggregation tasks. The current smart object design comprises of two components in its Content Aggregation model. The smallest learning entity that serves to provide a learning experience to a user is called an *element*. Elements are represented on a file system as individual files. Groups of elements are aggregated to be called *packages*. Packages are mapped to directories on a file system. The development tool must support the creation of such a representation of learning components along with a graphically appealing representation of the content aggregation supported by the smart object.

5.2.2 Metadata Extraction

Metadata is considered to be the face of the content. It can be used by a digital library for information discovery purposes as well as by a learner in a pre-learning decision process. By design, every component of the smart object can have metadata. This metadata can be as descriptive as possible and can encompass the entire range of elements specified by the IMS metadata specification. The prototype tool should have the ability to collect and retrieve element and package metadata from the manifest instance (*imsmanifest.xml*) or from individual metadata instances stored alongside the elements or packages. The nature of metadata collected should be reflective of the general, technical and educational characteristics of a learning resource or package. In addition, a degree of abstraction must be provided to ease the technical know-how requirements about XML and the underlying smart object implementation from a contributor.

5.2.3 Support for Methods

Smart objects demonstrate self-sufficiency, aggregation and object-orientedness by using methods; software components that can be called upon to perform a certain action on the learning resource while being deployed on the web. The method called determines the type of action being performed and the set of modules invoked thereof. The tool must have an interface which allows a contributor to select the methods to be supported by the smart object. Also, a scalable approach to displaying the supported methods should be adopted to support future changes to method types and names.

5.2.4 User repository and Access Control

The smart object model supports maintaining a user repository and access control lists. A user repository defines a set of users (and groups) that are eligible to access a smart object and invoke its methods. The user repository can ideally be a database or a file instance maintaining a list of user/password combinations. In addition to this, a user repository may also indicate user group information assisting in decision making by modules.

The access control list on the other hand defines user permissions on various smart object components. As defined in 4.4.4, every smart object component (*element/package*) can have read/write/modify permissions assigned to a user and is stored in the configuration instance.

These two characteristics necessitate an interface wherein the user repository can be populated along with permissions being defined for every smart object component.

5.2.5 Content Packaging

The smart object development process involves metadata harvesting, content aggregation and user repository and access control definition operations. Once these tasks have been successfully completed, the smart object is ready to be deployed. The pre-deployment process involves transporting the smart object across domains as a single unit. The smart object design adheres to the IMS Content Packaging Specification[13] which facilitates the exchange of learning resources as a *package interchange file* (a compressed .zip/.tar file) containing data (learning components) and metadata (learning information). Such a packaging scheme should be implemented in the prototype tool facilitation the compression of aggregated learning resources, resource metadata and corresponding methods that provide intelligence and self-sufficiency to the smart object.

5.2.6 Platform Independence

Little needs to be done to overemphasize the need for platform independence in any learning object model. Since smart objects are mobile, free-standing agents with learning value their deployment should be independent of platform considerations. Similarly, the tool used to develop such learning objects should not be constrained to specific platforms. This requirement necessitates the development of smart object to be supported on multiple platforms.

5.3 Tool Architecture and Layout

A three-tier architecture[42] was found to be most suitable to meet the operational requirements of the development tool discussed in the previous section. The architecture (shown in figure 5.1) has three components to it, a data layer, shielded by the business logic layer accessed via the presentation layer. Each tier is indicative of a certain set of actions/functions performed by the tool. A brief description of each of these components and sub-modules within them follows:

Tier III: The modules of this layer are responsible for providing the look and feel and all other presentation functions facilitated by the tool. The *GUI Builder, Main Internal Frame* and *Tool Base Panel* modules are responsible for instantiating and terminating all the user-interfaces. In addition, the user interfaces include a *Resource Tree Panel* for displaying content aggregations

(elements and packages), *User Access Panel* for displaying user repository information and package/element access control information, *Methods Panel* for displaying the supported smart object methods, *Manifest Panel* for displaying the content and structure of the manifest instance (*imsmanifest.xml*) and *Metadata Panel* for displaying/collecting metadata of elements and element aggregations (packages). User interactions with any of these components are propagated to Tier II and then onto Tier I. A pictographic representation of the user interface is shown in figure 5.2.

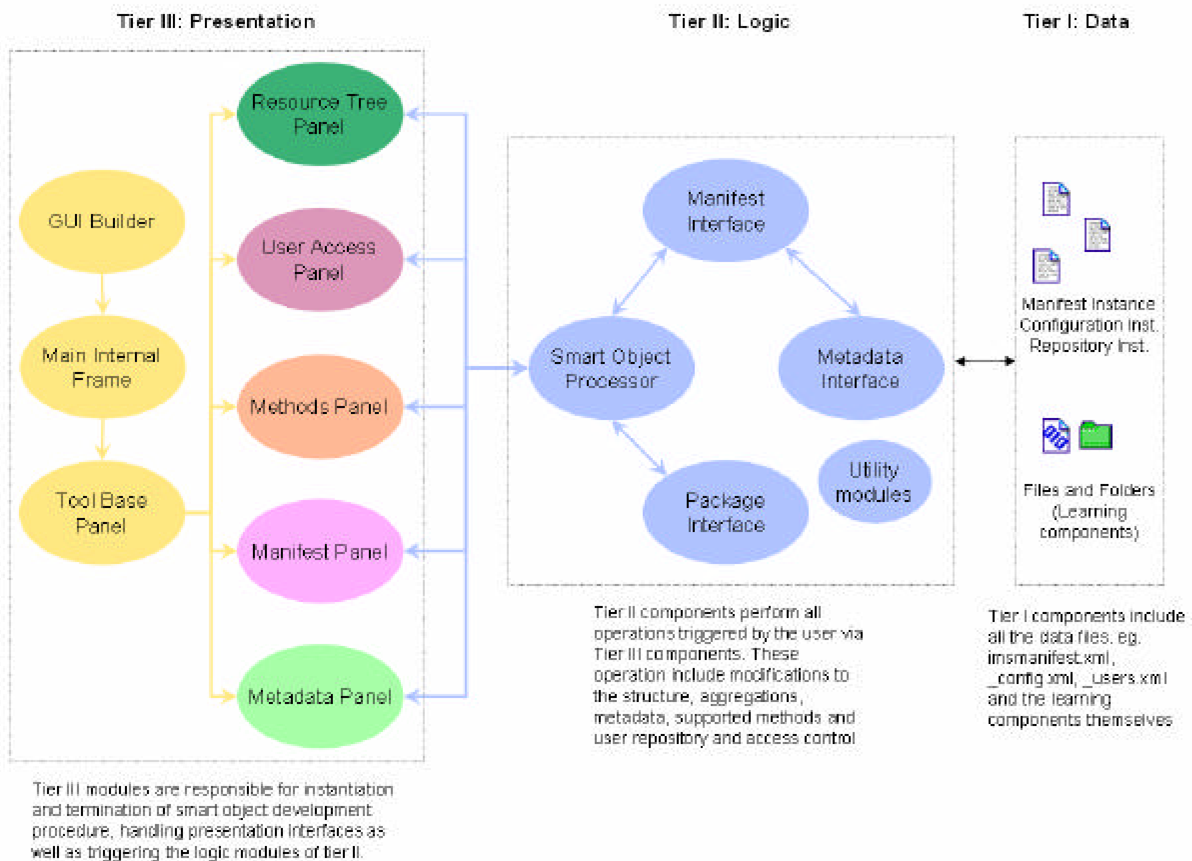

Figure 5.1. Three-tier architecture of the Smart Object Development Tool

Tier II: This tier consists of modules that provide core functionality for actions/events triggered by the presentation layer. These actions include requests for addition/deletion of elements/packages, manipulation of component metadata and support for methods. The *Smart Object Processor* acts as the gateway to the logic tier and redirects tasks to the *Manifest Interface* and *Package Interface* modules. The *Manifest Interface* module is responsible for making metadata-related modifications and for recording changes to the structure of the smart object.

The *Package Interface* on the other hand is responsible for recording changes in structure and behavior being triggered by a user. Such changes include addition or removal of packages/elements, modifications to the user repository and access control list and supported methods information. Tier II also includes key modules which provide access to the data modules of tier I such as XML parser modules, file access handlers etc.

Tier I: This tier comprises of the components that are manipulated by the above tiers. For the smart object model, these include the manifest instance, configuration instance and the user repository instance as mandatory items, optional data items (these include files and folders that are useful in learning) and methods which provide intelligence to the smart object. Modules from tier-II modify/collect these data files during the development process and use them to create a learning package.

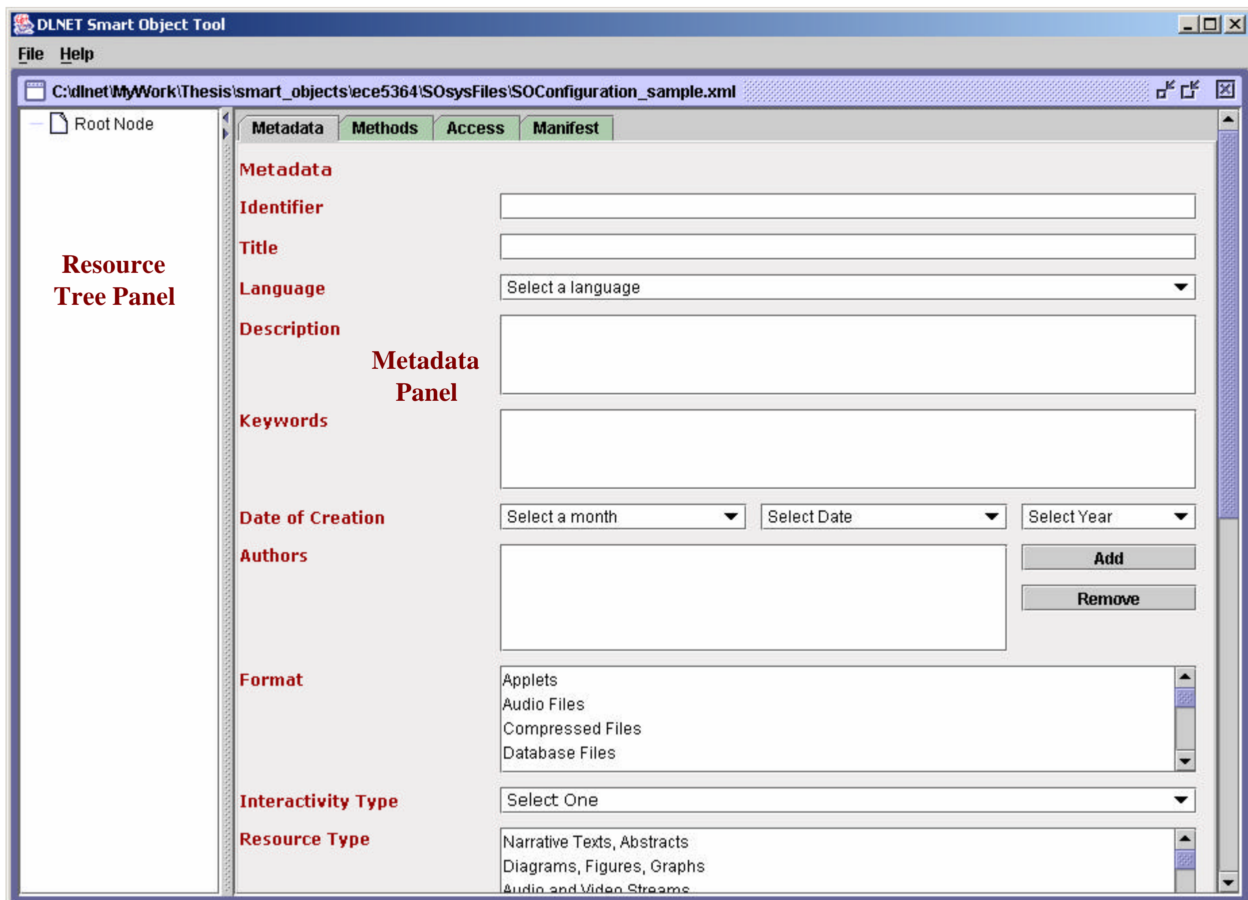

Figure 5.2. Graphical User Interface of the Smart Object development tool

5.3.1 Content Aggregation

Content Aggregation is an inbuilt feature in smart objects. Content builders can take advantage of this feature to aggregate items with similar characteristics as packages. The smart object development tool allows for content aggregation via the *Resource Tree Panel*. Contributors can choose to build smart objects in a highly structured manner or in a flat un-structured format. As discussed before packages map to a directory on the file system. A package can be added/deleted via the Add/Delete Package options available in the *Resource Tree Panel*. Any aggregation action invokes the corresponding action handler in the *Smart Object Processor* module of tier-II. A sample aggregation is shown in figure 5.3. In this example, a new package is being added to an already existing package named 'Course Text'. Course Text corresponds to a directory in the smart object development directory. The figure also shows a number of other aggregations each indicative of the characteristics of learning resources contained within them (for example, the package 'Lecture Audios' may indicate a package containing numerous audio lectures as opposed to the package 'Exams' which could contain sample exams as PDF files).

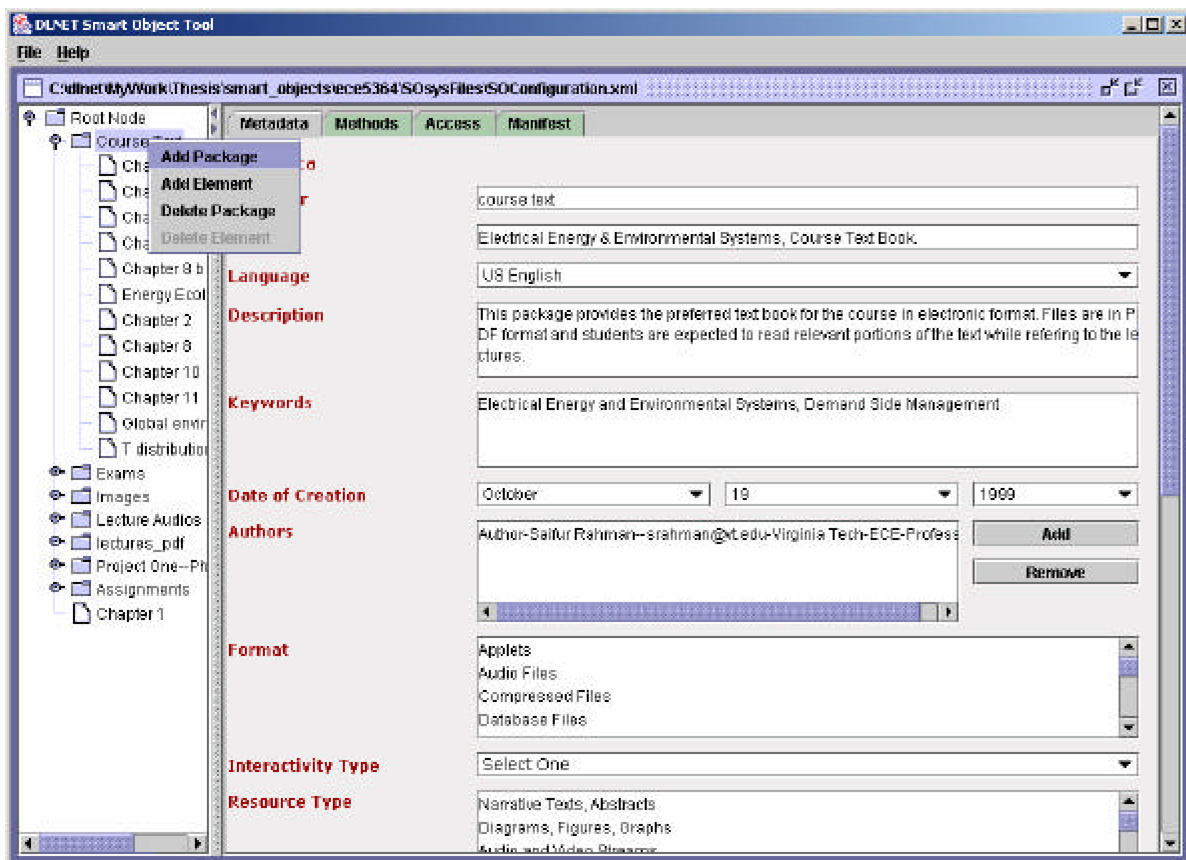

Figure 5.3. Content Aggregation interface

5.3.2 Metadata Extraction

The smart object model allows for defining metadata for every learning component. Similarly, the smart object development tool allows for defining metadata for elements and packages of the smart object. The current listing of metadata for any component can be recalled by selecting the component in the *Resource Tree Panel*. This triggers a method in *Smart Object Processor* that retrieves and renders the component metadata to the user interface. Changes to the metadata can be made and applied via this interface. The variety of metadata currently harvested encompasses a wide selection from the IMS metadata specification[14]. The current implementation requires mandatory metadata fields like the *title*, *description*, *language*, *keywords*, *date of creation*, *author*, *format* and *classification* to be specified for every component. Metadata can be applied via the Save button at the bottom of the Panel. A sample of the metadata harvesting process is shown in figure 5.4.

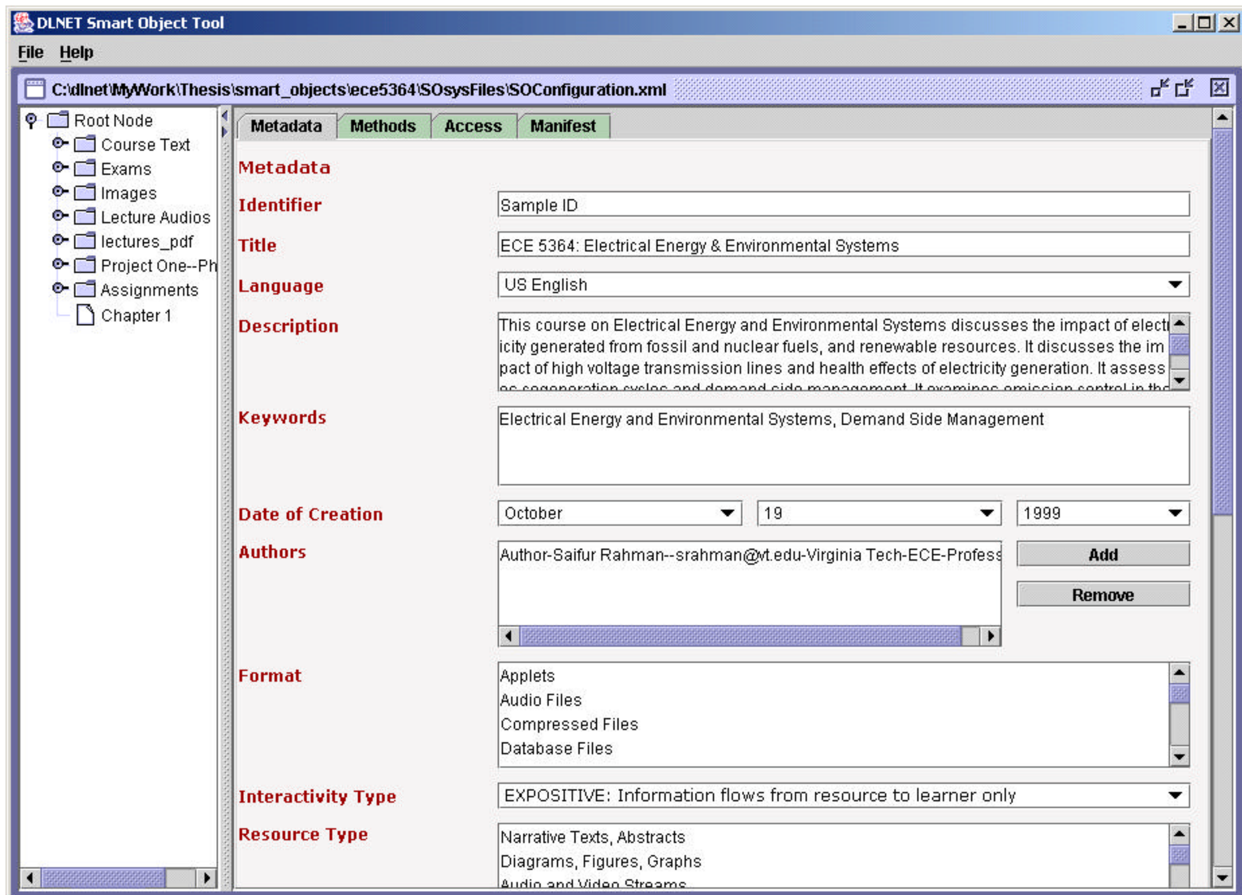

Figure 5.4. Metadata harvesting interface

5.3.3 Supported Methods

Methods define the actions that can be performed on a smart object. As discussed in section 5.2.3, methods define self-sufficiency scope. The number of methods supported reflects the variety of actions that a smart object supports. The most basic method required to realize web-based learning is the *display* method. As discussed in chapter 4, this method is responsible for providing learning information (general, technical, educational, technical etc.) to help the user in a pre-learning decision process. Other methods support a wide variety of operations to be performed on the smart object and include those that allow changing structuring (*add/remove package/element*) and learning information (*add/remove metadata*). Method selections are handled by the *Methods Panel* and passed onto the *Smart Object Processor* prior to the content packaging stage. Unlike the *Metadata Panel*, the *Methods panel* is referenced only once before the package creation state. Details about the supported methods are included into the configuration instance along with the software components needed to make them work before creating a smart object package. Supported methods are selected using the selection interface provided in the Methods Panel (figure 5.5).

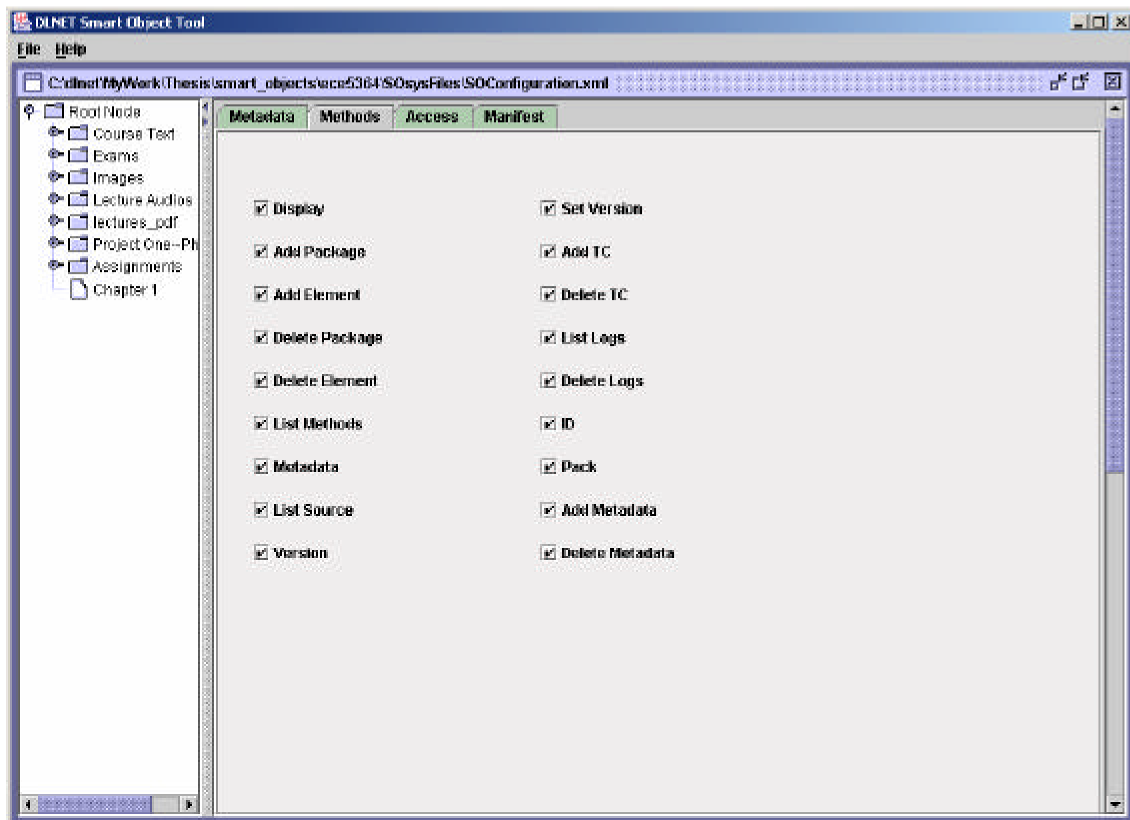

Figure 5.5. Supported methods panel

5.3.4 User repository and Access Control

To meet the requirements discussed in section 5.2.4, the development tool allows for populating a user repository and maintaining access control restrictions on the learning components of the smart object. The user repository is an XML instance with '*user name-password-user type*' combinations stored along with the configuration instance (*_config.xml*). The format and structure of the user repository and configuration instance are discussed in sections 4.4.11.4.4. The tool provides an interface to manipulate the smart object user repository and the access control list for each learning component. A sample user repository and access control view is shown in figure 5.6. The user repository instance can be manipulated by using the add/remove button combination available next to the *Smart Object Users* listing. The *Smart Object Processor* module of the *logic layer* handles the addition/removal of users to the repository. In addition, access control restrictions can be imposed on specific components of the smart object (*packages/elements*).

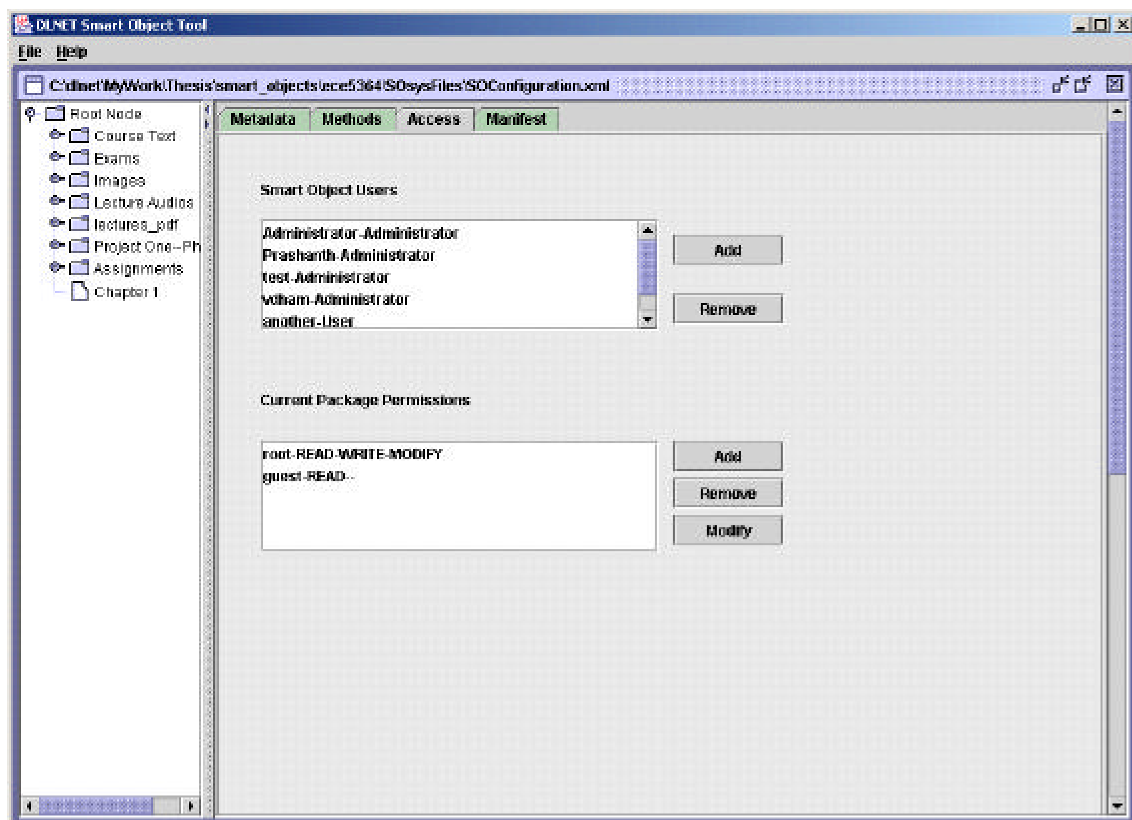

Figure 5.6. Sample user repository interface

The *Current Package Permissions* section can be used to manipulate the level of control a smart object user can have over a particular learning component. The learning component is determined by the selection made in the *Resource Tree Panel*. Details about permissions are discussed in section 4.4.4. The *User Access Panel* triggers methods in the *Smart Object Processor* that can manipulate the access restrictions for a package or element in the configuration instance.

5.3.5 Content Packaging

To meet the packaging requirements discussed in 5.2.5, the smart object and the development tool use IMS Content Packaging Specification (v1.1.2). As discussed is chapter 3, content packaging is used to define a standardized set of structures that can be used for transporting content across digital domains[13]. Content is exchanged as a Package Interchange File(PIF), a concise web delivery format file containing structuring information, along with the learning items themselves. Structuring information is stored in a manifest file in the root folder of the package (usually a .zip or a .cab file) including the metadata information of the content.

Once the content has been aggregated, metadata corresponding to every component of the learning object collected, access control restrictions specified and methods that provide functionality/intelligence to the smart object selected a package conforming to the content packaging specification is created. The package created by the tool is of a *.zip* format accessible via any popular decompression tools. Like is the case with other functionalities, the *Smart Object Processor* module triggers the validation and compression sequence. The validation process involves verifying the existence of all the resources listed in the *Resource Tree Panel* and performing a discrepancy check on the metadata structuring, user profile repository and access control functionalities. Once these tests have been successfully performed, a PIF file (.zip format) is generated containing the smart object content structured according to IMS-CP specifications. This file can then be transported across domains to be hosted by a digital library or alternately be hosted as a standalone object.

5.3.6 Platform Independence

To address the issue of platform independence, the smart object development tool has been built using Sun's Java 2 platform (J2SE 1.4.0)[17]. The graphical user interface has been developed using components from Java's lightweight Swing framework[19]. In addition, the core

functionality of the tool includes modules from Sun's Java API for XML processing[18] and Apache's Xerces2 Java Parser[3]. The tool has been successfully tested on Windows 2000/XP[28] and Red Hat Linux 8.0[30] platforms.

5.4 Creating and deploying Smart Objects - An example

Creating a Smart Object: The previous sections focused on the design requirements and architecture of the smart object development tool. To exemplify the operation of the tool, this section describes the tasks involved in creating a smart object and deploying it on the web.

- The tool is distributed as a compressed (*.zip* or *.tar*) file. Using any well known decompression utility, the contents of the file can be saved to a working directory for the tool. The tool can be invoked via a batch file on windows (*runME.bat*) or a script file in a Unix platform (*runME.sh*). These are the only two files in the root folder of the distribution.
- Once the tool has been invoked the user has the option of choosing to develop a new smart object or continue developing an existing object. These options are available via the 'File' menu on the application. On selecting the 'New' option, the user is prompted for the selection of a *working directory* (here after referred as: <*working_dir*>). The *working directory* is located on the file system of the user and contains a folder and file listing. Upon completion of this selection, the tool initiates the *presentation tier* modules. This would involve creating the *Resource Tree Panel*, *Metadata Panel*, *User Access Panel*, *Methods Panel* and *Manifest Panel*. Also, a manifest instance (*imsmanifest.xml*) is created in the working directory along with empty configuration (<*working_dir*>/_sysFiles/_config.xml) and user database (<*working_dir*>/_sysFiles/ _users.xml) instances. A screenshot of the tool's user interface at this point is shown in figure 5.2.
- Since smart objects support content aggregation, a contributor can utilize this feature to intuitively arrange learning resources in a hierarchical fashion. For example, lectures audio files may be grouped under a folder named *lectures_audio* while class notes and sample exams can be aggregated under folders named *lectures_pdf* and *Exams* respectively. The structuring can also be indicative of the file characteristics

(*lectures_audio* indicates lectures as audio files, while *lectures_pdf* indicates lecture notes as pdf files).

- Once the structure has been setup, the user can fill out metadata details for the smart object. The Metadata Panel can be used for this purpose. Mandatory metadata items include Title, Description, Keyword information, Author information, Language, Copyright information and Classification. Once these fields have been filled out, the tool stores this data in a temporary manifest instance created in <*working_dir*>. A screenshot of the root metadata is shown in figure 5.7.

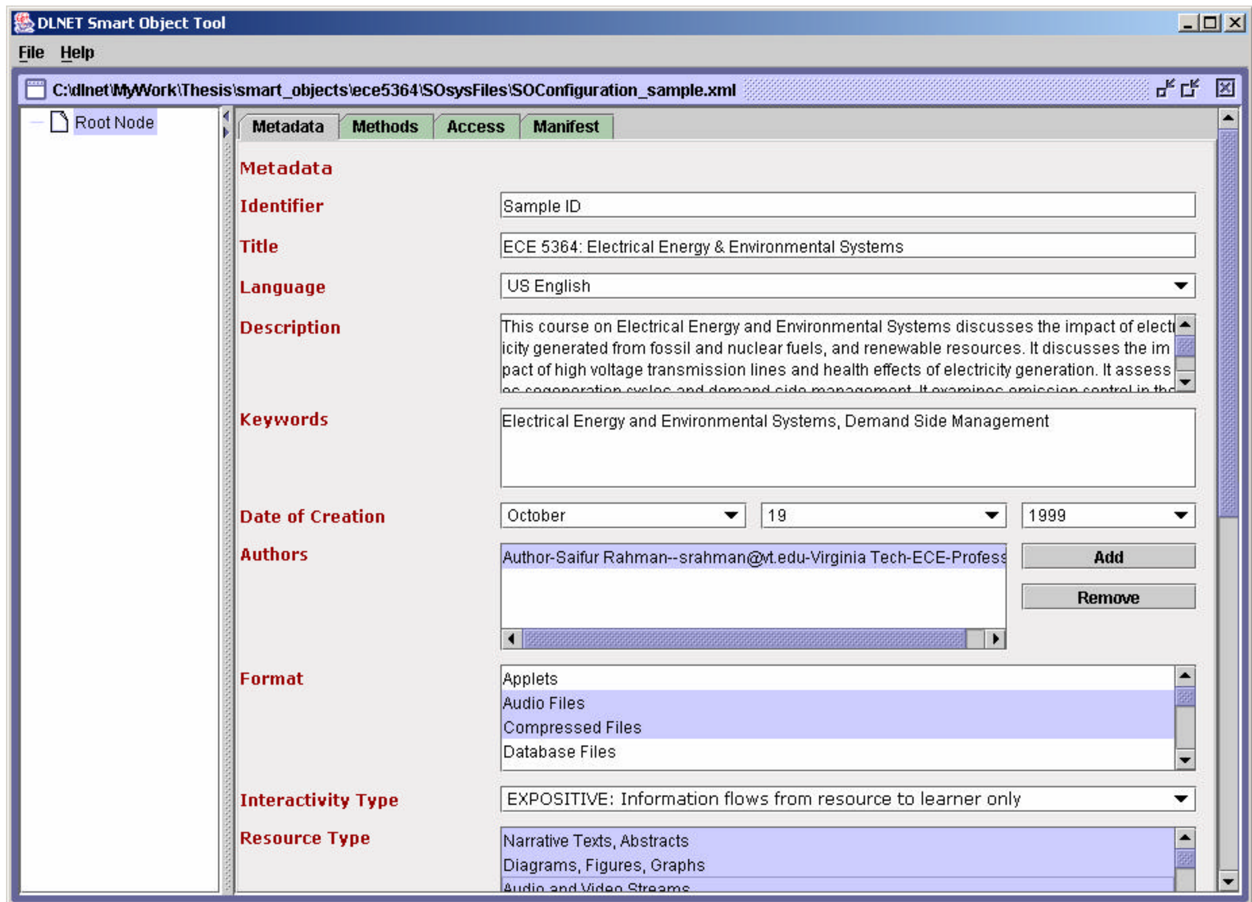

Figure 5.7. Smart Object example: Root metadata

- The user can then select the methods to be supported by the smart object. This can be done by choosing the Methods Tab (invokes the *Methods Pane*). At the time of documentation, the smart object model has 20 supported methods. Information about the working of each method has been discussed in section 4.5.
- The user can populate the user database via the *User Access Panel.* The panel comprises of two sections, the *Smart Object Users* section which lists the current entries in the user repository (<*working_dir*>/_sysFiles/_users.xml). The second section contains user access information to a particular package or element. Since the smart object currently has no components, the entries would indicate access restriction to the root package. Users can be added to the repository by choosing the Add Button. A screenshot showing the addition of a user 'Administrator' with Administrative privileges is shown in figure 5.8.

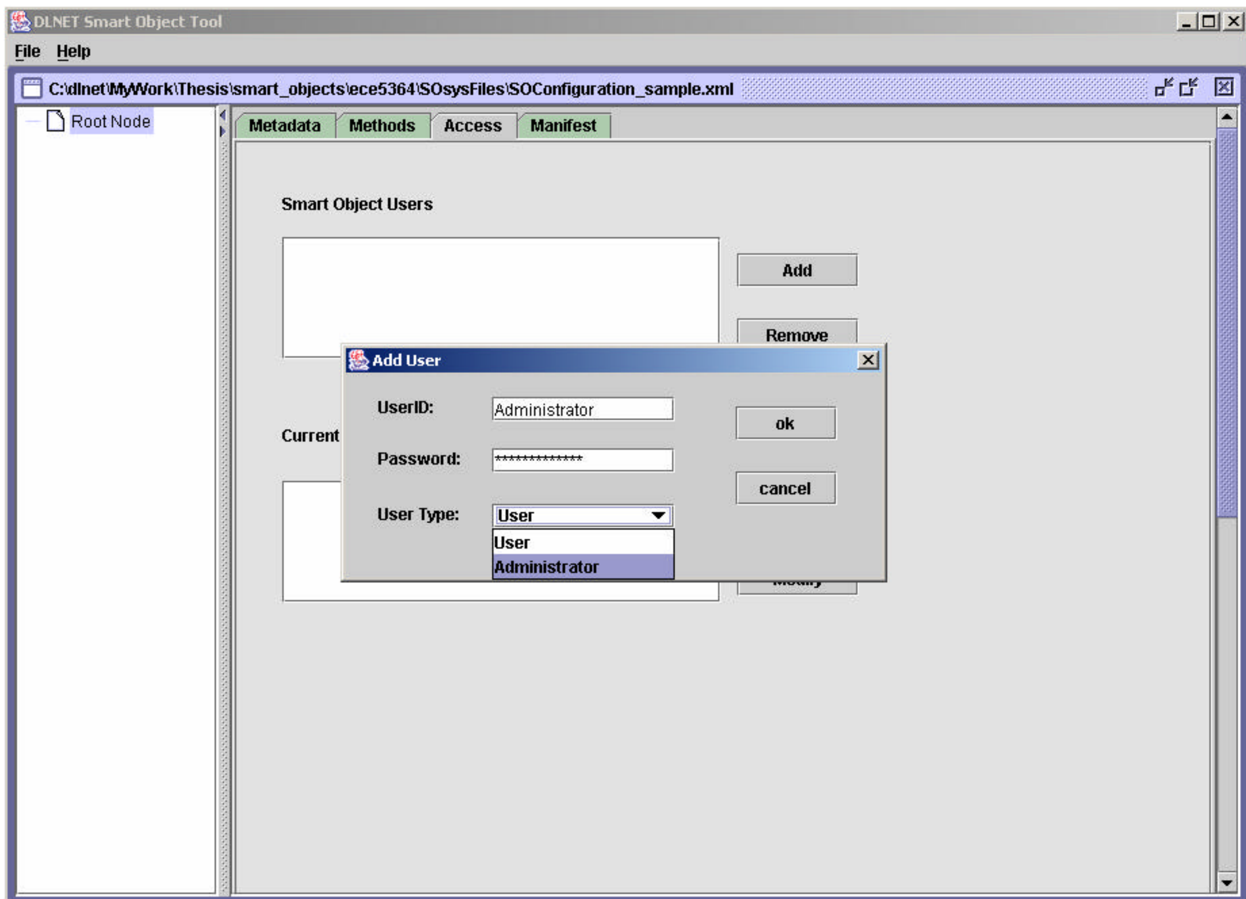

Figure 5.8. Smart Object example: User Access Panel (Add User)

- Following the above procedure, an entire repository of users can be created. Upon completion of this task, access control restrictions can be imposed on users in the repository. Hence permissions can be set for users for accessing the root package. This can be done by using the 'Current Package Permissions' section of the *User Access Panel*. The current smart object release supports read/write/modify permissions to be assigned to any component.
- Once permissions have been assigned for the root package, content aggregations can be added to the smart object. In case the user decides on a flat structure, then elements can be added to the root package. For adding a *package* or *element* to an existing smart object component, the user can right-click on that component and choose the appropriate action. For adding a package to the root package, the Add Package option can be selected. Once this option is selected, a window listing the directories under the selected directory (<*working_dir*>) appears. The user can then select a directory to add as a package to the root package. A screenshot displaying this information is shown in figure 5.9. In this case, the folder *lectures_pdf* is being added to the root package. The display name and Package name are optional items and are discussed in section 4.5.2.

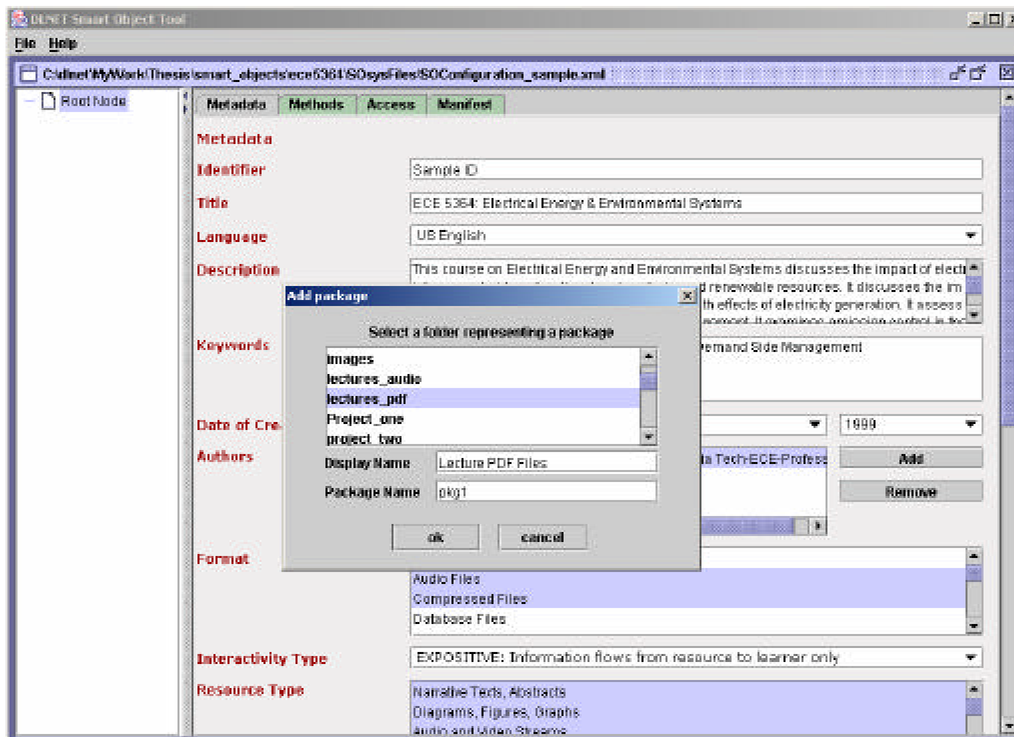

Figure 5.9. Smart Object example: Add Package

- Similarly, additional packages or elements can be added to the smart object structure along with metadata information being filled out in the *Metadata Panel* and access control being specified in the *User Access Panel* (see figure 5.10). Once the smart object has been completely built, the package option can be selected from the menu items to prepare the smart object for packaging. The packaging process involves collecting and preparing elements and packages, validating metadata, configuration instance and user repository. A smart object package is then created in the *<working_dir>*. This package interchange file is now ready to be deployed or transported across domains.

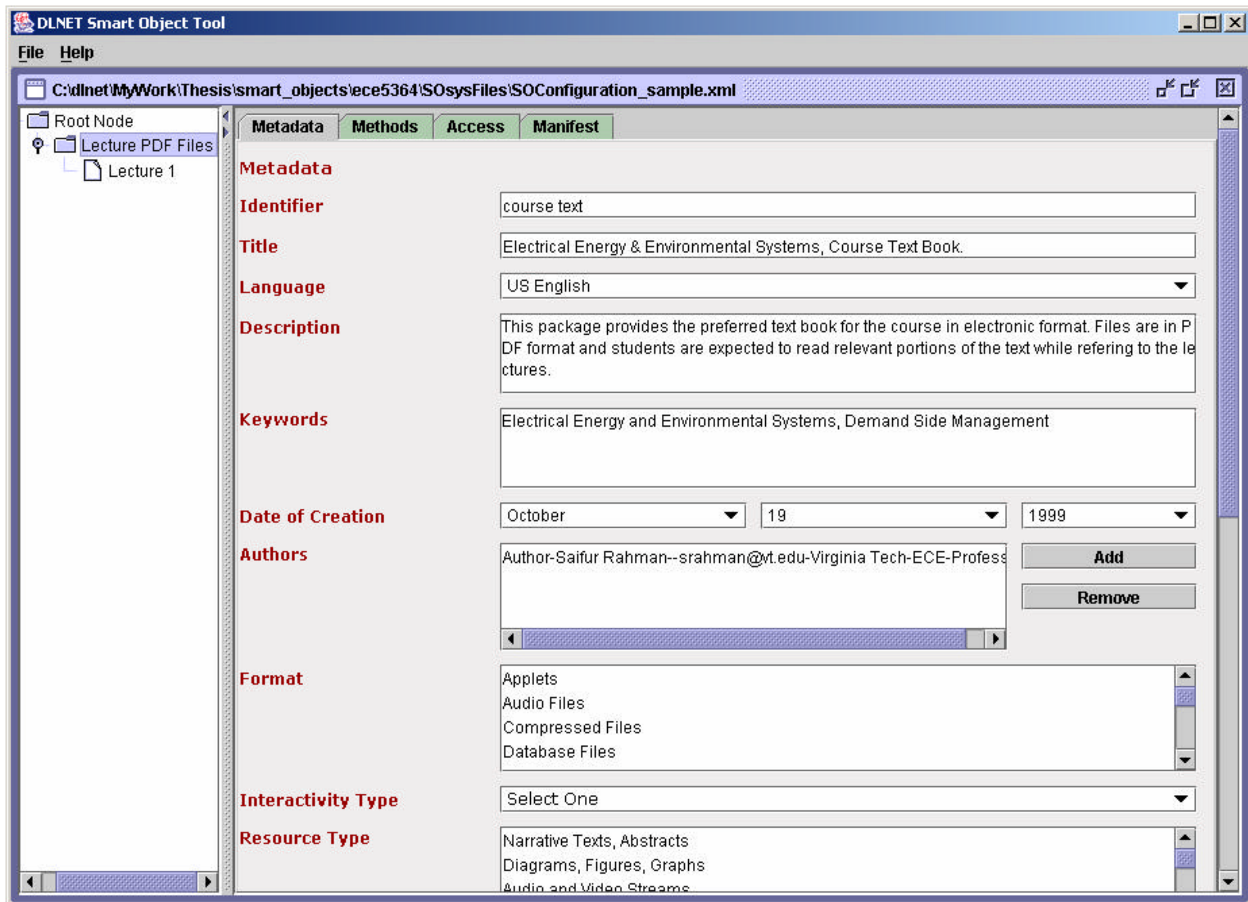

Figure 5.10. Smart Object example: Package/Element structuring information

Deployment: Once the smart object package has been created, it can be transported to the destination digital domain as a single unit (*.zip* or *.tar* file). As discussed in chapter 4, smart objects are platform-independent and operate inside any java-based web-container that provides a computing environment. For standards conformance purposes, smart objects have been

deployed and tested on containers conformant to the Java Servlet Specification[22] (version 2.2 and 2.3). Smart Objects have been deployed and tested on two different operating platforms (Windows 2000/XP and Red Hat Linux 8.0). Following are the instructions for deploying smart objects on Apache Tomcat, an open source implementation of the Java Servlet Specification. Note that these instructions specifically relate to the deployment of smart objects in a container and not to the general web server configuration. Web server documentation and manuals can be obtained

- The Tomcat container serves web resources based on 'Context's. A context, simply put is a web application running on a virtual host. It is represented by a 'Context' tag in the web container's configuration instance. A sample configuration snippet is shown in the figure below. Figure 5.11 shows two context definitions for Windows and Linux operating systems. The variable 'docBase' indicates the smart object deployment directory.

```
<!-- A sample context for hosting a Smart Object on Windows-->
<Context path="" docBase="C:/dlnet/MyWork/Thesis/smart_objects/ece5364" debug="0" reloadable="true"/>
<!-- A sample context for hosting a Smart Object on Linux -->
<Context path="/temp" docBase="/opt/smart_objects/smart_object_001" debug="0" reloadable="true"/>
<!-- Tomcat Examples Context -->
<Context path="/examples" docBase="examples" debug="0" reloadable="true" crossContext="true">
    <Logger className="org.apache.catalina.logger.FileLogger" prefix="localhost_examples_log." suffix=".txt" timestamp="true"/>
    <Ejb name="ejb/EmplRecord" type="Entity" home="com.wombat.empl.EmployeeRecordHome" remote="com.wombat.empl.Emplo
```

Figure 5.11. Sample Apache Tomcat Configuration instance

- Once the configuration instance has been modified, the compressed smart object package is decompressed into the *docBase* directory. Upon completion of this procedure, the container service will have to be restarted for the changes to take effect. The smart object should now be available at a URL: *http://foobar.org/index.jsp* for the windows system and *http://foobar.org/temp/index.jsp* for the Linux deployment.

Chapter Six

Conclusions and Future Work

6.1 Summary

Learning objects are key components to realizing digital pedagogy. Most of the learning object models currently available are built on the fundamental assumption that they would remain in the shadow/shelter of the parent digital library. In most cases this ideology is valid. However, such an assumption reduces learning objects to no more than passive, slave-like entities without any self-governing characteristics. In case their association with the parent library breaks for some reason (either due to learning object migration to a different domain or due to disappearance of the digital library), learning objects would then be no more than dumb entities (sequences of hypertext files) with no sustainability features. Over time, this would reduce the learning value of such objects ultimately rending them useless for pedagogy.

To address this shortcoming and to relieve the high degree of dependence of learning objects on digital libraries, this work has proposed a scalable, self-sufficient, interoperable, intelligent, technology-conformant learning object model. This work is based on a previously proposed learning object model[26] which suffered from scalability, interoperability and technology conformance issues. The smart object model proposed here has been built using Sun's Java programming language[20] and hence inherits Java's platform independence. It uses XML for storing metadata and object structuring and configuration information. The metadata conforms to the IMS Metadata Specification [14] while the content packaging scheme conforms to the IMS Content Packaging Specification [13]. Together, these two specifications lead the way to a popular, interoperable learning resource description framework and are widely accepted in the e-learning community. We believe that the adoption of these two specifications will encourage the inception of smart objects into already existing digital library frameworks. In addition, smart objects have at their disposal, a customized user repository stored as an XML instance which is used to specify access control restrictions on smart object components. Access control information is stored in a configuration instance along with structural and method information about the smart object.

Another important contribution of this work has been in the development of a client-utility that can be used to develop smart objects. The prototype tool is a 3-tier application with a highly modular architecture. It assists in the creation and/or modification of smart objects. It helps in realizing content aggregation, as well as defining the degree of intelligence and self-sufficiency supported by the smart object. It also assists in harvesting metadata for every smart object component along with assisting in building a user repository and defining access control restrictions on individual components.

6.2 Conclusions

A digital library's ability to formulate dynamic learning environments determines its operational effectiveness. Such an environment in the context of a non-warehouse library (i.e. a library that is more than just a data warehouse) would include dynamic learning resources the contents of which can be modified/restructured to stay in tune with technological developments and user interests. This thesis has developed and demonstrated a working model to realize this objective. Using this model, learning objects can be created to work under the umbrella of a digital library as well as be involved in an ad-hoc learning process without having to depend on the library's ability to provide learning and administrative services. This dissociates the longstanding dependence of learning objects on digital libraries for providing an environment of operation and pedagogy. In addition, the technology-conformance and modular design approach of this model (to the most popular metadata and content packaging schemes) enables its seamless integration into existing digital library infrastructures and customized learning environments.

6.3 Future Work

While this thesis has produced a scalable, intelligent learning object model and a prototype development tool to build smart objects, there is much scope for future research on this topic. Some of the potential areas of improvement to this work and possible extensions to the existing model are discussed below:

6.3.1 Additional Functionality

This work has given way to a self-sufficient, aggregate, intelligent, platform-independent and specification-conforming learning resource model which fits into existing digital library frameworks. Smart objects attain self-sufficiency and intelligence from using software

components called methods, which make them active entities as opposed to traditional passive learning objects. The more the number of methods available at its disposal, the more intelligent a smart object becomes. This work has remodeled more than half of the methods available in the original bucket model[26]. The recreated methods are thought of as being the keys to realizing the objective of this work (modeling learning objects capable of providing dynamic learning environments). In addition to these, we strongly feel the need to develop methods for gathering the runtime statistics (see 6.3.4) of the smart object. These methods combined with the runtime API of SCORM[37] would create a learning object that can act as a Learning Management System (LMS) and gather user learning statistics, thus contributing towards enrichment of its own metadata.

6.3.2 Centralized Policy Distribution

Smart objects provide a methodology for providing action-oriented services (*display* metadata, *add package*, *delete element* etc.). However, policies that determine operation and access restrictions (what methods to support and which user can access what) are left to the discretion of the administering authority. This authority can either be a digital library or an author. It is infeasible to individually administer these objects in a digital library environment. To alleviate this difficulty, a management tool (an extension of the development tool) can be built with the capability to connect to multiple smart object URLs simultaneously and automatically distribute operational preferences and policies.

6.3.3 Increased Security

Another area that requires constant work and contribution in smart object development is security. Web applications are constantly subject to unscrupulous attacks and poor design and security practices could very easily lead to an unsafe computing/learning environment. The current smart object model minimizes risk to the container hosting the smart object and to the smart object itself by following some basic security procedures. To the best of the author's knowledge, safe programming practices have been followed so as to minimize security risks. In addition, an authentication and authorization scheme prevents unauthorized access to learning resources. The user repository contains passwords stored using one-way encryption schemes (MD5[41] or SHA-1[9]). Additionally, access to protected resources can be restricted using the

access control feature available in the configuration file. This is a primitive scheme and can be built upon to provide a reasonable level of security for learning components.

Another realm of security for learning environments is the issue of copyrights. By providing a primitive user database (to specify access control restrictions on resources), we have laid a foundation stone to protecting learning resources from unauthorized access. However, to boast of complete copyright protection would be premature and unhealthy. Additional work in this area would involve studying the applicability of protocols like X.509[33] and digital signature[8] schemes to protect unauthorized/time-based access to learning resources.

6.3.4 SCORM conformance

This work started off with the objective of remodeling buckets[26] by providing an alternate implementation which is both scalable and is conformant to well-known standards and specifications in e-learning. Preliminary studies of scalable learning objects indicated a strong relation of the proposed smart object model to a widely accepted scalable, aggregate, interoperable learning resource model heading towards standardization, the SCORM[37]. SCORM, a DoD initiative, is a liaison of leading industrial, educational and government agencies (ADL initiative,[2]) working towards the development and improvement of e-learning standards and specifications. The SCORM has been discussed in detail in chapter 2. To summarize, a learning object is deemed SCORM-compliant if it provides a Content Aggregation Model to represent learning content and a Runtime Model to communicate with a Learning Management System. This thesis comes close to providing a SCORM-like Content Aggregation Model[36] to Smart Objects. It provides a content aggregation (content structuring) scheme for learning content by defining individual learning items as *elements* and aggregated learning components as *packages* (SCORM calls these *Assets* and *Sharable Content Objects* respectively), IMS/IEEE metadata scheme[15,14] and an IMS content packaging scheme[13]. Future work is envisioned to develop a Runtime environment enabling smart objects to be able to communicate with an LMS via a learner. This would result in SCORM objects with intelligence and self-sufficiency to go with aggregation and interoperability and would lead to the development of SCORM-compliant Smart Objects (ScSO).

Appendix – Smart Object API

Smart Object Method	add_element		
Arguments	*method*	String type with value 'add_element'	*Required*
	package	String type with value set to package name to add element to	*Optional*
	new_element_name	String type representing the new element name	*Required*
	new_element_display_name	String type representing a display name for the element	*Optional*
	element_file	File object to be added	*Required*
MIME Type	text/html		
HTTP Methods	GET – provides interfaces/forms for adding element POST – posts element related data to the smart object		
Example	*http://foobar.edu/some_path/index.jsp?method=add_element&package=pkg1&new_element_name= sample_element& new_element_display_name='A Sample Element'&element_file=<file>*		
Description	The add_element method can be used to add new elements to a smart object. The HTTP GET method can be used to retrieve current element and package information while the POST method can be used to execute an element addition. The user adding an element to an existing smart object structure should have write permissions to the package.		

Smart Object Method	add_metadata		
Arguments	*method*	String type with value ‘add_metadata’	*Required*
	package	String type representing the name of the package (eg. pkg1.pkg2)	*Optional*
	element	String type representing the name of the element (eg. elem1)	*Optional*
	…………………	Other custom metadata parameters	*Custom*
MIME Type	text/html		
HTTP Methods	GET – provides interfaces/forms for displaying package/element metadata. POST – submits package/element metadata modifications. (ACL enabled)		
Example	*http://foobar.edu/some_path/index.jsp?method=add_metadata&package=pkg1.pkg2&Title='A Sample title'&Description='………'………*		
Description	The add_metadata method can be used to add metadata to a particular component of the smart object. This component can be a package, and element or the smart object itself. Since metadata is presented on a form-based interface, a number of customizations are possible. The metadata collected can vary depending on application. To invoke this method requires that a user have modify privileges on the component being modified.		

Smart Object Method	add_package		
Arguments	*method*	String type with value ‘add_package’	*Required*
	package	String type representing the name of the package (eg. pkg1.pkg2)	*Optional*
	new_package_name	String type representing the name of the package to be added (eg. new_package)	*Required*
	new_package_display _name	String type representing the display name of the new package (eg. ‘A sample chapter’)	*Optional*
MIME Type	text/html		
HTTP Methods	GET – provides interfaces/forms for traversing/adding packages. POST – posts new package information to be created (ACL restrictions)		
Example	*http://foobar.edu/some_path/index.jsp?method=add_package&package=pkg1 &new_package_name=sample_package& new_package_display_name=’A Sample Package’*		
Description	The add_package method can be used to add a new package to a smart object. The HTTP GET method can be used to traverse to the desired package while the POST can be used to trigger the addition of a new package. Note that the GET method can also be used to the same effect as the POST method		

Smart Object Method	add_principal		
Arguments	*method*	String type with value ‘add_principal’	*Required*
	principal	String type representing the new user id to be added	*Required*
	passwd	String type representing the password of the new user	*Required*
	UserType	String type representing user type (User/Administrator)	*Required*
MIME Type	text/html		
HTTP Methods	GET – provides interfaces/forms for adding a new user/principal POST – posts data for adding a new user (Administrative restrictions apply)		
Example	*http://foobar.edu/some_path/index.jsp?method=add_principal&principal=Admin&passwd=Admin_pwd&UserType=Administrator*		
Description	The add_principal method can be used to make additions to the user repository that every smart object is equipped with. The execution of this method requires administrative privileges. Execution can be effected via the GET or POST methods.		

Smart Object Method	add_tc		
Arguments	*method*	String type with value ‘add_tc’	*Required*
	blockedid	String type representing an ID whose login attempts should be denied	*Required*
	package	String type representing the package whose permissions need to be changed	*Optional*
	element	String type representing the element whose permissions need to be changed	*Optional*
	adduserid	String type representing the UserID whose permissions on a package/element are being modified	*Required*
	permission	Strung type with three binary characters representing permissions	*Required*
MIME Type	text/html		
HTTP Methods	GET/POST – provides interfaces/forms for adding terms and conditions.		
Example	*http://foobar.edu/some_path/index.jsp?method=add_tc&blockedid=Administrator* *http://foobar.edu/some_path/index.jsp?method=add_tc&package=pkg1.pkg2&adduserid=sample&perm=100*		
Description	The add_tc method can be used to specify access control restrictions on smart object components. The currently implementation supports denying a user access to certain components and actions of the smart object as well as rejecting remote access features for certain users. The above example illustrates the two terms and conditions currently supported.		

Smart Object Method	delete_element		
Arguments	*method*	String type with value 'delete_element'	*Required*
	package	String type representing the package which contains the to be deleted element	*Optional*
	delete_element_name	String type representing the element name to be deleted from a package	*Required*
MIME Type	text/html		
HTTP Methods	GET/POST – provides listing of elements and sub-packages of a package, triggers the deletion of an element (ACL restrictions apply)		
Example	*http://foobar.edu/some_path/index.jsp?method=delete_element&package=pkg1&delete_element_name=elem1*		
Description	The delete_element method can be used to delete existing elements of a smart object. To execute such a deletion, the logged in user must have modify privileges over the parent package containing the element to be deleted.		

Smart Object Method	delete_package		
Arguments	*method*	String type with value 'delete_package'	*Required*
	package	String type representing the package which contains the to be deleted package	*Optional*
	delete_package_name	String type representing the package to be deleted from a package	*Required*
MIME Type	text/html		
HTTP Methods	GET/POST – provides listing of elements and sub-packages of a package, triggers the deletion of a package (ACL restrictions apply)		
Example	*http://foobar.edu/some_path/index.jsp?method=delete_package&package=pkg1&delete_package_name=pkg3*		
Description	The delete_package method allows for the deletion of existing packages of a smart object. To delete an existing package, the logged in user must have modify privileges over the parent package.		

Smart Object Method	delete_principal	
Arguments	*method*	String type with value 'delete_principal' *Required*
	principal	String representing UserID to be deleted *Required*
MIME Type	text/html	
HTTP Methods	GET/POST – provides interfaces/forms for deleting a user from the repository	
Example	*http://foobar.edu/some_path/index.jsp?method=delete_principal&principal=sample_id*	
Description	This method can be used to modify the user repository of a smart object. Existing users can be deleted by a user with administrative privileges. Deletion of a user from the repository also assures deletion of all access control details relating to the deleted user from the smart object configuration instance as well.	

Smart Object Method	delete_logs		
Arguments	*method*	String type with value ‘delete_logs’	*Required*
	date	String representing the log date to be deleted (in YYYY-MM-DD format)	*Required*
MIME Type	text/html		
HTTP Methods	GET/POST – provides interfaces for listing the log files collected by the smart object as well as for deleting them		
Example	*http://foobar.edu/some_path/index.jsp?method=delete_logs&date=2003-03-04*		
Description	The delete_logs method can be used to delete logs generated by the smart object. Logs contain information pertaining to the smart object methods accessed and user information. This method is again an administrative action and hence can be performed only by users that are smart object administrators. Logs are created, identified and deleted based on the date.		

Smart Object Method	delete_metadata		
Arguments	*method*	String type with value 'delete_metadata'	*Required*
	package	String representing the package containing the sub-package/element whose metadata is to be deleted	*Optional*
	element	String representing the element whose metadata is to be deleted	*Optional*
MIME Type	text/html		
HTTP Methods	GET – provides interfaces for deleting the metadata of a particular component of a smart object		
Example	*http://foobar.edu/some_path/index.jsp?method=delete_metadata&package=pkg1.pkg11&element=elem1*		
Description	The delete_metadata method can be used to delete metadata pertaining to a particular component of the smart object. The operation requires at least modify privileges on the component being modified.		

Smart Object Method	delete_tc		
Arguments	*method*	String type with value 'delete_tc'	*Required*
	blockedid	String type representing the UserID that has to be unblocked	*Required*
	package	String type representing the package name whose permissions need to be changed	*Optional*
	element	String type representing the element name whose permissions need to be changed	*Optional*
	deleteuserid	String type representing the UserID whose permissions need to be reset (deleted)	*Required*
MIME Type	text/html		
HTTP Methods	GET/POST – provides an interface for deleting existing terms and conditions		
Example	*http://foobar.edu/some_path/index.jsp?method=delete_tc&package=pkg1.pkg2&deleteuserid=sample*		
Description	The delete_tc method can be used by an administrator to delete previously specified terms and conditions of operation of a smart object. Access control restrictions can be undone using this method. Users can be unblocked from logging into the smart object and access control restrictions on certain smart object components can be lifted using this method.		

Smart Object Method	display		
Arguments	*method*	String type with value 'display'	*Required*
	package	String type representing the name of the package to be displayed (eg. pkg1.pkg2)	*Optional*
	element	String type representing the name of the element to be displayed (eg. elem1)	*Optional*
	start	Boolean value indicating if the element should be played	*Optional*
MIME Type	text/html		
HTTP Methods	GET/POST – retrieves metadata for any component of the smart object. Also responsible for retrieving the learning components (elements) and presenting to the user.		
Example	*http://foobar.edu/some_path/index.jsp?method=display&package=pkg1&element=elem1&start=true*		
Description	The display method is responsible for retrieving metadata information of a smart object component. This information can be used in the learning process of a user. The method also delivers the end resource to the user. Access control restrictions on components can be realized using this method.		

Smart Object Method	id		
Arguments	*method*	String type with value 'id'	*Required*
	package	String representing the package whose ID is to be retrieved	*Optional*
	element	String representing the element whose ID is to be retrieved	*Optional*
MIME Type	text/html		
HTTP Methods	GET/POST – retrieves the ID of the smart object or a component		
Example	*http://foobar.edu/some_path/index.jsp?method=id&package=pkg10.pkg20*		
Description	The id method retrieves the identifier assigned to the smart object or a component within it and displays it to a user. Every component of the smart object can be assigned an identifier, either local or global (CNRI).		

Smart Object Method	list_logs		
Arguments	*method*	String type with value 'list_logs'	*Required*
	date	String representing the date in YYYY-MM-DD format	*Required*
MIME Type	text/html		
HTTP Methods	GET/POST – lists the logs that were collected by the smart object		
Example	*http://foobar.edu/some_path/index.jsp?method=list_logs&date=2003-04-04*		
Description	The list_logs method can be used by an administrator to monitor the usage statistics. Smart objects record request and response information which can then be audited by an administrator. Logs are stored and retrieved based on the dates of access.		

Smart Object Method	list_methods		
Arguments	*method*	String type with value ‘list_methods’	*Required*
	xml	*Boolean value to force output as XML*	*Optional*
MIME Type	text/html		
HTTP Methods	GET/POST – lists the methods that are supported by the smart object either as HTML or as XML		
Example	*http://foobar.edu/some_path/index.jsp?method=list_methods&xml=true*		
Description	The list_methods method is responsible for listing the currently supported smart object methods. This listing can be presented to the user either as hypertext or XML markup.		

Smart Object Method	list_source		
Arguments	*method*	String type with value 'list_source'	*Required*
	action	String representing the method whose source is to be retrieved	*Required*
MIME Type	text/html		
HTTP Methods	GET/POST – lists the source code of a particular smart object method		
Example	*http://foobar.edu/some_path/index.jsp?method=list_source&action=add_package*		
Description	The list_source method lists the source code that enables actions to be performed on a smart object. The method lists Java code of all the modules of a smart object method.		

Smart Object Method	metadata		
Arguments	*method*	String type with value 'metadata'	*Required*
	package	String representing the package whose metadata is to be retrieved	*Optional*
	element	String representing the element whose metadata is to be retrieved	*Optional*
	format	String representing the output metadata format (eg. dc, ims etc.)	*Required*
MIME Type	text/xml		
HTTP Methods	GET/POST – retrieves the complete metadata of a particular component of the smart object as XML.		
Example	*http://foobar.edu/some_path/index.jsp?method=metadata&format=dc*		
Description	The metadata method can be used to obtain metadata information about a particular package or element of the smart object. It can also be used to obtain the metadata of the entire smart object. the output format is currently xml and can be formatted according to various metadata specifications (dc, ims are currently supported)		

Smart Object Method	pack		
Arguments	*method*	String type with value 'pack'	*Required*
MIME Type	text/html		
HTTP Methods	GET/POST – streams the smart object in compressed format according to IMS content packaging specification		
Example	*http://foobar.edu/some_path/index.jsp?method=pack*		
Description	The pack method can be used for inter-domain smart object transportation or replication. It sends out a compressed smart object byte stream which is compliant to IMS content packaging specifications.		

Smart Object Method	set_version		
Arguments	*method*	String type with value 'set_version'	*Required*
	version	String type indicating a new version	*Required*
MIME Type	text/html		
HTTP Methods	GET/POST – can be used to set the version of a smart object (Administrative access only)		
Example	*http://foobar.edu/some_path/index.jsp?method=set_version&version=1.0.J.B*		
Description	The set_version method can be used to change the smart object version. A change may be needed if the modules of the smart object are updated or new methods are added. The execution of this command requires administrative access to the smart object		

Smart Object Method	version		
Arguments	*method*	String type with value ‘version’	*Required*
	xml	Boolean value (true/false)	*Optional*
MIME Type	text/html, text/xml		
HTTP Methods	GET/POST – retrieve smart object version information as html or xml		
Example	*http://foobar.edu/some_path/index.jsp?method=version*		
Description	The version method retrieves the current smart object version either as hypertext or xml and displays it to the user. This method does not require any administrative authority.		

References

[1]. A format for Bibliographic records, RFC 1807, http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc1807.html.

[2]. Advanced Distributed Learning Initiative, http://www.adlnet.org.

[3]. Apache Xerces2 Java Parser, http://xml.apache.org/xerces2-j/index.html.

[4]. Brandon Hall, New Technology Definitions glossary, http://www.brandonhall.com/public/glossary/glossary.html#Learning Management System (LMS).

[5]. Corporate for National Research Initiatives, The Handle System, http://www.handle.net/.

[6]. Diane Hillmann, NSDL Metadata Primer, http://metamanagement.comm.nsdlib.org/outline.html.

[7]. Digital Library Network for Engineering and Technology, http://www.dlnet.vt.edu.

[8]. Digital Signature, Definition, http://whatis.techtarget.com/definition/0,289893,sid9_gci211953,00.html.

[9]. D. Eastlake, P. Jones, US Secure Hash Algorithm 1 (SHA1), Request for Comments: 3174, http://www.faqs.org/rfcs/rfc3174.html.

[10]. Dublin Core Metadata Initiative (DCMI), http://purl.org/metadata/dublin_core.

[11]. Edward T. O'Neill, Brian F. Lavoie, Patrick D. McClain, Web Characterization Project: An Analysis of Metadata Usage on the Web, http://www.oclc.org/research/publications/arr/1998/oneill_etal/metadata.htm.

[12]. Encoding Dublin Core Metadata in HTML RFC-2731, http://zvon.org/tmRFC/RFC2731/Output/index.html.

[13]. IMS Content Packaging Specification version 1.1.2.

[14]. IMS Metadata Specification version 1.2.1.

[15]. IMS/IEEE Learning Object Metadata standard version 1.0.

[16]. Java 2 Enterprise Edition (J2EE), http://java.sun.com/j2ee/

[17]. Java 2 Standard Edition, http://java.sun.com/j2se.

[18]. Java API for XML Processing, http://java.sun.com/xml/jaxp/index.html.

[19]. Java Foundation Classes: Cross-Platform GUIs & Graphics, http://java.sun.com/products/jfc/.

[20]. Java Programming Language, http://java.sun.com.

[21]. Java Server Pages (JSP), http://java.sun.com/products/jsp/.

[22]. Java Servlet Technology, http://java.sun.com/products/servlet/.

[23]. Learning Object representation in DLNET, White Paper. URL: http://www.dlnet.vt.edu.

[24]. Metadata - A primer, http://www.clubi.ie/webserch/metadata.htm.

[25]. Michael Lesk, Why Digital Libraries? Bellcore and University College London, UKoln University, June 1995.

[26]. Michael Nelson, PhD dissertation, Buckets: Smart Objects in Digital Libraries. August 2000.

[27]. Microsoft Learning Resource Interchange (LRN) 3.0 Toolkit, http://www.microsoft.com/technet/itsolutions/education/deploy/lrntoolkit/toolkitl.asp

[28]. Microsoft Windows Operating System, http://www.microsoft.com/windows/default.mspx

[29]. Miltiadis D. Lytras, Georgios I. Doukidis, Expanding e-learning effectiveness.The shift from content orientation to knowledge management utilization, ED-MEDIA 2001, Word Conference on Educational Multimedia, Hypermedia and Telecommunications, Tampere, Finland, June 2001.

[30]. Red Hat Linux 8.0, http://www.redhat.com.

[31]. National Science Digital Library Initiative, http://www.nsdl.org

[32]. Pennsylvania Spatial Data Access system (PASDA), http://www.pasda.psu.edu/.

[33]. Public-Key Infrastructure (X.509) (pkix) Charter, http://www.ietf.org/html.charters/pkix-charter.html.

[34]. Robert Kahn, Robert Wilensky, A Framework for Distributed Digital Object Services, May 13, 1995;cnri.dlib/tn95-01., http://www.cnri.reston.va.us/home/cstr/arch/k-w.html

[35]. R.Rivest, Network Working Group, Request for Comments: 1321, http://www.faqs.org/rfcs/rfc1321.html

[36]. Sharable Content Object Reference Model (SCORM), Content Aggregation Model, Advanced Digital Library Initiative, 2001.

[37]. Sharable Content Object Reference Model (SCORM), The SCORM Overview, Advanced Digital Library Initiative, 2001.

[38]. S. Weibel, J. Kunze, C. Lagoze, M. Wolf, Dublin Core Metadata for Resource Discovery, RFC2413, Sept 1998.

[39]. The Digital Object Identifier (DOI) System, http://www.doi.org

[40]. The Digital Object identifier Handbook, http://dx.doi.org/10.1000/182

[41]. The MD5 Message-Digest Algorithm, R.Rivest, Network Working Group, Request for Comments: 1321, http://www.faqs.org/rfcs/rfc1321.html

[42]. Webopedia's definition of a Three-tier architecture, http://www.webopedia.com/TERM/T/three_tier.html.

[43]. White Paper, Using Microsoft Technology to Build a Learning Organization, http://www.microsoft.com/elearn.

[44]. William Yeo Arms, Digital Library Research in the United States, Corporation for National research Initiatives. UKoln University, September 1995.

Vitae

Vara Prashanth Pushpagiri was born on November 18, 1978 in Nellore, India. He earned a Bachelor's in Electrical and Electronics Engineering for the Jawaharlal Nehru Technological University, Hyderabad in June 2000. He then joined the Bradley Department of Electrical and Computer Engineering at Virginia Tech in Fall 2000. In Spring 2001, he moved to the Alexandria Research Institute (ARI), where he worked on the Digital Library Network for Engineering and Technology (DLNET) under the supervision of Professor Saifur Rahman, director of the institute. He completed his thesis at the ARI in Summer 2003 and is interested in pursuing a research-oriented career in Computer and Network Security.

He is a student member of the IEEE and the IEEE Computer Society.