INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

The Web Interfacing Repository Manager: A Framework for Developing Web-Based Experiment Management Systems

by

Rex M. Jakobovits

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1999

Program Authorized to Offer Degree:
Department of Computer Science and Engineering

UMI Number: 9944128

UMI Microform 9944128
Copyright 1999, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

In presenting this thesis in partial fulfillment of the requirements for Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of the dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copies or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature ______________________

Date 7/14/99

University of Washington

Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Rex M. Jakobovits

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of Supervisory Committee:

James F. Brinkley

Reading Committee:

Linda G. Shapiro

Cornelius Rosse

Date: July 14, 1999

University of Washington

Abstract

The Web Interfacing Repository Manager: A Framework for Developing Web-Based Experiment Management Systems

Rex M. Jakobovits

Chairperson of the Supervisory Committee:
Professor James F. Brinkley
Departments of Biological Structure and Computer Science & Engineering

Recent advances in tools for scientific data acquisition, visualization, and analysis have lead to growing information management problems for research laboratories. An exponential increase in the volume of data, combined with a proliferation of heterogeneous formats and autonomous systems, has driven the need for flexible and powerful Experiment Management Systems (EMS). This dissertation provides a detailed analysis of the informatics requirements of an EMS, and proposes a new type of middleware called an EMS-Building Environment (EMSBE), which enables the rapid development of web-based systems for managing laboratory data and workflow. A new data model and design methodology for modeling experiment data as hierarchical views is described, which provides a framework for the rapid development of drill-down navigation systems that adapt themselves to the context of the user. A layered software architecture for an object-relational EMSBE is then described. Based on that architecture, an application server has been implemented, called the Web-Interfacing Repository Manager (WIRM). As a demonstration of the effectiveness of the data model and methodology, WIRM is used to build a working EMS that manages the complex data and workflow of a neuroscience research project.

TABLE OF CONTENTS

LIST OF FIGURES .... **iv**

LIST OF TABLES .... **vii**

Chapter 1: Introduction **1**

Chapter 2: Requirements Analysis **8**

2.1 Seven Roles of an Effective EMS 9

2.2 Ten Requirements of an EMSBE 22

Chapter 3: Data Model **30**

3.1 Repository Schema Model 31

3.2 Query-By-Context View Model 33

Chapter 4: WIRM Architecture **43**

4.1 Architectural Overview 43

4.2 Server Components 45

4.3 Built-in Classes 51

4.4 System Wirmlets 53

4.5 Constructing a Web View 56

Chapter 5: WIRM Implementation **61**

5.1 Persistent Perl 61

5.2 Server Component Implementation 63

Chapter 6: Methodology **72**

6.1 Designing Schemas 73

6.2 Modeling Class Definitions 77

6.3 Developing Custom Wirmlets 91

6.4 Evolving Schemas and Wirmlets 102

Chapter 7: Results & Experience **107**

7.1 Overview of the Brain Mapper Experiment 107

7.2 History of the EMS Development 109

7.3 The Brain Mapper Schema 114
7.4 The Brain Mapper Wirmlets 129
7.5 Evaluation of the Benefits of WIRM 138
Chapter 8: Related Work 142
8.1 Existing Commercial Systems 142
8.2 Research Systems for Web Site Management 143
8.3 Research Systems for Laboratory Information Management 146
Chapter 9: Future Work 149
9.1 Adopt a True Client-Server Architecture 149
9.2 Heighten Context Sensitivity 150
9.3 Become Purely Object-Oriented 150
9.4 Improve Support for Aggregate Types 151
9.5 Integrate with more Applications and Data Types 152
9.6 Provide Transparent Swizzling 153
9.7 Adopt a Better Transaction Model 155
9.8 Separate Content from Presentation 156
9.9 Refine Workflow Tools 156
9.10 Support an ASP Approach 157
Bibliography 160
Appendix A: WIRM API Reference 170
A.1 FSA Controller 170
A.2 Visualization Cache Manager 171
A.3 Table Manipulator 175
A.4 Repository Object Interface 178
A.5 HTML Generator 185
A.6 Gateway Interface 189
Appendix B: Using WIRM to Build Portals 195
B.1 NeuroPortal: a Gateway to the Human Brain Project 195
B.2 The Need for a Gateway 197

B.3 The Need for a Knowledge Base 199
B.4 The Need for a Forum 200
Appendix C: Deblobification 202

LIST OF FIGURES

Figure 1: Roles of a Repository Manager 3
Figure 2: Three-Tiered Software Development Approach 5
Figure 3: A Uniform Query Interface for Brain Mapping Data 12
Figure 4: User Authentication 14
Figure 5: Making an Annotation 15
Figure 6: Uploading a Photo 16
Figure 7: File Metadata 17
Figure 8: Spatial Navigation of Language Site Data 19
Figure 9: MR Image Converted for Web Viewing 25
Figure 10: Schema Evolution 26
Figure 11: Repository Explorer 27
Figure 12: Multiple Context Dimensions 34
Figure 13: WIRM Architecture 44
Figure 15: Repository Object API 48
Figure 16: Gateway API 50
Figure 17: Generic WIRM Console 54
Figure 18: Invoking a Wirmlet 57
Figure 19: Retrieving a Patient 58
Figure 20: Constructing a Page View 60
Figure 21: Embedding an Image in a View 60
Figure 22: Persistent Perl 62
Figure 23: Defining the Book Schema with the Schema Definition Tool 75
Figure 24: Making a Book with the Object Maker 77
Figure 25: Default Row View for Book Class 80
Figure 26: Row View Using Custom Label 81
Figure 27: Custom Row View for Book 83
Figure 28: Custom Page View for Book 85

Figure 29: Default Make Prompt for Loan 86
Figure 30: Custom Make Prompt for Loan 88
Figure 31: Result of Making a Loan 90
Figure 32: Default Main Menu 91
Figure 33: Customized Main Menu 95
Figure 34: Prompt for Book Return 97
Figure 35: Book Return Results 98
Figure 36: Upload Cover Prompt 100
Figure 37: Adding an Attribute 103
Figure 38: Context-Sensitive Main Menu 106
Figure 39: Multimedia Patient Data 108
Figure 40: Creating a 3D Model 109
Figure 41: Workflow Management 110
Figure 42: Web-Database Connectivity Solution 112
Figure 43: The WIRM Web Site 113
Figure 44: Brain Mapper Class Hierarchy 114
Figure 45: Two Views of a Patient 116
Figure 46: Photo View and File View 119
Figure 47: Exam View 120
Figure 48: Series View 121
Figure 49: MR Slice View 122
Figure 50: Model View 124
Figure 51: Multiple Renderings 125
Figure 52: Study Record (Main View) 126
Figure 53: Study Record (Detailed View) 127
Figure 54: Brain Mapper Main Menu & Patient Browser 129
Figure 55: Patient Workflow Console and Make Patient Wirmlet 131
Figure 56: Patient Update 133
Figure 57: Uploading a Spreadsheet 134

Figure 58: Editing Trials By Hand 135
Figure 59: Photo Update Wirmlet 136
Figure 60: Data Analysis Console 137
Figure 61: Araneus Data Model 143
Figure 62: Restructuring a Web Site with Araneus 144
Figure 63: Zoo Architecture 147
Figure 64: Knowledge-Based Deblobification 204

LIST OF TABLES

Table 1: Brain Mapper Information Sources 12
Table 2: Brain Mapper Access Regulation 22
Table 3: Query-By-Context Notation 35
Table 4: File Schema 51
Table 5: Annotation Schema 52
Table 6: User Schema 53
Table 7: Book Schema 74
Table 8: Loan Schema 74
Table 9 : Lines of Code in Brain Mapper EMS 140
Table 10: Lines of Code in WIRM API's 140
Table 11: Size of WIRM and Brain Mapper EMS 141

Acknowledgements

I owe many thanks to a number of people who have been inspirational to me throughout my graduate studies. I'd like to thank my advisors Jim Brinkley and Linda Shapiro, who have taught me a tremendous amount about conducting research and have guided my growth as a graduate student. I would also like to thank Cornelius Rosse, who inspired me to cultivate my sitzfleisch when I needed it most.

I would also like to thank my family: sister Joy who through her amazing courage has shown me what is important in life, Mom & Chet who have been so generous and kind, and Dad and Dina, who have taught me how to be a scholar.

And I am ever in debt to my wonderful friends and colleagues, Derrick Weathersby, Marc Fiuczynski, and Kevin Hinshaw, who have always been there for me throughout these long years of graduate school. Now it's your turn, guys!

Finally, I would like to thank my cherished wife-to-be, Allegra Giacchino, whose constant love and support has given me the strength and confidence to go all the way.

Dedication

To Allegra

Chapter 1: Introduction

In the past decade, new generations of research tools have given rise to an explosion of data to be managed by scientists in every discipline. Advances in hardware and software applications for data acquisition, visualization, and analysis have lead to growing information management problems both within experiments and across multiple projects. Experiments are becoming harder to manage because of four basic trends:

1) *Exponential increase in volume of data* – advances in technology are allowing scientists to record and study natural phenomena at an increasing level of precision and depth, which has lead to a corresponding increase in the volume of data collected, often in the form of multimedia files [Sha94].

2) *Proliferation of formats* – the increase in data volume has been accompanied by an increase in data formats for storage, organization, and dissemination of heterogeneous data. These data files may consist of a wide range of image, sound, and video types, custom-formatted binary data, and ASCII dumps of alphanumeric tables. Many formats are proprietary, and the convergence towards compatible standards is slow. In addition to the actual data files, experiment management involves handling a wide range of ancillary file objects that aren't traditionally considered to be ''data'', such as the procedural scripts used to generate the data, documentation, etc.

3) *Trend towards distributed heterogeneous systems* – experiments are becoming increasingly reliant on the interoperability of uncoordinated software applications. Unlike earlier generations of experiments that ran on single monolithic systems, today's applications often run on multiple platforms and require complex data transformations to work together.

4) *Increased collaboration* – there is a growing trend towards consortium-based projects, in which experiments are performed by teams of collaborating scientists from multiple disciplines across institutions, rather than within single labs.

These four trends combined result in experiments that require a far more sophisticated data management process than their predecessors. A typical example is the Structural Informatics Group's Brain Mapping Project [Mod97b, Sig-URL], being developed by the University of Washington as part of the national Human Brain Project [Kos97]. The project's goal is to develop an information framework for managing cortical stimulation data obtained during neurosurgery [Oje89]. An interactive Brain Mapping tool allows a neurosurgeon to visually match an intra-operative photo of a patient's brain surface to a 3-D MRI-based visualization of the patient's brain. The experiment requires fine-grained collaboration and data sharing among radiologists, neurosurgeons, neuroscientists, statisticians, computer vision experts, database administrators, and a number of technicians, students, and assistants. Functional brain mapping data consists of thousands of ordered MRI slices grouped into exams, 3-D rendered brain images, digitized photographs, lists of identified site coordinates, and alphanumeric tables of patient demographics. A wide range of heterogeneous software applications is called upon to interact in a complex workflow process.

Without the proper tools, experiment management becomes an overwhelming task as the amount of file-based information increases. Four major requirements need to be addressed when handling scientific multimedia data: metadata management, query support, user interface construction, and application interfacing.

Each of these requirements can be addressed by a number of popular technologies. Metadata management can be aided by enforcing strict directory maintenance through a revision control system [Tic85]. Queries can be supported by importing tabular data to a relational database. User interfaces can be constructed as CGI or Java applications. Files

can be interfaced to applications by Perl scripts, the language of choice for managing processes and handling files [Wal91].

A number of competing off-the-shelf products attempt to provide solutions that are more complete by integrating these features within a single tool (see Figure 1). Commercial repository systems such as SAP/R3 [Wil99] are marketed as "enterprise solutions", aimed at large corporations. The drawbacks of these commercial systems are that they are geared towards business data, and they require a significant investment in monetary and personnel resources. In addition to the high price tag of the software itself (often costing thousands of dollars), the laboratory may need to invest in new hardware and personnel training. Furthermore, when the data management application is tied into a proprietary vendor's system, the software no longer can be freely distributed within the research community.

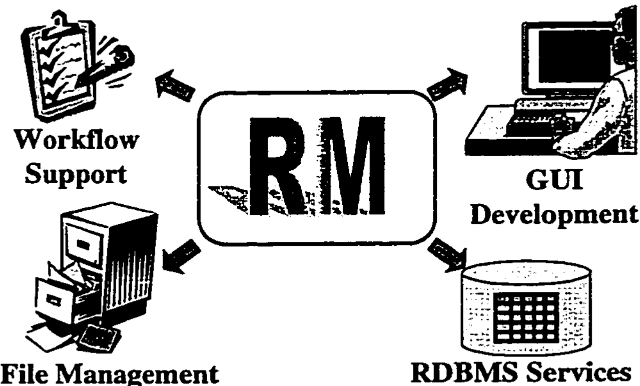


Figure 1: Roles of a Repository Manager

What is really needed is a class of "enterprise software" geared towards the laboratory, which can manage all aspects of the experiment lifecycle, including the design, collection, and exploration of data. Such systems are known as "Laboratory Information Management Systems" (LIMS) or "Experiment Management Systems" (EMS). A number of vendors supply LIMS aimed at large clinical laboratories [Lim-URL]. These systems tend to provide customized solutions for commercial industries such as utilities

or pharmaceuticals. However, there are no commercial products which adequately serve the needs of smaller research labs, whose IT budgets can't afford a commercial LIMS, and whose data types may be far more complex than those of larger commercial labs. There has been a surprising lack of general-purpose research in this area. One leading group in this area is the Zoo project at the University of Wisconsin [Ioa96]. They point out that "little attention has been devoted to small teams of scientists... there are no adequate experiment management tools that are powerful enough to capture the complexity of the experiments and yet at the same time are natural and intuitive to the non-expert."

The reason for this dearth of solutions is the diverse nature of experiment data and interface requirements. The variability and complexities of experiments prohibits the feasibility of building a general-purpose solution. There can be no "silver bullet" EMS which adapts itself for a wide range of domains. Instead, each EMS must be custom-built for a single problem domain.

To develop an EMS from scratch is very hard. The developer must interface to non-standard components, manage extremely complex data, and provide multiple interfaces for diverse classes of users. The type of programmer needed to build these solutions from the bottom up is prohibitively expensive. To make EMS-building feasible, a three-tiered approach must be taken: data and files should be kept separate from the applications, and a suite of high-level application programming interfaces (middleware) should be used to tie them together (see Figure 2).

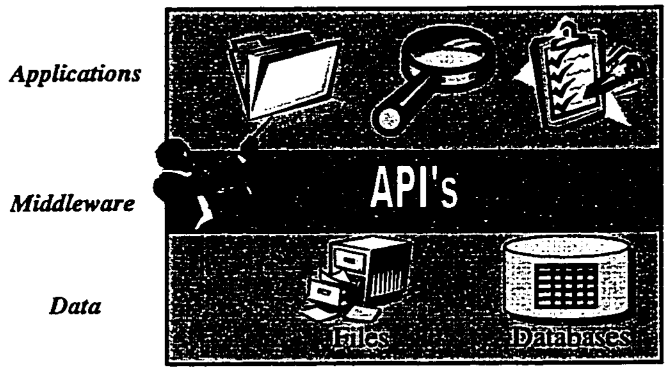


Figure 2: Three-Tiered Software Development Approach

This dissertation identifies a new class of middleware: the EMS-Building Environment (EMSBE) and describes a working EMSBE called the Web-Interfacing Repository Manager (WIRM) [Jak97b]. The EMSBE is an application server, which facilitates the rapid development of an EMS. WIRM has the following features:

1) *Web-Based:* An EMS developed with WIRM uses the Web as a front-end, so it can be accessed from anywhere on any platform.
2) *Object-relational:* WIRM is based on an object-relational data model, which can be extended to handle complex domain types.
3) *Rapid Application Development:* An EMS built on WIRM uses the high-level Perl scripting language, which provides powerful constructs for rapid prototyping and interfacing.

4) *Context-Sensitive View Definitions:* WIRM supports adaptive, customizable "context-sensitive" view definition.
5) *Freeware Components:* unlike commercial repository systems, WIRM leverages off of a huge base of free open-source software, so an EMS built in WIRM can be freely distributed.

Applications, which use the Web as a front-end, are becoming more common. Inevitably, complex applications are moving towards the Web-Information System (WIS) model, in which the data and logic of the application reside on a remote server, accessed by a thin client. This has the natural advantage of freeing the end-user from application and data maintenance, and allows the resource-intensive tasks to be outsourced to an application service provider. Many experts agree that "Web-based information systems will become more pervasive than client/server systems did a decade ago, with an exponentially higher impact on our lives..." [Isa98]. The lack of mature WIS-building tools is holding back this evolution, so the development of application servers such as WIRM is an important new area of research.

This dissertation is an interdisciplinary work that contributes to three fields of research: computer science, medical informatics, and neuroinformatics. The major contributions are as follows:

1) A detailed analysis of the informatics requirements of an Experiment Management System, and the proposal of a new type of application development tool (EMSBE), which has the necessary features to build systems that meet those requirements.

2) A new data model and design methodology for modeling experiment data as hierarchical, context-sensitive views, providing a framework for the rapid development of drill-down navigation systems that meet the requirements of an EMS.

3) A layered software architecture for an Experiment Management System Building Environment.

4) A detailed implementation of an Experiment Management System for managing neuroscience data and workflow.

As a demonstration of the architecture, I describe the implementation of a Web-Interfacing Repository Manager, a Perl-based EMSBE. As a demonstration of the effectiveness of the data model and methodology, I use them to design a working web-based EMS to manage the complex data and workflow of a neuroscience research project.

Here is a walkthrough of the remainder of this dissertation. Chapter 2 describes the seven roles of an effective EMS, and then specifies ten requirements that an EMSBE must have to build systems that meet these roles. Chapter 3 presents a formal data model for modeling complex scientific data (the Repository Schema Model) and a model for defining web interfaces as a query-driven network of context-sensitive views (the Query-By-Context View Model). Chapter 4 describes the components of the layered WIRM architecture, including a detailed description of how they interact to generate web views. Chapter 5 describes a working implementation of an EMSBE, including an in-depth review of the interfaces that are provided. Chapter 6 presents a practical methodology based on the data model for building a custom EMS, illustrated with a systematic walkthrough of designing a simple example domain. Chapter 7 describes a working EMS for the UW Brain Mapper project, discusses my experience building it, and summarizes the benefits of WIRM. Chapter 8 compares WIRM to related commercial and research systems. Finally, Chapter 9 proposes ten directions in which WIRM can be extended and improved. The appendices include a detailed reference manual to all of WIRM's API's, a discussion of how WIRM could be used to build web portals for scientific communities, and a treatise on modeling the symbolic content of multimedia objects.

Chapter 2: Requirements Analysis

There have been a number of efforts to identify the requirements of an EMS, although none have been comprehensive, nor have they discussed the impact of the Web as a delivery technology. Furthermore, there has never been an analysis of the specific requirements of an EMSBE, which is an important contribution of this dissertation. Existing efforts to identify requirements have mostly been driven by a domain-specific EMS, e.g. [Ahr96]. Ahrens focuses mainly on scheduling issues and how they relate to parallel processing. Another analysis can be found in [Ioa96], in which three main requirements for an EMS are identified:

1. *Uniform Interface* – scientists should be provided a consistent user interface.
2. *Transparently manage data* – users shouldn't have to deal with tedious data management issues.
3. *Hide details of underlying software* – the system should buffer the user from the idiosyncrasies of the software.

Although these requirements are important, they can be regarded as facets of a single feature, which is to provide an abstraction layer for the users. From my experience, I have identified many more requirements of an EMS that need to be addressed, especially in the presence of remote applications that must work across machines, locations, and disciplines. I developed the following requirements lists based on my extensive experience building a complex EMS, the UW Human Brain Mapper, and from experience developing a Data Environment for Vision Research (DEVR), which was intended to be a framework for managing computer vision experiments. In addition, I have collaborated with numerous other informatics projects (e.g. [Mar96]), and have explored the issues of data integration [Jak97a] for the nationwide Human Brain Project consortium. From this

hands-on experience, I have distilled a detailed set of requirements that are common for most experiment management issues, which will be presented in the section that follows.

It is important to distinguish between two very different sets of requirements: those of an EMS, and those of an EMSBE. The EMS requirements are driven by the needs of end-users (scientist) and the experiments themselves, whereas the EMSBE requirements are driven by the needs of an EMS Developer (programmer) who must deal with the technical issues of creating *any* EMS, regardless of domain. The EMSBE requirements are directly driven by the requirements of an EMS, but the focus is on implementation rather than features for the end-user. The first subsection specifies the seven roles of an effective EMS, and evaluates the EMS requirements. In the second subsection, identifies ten requirements of an EMSBE and shows how they are driven by the EMS requirements.

2.1 Seven Roles of an Effective EMS

For an EMS to be effective, regardless of the domain, it must offer a wide range of features, which I have organized into the following seven categories: *systems integration, data integration, workflow support, remote collaboration facilities, advanced data type management, intelligent navigation,* and *adaptive user interfaces.* I explore each of these categories in turn.

2.1.1 Systems Integration

As described in the previous section, one of the main challenges of today's scientists is dealing with a wide range of heterogeneous software tools that are used in the acquisition, processing, and management of experiment data. These programs often require tedious, repetitive, and complex handling, and typically present the user with limited, cryptic, and non-standard interfaces. Furthermore, they often require data to be

translated between multiple formats in order to work together. As emphasized by the ZOO group, one of the main roles of an EMS is to insulate the scientist from the complexities and idiosyncrasies of these applications by hiding the details of managing the data flow between them. The EMS should play the role of Systems Integrator, providing the end-user a high-level console for launching and interfacing with heterogeneous external applications. The EMS should serve as a wrapper around the applications, in which the details of each program are handled by the EMS programmer, rather than the EMS user.

For example, consider the process of constructing and viewing a three-dimensional model of a patient's brain in the Brain Mapper experiment. Before the EMS, users had to interact directly with the radiology department's image server to fetch the MRI exam into the local machine. The interface to the image server is a clunky text-based FTP program, activated by a cryptic command-line interface requiring the user to specify the domain name of the radiology department's ftp server and even the port number. Then the user had to navigate through a number of text menus (without the aid of a mouse-driven GUI) to initiate the file transfer. After the transfer, the user must operate an in-house image-processing program to align and crop the MR slices before they can be visualized as a 3D model. The interface for running this program is a non-intuitive process in which the user must edit a Lisp program by hand to indicate the location and size of the image sequence, save the changes, and then run a command-line interface to interpret the Lisp program and execute the processor. The resulting intermediate files must be explicitly stored in a user-specified location, and then fed to another invocation of the visualizer program to generate the 3D model. The user is required to learn three separate interfaces, each with their own quirks and under-documented procedures.

Now consider the same operation as handled by an EMS. To fetch the exam, the user should be able to request that operation from a well-defined menu in the EMS console, select the patient from a list, enter the minimal required information, and then submit the

request. The EMS should then interface with the FTP program and retrieve the files to a program-specified location, perhaps importing them directly into an EMS-controlled file repository. The user doesn't need to learn or deal with the text-based radiology server, nor does he need to know where the files are stored. Then, to visualize the image, the user should be able to select that option from the console and have the EMS automatically generate the necessary Lisp script to perform the image processing, and then launch the appropriate visualization software. In practice, the automation of these stages occurs in a step-wise manner.

2.1.2 Data Integration

In addition to providing a uniform interface to multiple external *programs,* the EMS should provide a uniform interface for querying and browsing experiment *data,* regardless of the source of that data. This can be done in two ways: either providing a central data warehouse which contains copies of data extracted from the various sources, or by mediating queries to external sources on the fly, and returning results in an integrated manner. Often, a combination of both methods is used. Regardless of the underlying mechanism, the end-user should be presented with a unified query interface that allows them to perform ad-hoc queries on the data, without having to worry about where the data came from.

For the UW Brain Mapper, the EMS provides a repository that acts as a multimedia warehouse consisting of an instantiated view over a wide range of data sources:

Table 1: Brain Mapper Information Sources

INFORMATION SOURCE	DATA
Radiology image server	MRI slices
Camera, photo lab, scanner	Intraoperative photo
Excel spreadsheet	Study data
Patient record (hard copy)	Patient demographics
Skandha (graphics application)	3D Models and Renderings
Brain Mapper	Map data

The EMS provides a single instantiated view over the multiple sources, allowing them to be retrieved and organized at the user's request, into customized, unified graphical views over the integrated data, facilitating data mining and dissemination, as shown in Figure 3.

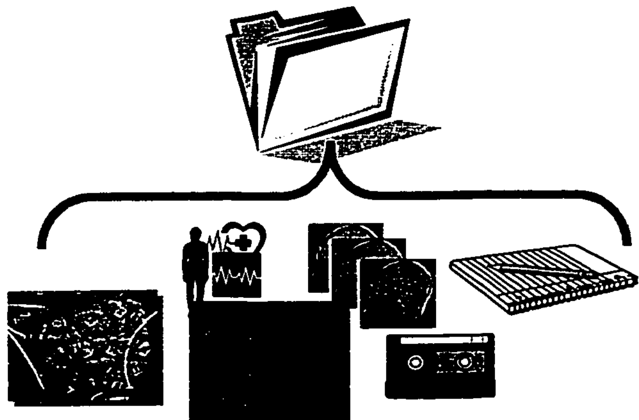


Figure 3: A Uniform Query Interface for Brain Mapping Data

2.1.3 Workflow Support

An important role of the EMS is to assist in the management of the stages of data acquisition and processing [Ail98]. The stages of workflow should be modeled in the EMS, and it should keep track of what's been done on each data object, who did it,

what's left to do, etc. The workflow support should be tightly integrated with the Systems Integration facilities, enabling workflow tasks to be automatically recorded and tied to launching the appropriate programs.

In the Brain Mapper, the workflow console is *patient-centric*, that is, all tasks are viewed as parts of the job of acquiring and processing a Patient. The main workflow console allows users to register new patients, fetch their exams, etc. In addition, from the patient view, the user can determine what stages of workflow have been accomplished. For example, if the Photo has been uploaded, then it's icon appears. This kind of integration of workflow monitoring within data views can be useful, although a master workflow view that displays the status of every patient would be desirable, and is discussed in the future work section.

2.1.4 Remote Collaboration Facilities

An EMS should provide the ability for users to share data and programs across facilities. It should serve as a document delivery system, allowing users to retrieve any kind of file from anywhere. The Web is the natural platform for this, as it is virtually omnipresent in today's laboratories and offices. The EMS should provide high-level facilities for publishing experimental data on the Web, both for sharing by collaborators and external users.

To facilitate multi-user collaboration, the EMS should provide a *user authentication* facility, which allows users to log in (see Figure 4), and audit which users have authored what data.

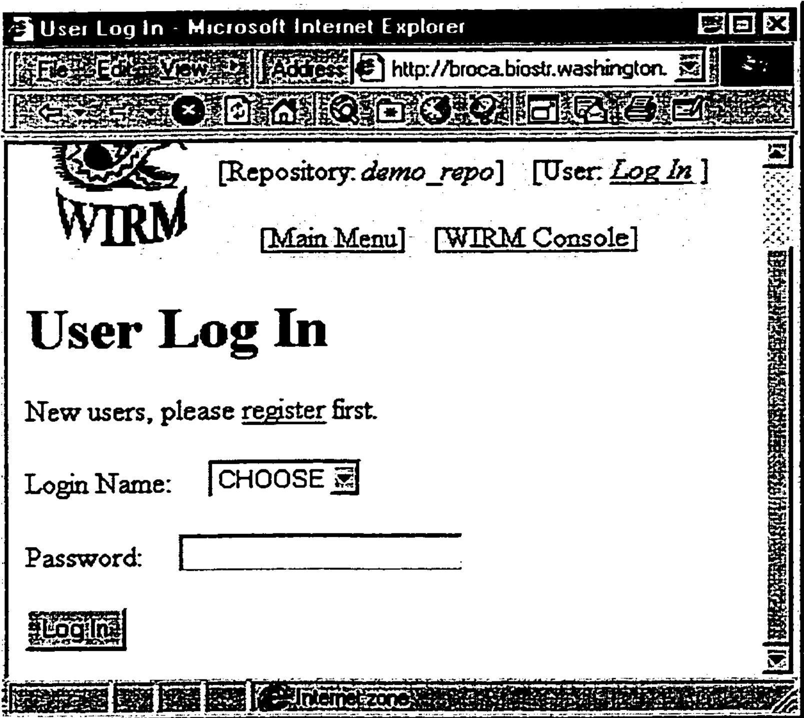


Figure 4: User Authentication

In addition, the EMS should provide facilities for users to post annotations on data objects for other users to read. The Brain Mapper allows users to make annotations on any data object in the system, as depicted in Figure 5.

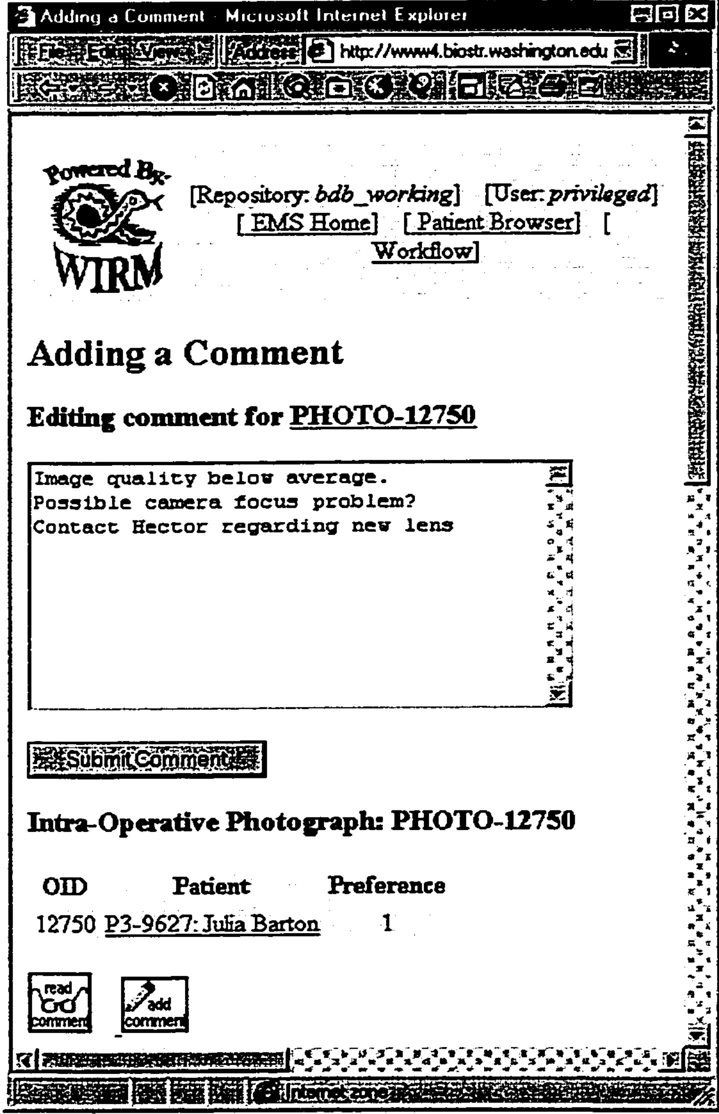


Figure 5: Making an Annotation

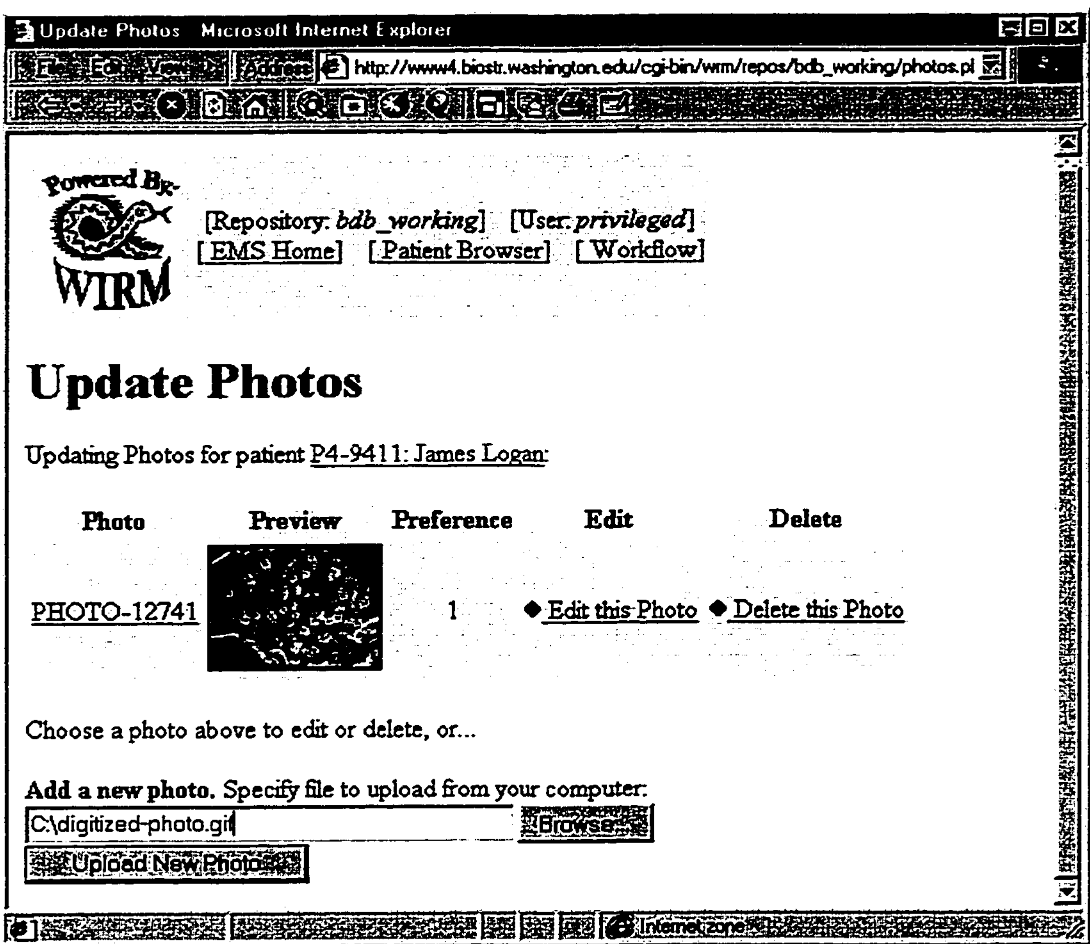


Figure 6: Uploading a Photo

A Web-based system ensures that users can access the EMS from any browser. For example, the technician may upload the digitized file from one lab, and the surgeon's assistant may upload the study file from another lab (shown in Figure 6), and both these items may be accessed from a third location.

2.1.5 Advanced Data Type Management

Unlike enterprise management systems, the scientific EMS must be capable of handling very complex data types. The typical experiment may consist of dozens of different types of objects, of a very heterogeneous nature. For example, brain data may consist of

a wide range of image, sound, and video types, custom-formatted binary data, and ASCII dumps of alphanumeric tables. Standard relational data types are not sufficient. The underlying database for the EMS should be an object-oriented or object-relational system. The data model should allow arbitrarily complex user-defined types, and have built-in support for managing File types and handling image conversion.

To handle multimedia types, the system should be able to visualize them (e.g. display an

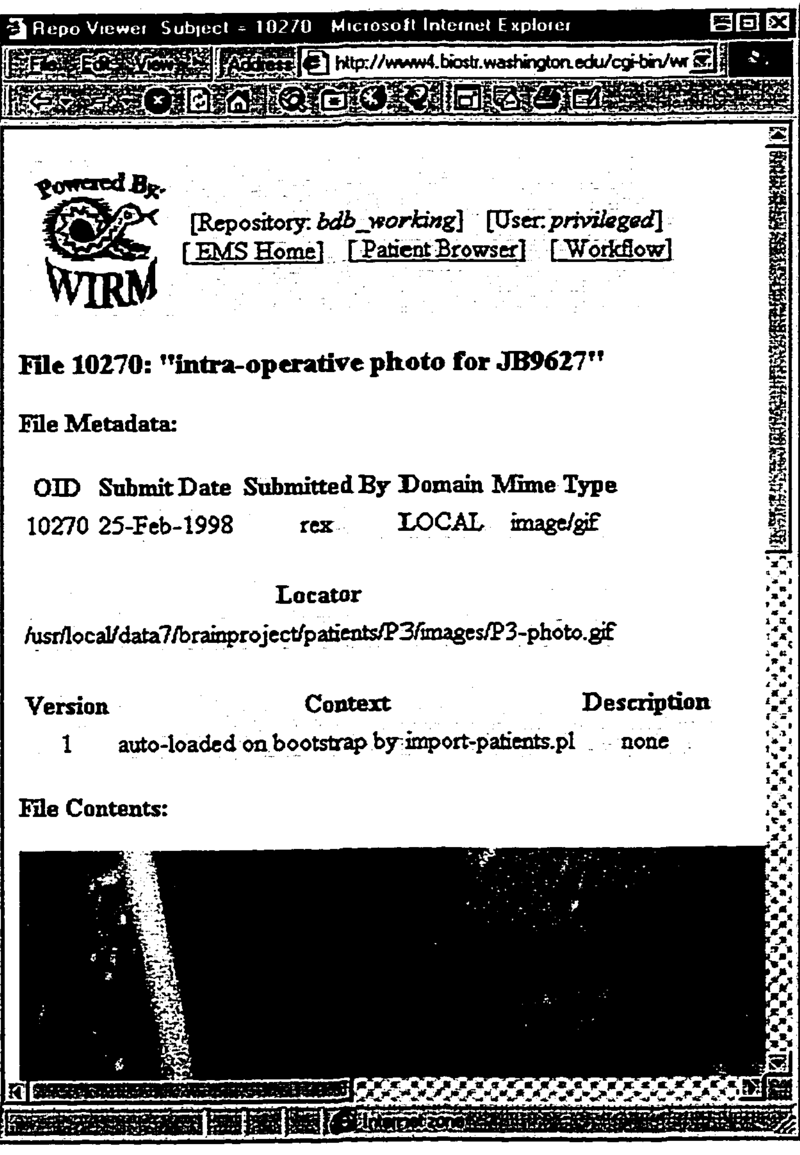


Figure 7: File Metadata

image or play a sound file), and to manage the *metadata* associated with each object. For example, the date, size, type, and descriptive information of an object should be stored. The Brain Mapper EMS manages the metadata for all files registered in the system. Figure 7 shows the metadata view for a registered photograph.
Another important feature is the ability to support an evolving domain model. Users may want to add a new attribute, or rename an existing attribute, or even define a new data type. The system should supply features for these operations. The Brain Mapper EMS (in fact, every WIRM-based EMS) provides a GUI for defining new data types and evolving existing types.

2.1.6 Intelligent Navigation

The EMS should provide a navigational and organizational structure that is consistent and predictable. Much of this falls under the auspices of information architecture design, but the EMS itself should provide basic capabilities, such as the ability to situate related materials together on the same page, either by embedding the information in a single view or creating hyperlinks. The importance of hyperlinking can not be overemphasized: the ability to see an overview of an object and then retrieve detailed information on a part of that object by clicking on it gives users a much greater ability to get the information they need. As much scientific data is hierarchical in nature [Jak95], the EMS should allow users to traverse data hierarchically. For example, in the Brain Mapper, the users may be presented with a list of patient objects, from which they can click on a single patient to see an overview of that patient, from which they can access a detailed view of any part of that patient, such as the patient's MRI exam, which is a hierarchical object in its own right, consisting of multiple series which in turn consist of multiple Slices. This kind of navigation is known as *drill-down*.

Another kind of navigation that is useful is *spatial linking,* where the user can click on an image-map to retrieve data related to regions on the image. For example, the Brain

Figure 8: Spatial Navigation of Language Site Data

Mapper allows a user to retrieve information about language sites by clicking on them in the 3D model, as depicted in Figure 8.

2.1.7 Adaptive User Interfaces

Because today's experiments involve multiple classes of users and the data is intended for multiple classes of audience, the EMS should provide multiple, customized interfaces for each class of user. For example, the Brain Mapper has the following kinds of users:

- Surgeons, interested mostly in patient demographics, photos, and identifying anatomical structures. Refer to patients by a surgery research number.

- Radiologists, interested in MRI exams and visualized 3D images generated from them. Refer to patients by exam number.

- Computer Vision Experts, interested in 3D models and their associated internal model files. Refer to patients by name or initials, or by an internally assigned patient ID.

- Neuroscientists, study all parts of the data, especially the relationship between language sites and anatomical structures.

- Technicians, interested in technical details, such as the format of a file, or the codes used in a study spreadsheet, etc. May refer to patient by name, research number, exam number, or even a database-specific Object ID.

Each of these groups has different priorities when viewing a patient, and thus would like different parts of the record emphasized. Furthermore, each refers to a patient by a different preferred label, so the system should present the preferred identifiers. This imposes a complex requirement on the EMSBE, but I feel it is important enough to include as an essential role of the EMS.

Ideally, the Patient View should be customized for each group, hiding the unnecessary details from some users and emphasizing different parts of the patient record based on anticipated needs. The goal is to adhere to the principle of *economy of attention:* the user can only focus on a limited amount of objects on a single screen without becoming overwhelmed, and the system should try to limit the amount of information to the minimal amount necessary for the majority of users in a given class. Other detailed information is still accessible, but it requires navigation to get there rather than putting it all on the screen at once [Roz98].

A common theme at the 98 SIGMOD was the notion that *less is more* in terms of data management (e.g. [Cha98]). There is a growing research focus on ways to "reduce the knobs" of information management applications. As some users want technical information, while others prefer high-level overviews only, the EMS should support customizable views, rather than relying on a single, compromised view for all users. The system ideally should support multiple interfaces at the granularity of a data object view, rather than requiring the designer to create completely separate sites for each user class. This will be described in detail in the next section.

It is important to acknowledge the tradeoff between the principle of economy of attention, and the *principle of fewer clicks*. The designer should be careful to avoid compartmentalizing information to such a degree that the user is required to click through too many screens to find the information that they want. A balance must be achieved.

Another important issue is *access regulation*. The system should provide access control over data at an arbitrarily fine granularity. There are two main motivating factors behind this requirement: protecting the privacy of subjects, and protecting proprietary data from competitors. Privacy is a central issue in medical informatics, and there is a growing body of research in this area. From the perspective of an EMS, the enforcement of privacy can be considered an aspect of the adaptive user interface. In addition to classifying users by type or interest, the system should classify users according to their privilege level. The Brain Mapper EMS provides three levels of access: *privileged,* who have access to everything in the database, *collaborator,* who have access to everything except patient identification, and *public*, who have access limited to certain parts of a subset of patient records. The distinction between public and collaborator allows us to withhold unpublished data from the general public. The system allows access regulation to be defined at an arbitrarily fine granularity.

To protect proprietary and unpublished research data, we have plan (in future work) to support three categories of patient data: *demo, published,* and *unpublished.* Every aspect of the demo patients can be viewed by all users. The published patients allow only a subset of data to be viewed by the public (i.e. that data which has appeared in a journal). The unpublished patients are completely hidden from the public. Table 2 summarizes the access regulation guidelines.

Table 2: Brain Mapper Access Regulation

	PUBLIC	COLLABORATOR	PRIVILEGED
DEMO	All, use pseudonyms	All but names	All data
PUBLISHED	Selected data	All but names	All data
UNPUBLISHED	Nothing	All but names, possible restrictions on site data.	All data

2.2 Ten Requirements of an EMSBE

As demonstrated, the EMS and the EMSBE are distinct concepts. Ideally, one could blur the line between the EMSBE and the EMS if the EMS design tools were accessible to non-programmers. If the EMSBE tools took the form of visual languages, templates, and wizards, the EMS end-user could in theory play the role of EMS designer. But the complexities and variability of experiments precludes this possibility, as high-level non-programmer tools are by nature restricted in their operations, and could not be made general enough to handle arbitrary EMS domains. In practice, scientific projects tend to hire programmers to build interfaces from the bottom up, creating custom software to tie their tools together. The task of creating an EMS from scratch is a very large undertaking, requiring many hours of expensive development time. The presence of a good EMSBE can greatly reduce the time and effort in building and maintaining an EMS. Here I have identified the requirements of an EMSBE that can be leveraged by a

programmer to create an EMS with a limited amount of effort. I relate each requirement to the EMS roles that it supports.

2.2.1 General purpose programming environment

For an EMSBE to truly meet the needs of all possible systems, it must be an extension of a general purpose, procedural programming environment. Declarative templates and visual tools are limited in their scope and power. The programming language should preferably include functions or libraries for handling strings and processes. WIRM is based in Perl, which is well suited for manipulating programs, performing translations, and generating interfaces. By providing interfaces that allows data structures to be transparently saved to the database, I have essentially created a Persistent Perl programming language. Other possible candidates are C++ or Java.

This requirement pertains to all seven of the EMS Roles, especially systems integration and advanced data type support.

2.2.2 Data model that supports complex objects & relationships

The EMSBE should be based on a data model that can support the complexity inherent in scientific data types and the relationships between them. The model should be object-oriented, providing encapsulation, inheritance, etc., but also provide support for database queries. I have defined a new database model, the Repository Schema Model, which is based on the object-relational data model and includes explicit constructs for representing users, workflow, and Web views. It employs a hierarchical structure, which translates naturally into creating drill-down views. Chapter III explores the Repository Schema Model in detail.

2.2.3 Built-in support for managing file metadata

Document management is a central requirement of any EMS, so there should be explicit support for it built into the EMSBE. The EMSBE data model explicitly represents the concept of file objects and all common metadata associated with them. It should provide a repository for storing files uploaded into the system, as well as manage the metadata of external files registered with the system. It should provide query facilities for retrieving documents, and the ability to create compound documents based on multiple files. File management should be built into all layers of the programmer's interface. This requirement is necessary to support the advanced data types role, as well as the remote collaboration role of the EMS. The WIRM EMSBE provides support for three classes of files: files copied into the protected file repository, external files accessible from the server's file system, and remote files on internet-accessible machines.

2.2.4 Tools for storing, visualizing, & manipulating arbitrary multimedia types

The EMSBE should provide support for storing, visualizing, and manipulating arbitrary multimedia types. All the common image formats should be supported, as well as functions for converting between them. WIRM uses third-party modules for handling common data formats. For example, image management facilities are provided by the Image::Magick module [Mag-URL].

Because the range of types can't be predicted in advance, the EMSBE facilities for handling multimedia should be easily extensible. WIRM adopts the notion of a Media Module, similar to the *datablade* concept of Illustra [Sto96], which allows the programmer to install new type-specific functions into the system, utilizing a mime-type identifier system. For example, I have developed a special media module for handling the data type GE/MR image. Figure 9 shows an MR view, which used the MR Media Module to translate the MR image into a JPEG suitable for viewing through a Web browser.

The multimedia management tools support the advanced data types requirements and the remote collaboration facilities of the EMS.

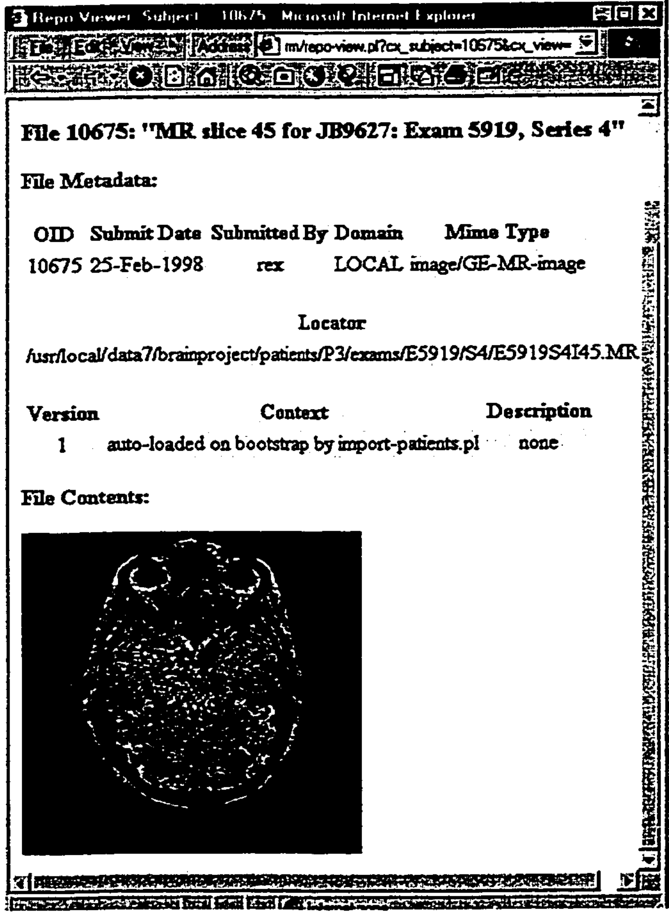


Figure 9: MR Image Converted for Web Viewing

2.2.5 Dynamic Schema Evolution

As expressed in the previous section, dynamic schema evolution is highly desirable, as it allows a much more flexible approach to managing data types. The EMSBE should facilitate changing schema on the fly, without having to go through complex recompilation procedures or having to worry about updating existing objects to include the changed attributes. WIRM provides such a facility (shown in Figure 10), which has proven to be indispensable in supporting the dynamic nature of experiments.

2.2.6 Database Interfacing

The EMSBE should have built-in controls for interfacing with relational databases. A declarative query language should be provided (e.g. SQL), as well as features for iterating over query results and marshalling them into local data structures, and functions for committing changes back to the database. By integrating these features into the programming language, the user is provided with a *persistent programming language*,

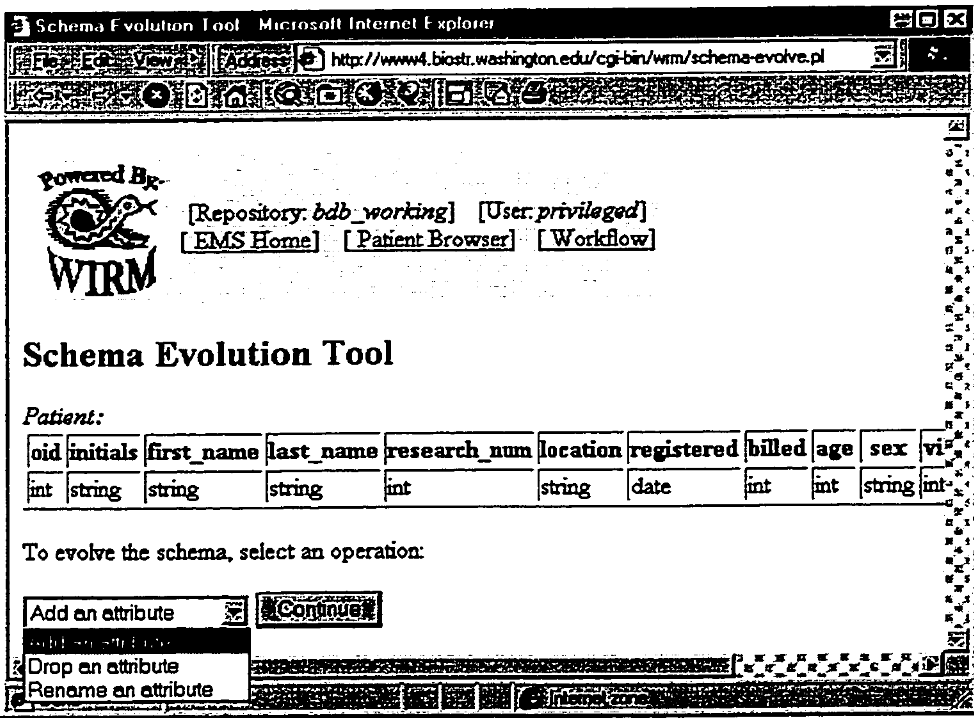


Figure 10: Schema Evolution

similar to that exported by an object-oriented database such as Objectstore [Ore92].
A generic GUI for issuing queries over the database should be part of the EMSBE and included with every EMS. Figure 11 shows WIRM's Repository Explorer, which allows the user to issue Boolean queries over any data type.

This requirement is especially important for the data integration role of the EMS.

2.2.7 Tools for rapidly creating Web-based GUIs

Creating user interfaces has proven to be the single most resource intensive task in building an EMS. Thus, the EMSBE should provide a significant suite of tools for generating user interfaces, preferably of the "rapid prototyping" flavor. Functions for visualizing data objects as Web pages should be powerful and flexible. Functions for creating input elements (e.g. HTML form elements) and parsing them to interact with user input are also required.

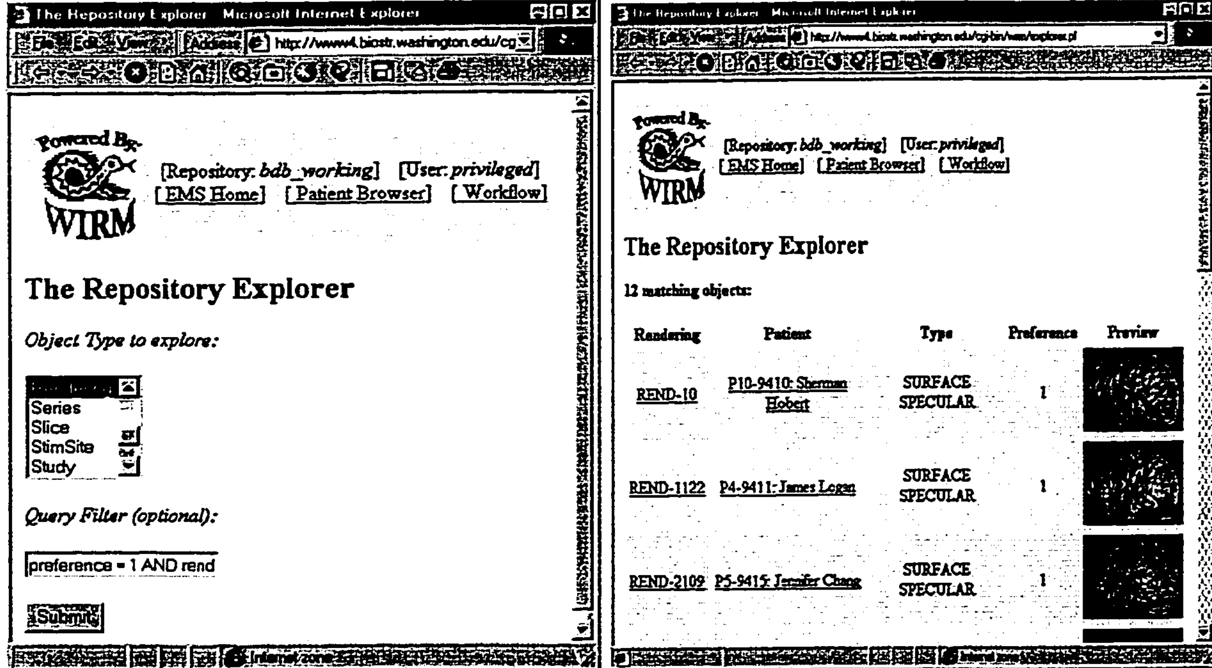


Figure 11: Repository Explorer

As I have explained, the Web is the most important platform for interfacing with users, so the EMSBE should have HTML-specific generators. WIRM provides high level template-based functions for creating tables, lists, hyperlinks, and other HTML objects, as well as all form elements (e.g. popup menus, etc.).

This requirement is essential for providing remote collaboration facilities, as well as meeting the intelligent navigation facilities described above.

2.2.8 External application interfacing and workflow processing

To support systems integration, the EMSBE should have facilities for interfacing with external applications. As experiments often require the tying together of a wide range of unstable, custom research applications, the EMSBE should be able to easily parse and translate streams of data. The vast majority of programs export their data in ASCII format, so the EMSBE should be adept at text manipulation. This makes Perl an ideal platform for the EMSBE, as it supports a very powerful regular expression parser. Furthermore, the EMSBE should be capable of launching and controlling external processes, another feature in which Perl excels.

2.2.9 High-level tools for defining workflow processes

To support the Workflow role of an EMS, an EMSBE should provide high level tools for defining workflow processes. A prototype version of WIRM includes a built-in type WorkStep, which can be used to identify a piece of workflow action. I am developing ways to specify dependencies between WorkSteps. In addition, WIRM allows the user to encapsulate bits of work logic as Wirmlets, which can be executed independently or linked together in a workflow console.

2.2.10 Support for creating adaptive context-sensitive interfaces

This requirement involves modeling the user explicitly as a built-in class, providing facilities for users to log in, and keeping track of query sessions, so the system can support context-sensitive adaptive interfaces. These features will be explored in detail in Chapter V.

Chapter 3: Data Model

As described in the Requirements chapter, the data model should be sufficient to capture the complex *structure* inherent in scientific domain data, i.e., the compositional hierarchy of domain concepts. The *structural informatics* approach [Bri93] emphasizes the importance of explicitly encoding the structure of real world objects as explicit knowledge within the schemas of the data model. The structural composition of the knowledge is necessary for organizing the data in the following ways:

- Providing reference and educational resources that impart a better understanding of domain knowledge (e.g. the UWDA [Bri97, DA-URL]).
- Integrating information from multiple sources.
- Organizing and indexing domain content for online retrieval (e.g. patient records databases, etc).
- Visualizing multimedia information.

The WIRM data model embraces the structural informatics approach by enabling the encoding of structure at two distinct levels: at the *schema level*, in that schemas may contain composite *parts* which are references to other schemas, and at the *method level*, in that class methods may be composed of sub-methods (e.g. a row view may contain a label view). The schema level structural requirements are met by the Repository Schema Model (RSM), an object-relational approach to modeling the attributes of a domain. The method level requirements are met by the *Query-By-Context* model (QBCx) [Jak98], a query model for capturing the views and operations of a domain. In the next section, each model will be explored in turn.

3.1 Repository Schema Model

Every data object in a WIRM system inherits its structure from a schema defined in the Repository Schema Model. The RSM includes three general classes of data types: atomic types, composite types, and aggregate types. *Atomic types* are the typical relational data types: strings, integers, real numbers, dates, etc.

Composite Types are complex types, which can be composed of multiple atomic types or other composite types. Every composite type must include an *Object Identifier (OID)*, an integer which allows instances of those types to be uniquely distinguished from any other object in the system. This allows the property of *object identity*, which is necessary for any object-oriented system.

Composite types come in two categories: *system defined* and *domain specific*. System defined types are domain-independent types built into the data model, and include the important concepts of *File* and *User* (other system-defined types exist, such as *Annotation)*. The underlying representation for all composite types is a relational table. The File type consists of a binary file object, a unique file ID, and metadata about that file, such as its external name, format, author, date of creation, version number, etc. The file can be of any type, such as an image, an audio or video clip, an HTML document, configuration file, script, or even a binary executable. As part of future work, we may consider implementing a revision control system within WIRM for managing source code.

The domain-specific types are defined by the repository developer for a given EMS, and consist of an ordered list of attributes. For example, the Brain Mapper might define a *Patient* type consisting of three attributes: *name* (a string), *photo* (a file), and *exam* (a reference to another composite object).

An *Aggregate Type* is a collection of references to Repository Objects. Two types of aggregate objects are Lists (ordered) and Sets (unordered). This class of object is useful for representing aggregate relationships, but it can be implemented using non-aggregate types in a standard relational manner. Therefore, I have chosen to limit the current implementation of the RSM to not include explicit support for Aggregate Types. Future implementations will allow it, and the Repository API will provide methods for creating, manipulating, and iterating over Lists and Sets.

The question remains: why does WIRM use the object-relational model, rather than a pure object-oriented one? It is clear that a pure relational approach would have been insufficient to model the complex types of scientific experiments, but a pure object-oriented approach would have sufficed [Jak93]. One reason is that the object-relational model combines the advantages of SQL query support with the benefits of object oriented modeling (encapsulation, inheritance, and reference-oriented navigation) [Sto96]. In addition to the natural speed of optimized relational query engines, the notion of being able to treat your data as tables has the added benefit of portability: object-relational data is easily transferable between systems. A somewhat rigid model encourages uniformity across applications, whereas the less structured object-oriented model may allow more natural modeling but leads to more diversity across applications, so it is harder to achieve interoperability. The object-relational model promotes clean separation between code and data, and imposes a simple table structure to the underlying data. By limiting the flexibility of the data model, one can achieve the same benefits that arise from "reducing the knobs" of an interface: simplicity affords control. Data modeling without the rigid structure of the relational model gives rise to difficulties coordinating data transfer between applications. As the poet Robert Frost once put it, "poetry without rhyme is like playing tennis without a net".

This principle applies not just to the data model, but to the entire WIRM methodology. The Wirmlet Approach, defined in the next chapter, breaks down the problem of designing an information system into logical, coherent, manageable steps, guided by the imposed structure.

3.2 Query-By-Context View Model

An important contribution of the Query-By-Context data model is addressing the issues of user-interface development directly in the data model. The key idea is that user interaction with the server should be integrated at the data model level, not added to the design as an afterthought. User interface issues are becoming increasingly relevant to database research, as Stonebraker makes clear in Readings in Database Systems [Sto94]: "the needs of user-interface programs drive requirements for function in database systems, and the 'value-added' that a database system provides will increasingly be in the front-end tools that go with it, because a high-performance SQL engine will increasingly be a commodity product."

An inherent assumption about the Query-By-Context model is the inclusion of the user and the GUI as explicit parts of the model. Specifically, I consider Web sites to be views over databases & document repositories. These views are ultimately the results of executing context-sensitive relational queries over a database. The elements that make up a Web View, which I call *Document Elements*, are generated by methods on objects in a hierarchy of classes.

Including the user in the model allows GUI issues to be addressed at every stage of the information modeling process. According [Goy96], "programming nontrivial GUI applications currently is an arduous task requiring significant time and effort to build and even more so to maintain and extend... a major redesign of the system is required to make a change to the GUI."

QBCx simplifies the process of creating context-sensitive views over multimedia data by providing a framework for designing those views. An important aspect of this process is the notion of *context-based queries*. In QBCx, a user's interactions with the system are represented as a Query Session. This involves logging in, accessing a Wirmlet, and performing a sequence of submission-response actions. Every query session carries with it a *user context* (also called a *perspective*) which is assigned when a query session commences, and is carried throughout the query. Some examples of perspective contexts are the privacy-level and type of user. The composition and content of the Web View are partly determined by the perspective.

In addition to the viewer, Web Views are determined by the *subject context* (the identity of the object being viewed) and the *view class context* (the aspect of the subject which is of interest). These three contexts (user, subject, and view class) represent three dimensions in which all Web Views are defined. Figure 12 demonstrates the navigation space in which Web Views exist. Each point in the space represents a single Web View. The vertical axis represents the view class, and the horizontal axis represents the subject

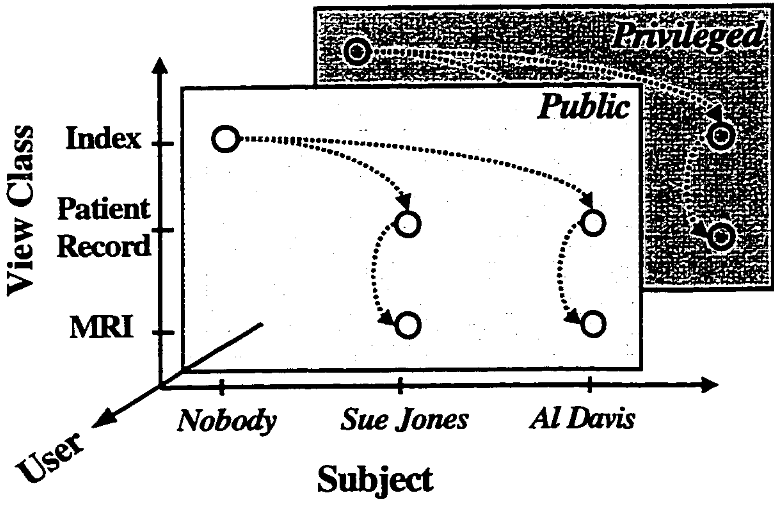


Figure 12: Multiple Context Dimensions

being viewed. As a user navigates the Web site, he moves in a plane created by these two axes. For example, a public user is restricted to the green plane, in which he or she may move vertically to view different parts of a single patient's record, or horizontally to view multiple patients. Traditional template-based database-driven Web sites are restricted to a single plane. However, WIRM provides the third dimension, enabling views that adapt for a specified user class.

In the following paragraphs, I present a formal definition of the Query-By-Context view model. The following notation will be used:

Table 3: Query-By-Context Notation

<table>
<tr><td>&x</td><td>reference to an entity (as opposed to the entity itself)</td></tr>
<tr><td>x → (y, z)</td><td>x is composed of y and z (like BNF)</td></tr>
<tr><td>{x}</td><td>set of one or more instances of x</td></tr>
<tr><td>[x | y]</td><td>x OR y</td></tr>
<tr><td>a X b</td><td>cross-product of a and b</td></tr>
<tr><td>x = F(y)</td><td>x is the result of function F applied to y</td></tr>
<tr><td>r = x.M(y)</td><td>r is the result of executing method M on object x.</td></tr>
<tr><td>italicized</td><td>entity resolves as a simple string</td></tr>
</table>

In the Query-By-Context view model, the *User* (client) and the *Repository* (server) are included as explicit parts of the model. The user navigates through a *Virtual Navigation Space* (VNS) of document nodes which are generated by interacting with the server through the *Wirmlet Web*, which consists of a collection of discrete procedural scripts called *Wirmlets*. The nodes of the VNS are formatted documents called Web Views, which are the basic units of interaction between the User and the Repository. Each Web View is a user-readable document generated by the Repository Web upon receiving a *Client Request* by the user. From an implementation perspective, Web Views are HTML

pages, Wirmlets are special CGI scripts, and Client Requests are standard HTTP requests of the WIRM server, typically issued via clicking on a hyperlink or a SUBMIT button from the client's browser.

Unlike static Web sites, the pages served by a Repository Web are generated dynamically at *click-time*, by invoking Wirmlets on the fly. The Virtual Navigation Space is the infinite network of all possible pages generated by all Wirmlets in the system, defined as:

VNS = Wirmlet-Web X Form-States X Repository-States X Context-States

Wirmlet-Web is the set of all Wirmlet scripts that make up the Repository Web, both system-defined and user-defined. For example, in the Brain Mapper EMS, the Wirmlet-Web consists of the Patient Browser, Data Analysis Console, Patient Workflow scripts, and the standard Wirmlets provided by the Generic WIRM Console. *Form-States* is the set of all possible user-supplied inputs to every Wirmlet. *Repository-States* is the infinite set of all possible states of the repository, including the class definitions and the contents of the databases and file system. Finally, the *Context-States* are the set of all possible Contexts, which identifies the User and the Subject of any given query session. For a given moment in time, the values of the three States collectively comprise the *Session-State* of the repository system.

It is important to note that the VNS is composed of the set of all possible Web Views, in which each WV is uniquely defined by four elements: the Wirmlet being activated, the current state of the repository, the current state of the form, and the current Context-State. Therefore, a given Web View can be uniquely specified by the following tuple:

WV = (Wirmlet, Form-State, Repository-State, Context-State)

For example, consider the Web View generated when a user clicks on a language site in a patient's study, as previously shown in Figure 8 in Chapter 2. The Wirmlet is the Repository Object Viewer. The Form-State includes variables Click.X and Click.Y, which contain the pixel coordinates of the cursor at click-time. The Repository-State includes the current Class Definition file (which defines the procedural method for viewing a Study), the contents of the database (Patient table, Study table, Stimulation Site table, etc.), and the contents of the FSA (including the particular rendering file being displayed). The Context-State includes the ID of the current user, and the ID of the current Subject (an instance of the Study class).

From a compositional perspective, all Web Views are instantiated as a sequence of *Document Elements*, which are physically made up of text, images, hyperlinks, form elements, and presentation markup. The Document Elements are emitted when the user executes a Wirmlet. A Wirmlet is defined to be a script with a unique Name, and a user-accessible location (specified by a URL), which encapsulates a piece of procedural logic, performing a discrete action that meets a workflow or data access requirement. Specifically:

Wirmlet → (Wirmlet-Name, URL, WorkLogic)

WorkLogic is a function that takes as input the Session-State and emits a Web View as output. For example, the Repository Object Viewer for the Brain Mapper EMS has the URL http://www4.biostr.washington.edu/cgi-bin/wrm/repo-view.pl. The Object Viewer's WorkLogic function takes an OID as a parameter, retrieves that object from the database, and invokes the View Page method on it. In the example above, the Study object's View Page method includes a spatial-processing algorithm that determines which language site is closest to the user's click. The WorkLogic function has access to the Session-State. In the example, this allows the View Page method to look up the coordinates of Stimulation Sites in the Repository-State to determine which is closest to

the coordinates specified in the Form-State, and to display the resulting Site using the customized view preferred by the current user specified in the Context-State.

The Web View can be functionally described by the following signature:

WV = WorkLogic(Repository-State, Context-State, Form-State)

In addition to emitting the Web View, the WorkLogic may also arbitrarily alter the current Session-State.

The key advantage of this view-centric model is that it allows the designer to describe the system in terms of context-sensitive web views, which provides a logical framework for creating adaptive web sites. More details on the possible forms and functions of Wirmlets will be given later in this chapter.

Now I will expand on the definitions of the three state concepts that make up the Session-State. To begin, consider the *Context-State*. This is a set of globally accessible variables for carrying the context of a Web View. The context variables are used by the Wirmlets and Object methods to modify the queries performed on the repository, and to filter the data that is to be included in the Document Elements.

Currently, the model supports two kinds of context: the identity of the current User, and the identity of the current Subject:

Context-State → (&User, &Subject)

Note the ampersand syntax, which indicates a *reference* to the specified entity, rather than the entity itself.

Later, I will demonstrate the use of Context-State. For now, it is sufficient to simply know that this state exists and is available to the Wirmlets' WorkLogic functions as well as the class methods of the Repository Object definitions.

The second kind of state is the *Form-State*, which consists of an arbitrary list of *Form-Parameters:*

Form-State → ({Form-Parameter})

The Form-Parameters are text variables tied to special Document Elements in a Web View, generated by the Wirmlets. Each Form-Parameter has a Name and a Value, which can be accessed by the Wirmlet that generated it:

Form-Parameter → (Parameter-Name, Parameter-Type, Parameter-Value)

The Parameter-Type refers to the specific kind of Document Element tied to the Form-Parameter, and can be a *choice-element* (a drop-down menu, scrollable list, group of radio buttons, etc.), or a *free-element* (text field or a scrollable text area). The Parameter-Name is an arbitrary string assigned to the Document Element by the first invocation of the Wirmlet code to be used as a handle for retrieving the contents on the second invocation of the Wirmlet code. For example, the Form-Parameter in the Study View described above has Parameter-Name "Click", Parameter-Type "Image-Button", and Parameter-Value containing the X and Y coordinates of the user's cursor.

The Form-Parameter is rendered as graphical interface controls that the user can manipulate with the client browser. When the user submits the form (thereby invoking the Wirmlet for a second time), the user's input is passed to the Wirmlet code as the Parameter-Value, which can affect the actions of the Wirmlet and thus affect the

subsequent Web View. I will show more examples of Form-Parameters when I discuss the Wirmlet definition process later in this chapter.

The third kind of state is the *Repository-State*. This is by far the most complex and significant part of the Session-State. The Repository-State consists of three parts: the repository's Class Definitions, the combined extent of all Repository Object Instances in the database, and the set of all File objects that have been registered with the repository:

Repository-State → ({Class-Def}, {Repo-Instance}, {File-Data})

I will now expand on the definitions of these three components in turn. The *Class-Defs* make up the object-relational Types known by the repository, both built-in and user-defined. The user-defined Class-Defs encapsulate the domain knowledge that represents the real world concepts managed by the EMS. As the repository class definitions are dynamic, the set of Class-Def elements is subject to constant change, which is why I include it as part of the *Session-State*. Each Class-Def has the following form:

Class-Def → (Class-Name, Schema-Def, Method-Defs)

For example, consider the Class-Def for a patient. The Class-Name is "Patient"; the Schema-Def is a description of the patient's attributes (such as age, sex, etc.). The Method-Defs is a set of functions that can operate on a Patient, such as how a Patient is displayed, how to create one, how to edit one, etc.

The repository developer assigns the Class-Name when a new class is defined. The *Schema-Def* defines the attributes of the class, which will be represented as an object-relational type definition, defining the structure of the underlying table in the repository database. The Schema-Def consists of a list of Attribute Definitions, with the following form:

Schema-Def → {Attribute-Def}

Attribute-Def → (Att-Name, Att-Type)

Att-Type = [char(len) | int | real | date | &Class]

That is, each Attribute-Def has a name and a type, which is either a basic relational type (character string, number, or date), or a reference to a complex type. Note that the complex type can be either a built-in class or a user-defined class. The built-in classes are User, File, and Annotation. These will be explained in detail later in this chapter.

In addition to the Schema-Def, a Class-Def may have a set of procedural Methods defined for it, as follows:

Method-Defs → ({View-Fn}, Make-Fn, Edit-Fn, Delete-Fn, {Misc-Fn})

The *Methods-Def* is a set of functions belonging to the class, which operate on data instances of that class. As specified above, classes may have five types of methods: *View-Fn,* which emits a sequence of Document Elements to be included in a Web View, *Make-Fn*, which allows users to create new instances using the Form-State, *Edit-Fn,* which allows users to update existing instances, *Delete-Fn,* which removes an instance from the repository (including necessary parts), and *Misc-Fn*, which can be any general purpose function. Default View-Fn, Make-Fn, Edit-Fn and Delete-Fn methods are all inherited if the methods are not explicitly defined for a class.

I will now discuss the View-Fn methods, which can take multiple forms:

View-Fn → [View-Page-Fn | View-Table-Fn | View-Label-Fn |View-Other-Fn]

View-Table-Fn → (View-Table-Header-Fn, View-Table-Row-Fn)

The View-Fn includes three types of methods that declare how a subject should appear at three abstraction levels: *View-Label-Fn* (used whenever an object is named), *View-Table-Fn* (used when viewing an object within a collection), and *View-Page-Fn* (used when an

object becomes the focus object in the VNS). As each of these elements is a procedural function, it is useful to look at the signatures of these functions:

Link = Instance.View-Label-Fn(Session-State)
[V1, V2, ...] = Instance.View-Table-Row-Fn(Session-State)
DE = Instance.View-Page-Fn(Session-State)

where [V1, V2, ...] are attribute values (raw or computed), and DE is a Document Element, as defined above.

View-Label-Fn returns a hyperlink which leads to the object's Full-Page view, using a descriptive text label (e.g. a patient's name). View-Table-Fn is composed of two methods: *View-Table-Header-Fn*, which returns a list of header strings, and *View-Table-Row-Fn,* which returns a list of values (to be used in a Table Document Element). These values are transformations between the attribute values of the instance and Document Elements (e.g. HTML renderings of the values). The third method, View-Page-Fn, returns a full-page Document Element. Note that the View methods can be constructed hierarchically. For example, the View-Table-Row-Fn of an object often invokes the View-Label-Fn for the first item in the row.

As will be demonstrated in the Methodology chapter, the Repository Object Model provides a useful abstraction for designing hierarchical navigation systems.

Chapter 4: WIRM Architecture

4.1 Architectural Overview

The components of this architecture, as shown in Figure 13 below, are:

- *Client Browser*, any internet browser (Netscape, Internet Explorer, etc.) used by the user to send HTTP requests over the internet.
- *Web Server*, a standard Web server that handles requests and activates CGI scripts.
- *Wirmlets*, CGI scripts, either system-supplied or user-defined.
- *Web View*, an HTML document generated by the Wirmlet which is passed back to the user's browser.
- *WIRM Server*, which handles requests from the Wirmlet.
- *Class Definitions*, which specify how repository objects should be viewed and other methods.
- *Database Server*, which handles SQL queries and access the database.
- *Database*, any standard relational database (collection of tables residing on disk).
- *File Data*, either protected by the repository (FSA) or any files on the server machine.
- *Visualization Cache*, a Web-viewable area for staging files.

The user submits a request through the client browser, which is transmitted across the Internet to the Web server that serves the EMS. The client's request contains the following information:

- the URL of the target Wirmlet
- the Context-State, which includes the User ID and possibly the Focus Subject.

- Any Form-State, either data being carried over from a previous invocation of the Wirmlet, or the values of form parameters that have been entered by the user into the client browser.

The Web server invokes the target Wirmlet, passing it the Context-State and Form-State. The Wirmlet is a CGI script that accesses the WIRM API's. Conceptually, the WIRM API is considered a server that handles requests from the Wirmlet. However, in the actual implementation, the server may be a library of functions that are compiled as part of the CGI script. The difference is not critical, so I will refer to the WIRM API's as a separate Server.

The Wirmlet makes requests to the WIRM server, based on the actions specified in the Wirmlet's WorkLogic function. The requests may involve queries to the database, which are translated into SQL and passed to the Database Server. The results are returned and wrapped in HTML, so they may be inserted into the Web View document. Other requests may retrieve Files from the File Data area, which may cause them to be converted into Web-viewable objects in the visualization cache, which are then included in the Web View. Other requests may access methods of the Class Definitions, which

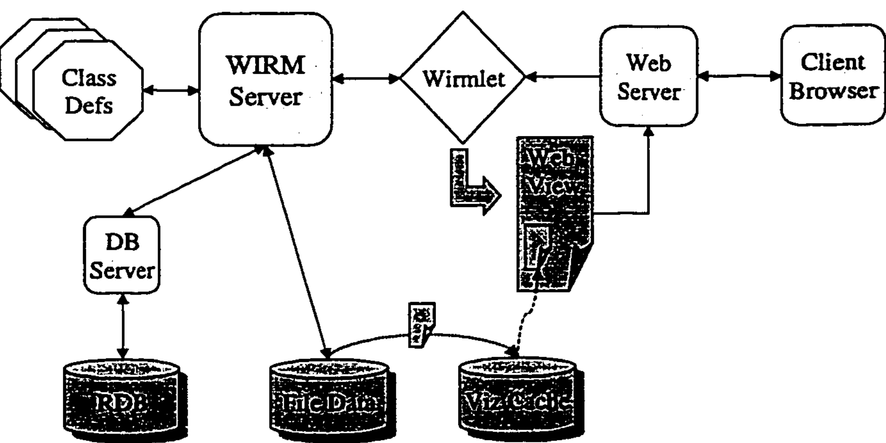


Figure 13: WIRM Architecture

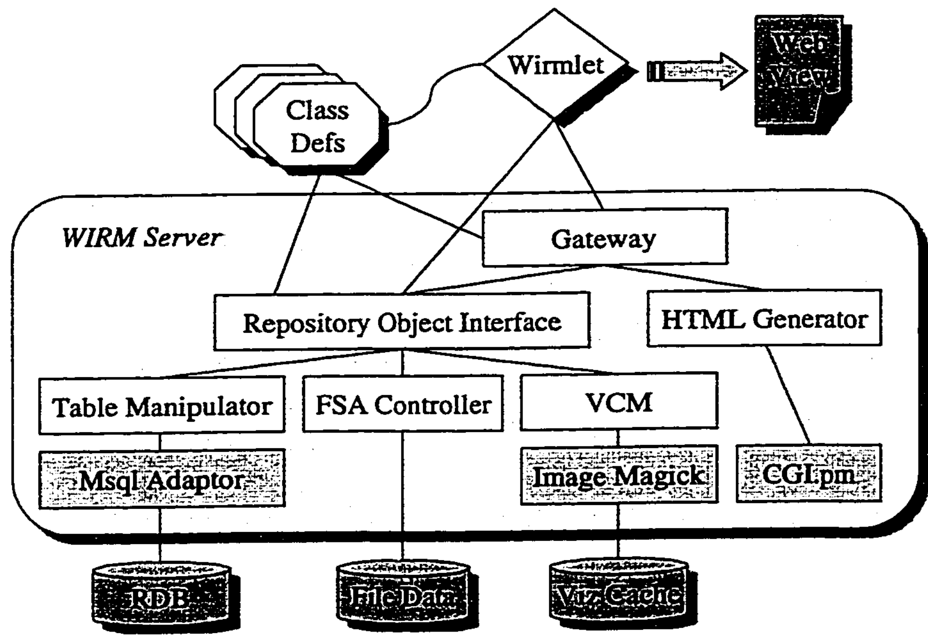


Figure 14: WIRM Server Components

generate pieces of the Web View document. Finally, the completed Web View is returned to the Web server, which in turn passes it back to the waiting Client Browser.

4.2 Server Components

The WIRM Server is built from a layered architecture of components, each of which handles a specific function. There are six component libraries divided into two categories: the *Developer API's,* which includes the Gateway, Repository Object Interface, and HTML Generator, and the *Internal Interfaces,* which include the Table Manipulator, the FSA Controller, and the Visualizer. The Developer API's are used by the EMS designer to define Wirmlets and the methods for the domain-specific Class Definitions, whereas the Internal Interfaces are used by the WIRM developers to encapsulate access to the various repository resources. The WIRM Server components are shown in Figure 14.

Each component will now be described in turn.

4.2.1 FSA Controller

The FSA Controller regulates access to the File Storage Area (FSA), which is an internal repository of files managed by WIRM. Wirmlet developers will not normally access the FSA Controller directly. Instead, they should use the file handling functions in the repository object API or the higher-level file visualization methods in the Gateway API.

There are three classes of file handled by WIRM:

- FSA Files, which are physically managed by WIRM in the FSA
- Local Files, which exist on a file system directly accessible by WIRM's server but not in the FSA.
- Remote Files, which have been registered with WIRM but are maintained on a remote machine.

Using the FSA Controller, files may be copied into the storage area from a file handle, and file locations may be looked up based on file id. Other than file location, the interface does not maintain any file metadata. That task is managed by the Repository Object Interface (see below), which builds an additional level of abstraction on top of the FSA Controller, allowing the Wirmlet developer to treat files as objects complete with metadata rather than just blobs in the file system.

The FSA has two partitions: the Custom area and the Default area. In the Default area, files are assigned a location based on their file id (hashed across multiple directories to avoid the inefficiency of having thousands of files in a single directory). This frees the programmer from having to identify a unique name for every file registered in the system. In the Custom area, files are organized in human-readable paths defined by the programmer who submits the file. This allows the user to keep some metadata redundant in the file path, which allows files to be recognized without the accompanying metadata in the database.

4.2.2 Visualization Cache Manager

As described in the previous section, Files managed by WIRM can reside in the File Storage Area or in other locations accessible by the WIRM server. These locations are not necessarily accessible from the Web, however. The Visualization Cache Manager (VCM) provides Web access to multimedia files managed by WIRM by maintaining a Web-accessible cache (called the ''Viz Cache'') on the server machine. When a user requests a file through a WIRM-created Web page, the Repository Object API and the Gateway Interface use the VCM to convert the file into a Web-viewable type and then copy the file into the Viz Cache. If this process has already occurred for the desired file, the existing copy is used. The Wirmlet developer should use the Gateway and Repository Object interfaces, and not the Visualization Cache Manager directly.

The conversion process is automatic: when a file is requested for Web viewing, the VCM determines a file's type and then converts it to a Web-favorable format using a built-in conversion routine or a user-supplied module. The built-in routines should support all common image formats, which are converted to JPEG's. The ability to supply user-defined routines is analogous to the use of datablade modules by Illustra [Sto96] for extending their type system. For example, the Brain Mapper EMS extends the VCM to include a module for converting MR images to JPEG's.

If a file is of an unrecognized type or is already in a format acceptable by the browser, the VCM copies it from its source location into the Viz Cache and provides a hyperlink to the unchanged copy, which the user may retrieve through their browser.

4.2.3 Table Manipulator

The Table Manipulator Interface provides functions for creating and deleting tables, inserting and removing records, accessing rows of object attributes by ID, formulating

SQL queries, and retrieving the results of a query into a tabular data structure accessible by the Wirmlet manipulation language. Like the FSA Controller and the VCM, the Table Manipulator is an internal API, not intended to be used by the Wirmlet developer, whose functionality is exported through the developer API's.

The Table Manipulator connects to the relational database server and exports a table-level interface to repository data. It handles SQL-like queries from the calling environment, which return a *statement handle* from which results can be retrieved one at a time using a standard cursor traversal mechanism.

4.2.4 Repository Object Interface

The Repository Object Application Programmer's Interface (REPO API) provides an object-relational data model to the Wirmlet developer by abstracting away the relational statement handles of the Table Manipulator and allowing the data to be viewed as

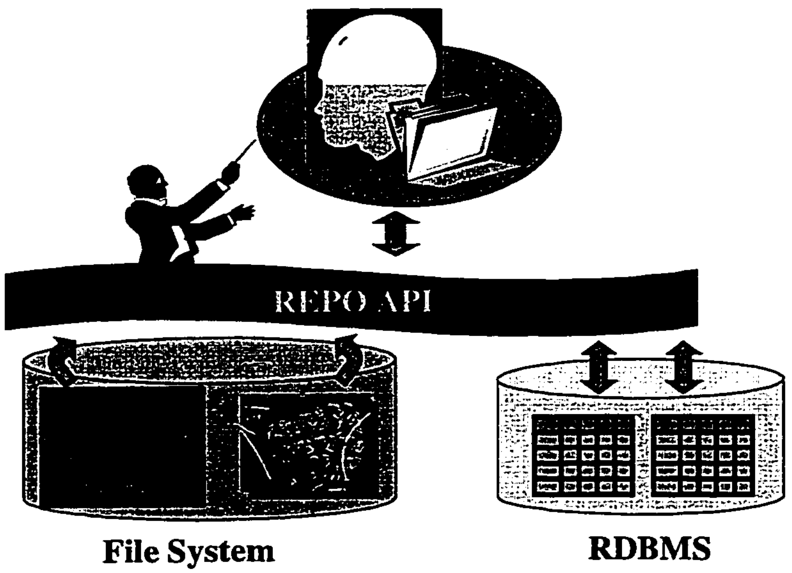


Figure 15: Repository Object API

collections of objects that conform to the Repository Object model as defined in the previous chapter. Each class is represented by a table in the relational database whose columns match the schema attributes, and whose rows comprise the instances of that class. By using the REPO API, The Wirmlet developer operates on object-oriented data structures rather than tables, navigating through networks of objects and their attributes rather than cursing through rows of data.

The REPO API is a collection of functions which allow the programmer to define new object types, import or create instances, edit existing instances, manipulate sets of instances, and pose queries over the data. An object may have more than one physical component, but the Object API allows the Wirmlet developer to refer to the object as a single entity. For example, an MR-image object consists of a file in the FSA and descriptive information in a database table. A request to retrieve an image by name would invoke an Object API function that looks up the image ID in the File Description Table, then retrieves the appropriate file from the FSA.

In addition to providing a convenient interface for the Wirmlet developer, the REPO API enforces consistency between the FSA and the associated metadata in the DBMS.

4.2.5 HTML Generator

The HTML Generator is a developer's interface that encapsulates some common user-interface constructs in high-level functions, which emit HTML strings, allowing the rapid development of Web documents. It is heavily used in constructing the Web View, by the View Methods of Class Definitions, and by the Wirmlets themselves. There are no methods in the HTML Generator which depend on the repository object model, so this API can be used independently of a WIRM repository. User interface methods which deal directly with Repository Objects can be found in the Gateway Interface.

The HTML Generator provides a suite of functions for creating and parsing interactive form elements (e.g. popup menus, scrollable lists, etc.), and other shortcuts for generating HTML syntax (e.g. turning an array into a formatted HTML table, handling document layout, displaying a thumbnail image, etc.).

4.2.6 Gateway Interface

The Gateway Interface provides a high-level interface for displaying repository objects as Web pages. There are generic methods for viewing objects as labels, rows, and full pages. The Gateway provides methods for generating HTML tables of objects, and for generating form elements for choosing from a list of objects. There are methods for creating JPEG versions of image files, and for creating thumbnails and clickable image maps.

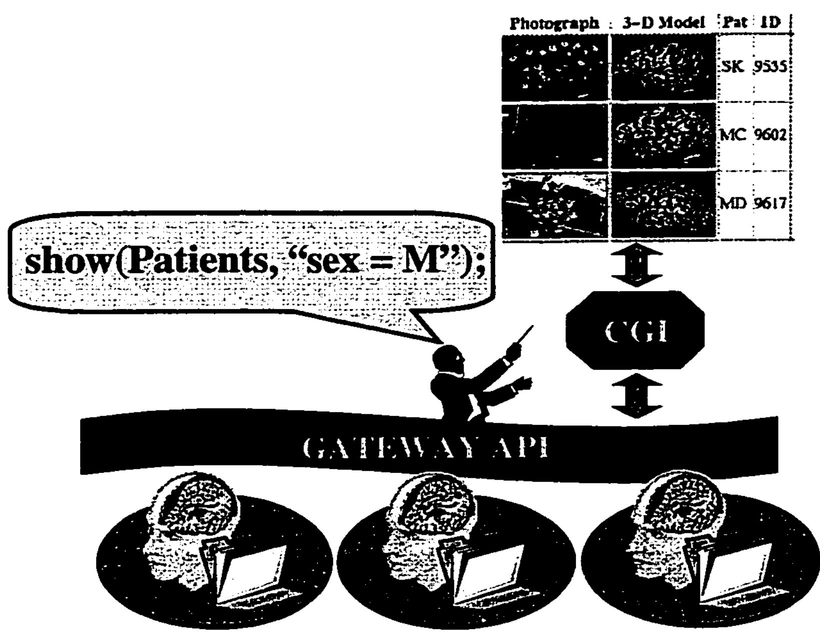


Figure 16: Gateway API

The Gateway is used along with the HTML Generator for developing Wirmlets and for defining the View Classes of repository object types. The details of the Gateway Interface shall be described in the Implementation chapter that follows.

4.3 Built-in Classes

In addition to the atomic classes (char, int, real, time, date), WIRM provides three built-in complex classes: File, Annotation, and User. The classes are described below.

The File class is used to contain the metadata (including location) of files registered into

Table 4: File Schema

FIELD	TYPE	DESCRIPTION
label	Char(100)	text label for identifying file
domain	Int	'FSA', 'SERVER', or 'URL'.
locator	Char(200)	URL or filename.
mime_type	Char(100)	mime-type label
submit_date	Date	date that file was imported
submitted_by	Char(10)	login name of submittor
version	Char(10)	for version history, if maintained
context	Char(100)	domain-dependent context info
description	Char(400)	text describing this file

the repository. It is used for all three categories of file: FSA files, which have been imported into the protected File Storage Area, LOCAL files, which reside on the server's file system but not in the FSA, and REMOTE files, which can reside on any network-accessible machine (including any file on the internet). The attributes maintained by the File class are shown in Table 4.

Table 5: Annotation Schema

FIELD	TYPE	DESCRIPTION
subject	any repo	OID of the target subject.
author	User	OID of the User who created the Annotation.
created	date	Date the Annotation was created.
body	char(500)	Text of the Annotation.

The File class includes a custom Make method that prompts the user for metadata (and automatically assigns the mime_type, submit_date and submitted_by values), and provides a facility for uploading files into the FSA. In addition to the standard View methods, the File class includes methods for viewing the contents of a File (without metadata), or an iconic version of the File.

The Annotation class supports the system-wide facility to attach a comment to any repository object. Annotations can be created using the Object Maker Wirmlet, which invokes a custom Make method that requests the user to specify the OID of a target object. The attributes of the Annotation schema are defined in Table 5.

The Annotation class includes methods for creating a "Make Annotation" icon that generates an annotation when clicked, which can be displayed in the view of any object. Also included is a method for creating a "Read Annotations" icon, which appears whenever an object has annotations.

The User class holds information about individual users, and allows the system to maintain query session state and provide customized views based on the user's group. The attributes of the User class are shown in Table 6.

Table 6: User Schema

FIELD	TYPE	DESCRIPTION
login	char(20)	Unique login name identifying the user.
first_name	char(50)	User's first name.
last_name	char(50)	User's last name.
email	char(100)	User's email address.
password	char(20)	Password chosen by user to authenticate login.
group	char(20)	Group assigned to user by administrator.

4.4 System Wirmlets

WIRM provides ten built-in Wirmlets, which come as part of any EMS. These Wirmlets encapsulate common functionality such as viewing or editing objects, which can be invoked directly from their URLs or programmatically as part of a view definition or custom Wirmlet. Custom links to the Wirmlets may be generated using special functions in the Gateway Interface. The ten Wirmlets are described below.

- **The Generic Console:** A top-level menu providing access to all generic repository functions, including user services (registering, logging in, etc.), browsing repository objects, performing updates, and defining or evolving classes. In addition to providing a menu of links to all the standard generic Wirmlets, it also offers parameterized links to those Wirmlets, such as the ability to register a file or make an annotation, both of which use the Object Maker Wirmlet with special parameters.

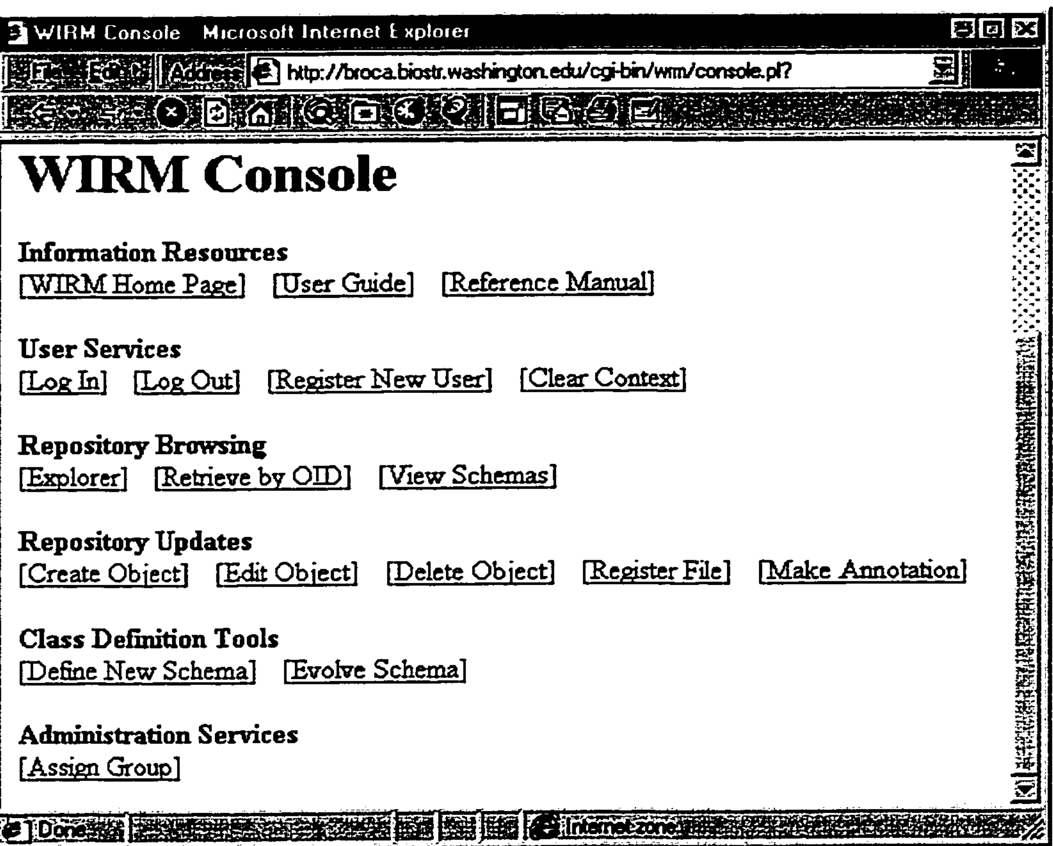


Figure 17: Generic WIRM Console

- **User Login:** Allows users to log in and out. Users logging in supply their username and password. Once a user logs in, their identity is maintained in the session context until they log out. Can be accessed via the repo_login_link function of the Gateway API.

- **Object Maker:** Creates new repository objects. The class name of the desired object is specified as a parameter. If a custom Make method exists for that class, the Object Maker invokes that method. Otherwise, it uses the default Make method, which supplies a prompt for every attribute in the class schema. If the supplied class is User, the Object Maker displays the New User registration screen. If the class is File, it displays the File registration screen, etc. If no class is supplied, presents a menu of

all repository classes to choose from. Can be accessed via the repo_mlink function of the Gateway API.

- **Object Editor:** Allows existing repository objects to be updated. The OID of the target object is specified as a parameter. If a custom Update method exists for that object's class, the Object Editor invokes that method. Otherwise, it uses the default Update method, which supplies a prompt for every attribute in the class schema, with the existing attribute values filled in. The user may make changes using the form elements, and the changes are committed to the database upon pressing submit. Can be accessed via the repo_elink function of the Gateway API.

- **Object Destroyer:** Deletes a repository object. The OID of the target object is specified as a parameter. The user is prompted for verification. If deleted, an object is permanently removed from the repository database.

- **Repo Viewer:** Displays the Page View of a specified object. This is the most commonly used Wirmlet, as every hyperlink that points to an object uses the Repo Viewer. All drill-down visualization goes through this Wirmlet. The OID of the target object is specified as a parameter. If a custom Page View method exists for that object's class, the Repo Viewer invokes that method. Otherwise, it uses the default Page View method, which displays an HTML table showing every attribute name and value. If no OID is specified, presents a menu for choosing an object by OID. Can be accessed via the repo_vlink function of the Gateway API.

- **Object Explorer:** Similar to the Repo Viewer, allows users to retrieve repository objects. While the Viewer retrieves objects by OID, the Explorer accepts an SQL query filter that is applied to objects of the specified class. The Viewer is appropriate for *known-item searching* (where the OID is known), while the Explorer is for browsing. In addition to the query filter, users may specify an *order-by* parameter

that identifies attribute(s) by which the results should be sorted. Matching results are retrieved from the database and displayed using the appropriate Row View.

- **Schema Maker:** A facility for defining new types in the database. Prompts for the new class name and the names and types of the attributes. They attribute types are presented in a popup menu which includes all atomic and complex types defined in the database, both user-defined and built-in. The user may also specify types that haven't yet been defined. This allows for circular type definition (e.g. type A has an attribute of type B, which in turn has an attribute of type A). When the user presses submit, the new type is created in the repository, and the type is immediately available for instance creation. Default View, Create, Edit, and Delete methods are automatically inherited by the class, and the user may override them with custom methods at any time.

- **Schema Viewer:** Displays all schemas in the repository, including attribute names and types. All schemas appear in one Web view, which is useful as a data dictionary.

- **Schema Evolver:** Allows any schema to be evolved in real-time. Provides operations for adding, deleting, and renaming attributes. Existing instances are automatically updated with the new schema. The underlying mechanism uses the following six steps: (1) create a temporary table based on the original table which includes the specified changes, (2) copy all the instances (including OID) from the original table to the temporary table, (3) delete the original table, (4) create a table with the original name that has the updated attributes, (5) copy all instances from the temporary table to the new table, and (6) delete the temporary table.

4.5 Constructing a Web View

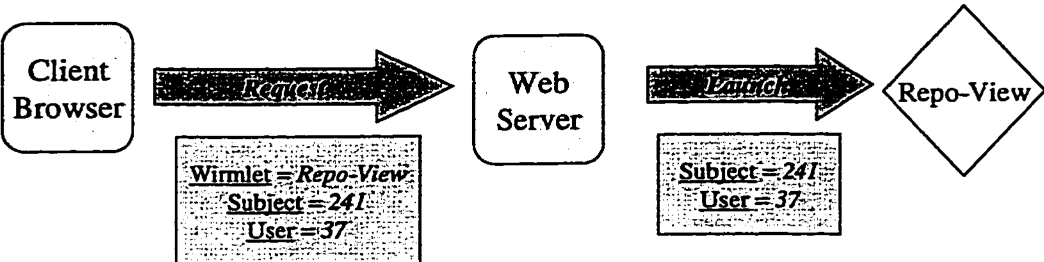


Figure 18: Invoking a Wirmlet

This section describes in detail the interaction between the various components and a Wirmlet in the construction of a Web View. I trace a single invocation of the *Repo Viewer* Wirmlet, activated on a Patient object with three attributes: the patient's name, age, and a photograph.

Figure 18 shows the client browser invoking the Repo Viewer Wirmlet. This would happen whenever the user clicks on a link to access the patient's Page View. The client's request to the Web server includes the URL of the target Wirmlet, the subject context (which is the OID of the Patient to be viewed) and the User context (which is the OID of the user currently logged in). The Web server invokes the Repo Viewer Wirmlet, forwarding the context parameters to the CGI script.

When the Repo Viewer Wirmlet is invoked, it accesses the Repository Object Interface to call the Repo Get function on the subject OID. Repo Get it looks up the subject's OID in the master index and determines that this subject is of the Patient class. It makes a DBSelect query to the Table Manipulator, using "Patient" as the table name and an SQL filter string to select the row whose OID column equals the subject's OID.

The Table Manipulator in turn accesses the Query Handler of the Msql Adaptor, which translates the request into an SQL query that is passed to the RDBMS server. The query is processed, and the resulting tabular data is returned through the Msql Adaptor and

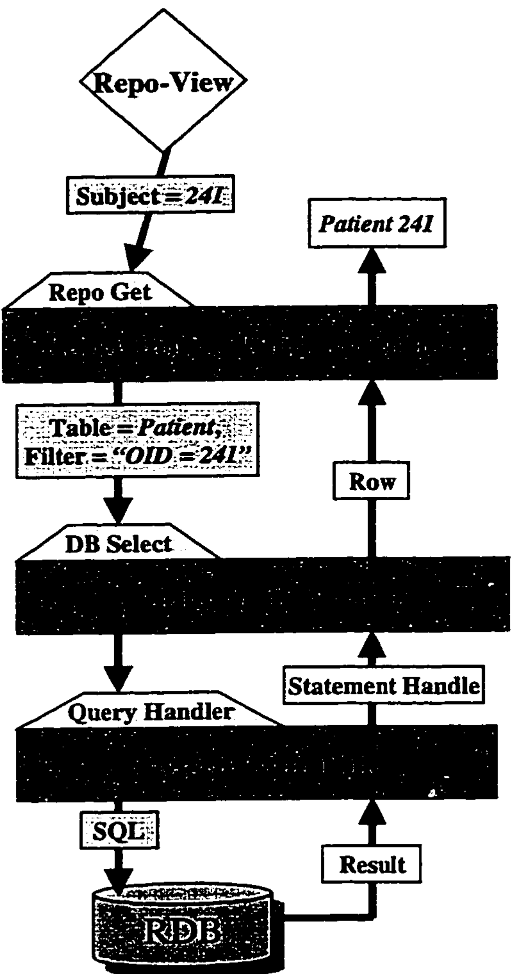


Figure 19: Retrieving a Patient

marshaled into a newly allocated Statement Handle data structure. A reference to the Statement Handle is passed back to the Table Manipulator, which extracts the row data into a simple Perl list and returns it to the waiting Repo Get function. The Repo Get function uses the Master Schema index to process the resulting list and generate a Perl hash structure whose keys are the attribute names and whose values are the values of the list. The hash structure serves as a Repo instance. A reference to the Repo is returned to the Repo Viewer Wirmlet, as depicted in Figure 19.

Once the Repo Viewer has the target Patient in memory, it then invokes the View Page method on that object, passing it the User context, as depicted in Figure 20a. If the

Patient class has a custom View Page method, it is used, otherwise the default View Page method is used. The View Page method accesses the Gateway interface to generate a Web view, and the attribute variables are filled in with the specific values of the current Patient, shown in Figure 20b.

The Web view includes the contents of the patient's photograph, which is currently not in memory. The value of the Photo attribute is the OID of a File object, so that File is retrieved with a second call to the Repo Get function of the Repository Object Interface. The File object (containing metadata and file location) is retrieved from the database and instantiated as an in-memory Repo.

The Patient View Page Method then invokes the View Contents method on the new File object, which calls an image display routine from the Gateway interface. The Gateway uses the FSA Controller to access the actual file from the File System, as depicted in Figure 21. The file is converted into a JPEG by the Visualization Cache Manager, and copied into a Web viewable area. An HTML image tag is generated for the JPEG image, and is returned by the File View Contents method to the Patient View Page method, where it is embedded in the Web View. The Repo Viewer Wirmlet emits the completed Web View and passes it to the Web server, which in turn transmits the HTML page to the waiting client browser.

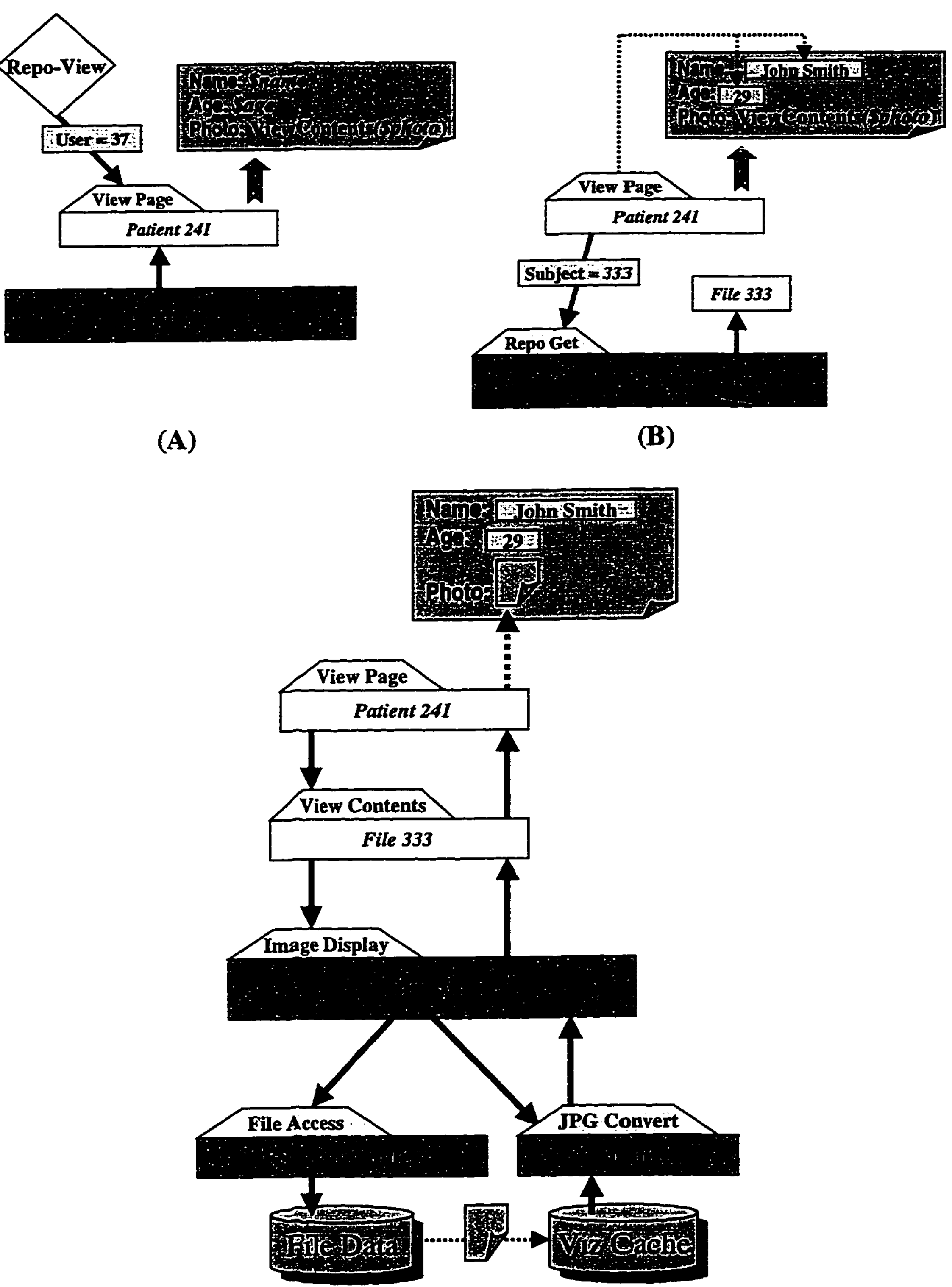


Figure 21: Embedding an Image in a View

Chapter 5: WIRM Implementation

In this chapter, I will describe how WIRM integrates the database with the programming environment, and give an overview of the implementation of each of the six application programmer's interfaces.

5.1 Persistent Perl

The WIRM Server modules have been developed in Perl [Wal91], a freely available, portable language designed for easy manipulation of text, files, and processes.
Perl's rich pattern matching and report processing features make it ideal for parsing forms and generating HTML documents. Perl's utilities for handling Unix processes and controlling the data flow between them makes it well-suited for interfacing the EMS with external tools.

Each Wirmlet is implemented as a Perl script residing in the CGI directory of the Web server. Because Perl does not require a separate compilation stage, it facilitates fast prototyping and testing of CGI scripts. For example, when adding a new feature to the console, the programmer merely edits a Perl script and clicks ``reload'' on the Web browser to instantly view the changes.

Along with Linux, Perl is the poster child for the free software movement. A large number of Perl modules have already been implemented for processing Web forms and accessing relational databases. These modules can be retrieved for free from the Comprehensive Perl Archive Network [Cpa-URL], and can be plugged directly into the WIRM architecture. CPAN also includes hundreds of other modules which can be seamlessly integrated into WIRM as "data blades" to extend the functionality of the server. For example, CPAN includes a set of modules called *LWP* which provide high level functions for performing http client requests and processing server responses. Incorporating LWP into WIRM would facilitate data integration with external Web sites.

The HTML Generator and Gateway Interface utilize a Perl module called *CGI.pm* [Ste-URL], which provides a simple interface for interpreting query strings passed to CGI scripts, and a rich set of functions for creating fill-out forms. The existence of CGI.pm saved huge amounts of development effort in designing the WIRM server.

WIRM exports the Repository Object Model to the Perl programmer. Repository Objects are implemented as a Perl Hash tied to a database row. When an instance of the type is created, a row is added to the table, and an associative array is allocated whose keys correspond to the column names, and whose values are bound to the row data. The Repository Object API defines query functions which utilize the relational query engine to retrieve rows which satisfy an SQL query, and then allocates an associative array for each row. The Perl programmer may iterate over the results, performing arbitrary computations, and updates are propagated back to the table, where they persist between invocations. Just as an object-oriented database can be described as *persistent C++*, WIRM can be described as *persistent Perl*, with the added benefit of a relational query engine and built-in multimedia support.

Although Perl provides object-oriented support, the current implementation of WIRM doesn't fully utilize it. Rather than having one module per class, all classes are defined in

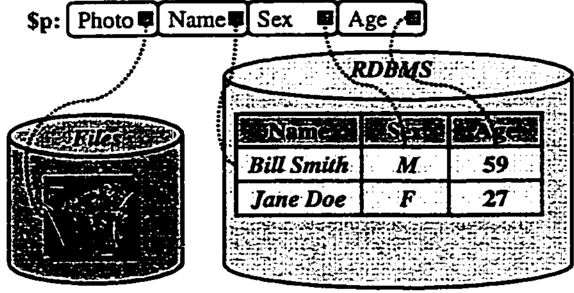


Figure 22: Persistent Perl

a single file (*class-defs.pl*). And rather than using Perl's built-in object-type invocation, types are explicitly managed at the interface level. For future implementations of WIRM, I plan to better use the object-oriented features of Perl.

5.2 Server Component Implementation

I have been calling WIRM a server, although in reality it is a set of scripts which are re-executed on every Web View, invoking the API's as included libraries rather than passing messages to processes. Conceptually, the Perl interpreter is a "server" that is recompiled on every Web View. The advantage to this approach is that it affords instant realization of changes to Wirmlets and Class Definitions, which makes a far superior debug cycle and simplifies the software development cycle by enabling real-time updates without bringing the system down. The obvious disadvantage is the overhead of compiling and loading every time, which results in a slower response time when the user requests a Web View. One solution to this problem would be to use Apache Mod-Perl, a library from the Apache/Perl Integration Project [Apa-URL], which compiles CGI scripts into the Web server, thereby speeding up response time. For most EMS systems, the advantages of a script-based approach outweigh the disadvantages.

The remainder of this chapter describes the implementation of each WIRM Server component, and gives an overview of each API. The individual functions are described in detail in the Appendix.

5.2.1 FSA Controller

The FSA Controller maintains the FSA Index, a Perl DBM file that associates File object IDs with file pathnames. The interface exports a File Path function, which encapsulates the index, allowing any file to be retrieved by ID.

Files can be imported into the FSA using either the Custom File Importer function (which allows the user to specify a destination path within the FSA for readability), or the Default File Importer function (which assigns a location automatically). The Custom File Importer copies binary data from a Perl file handle into a user-specified directory in the File Storage Area. The path directories are created if they don't already exist, and the FSA Index is automatically updated.

For files imported using the Default File Importer, a default destination directory is assigned by hashing on the file ID. This allows files to be distributed across multiple directories, rather than keeping them all in one central directory, which would cause performance degradation of the file system if thousands of files are maintained.

5.2.2 Visualization Cache Manager

The VCM is implemented as a Perl module that uses the third-party module *Image::Magick* [Mag-URL] to perform basic image manipulation. Image::Magick recognizes a wide range of image types, although users may extend the set of image types by implementing their own conversion routines, such as I have done with the GE MRI format. These extensions must be incorporated into the VCM Converter function.
The VCM provides functions for generating full-sized web viewable replicas of images, and for generating thumbnail versions. When a Wirmlet displays a repository file, it invokes the VCM's Cache Retrieval functions, which converts the file into the desired format and copies it into the cache (if it is not already there).

Currently, the VizCache consists of three partitions: one for thumbnails, one for full-size images, and one for other (non-image) file types. The VizCache location is specified in a configuration file, which allows the system to be customized according to the developer's requirements.

The location of a file in the VizCache is determined by the file ID (using a hash function), so that must be supplied when using the Cache Retrieval functions to request a file. Helper functions compute the actual and virtual location of a file's cached copy. (virtual paths are rooted at the hypertext document root, rather than the file system root, and are useful for generating Web Views).

5.2.3 Table Manipulator

Since the RDBMS is currently MSQL [Hug-URL], the Table Manipulator (TM) is built on the *msql-perl adaptor* [Koe-URL] as the interface to the database server. Future version of the Table Manipulator will use the more generic DBI instead, which can interface with a wide range of database systems. Note that the TM buffers the Repository Object Interface from changes to the underlying database adaptor, so migrating to DBI should only minimally affect the exported interface.

As explained in the Architecture chapter, results of queries to the database are returned as *statement handles* (2D arrays) by the Table Manipulator. A call to the database selector interface executes the query, and the resulting rows may be retrieved using the Fetch Row method. The selector methods accept an optional filter parameter which can constrain the query with a Boolean SQL WHERE clause (see reference manual for syntax). For example, the following code retrieves older male patients and prints out their names and ages:

```
$pats = db_select("Patient", "gender = 'M' AND age > 50");
while (@p = $pats->fetchrow()) {
  print "Name: $p[0], Age: $p[1]\n";
}
```

The next section demonstrates how the Repository Object API improves on the statement handle interface by abstracting away the table structure.

In addition to the selector facilities, the Table Manipulator exports methods for connecting to the database server, creating indexes to speed up query access, defining new tables, destroying existing tables, retrieving metadata about the names and types of table columns, joining multiple tables into a single statement handle, and operations for modifying table data (inserting, deleting, and updating rows).

5.2.4 Repository Object Interface

As explained in the Architecture chapter, the Repository Object API provides an object-relational data model to the CGI programmer by abstracting away the relational statement handles of the Table Manipulator and allowing the data to be viewed as collections of objects. The instances of an object type are maintained in a relational table, but the CGI programmer need not be concerned about the logical structure of the table representation, and instead operates on hashes whose values are bound to the row data.

The REPO API exports query functions that utilize the Table Manipulator to retrieve rows that satisfy an SQL filter clause, and then allocates an in-memory hash for each row. The CGI programmer may iterate over the results, performing arbitrary computations, and updates are propagated back to the table, where they persist between invocations.

The REPO API connects to two Perl DBM files: The Repo Types Index, which maps every OID to a type name, and the Schema Definition Index, which records the attribute names and types for each object class.

The methods of the REPO API are described in detail in the appendix. As a concrete illustration of the power of the REPO API, the remainder of this section includes a brief synopsis of the most important methods.

New schemas are defined using the *repo_define* function. Here is an example of how to define a Patient schema, which consists of a name, sex, age, and a reference to a surgeon object:

```
@patient_atts = ("name:char(20)", "sex:char(1)",
                 "age:int", "surgeon:repo");
repo_define("Patient", @patient_atts);
```

Instances are created using *object templates*, which are Perl hashes whose keys match the attribute names and whose values contain the initial values of the object to be created. The template is passed to the *repo_new* function as follows:

```
$template->{name} = 'Smith';
$template->{sex} = 'F';
$oid = repo_new("Patient", $template);
```

Repository Objects are retrieved from the database into memory using a variety of query functions, such as *repo_query* (to retrieve a group of objects using an SQL filter on their attributes), *repo_query_single* (to retrieve a single object), and *repo_get* (to swizzle an OID, replacing it with the object itself). Results are returned as a list of pointers to instantiated repository objects, which can be iterated over with a standard foreach loop. For example, the following code retrieves all male patients and prints their names and the names of their surgeons. Note the contrast to the Table Manipulator API:

```
$patients = repo_query("Patient", "sex = 'F'");
foreach $pat (@$patients) {
```

```
    $surg = repo_get($pat->{surgeon}); # retrieve surgeon object
    print "Patient: $pat->{name}, Surgeon: $surg->{name}\n";
}
```

Objects can be updated by directly manipulating the values in their hash, and then calling the *repo_update* function to commit the changes to the database. For example, the following code retrieves the patient named Smith, adds a year to his age, and then assigns him the surgeon named Jones:

```
$smith = repo_query_single("Patient", "name = 'Smith'");
$smith->{age} = $smith->{age} + 1;
$surg_id = repo_lookup_id("Surgeon", "name = 'Jones'");
$smith->{surgeon} = $surg_id;
repo_update($smith);
```

Other important methods provided by the REPO API are the Repo Evolver, which can be used to add, drop, or rename the attributes of an existing schema. Instances of the class will be automatically updated to reflect the changes. The original table is copied into a temporary table then destroyed & recreated with the changed columns, and the data in the temporary table are then copied back into the new table. For example, the following invocation renames a Patient's ''sex'' attribute to ''gender'', and adds a new ''age'' attribute:

```
repo_evolve("Patient", ("rename:sex:gender", "add:age:int"));
```

Other methods allow the user to remove schema definitions, retrieve information about the names and types of a schema's attributes, determine the type of an object reference, and translate between an object and an OID.

In addition, the REPO API provides methods for handling File objects. Eventually I will implement Files as a truly object-oriented class, but for now their methods are divided

between the REPO API and the Gateway API. The REPO API includes functions for importing file data into the repository and registering its metadata. Files are handled through a File Descriptor Template (FDT), whose fields are defined in the Architecture chapter.

The fields of the FDT are maintained in the relational database under a table called File. There are three classes of file that can be registered: *FSA*, *local*, and *remote*. FSA files are copied into the File Storage Area using the Import File facility, which has the advantage of assuring that the file will always be available for retrieval and that the file metadata will remain consistent with the file itself. Local files are accessible by the file system of the repository server, but are not copied into the FSA, so their metadata can become inconsistent if the file is moved or changed by an external agent. Finally, remote files are designated by a URL. The Gateway Interface section explores the implications of these three classes in terms of retrieval and storage.

5.2.5 HTML Generator

The HTML Generator is a developer's interface that encapsulates some common user-interface constructs in high-level functions that emit HTML strings, allowing the rapid development of Web documents. The HTML Generator is based on the popular Perl module, *CGI.pm* [Ste-URL], extending it with some higher-level abstractions. Unlike the Gateway Interface, which will be described below, the HTML Generator does not depend on the WIRM libraries and can be used as a stand-alone API for creating CGI-based web sites.

An important aspect of the HTML Generator is that the functions all emit strings instead of printing to standard out. This results in much cleaner code, in which the output can be spliced and prepared in any order before sending it to standard out.

The HTML Generator provides functions for generating pre-formatted form elements (e.g. *HT_date_choice*, which creates a popup menu of month names), navigational elements (e.g. *HT_href*, which generates a labeled URL), and other web document facilities. One of the API's most powerful features is the *HT_table* facility, which transforms arrays of data into formatted HTML tables. The programmer supplies a declarative *Table Template* to the function, which defines the table presentation style, including the colors of the rows, the alignment of the cells, and the style of the border.

5.2.6 Gateway Interface

The Gateway Interface provides a high-level interface for displaying repository objects as Web pages, including methods for generating HTML tables directly from sets of objects, and for generating form elements for selecting an object from a group. There are methods for creating JPEG versions of image files, and for creating thumbnails and live image maps.

The Gateway interface provides methods for maintaining the Context-State as defined in the Query-By-Context view model. The context values are stored in a Perl hash with descriptive keys such as "subject"', and "user", whose values are passed between Wirmlet invocations as special form parameters.

The Gateway functions are organized into four categories: generic object methods, image handling, form processing, and system class definitions (custom methods for Files, Annotations, and Users).

An important generic object method is *repo_view,* which takes a subject (or group of subjects) of any type and emits an html visualization of that object (or objects). The programmer may specify a specific view function to use, or the default views are used for the given object's class. Another important function is *repo_vlink,* which generates a

URL to a view of the given subject, using the Repo Viewer Wirmlet. Similar functions exist for the Make Object, Edit Object, and Delete Object Wirmlets.

Image handling methods include high level functions for utilizing the Visualization Cache Manager for generating thumbnails and full-size images from files. Other methods exist for creating image maps that return pixel coordinates when clicked.

Form processing methods include *repo_types_menu,* which generates a scrolling list of all possible repository object types, using the specified parameter name, *repo_choice,* which generates a popup menu whose choices are labeled by the default labels of a group of objects, and *repo_smart_prompt,* which automatically creates a form input element appropriate for a given attribute type.

Finally, custom class definitions are provided for the three built-in WIRM types (User, File, and Annotation), including methods for Label view, Row view, and Page view, as well as custom Make and Edit methods. In addition, more custom class methods are provided, such as methods for displaying the contents of a file (regardless of type), and a method for generating context-sensitive annotation icons. These and other methods may be viewed in detail in the appendix.

Chapter 6: Methodology

An EMS is defined by its domain knowledge and domain data. As specified by the Query-By-Context view model, domain knowledge is encoded in two forms: as Class Definitions (Schemas and Methods) and as Custom Wirmlets. The domain data is encoded as instances in the database and files in the repository. As part of the EMS design, domain experts articulate their domain knowledge to an EMS developer, who then uses the EMSBE to encode the domain knowledge as Class Definitions. In some cases, the domain experts may perform some of the encoding themselves, especially with regard to schema design. In any event, the process of encoding domain knowledge can be guided by following the WIRM EMS Development Methodology, as described in this chapter.

The task of building an EMS is really a data integration problem, as it involves resolving schematic and semantic conflicts across knowledge representation systems [Kim91]. As defined in Chapter II, a major role of the EMS is integrating heterogeneous systems and providing a uniform interface for accessing data from multiple sources. Each external application that interfaces with the system can be seen as an information producer or consumer. Some sources provide data, others provide knowledge, and many provide both knowledge and data. For each source, the key is to integrate the source's knowledge into a consistent framework, and map the data into the model using the knowledge framework as a guide.

The WIRM EMS Development Methodology first requires the EMS developer to work with the domain experts to develop a canonical modeling of real world concepts (knowledge) using the Repository Schema Model, and then map existing information systems to the canonical model, which facilitates the integration of data entities across systems.

In the EMS Development Methodology, it is recognized that schema and class definition must occur as a process of stepwise refinement. An EMS is constructed in many interleaving stages of experiment design and execution, and the classes and Wirmlets will naturally evolve as the EMS developer's understanding of the experiment data matures.

The methodology consists of the following steps:

- Designing Schemas in terms of the Repository Schema Model
- Modeling context-sensitive Class Definitions as defined in the Query-By-Context View Model
- Developing custom Wirmlets to interact with users and external applications
- Evolving schemas and Wirmlets as the EMS matures

The remainder of this chapter explores each of these steps in turn. To illustrate the methodology, I will use a simple example domain, the Lab Lending Library. The Lab Lending Library is a Web-based application that will maintain a repository of information about the books in a lab, keeping track of who's borrowed what. The application will define two simple classes: Book and Loan. In addition, the application will make use of WIRM's built-in User class for managing library users, and the File class for maintaining an image repository of book covers.

6.1 Designing Schemas

The domain experts should work with the EMS developer to identify the salient concepts in the domain, and to define attributes for each of those concepts in terms of atomic, composite, or aggregate types as allowed by the Repository Schema Model. Composite types can be system-defined or user-defined, as allowed by the object-relational data model. This process often involves observing the experiment data and classifying it according to semantic ontologies. It is often useful to pick a specific concept to be the

"central concept" and have all other concepts relate to it hierarchically. For example, in the Brain Mapper EMS, I chose the Patient concept as the central concept, and all other concepts were defined in relation to the patient.

It is common for two concepts to merge after awhile, or for a concept to split into separate concepts. The schema evolution capabilities of the EMSBE facilitate this.

Once the concepts and their attributes are identified, they should be implemented using the built-in point-and-click Schema Definition Wirmlet. The Schema Definition Wirmlet supports the creation of attributes that conform to the Repository Schema Model. A drop-down menu enables the user to choose from the allowable atomic types (strings, integers, etc.) or to choose a reference to a composite type. In the Lab Library Domain, the Schema Definition Wirmlet can be used to define a Book and a Loan schema, whose attributes are shown in Tables 7 and 8. Note the convention that class names are capitalized ("Book") and attributes are lower-case. The "status" attribute of the Loan schema will be either ACTIVE or CLOSED, depending on whether or not a book has been returned.

Table 7: Book Schema

NAME	TYPE	LENGTH
title	char	200
author_last_name	char	50
author_first_name	char	50
owner	User	--
cover	File	--
availability	char	20

Table 8: Loan Schema

NAME	TYPE	LENGTH
borrower	User	--
book	Book	--
date_borrowed	date	--
date_returned	date	--
status	char	20

The Schema Definition Wirmlet (depicted in Figure 23) recognizes all types that are known to the repository. For example, after creating the Book class, that becomes an option when defining the attributes for the Loan class, automatically appearing in the drop-down menu. For types which haven't yet been defined (or for circular type definitions), the Wirmlet allows the type to be written in using the "Other" field.

When the user presses SUBMIT in the Schema Definition Wirmlet, a new Schema is

Figure 23: Defining the Book Schema with the Schema Definition Tool

added to the Schema-Defs component of the Repository-State, consisting of a set of Attribute-Defs which conform to the (Att-Name, Att-Type) structure prescribed by the Repository Schema Model.

Now that the schemas have been defined, the repository is instantly ready to accept new data objects of these types. In this way, the object-relational characteristics of the Repository Schema Model are fully supported. The Repository Object Creator Wirmlet can be used to insert some Books and Loans by hand, as shown in Figure 24. Because the Objects have no Make methods defined in their Class definitions, the Object Creator uses the default Make methods to generate the attribute prompts. Note that the Web View shown in Figure 24 adheres to the Query-By-Context view model: the Wirmlet is acting on the current Repository-State, which includes the default Make Object method definition, and the user's input is being stored as the current Form-State . Notice the use of the *smart_prompt* function (as defined in the Gateway Interface) to create appropriate form elements for the various attribute types. When the user presses SUBMIT, the WorkLogic function of the Object Maker Wirmlet will be invoked, processing the current Form-State and Context-State, causing a new Book instance to be added to the Repository-State.

Part of the process of creating a Book is uploading a File into the FSA using the File Upload Wirmlet. All the metadata is prompted for or automatically assigned.
Objects may also be updated using the Object Update Wirmlet. The default Object Update Wirmlet looks just like the Object Maker Wirmlet in Figure 24, except that the attributes are filled in with their current values. In the next section, custom Make and Update methods will be defined to override these defaults.

Figure 24: Making a Book with the Object Maker

6.2 Modeling Class Definitions

The process of modeling class definitions involves creating *Class-Def* instances that adhere to the form defined in the Query-By-Context View Model. As dictated by the model, Class-Def instances are composed of Method-Defs which can be View Functions, Make Functions, Edit Functions, Delete Functions, or Misc. Functions. These are implemented as specially formatted Perl functions in a Class Definition file. The contents of the Class Definition file make up an important part of the Repository-State, which is part of the Session-State that determines the contents of every Web View. By adhering to this model, the EMS designer is given a natural framework in which to create

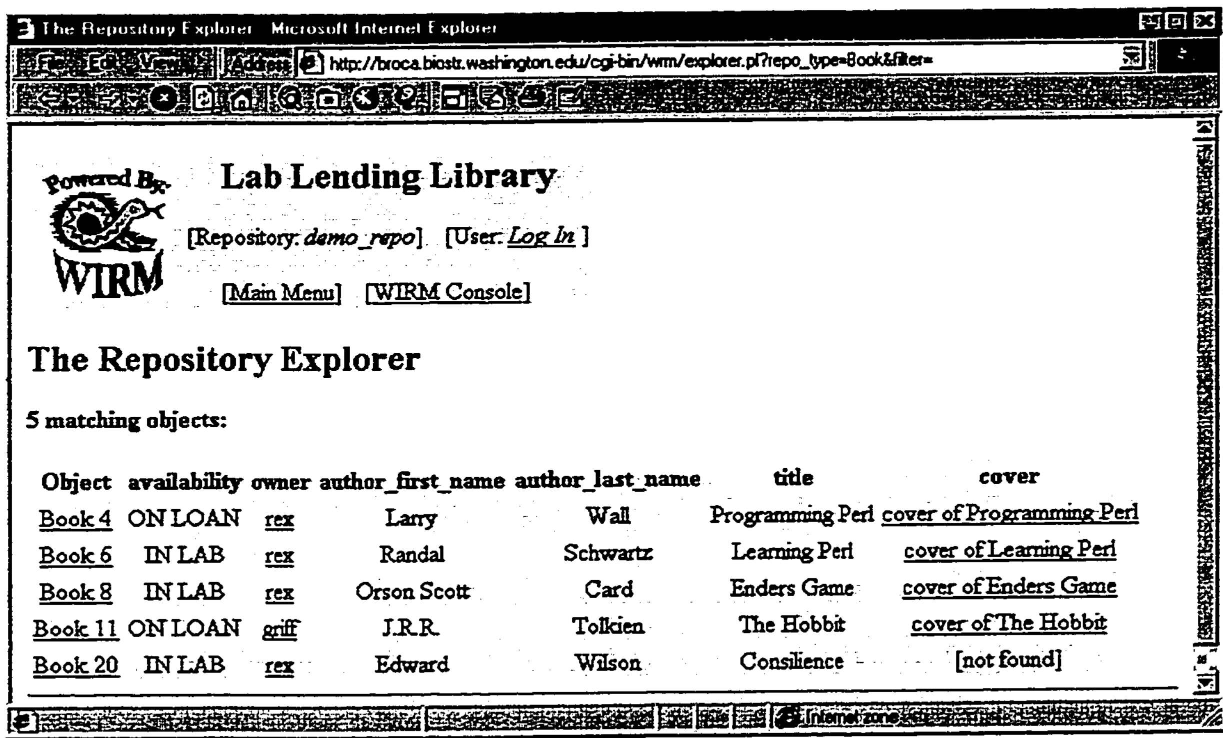
context-sensitive interfaces. By defining the prescribed View Functions (View-Page-Fn, View-Table-Fn, and View-Label-Fn), the Repo Viewer Wirmlet automatically enables a Virtual Navigation Space of drill-down visualization interfaces, as will be demonstrated in the examples which follow. Moreover, by defining the prescribed Edit-Fns and Make-Fns, the Edit Object and Make Object Wirmlets automatically enable a hierarchical workflow management interface.

Classes in the repository can be assigned arbitrary methods, but the Query-By-Context Model dictates that certain methods are inherited by every class if no custom versions are defined for them. By inserting a function starting with the class name and ending with a suffix matching the list below, that method automatically replaces the default method for that class (e.g. *Book_make overrides Class_make* for the book class):

- *Make-Fn:* implemented as a function called *"<class-name>_make"* which is called by the Object Maker Wirmlet. Should generate prompt and process fields to create a new object. The default simply prompts for every attribute (using smart prompts based on attribute type).
- *Update-Fn:* implemented as a function called *"<class-name>_update"* which is called by the Object Update Wirmlet. Should generate prompt and process fields to create a new object. Fields should be initialized with the existing values of the specified subject. The default prompts for every attribute.
- *Delete-Fn:* implemented as a function called *"<class-name>_delete"* which is called by the Object Destroyer Wirmlet. Should specify a repo_delete on the object, plus any sub-parts that should be cleaned up when the object is removed. The default removes the object only, leaving the sub-parts unchanged.
- *View-Label-Fn:* implemented as a function called *"<class-name>_view_label"* which is called whenever the object is to be displayed in label form. Should return a descriptive hyperlink that points to the Page view for this object. The default emits the class name followed by the OID.

- *View-Table-Row-Fn:* implemented as a function called *"<class-name_view_row* which is called whenever an object is displayed in the context of an HTML table. Used by the Repo Viewer Wirmlet and Explorer Wirmlet when displaying a set of objects. Should return a reference to a list of values (which can be either raw attributes or processed values). The default emits a reference to a list of every attribute value (unprocessed) in the order that they appear in the schema, with the first element being the Class_view_label for that object.
- *View-Table-Header-Fn:* implemented as a function called *"<class-name>_view_row_header"* which is called whenever a group of objects is to be displayed in table form. Used by the Repo Viewer and Explorer Wirmlets to generate the header for an HTML table of objects. Should return a reference to a list of strings that correspond to the attributes emitted by the Class_view_row method. Default is to emit a reference to a list of the attribute names as they appear in the schema, with the first element being the string "Object" for the Class_view_label column.
- *View-Page-Fn:* implemented as a function called *"<class-name>_view_page"* which is called by the Repo Viewer Wirmlet whenever a single object is displayed using the repo_vlink function. Should emit an HTML page showing the focus object. The default is to create an HTML table with a single row by executing the Class_view_row and Class_view_row_header methods.

As an example, Figure 25 shows the Explorer displaying a list of Books using the default *Class_view_row* and *Class_view_row_header* method. Notice the first column, "Object", which uses the default *Class_view_label* method. Although it uniquely identifies a book, it is not very user-friendly, so it will be overridden with a new function called Book_view_label in the class definition file.



Object	availability	owner	author_first_name	author_last_name	title	cover
Book 4	ON LOAN	rex	Larry	Wall	Programming Perl	cover of Programming Perl
Book 6	IN LAB	rex	Randal	Schwartz	Learning Perl	cover of Learning Perl
Book 8	IN LAB	rex	Orson Scott	Card	Enders Game	cover of Enders Game
Book 11	ON LOAN	griff	J.R.R.	Tolkien	The Hobbit	cover of The Hobbit
Book 20	IN LAB	rex	Edward	Wilson	Consilience	[not found]

Figure 25: Default Row View for Book Class

```
sub Book_view_label {
    my($r) = @_;

    $r = OBJ($r);
    return HT_href($r->{title}, repo_vlink($r));
}
```

Book_view_label first uses the *OBJ* function to "swizzle" the parameter $r, assuring that it is a reference to a repository object. This allows the function to accept both references and OID's as the parameter. For flexibility, all class methods should do this.

The call to *HT_href* generates an HTML hyperlink, whose label is specified as the contents of the title attribute, **$r->{title}**, and whose URL is specified by a call to the

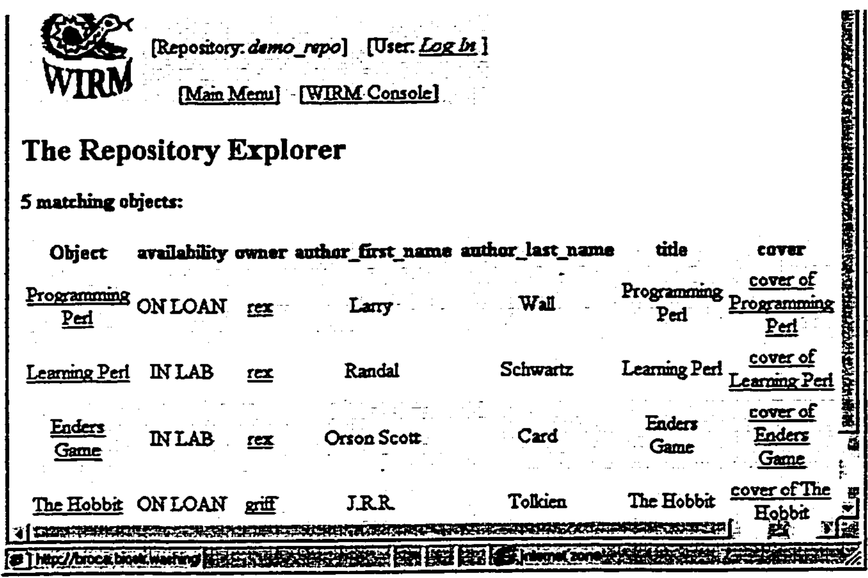


Figure 26: Row View Using Custom Label

repo_vlink facility. As expected, *repo_vlink* creates a URL to the Repo-View Wirmlet, using the current object (**$r**) as the subject. The result is a hyperlink that takes the user to the full *View_Page* view of the object.

Once the changes to the class definition file have been saved, they are immediately reflected in any new Web Views. Figure 26 shows the results of refreshing the Web View shown in Figure 25. Notice the first column now shows a hyperlink labeled by the book's title.

The rows in the table of books are being generated by the default *view_row* method, which now exhibits some redundant information (the title is repeated). The developer may define customized *Book_view_row* and *Book_view_row_header* functions to eliminate the redundant title column. Furthermore, the specialized row views can omit the **published** and **owner** attributes, since this detailed information is more appropriate for the *page view*.

```
sub Book_view_row_header {
    return ["Book",  "Author", "Cover"];
}

sub Book_view_row {
    my($r) = @_;
    $r = OBJ($r);
    return [
            Book_view_label($r),
            "$r->{author_first_name}
                $r->{author_last_name}",
            File_view_icon($r->{cover})
            ];
}
```

The first function takes no parameters and returns a reference to an array of strings, which will be used as column headers. Notice that the developer may choose arbitrary labels for the headers, and they don't have to correspond to the names of attributes in the schema. The second function takes a Book as a parameter, and returns a reference to a list of strings which will be used as row values in the construction of a table of Books. Note that the first column make use of the *Book_view_label* method, so if the label view is ever changed it will automatically be updated in the row view. This hierarchical approach to view construction, as prescribed by the QBCx data model, has huge implications in terms of saving time during interface evolution. The second item emits the Author field by concatenating the first_name and last_name attributes. The third item emits an iconic version of the cover.

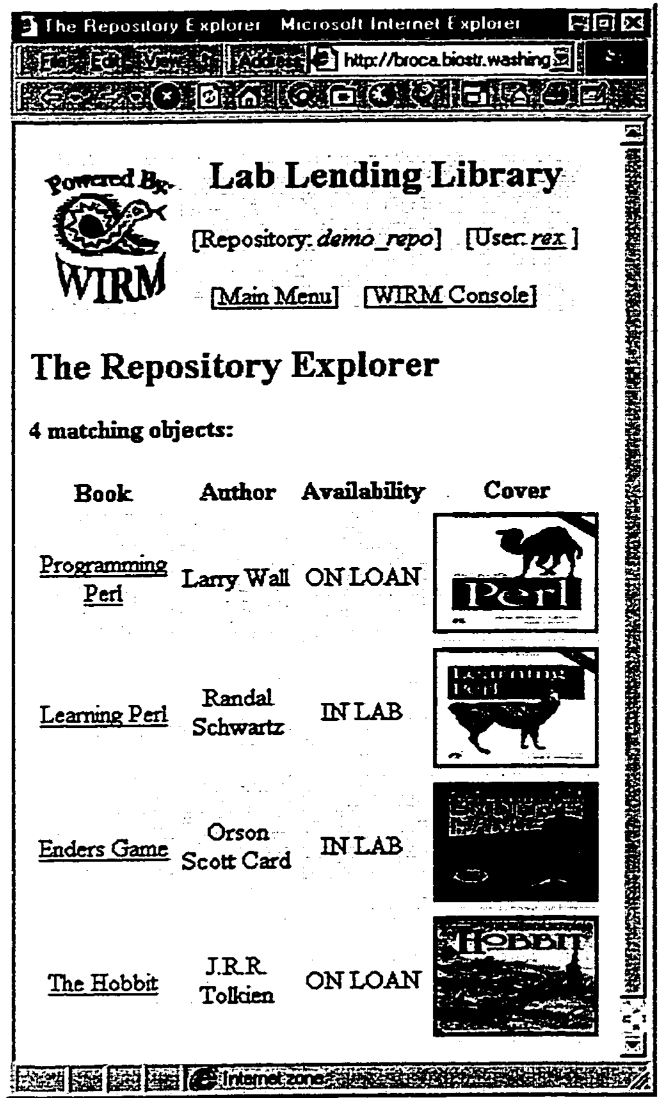


Figure 27: Custom Row View for Book

Figure 27 shows a screenshot of the customized row view.

The developer may also customize the *Page View* for the Book class, as follows:

```
sub Book_view_page {
    my($r) = @_;
    my($tt, $out);
```

```
    $r = OBJ($r);
    $out = h3("Title: $r->{title}");

    $tt->{HEADER} = ["Author", "Owner", "Availability",
                     "OID"];
    $tt->{ROWS} = [
        "$r->{author_first_name} $r->{author_last_name}",
        repo_view_label($r->{owner}),
        $r->{availability},
        $r->{oid}
      ];
    $out .= HT_table($tt);

    if ($CONTEXT{user}) {
        $out .= Annotation_view_icon_make($r);
        $out .= HT_space() . repo_edit_icon($r);
    }

    $out .= p() . b("Cover: ") . br();
    $out .= File_view_contents($r->{cover});
    $out .= p() . Annotation_view_icon($r);
 return $out;
}
```

The function returns an HTML string, which is used by the Repo View Wirmlet to render a Book as a Web page. The HTML string is composed in pieces and stored in the variable *$out*, which is returned at the end of the function.

The function first emits the Book title using <H3> font. Then an HTML table is generated, using a *table template*. A list of strings is assigned for the column headers, and

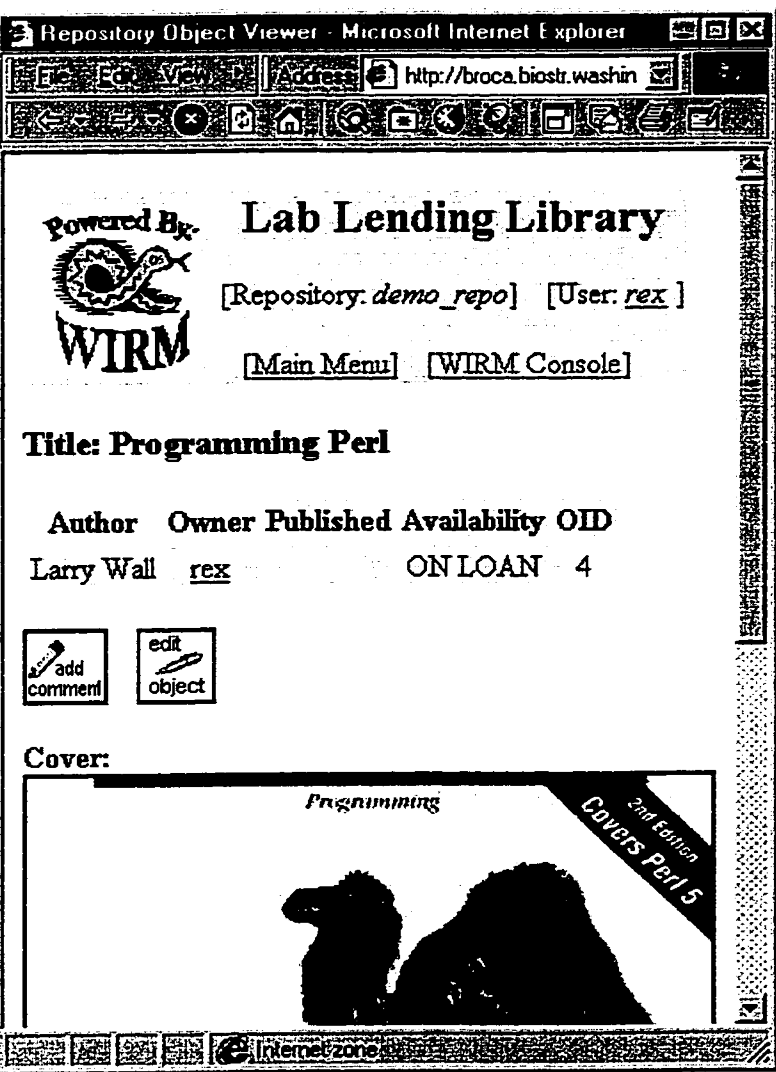


Figure 28: Custom Page View for Book

a row of values for the body of the table. The cover is displayed using the *File_view_contents* function. The *Annotation_view_icon* function emits the **read comments** icon if there are any comments. Finally, the **make comment** and **edit object** icons are displayed if the user context is defined (i.e. if the user has logged in). This is an example of how to use Query-By-Context to create a context-sensitive view.

The developer may also define a friendlier interface for checking out books. First, consider the default interface for creating a Loan object. Figure 29 is a screenshot of the

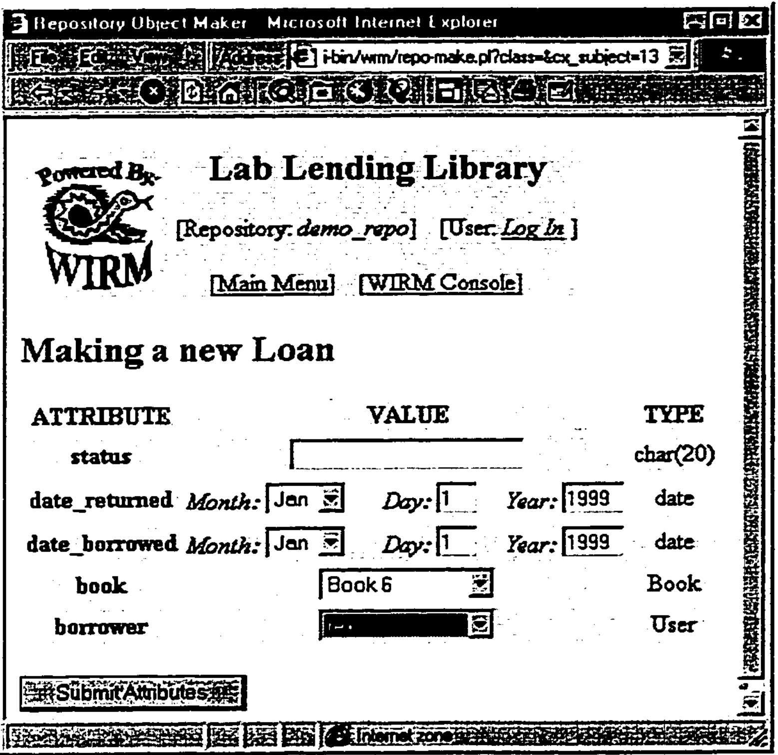


Figure 29: Default Make Prompt for Loan

Create Object Wirmlet used to create a Loan, calling the default Make method. Notice that the user is prompted for a status, date_borrowed, borrower, etc. It would be better if the Status was automatically set to "ACTIVE", the date_borrowed was automatically set to today's date, and the borrower was set to the current User. The only prompt should be which book to borrow. Furthermore, the act of making a loan should have the side effect of updating the *availability* attribute of the book being borrowed.

This can be achieved by adding the following method to the Loan class:

```
sub Loan_make {
    my($out);
    if (!$CONTEXT{user}) {
        return "You must first ", HT_href("log in", repo_login_link());
    }
```

```
    if (!param("Submit Attributes")) {
        $out = Loan_make_prompt();
    } else {
        $out = Loan_make_finish();
    }

    return $out;
}
```

The presence of this function will override the default make function when the Object Maker Wirmlet is activated on a Loan object. The first line requires that the user be logged in to make a loan. The function then checks the Form-State parameter *"Submit Attributes"*. If it doesn't exist, the user is prompted via the *Loan_make_prompt* function, defined below. Otherwise, the user has already pressed Submit, so the *Loan_make_finish* function is executed.

```
sub Loan_make_prompt {
    my($out, $avail_books);

    $out = h3("Borrowing a Book");
    $out .= em("Which Book are you borrowing?") . HT_space();

    $avail_books = repo_query("Book", "availability = 'IN LAB'");
    $out .=  repo_choice($avail_books, "book_choice") . p();
    $out .= submit("Borrow This Book");

    return $out;
}
```

The prompt is intelligent: it only shows books that are in the lab. The *repo_query* function makes a query on the Book class, retrieving all instances whose availability is "IN LAB", and storing them in a result variable named *$avail_books.*. The *repo_choice* function creates a popup menu based on the available books. The labels of the choices

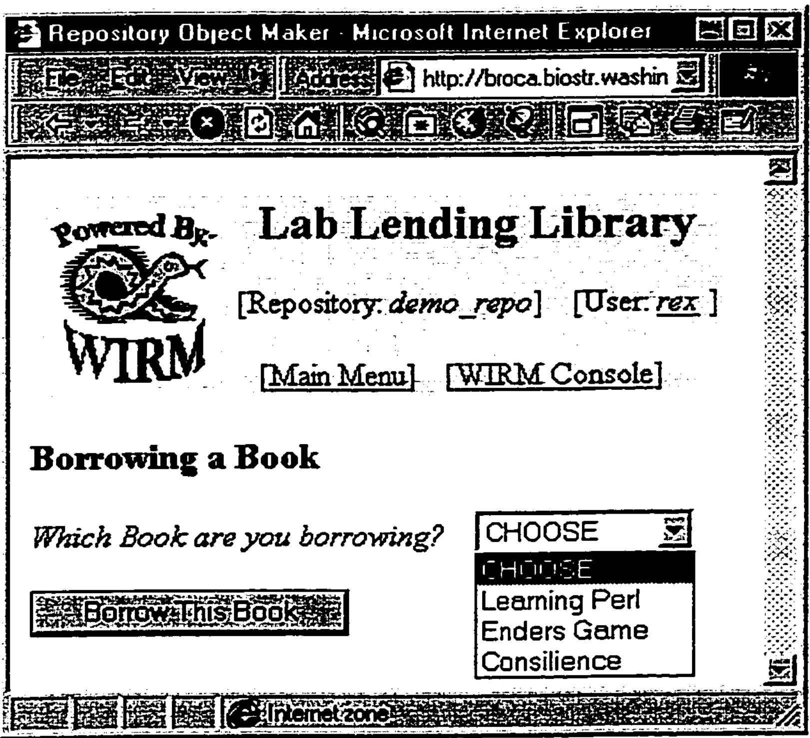


Figure 30: Custom Make Prompt for Loan

will be generated by applying the *Book_view_label* method on each item in *$avail_books*. When the user selects a book and presses the submit button, the selected object's OID is returned as a form parameter named *book_choice*. The custom Make Prompt is shown in Figure 30. Notice how improved it is over the default in Figure 29.

When the user presses the submit button, the Make Object Wirmlet is re-activated, and so *Loan_make* is executed again. This time, the form parameter "Borrow This Book" is defined, so *Loan_make* calls *Loan_make_finish:*

```
sub Loan_make_finish {
    my($out) = @_;
    my($book, $book_id, $loan, $loan_id);

    $book_id = param("book_choice");
```

```
    $loan->{borrower} = $CONTEXT{user};
    $loan->{book} = $book_id;
    $loan->{date_borrowed} = db_date_now();
    $loan->{status} = "ACTIVE";
    $loan_id = repo_new("Loan", $loan);

    # update book status
    $book = repo_get($book_id);
    $book->{availability} = "ON LOAN";
    repo_update($book);

    $out  .= "The following loan has been recorded: <P>";
    $out .= repo_view($loan_id);

    return $out;
}
```

First, the chosen book is retrieved into a variable called *$book_id*. Next, a Loan template is filled out. The *borrower* is set to the current User. The *book* attribute is set to the chosen book. *date_borrowed* is set to the current date, and the loan *status* is set to "ACTIVE". Then the *repo_new* function is called to create a new Loan object using the template. The OID of the new object is stored in the variable *$loan_id.*

Next, *Loan_make_finish* updates the availability of the book being borrowed. First, the book is retrieved using the *repo_get* function. Then the *availability* attribute is set to "ON LOAN". Finally, *repo_update* is invoked, which commits the changes to the database.

Figure 31 is a screenshot of the Create Object Wirmlet using the customized *Loan_make* after the user submits a request to borrow a book.

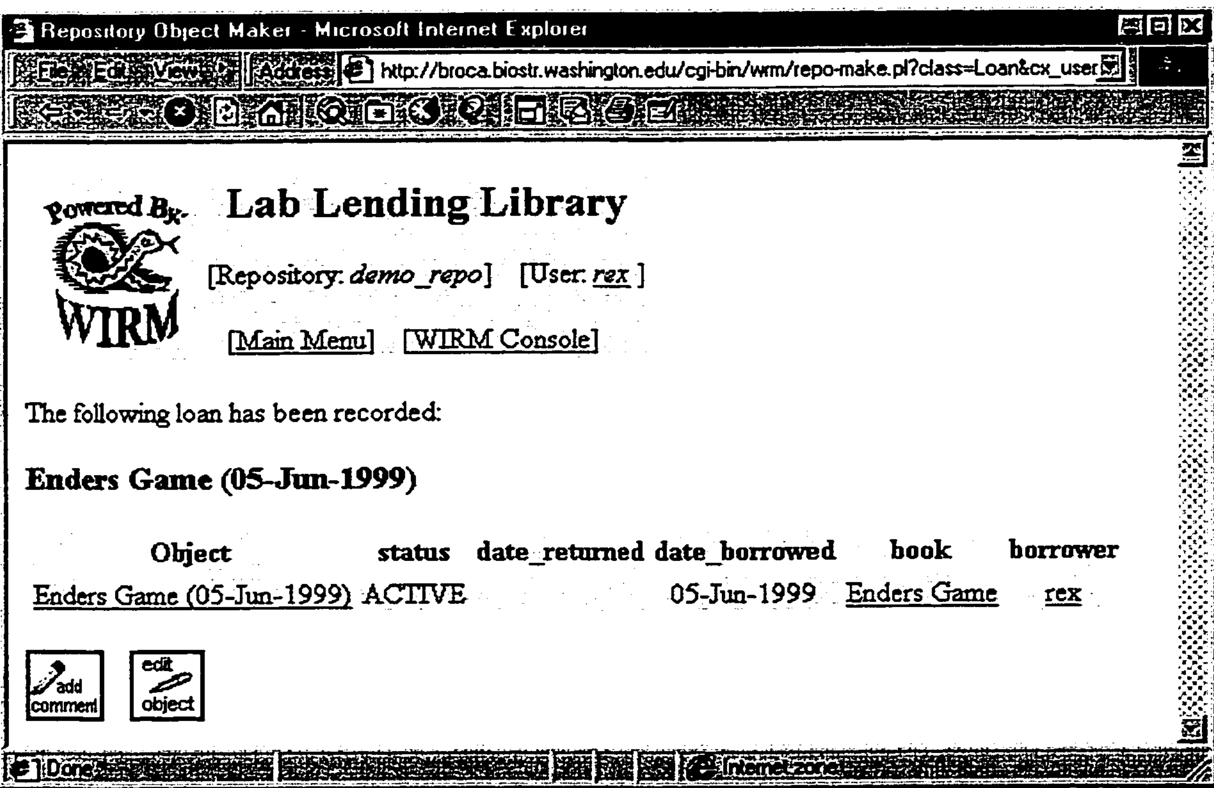


Figure 31: Result of Making a Loan

6.3 Developing Custom Wirmlets

Whereas the previous customizations were all made to the class definition file, defining a custom Wirmlet involves creating a new Perl script that resides in the repository-specific cgi-bin directory. All repository systems begin with two files in that directory: **main-menu.pl** and **generic-wirmlet.pl.**

The Main Menu is intended to be a domain-specific starting point for using the information system, in contrast to the **WIRM Console**, which provides access to general-purpose repository functions. The Wirmlet **main-menu.pl** is executed whenever the user clicks on the **[Main Menu]** link in the banner. By custom-tailoring the **[Main Menu]**, the developer can create a friendly interface that doesn't require end-users to work with the **[WIRM Console]**. For example, the Main Menu can have a link called **[Borrow Book]** which activates the *Object Maker* Wirmlet on the Loan class directly, rather than requiring the user to choose **[Create Object]** from the WIRM Console and then select the Loan class as a separate step. Figure 32 is a screenshot of the generic Main Menu Wirmlet, as provided with the system. The code for that Wirmlet is:

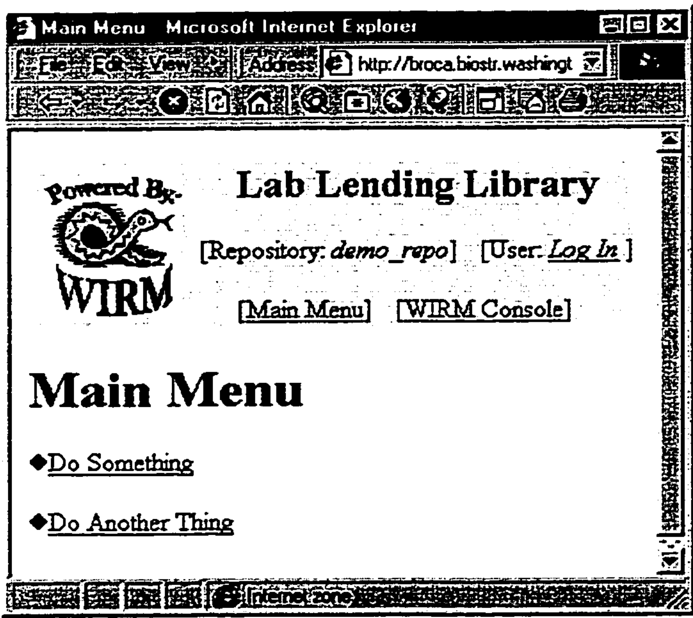


Figure 32: Default Main Menu

```
$items = [
        "Do Something", "wirmlet1.pl";
        "Do Another Thing", "wirmlet2.pl";
        ];

$title = "Main Menu";
print HT_form_init($title);
print repo_banner();
print h1($title);
print HT_list_diamond($items);
print HT_form_end();
```

To customize this Wirmlet, the developer may update the line where the $title is assigned, to something more descriptive:

```
$title = "Lab Library Front Desk";
```

The list of items may be changed to include specific actions available to the user, with parameterized links to the System Wirmlets, as follows:

```
$items = [
        "View Books", repo_explorer_link("Book"),
        "View Loans", repo_explorer_link("Loan"),
        "Register a New Book", repo_mlink("Book"),
        "Upload a Cover", upload-cover.pl,
        "Borrow a Book", repo_mlink("Loan"),
        "Return a Book", "return-book.pl",
];
```

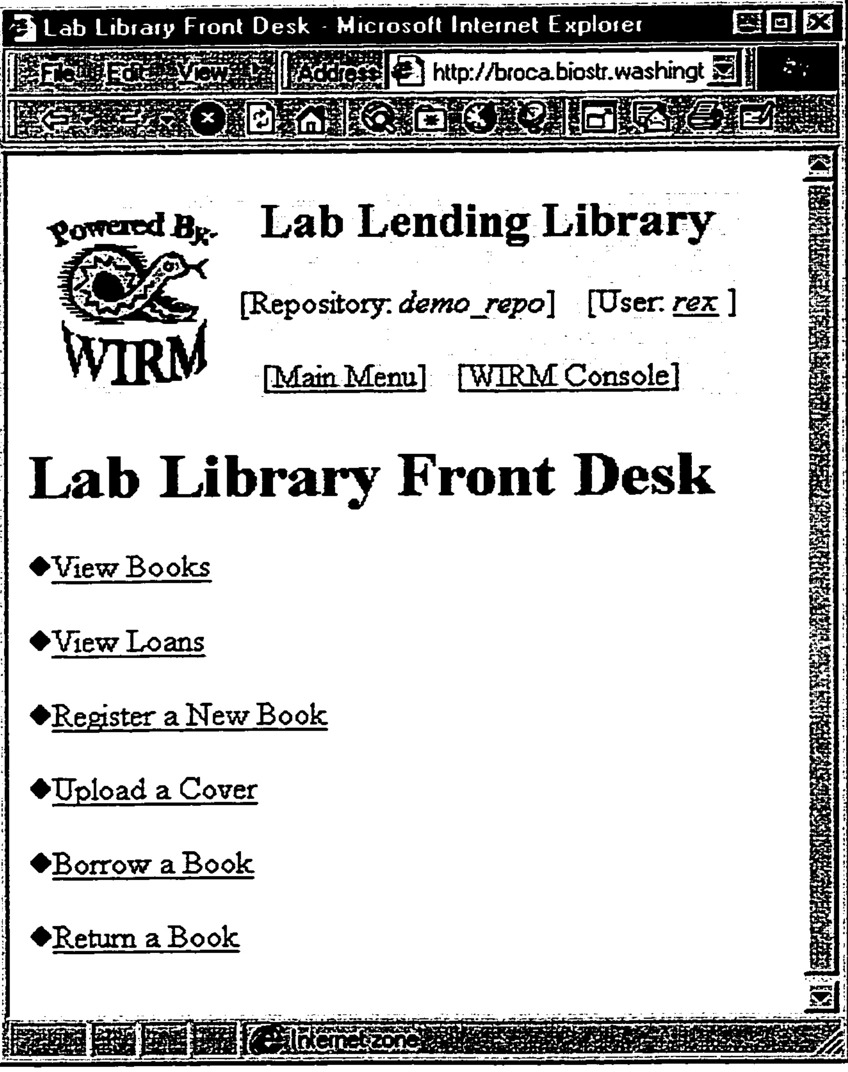


Figure 33: Customized Main Menu

For viewing books or loans, the *repo_explorer_link* function is used, which generates a URL to the Explorer Wirmlet, using the given class type. For registering a new book or borrowing a book, the *repo_mlink* function is used, which generates a URL to the Object Maker Wirmlet. The other two choices (*"Upload a Cover"* and *"Return a Book"*) point to custom Wirmlets that will be defined later. Figure 33 shows the updated main menu.

WIRM provides a template for creating domain-specific WIRMLETS called **generic-wirmplet.pl**. The template provides the general structure of a Wirmlet:

1. Initialize the HTML form, and display banner and title.
2. If no submit button has been pressed, display a prompt.
3. Otherwise, process the results.

Some Wirmlets have more a complex structure, such a multiple interactive processing stages, but they all follow this general format. The generic Wirmlet template code is shown below:

```
$title = "Generic Wirmlet";
print HT_form_init($title);
print repo_banner();
print h1($title);

if (!param("Submit")) {
    print show_prompt();
} else {
    print process_request();
}
print HT_form_end();

sub show_prompt {
    my($out);
    $out .= submit("Submit");
    return $out;
```

```
}

sub process_request {
    my($out);
    $out .= "request processed!";
    return $out;
}
```

Some operations don't fit well in the framework of the built-in Wirmlets. For example, returning a borrowed book. In these cases, the developer may design their own custom

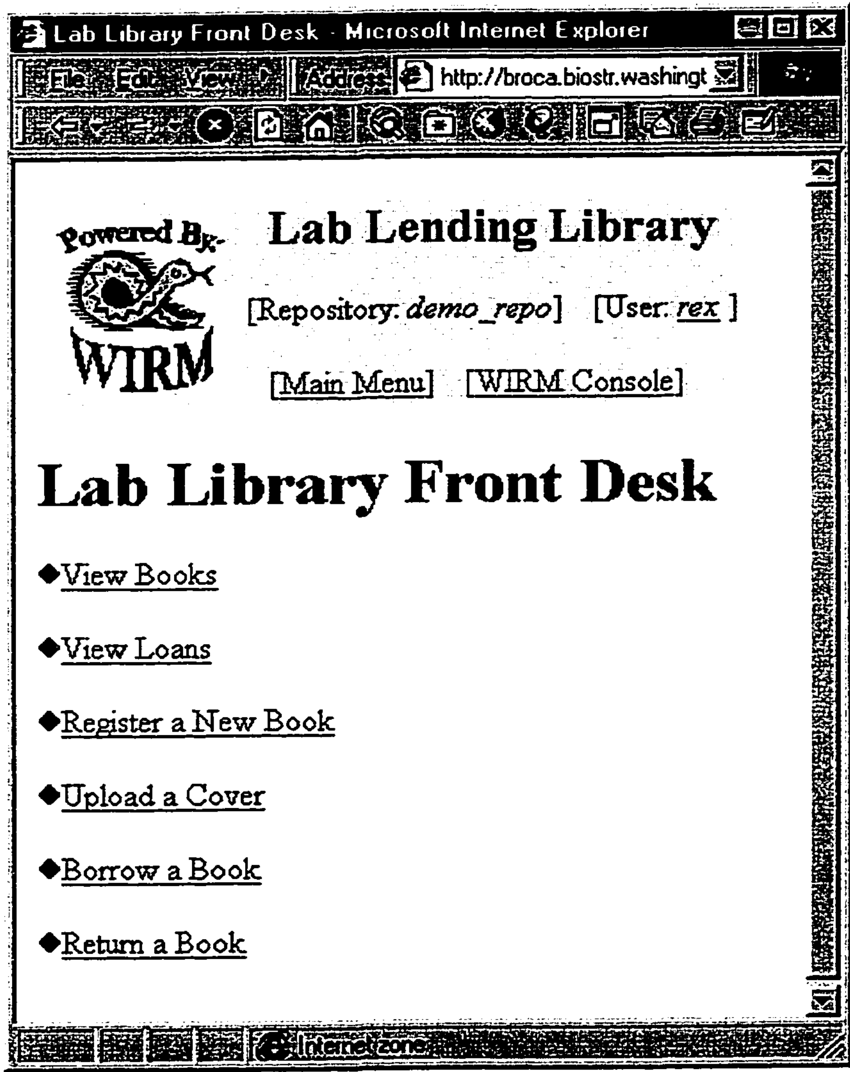


Figure 34: Customized Main Menu

Wirmlets by editing the generic Wirmlet.

The developer should change the title to be more descriptive than *"Generic Wirmlet", e.g.:*

```
$title = "Return A Book";
```

Then the developer should edit the *show_prompt* function. First, it should verify that the current User is logged in. Then, it should prompt for which Loan is being returned.

```
sub show_prompt {
    my($out, $loans);

    if (!$CONTEXT{user}) {
        $out .= "You must first "
             . HT_href("log in", repo_login_link());
        return $out;
    }

    $loans = repo_query("Loan",
              "status = 'ACTIVE' and borrower = $CONTEXT{user}");

    if (!$loans) {
        $out .= "User " . User_view_label($CONTEXT{user})
             . " has no outstanding loans!";
    } else {
        $out .= em("Select loan being returned:") . HT_space();
        $out .= repo_choice($loans, "loan") . p();
        $out .= submit("Submit");
    }

    return $out;
}
```

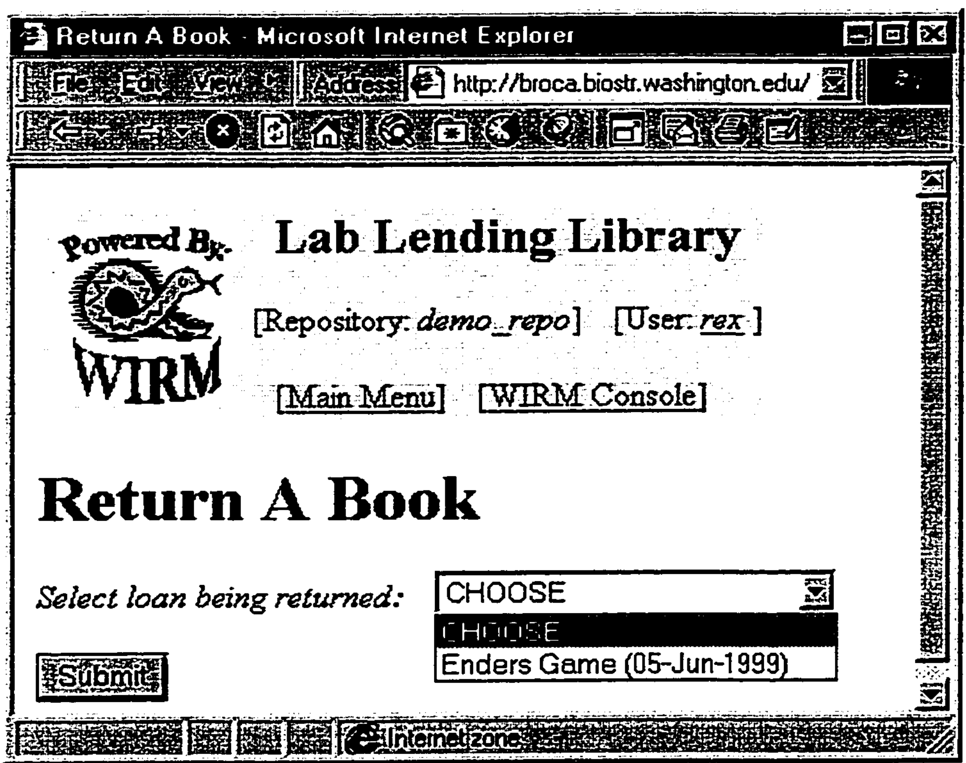


Figure 35: Prompt for Book Return

Notice that the prompt uses the *repo_choice* function to generate the popup-menu. The prompt is intelligent, limiting the possible choices to only the user's outstanding loans. This is done by filtering the *repo_query* function. The prompt also handles the special case where *repo_query* returns no results. Figure 34 depicts a screenshot of the **return-book.pl** Wirmlet prompt, demonstrating that the popup menu is limited to outstanding loans of the current user.

Next, the developer should edit the *process_request* function. It should update the return date and status of the specified Loan, and update the availability of the book.

```
sub process_request {
    my($out, $loan, $book);

    $loan = repo_get(param("loan"));
```

```
    $loan->{date_returned} = db_date_now();
    $loan->{status} = "CLOSED";
    repo_update($loan);

    $book = repo_get($loan->{book});
    $book->{availability} = "IN LAB";
    repo_update($book);

    print "The following Loan has been updated: <P>";
    print repo_view(param("loan"));

    return $out;
}
```

Figure 35 shows the result of returning a book.

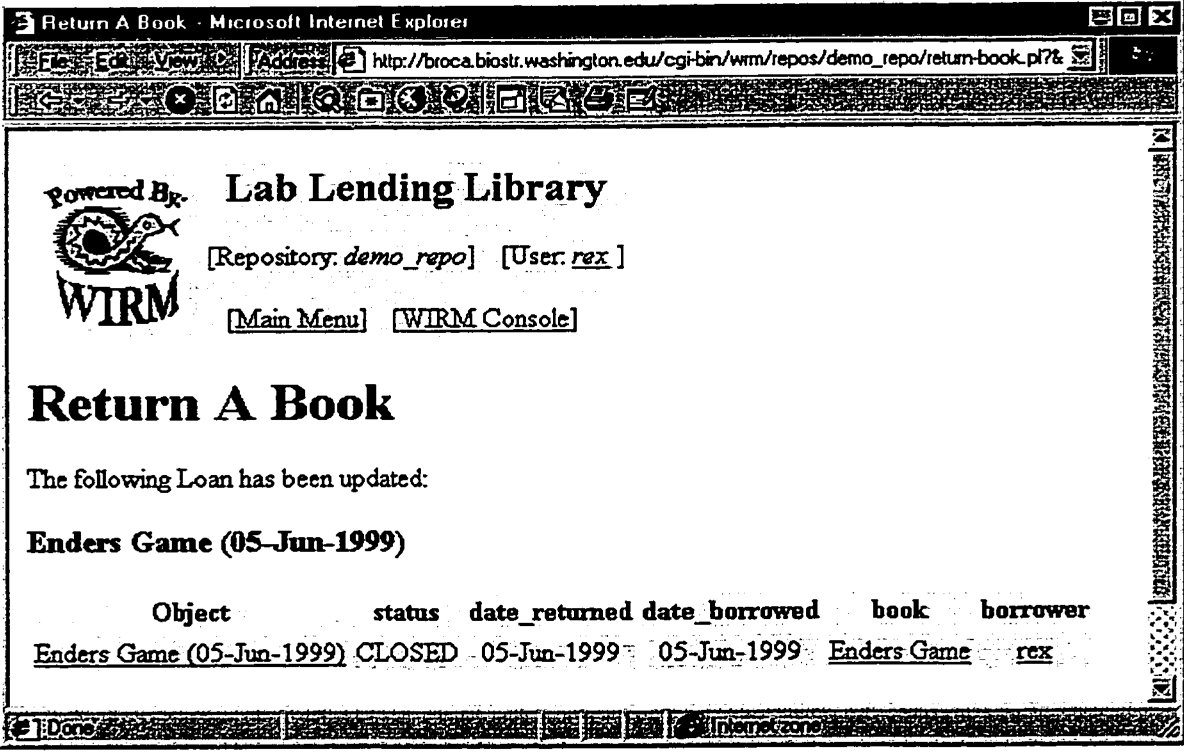


Figure 36: Book Return Results

A common bug is to forget to swizzle an OID. For example, in the above function, if *param("loan")* had been assigned to the *$loan* variable without applying *repo_get*, one would have encountered an error when trying to update the object's attributes, since it would be merely an OID (integer) instead of a reference to a repository object.

The next Wirmlet accessible from the Main Men is the Upload-Cover Wirmlet. Unlike previous Wirmlets, which were limited to two phases, the Upload Cover will consist of three phases: prompting for a book, prompting for a cover, and uploading the cover. The main body of the Wirmlet should test for three possible stages:

```
if (!param("book")) {
    print prompt_for_book();
} elsif (!param("cover")) {
    print prompt_for_cover();
} else {
    print process_request();
}
```

The function *prompt_for_book* should generate a popup menu of books to associate with the cover:

```
sub prompt_for_book {
    my($out, $books);

    $books = repo_query("Book");
    $out .= em("Select book: ") . HT_space();
    $out .= repo_choice($books, "book") . p();
    $out .= submit("Continue");

    return $out;
}
```

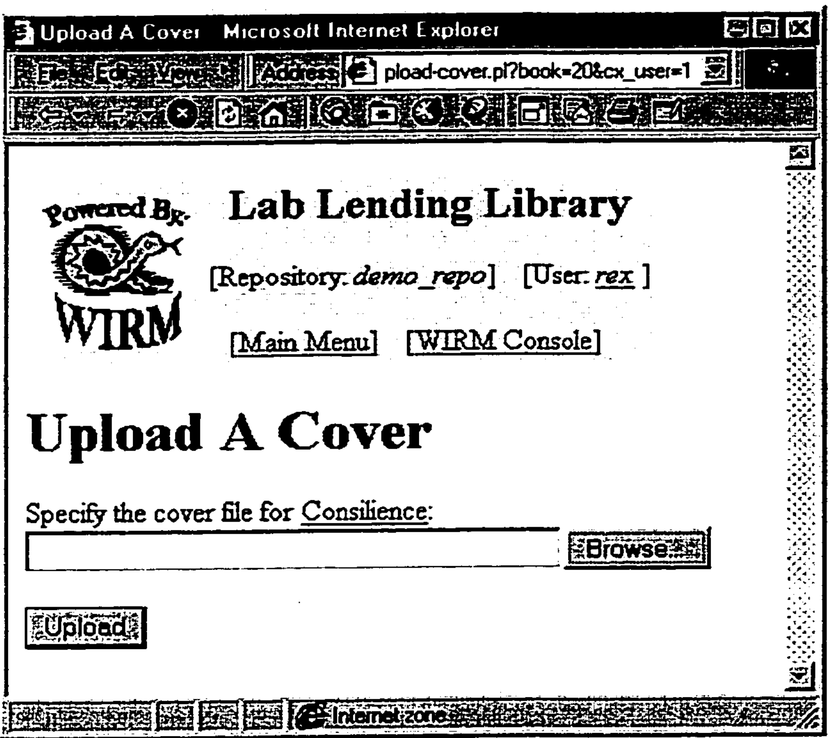


Figure 37: Upload Cover Prompt

The function *prompt_for_cover*, should use the *filefield* function to create a file upload field. This will automatically include a "browse" button that allows the user to navigate their local file system for the desired file. When the submit button is pressed, the selected file will be uploaded over the Internet to the Web server, and a filehandle to the file data will be returned as the "cover" parameter:

```
sub prompt_for_cover {
    my($out);

    $out .= "Specify the cover file for ";
    $out .= repo_view_label(param("book")) . ": <BR>";
    $out .= filefield(-name=>"cover", -size=>40);
    $out .= p() . submit("Upload");
    $out .= hidden("book", param("book")); # carry book parameter

    return $out;
```

```
}
```

Notice that the *book* value is carried as a hidden parameter. This allows the Wirmlet to remember the book for the third phase, *process_request.* The cover upload prompt is shown in Figure 36.

In the third phase, a File Descriptor Template is created and passed along with the file handle to *repo_import_file_from_handle*, which copies the file into the FSA and registers it. The resulting file OID is assigned to the cover attribute of the book object, and the book record is updated in the database:

```
sub process_request {
    my($out, $fh, $book, $fdt, $file_id);

    $book = repo_get(param("book"));
    $fh = param("cover");

    # create File Descriptor Template for importing cover image
    $fdt->{source} = "$fh";
    # translate backslashes to frontslashes
    $fdt->{source} =~ tr|\\|\/|;
    $fdt->{label} = "cover of $book->{title}";
    $fdt->{context} = "Lending Library: imported via cover-upload.pl";

    # import the image, which creates a new File object
    $file_id = repo_import_file_from_handle($fh, $fdt);
    if (!$file_id) {
        $out .= "Couldn't import file. <P>";
    } else {
        $book->{cover} = $file_id;
        repo_update($book);
        $out .= b("Cover uploaded into repository.");
        $out .= repo_view($book);
    }
```

```
    return $out;
}
```

6.4 Evolving Schemas and Wirmlets

Once the EMS has domain-specific schemas, custom class definitions, and a set of custom Wirmlets to handle the Workflow, the system is ready to support real users. Inevitably, system will become out of date as users request new features in the interface, or the experiment itself evolves. At this stage, the WIRM's ability to support dynamic schema evolution and class modification is called upon. In the example domain, consider

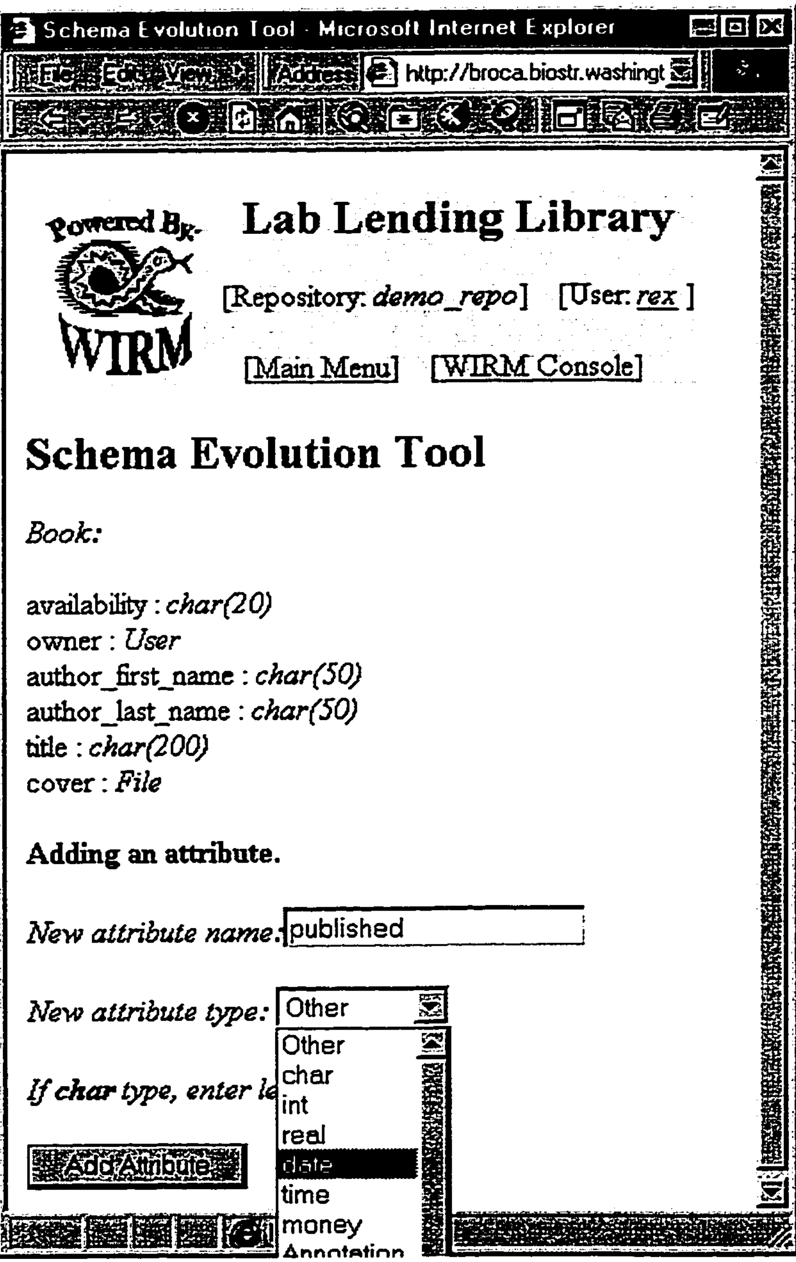


Figure 38: Adding an Attribute

the problem of adding a "date_published" attribute to the Book schema. This is easily accomplished with the built-in Schema Evolution Wirmlet, as shown in Figure 37.

Another common process as the EMS matures is to customize the interface for multiple user classes. In the example domain, the Main Menu presents a single view for all users. It would be better to hide certain operations (such as borrowing a book) from non-registered users. This is easy to accomplish by performing a test on the user context, and making the main menu context sensitive:

```
$guest_items = [
        "View Books", repo_explorer_link("Book"),
        "View Loans", repo_explorer_link("Loan"),
];

$member_items = [
  @$guest_items,
        "Register a New Book", "register-book.pl",
        "Upload a Cover", "upload-cover.pl",
        "Borrow a Book", repo_mlink("Loan"),
        "Return a Book", "return-book.pl",
];

$title = "Lab Library Front Desk";
print HT_form_init($title);
print repo_banner();
print h1($title);

if ($CONTEXT{user}) {
   print b("Hello, " . HT_textify(User_view_label($CONTEXT{user})) . "! ");
   print "Please make your selection from the choices below.";
   print p(), HT_list_diamond($member_items);
} else {
   print "For full access to Library resources, please "
      . HT_href("log in.", repo_login_link());
```

```
    print p(), "Guest access: ";
    print p(), HT_list_diamond($guest_items);
}

print HT_form_end();
```

The context sensitive menu is shown in Figure 38, with the guest user's view in the background and the registered user's view in the foreground.

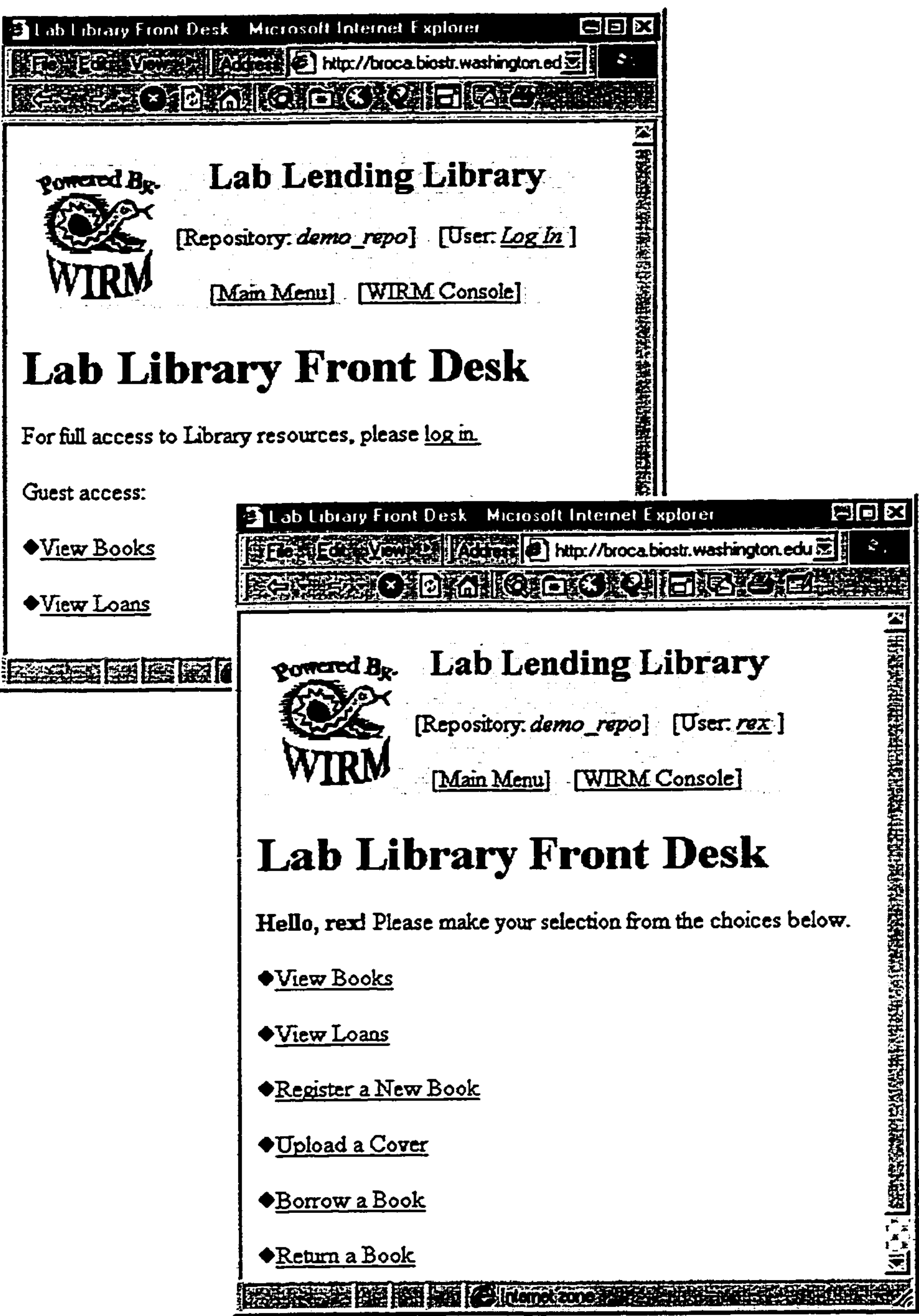


Figure 39: Context-Sensitive Main Menu

Chapter 7: Results & Experience

The WIRM has been used to implement the Brain Mapper Experiment Management System, including graphical data acquisition forms, scripts for importing existing file-based legacy data, a patient-centered browser for viewing photographs and renderings, a generic gateway for posing ad-hoc queries across all data types, and an interface for connecting to a remote image server to upload MRI slices. The EMS currently maintains multimedia data for thirty four patients, including demographics and surgery information, MRI slices, radiology exam parameters, digitized intra-operative photographs, 3-D surface and volume renderings, stimulation studies, and spatial mapping data from multiple authors.

In this chapter, the following topics will be discussed:

- Overview of the Brain Mapper experiment
- History of the EMS development
- The Brain Mapper Schema
- The Brain Mapper Wirmlets
- Evaluation of benefits of WIRM

7.1 Overview of the Brain Mapper Experiment

The University of Washington Human Brain Project is a multidisciplinary research project aimed at managing cortical stimulation data obtained during surgery. Neuroscience research has shown that language function is localized mostly in the left temporal area of the brain [Oje89]. Surgical stimulation studies have shown that the

specific locations of language sites are highly variable between patients. The goal of the Brain Mapping experiment is to explore correlations between the locations of language sites and patient demographics such as age, sex, and verbal IQ [Mod97b].

After the brain surface has been exposed, the patient is awakened and shown pictures of common objects. He or she is asked to name the objects while an electrical current is being applied to various regions of the exposed cortex. If the patient is unable to name the object while the current is applied, the site is considered essential for language. The naming task is recorded on audiotape. The locations of the stimulation sites are visually marked with paper labels, and the exposed cortex is photographed. The photograph is digitized and entered into an image database. Electro-encephalogram readings and other information are also recorded during the stimulation tasks.

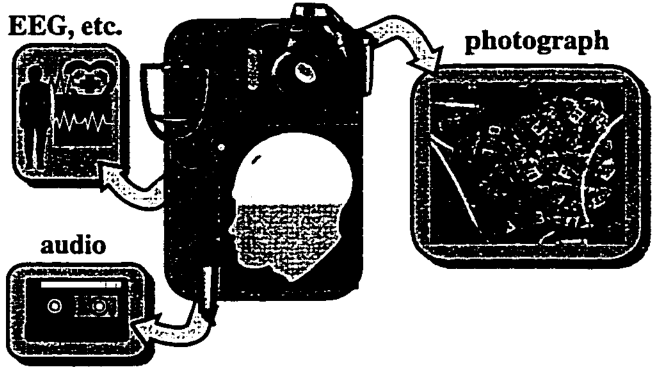


Figure 40: Multimedia Patient Data

Prior to surgery, the patient is given an MRI exam with multiple series optimized for veins, arteries, and cortex anatomy. The series are saved on disk and made available on a network-accessible image server. The MR slices are downloaded into the Structural Informatics graphics-processing server, where they are reconstructed into 3D models by computer scientists using in-house visualization software [Mod97a]. Detailed 2D images are generated from the reconstructions, which are stored in a Web-accessible image database and viewed by a wide range of scientists.

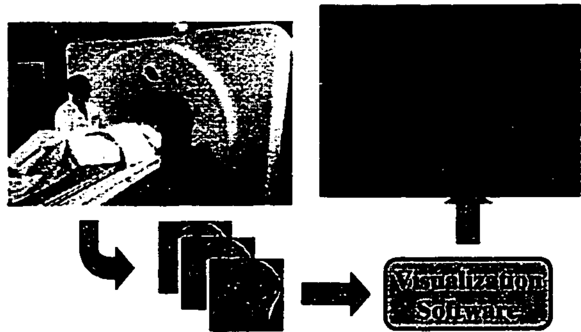


Figure 41: Creating a 3D Model

The 3D models and the digitized photograph are rendered side by side in an interactive application called the Brain Mapper, which allows a user to mark the locations on the 3D model that correspond to the labeled sites in the photograph. This allows the 3D coordinates of the sites to be registered in a database, which can then be used to compare essential locations across patients.

As evident from the above discussion, the Brain Mapper experiment involves a wide range of data types: patient demographics, digitized photographs, audio files, EEG readings, stimulation site data, sequences of MR images, 3D models, renderings, and mapping data. These data are stored in a variety of heterogeneous formats and have complex relationships at multiple granularity. The data are acquired from various physical locations (the operating room, the radiology lab, the computer science lab, etc.) and require the interaction of participants from various departments (radiologists, neurosurgeons, computer scientists, technicians, etc.). Furthermore, the data are produced and transformed by a complex network of uncoordinated software applications (network servers, visualization software, mapping programs, etc.).

7.2 History of the EMS Development

At first, the experiment was managed by hand without any supporting framework. The data were collected at the biological structure laboratory by an informal process, and file transfers were coordinated by email and word of mouth. Files were stored in an ad-hoc manner across a range of machines, and no formal directory existed. Much of the metadata about files was implicit in their pathnames or described in unstructured README files, and data retrieval often meant browsing through cluttered directories. When a complex workflow operation was to be performed (such as performing a mapping task) the files were collected by hand and copied into a working directory, and a custom script was created to run the necessary software.

It soon became obvious that this method would not scale well, and the increasing number of patients demanded a consistent organization for storing files, a structured metadata directory, and a protocol for handling the workflow. A significant effort was made in designing a directory structure and formalizing the metadata requirements. Files were moved from their multiple servers on to a single file system, and each patient was given a single directory in which to store all their data. A consistent naming convention was adopted (e.g. every patient directory was named by the patient's initials and research number), and a central file was created for keeping track of the workflow.

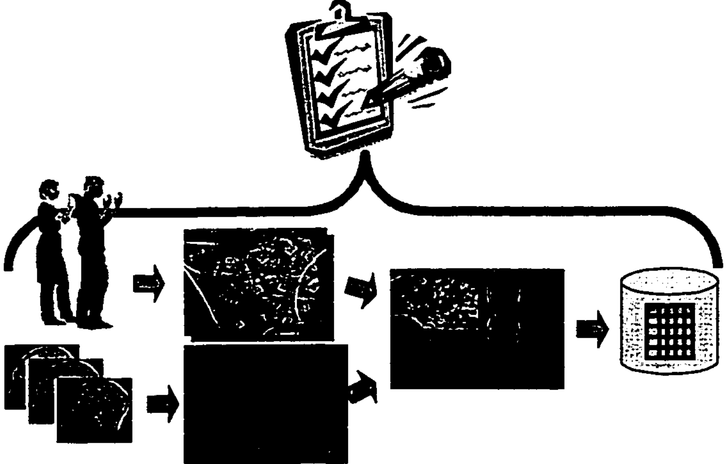


Figure 42: Workflow Management

This method reduced the problems of trying to locate data, but it still required an immense amount of maintenance by hand, and was subject to errors on repetitive tasks (such as copying long file pathnames into scripts). Furthermore, it was impossible to compare data across patients or make queries such as "retrieve all the digitized photographs" without going into every directory and copying out the file. More importantly, to retrieve any data the user had to understand the file hierarchy and naming conventions, and had to have access to the file system and understand how to surf directories in the Unix command line. In effect, the data was inaccessible to all but the computer scientists working in the Biostructure lab. We realized the need for a content delivery system, scripts to automate some of the workflow stages, and a database to store information about patients and files.

Each requirement was handled separately. Perl scripts were written to automate some of the workflow stages. A simple relational database was created for storing patient information and metadata. Images were converted into JPEG's and placed in directories where they could be viewed over the Web, where they could be easily viewed by collaborators in the experiment. Gradually, the Web interface became more comprehensive, so that some of the scripts could be launched from a CGI script, and some of the database tables were connected to the Web site. Perl took on a more central role in the entire process, as there were Perl modules available for connecting databases to the Web (Msql-Perl Adaptor) and processing Web forms (CGI.pm). Eventually, the system evolved so that all file access was performed through a query interface. The Web-database connectivity solution is shown below:

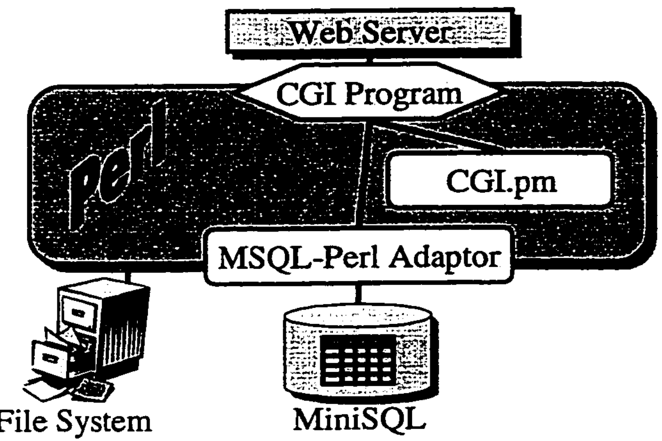


Figure 43: Web-Database Connectivity Solution

The Perl Web-database connectivity solution was serving as a repository manager. Our in-house solution made a lot of leverage off of existing free software components, allowed us to work in our own familiar programming environments, was independent of any specific hardware platforms, and was custom-tailored to our own experiment management requirements. The drawbacks of this system, as compared with a commercial repository manager, were that it offered no built-in file metadata support, used a primitive data model which didn't support user-defined types, and required the interface designer to explicitly work with the table structure of the relational database. Furthermore, there was no separation between the supporting framework and the EMS itself.

Gradually, I began to generalize the tools that were used in the various CGI scripts that made up the EMS, so they could be reused for building other parts of the EMS. I began to realize that the tools were a valuable commodity that could be used for other unrelated experiment management systems. The development of the WIRM API's proceeded in parallel with the implementation of its driving application. Wherever appropriate, I generalized solutions from the Brain Mapper to be part of the WIRM system. For

example, the Brain Mapper needed to display an HTML table of thumbnail images which could be clicked to view the full-sized images, and this function was added to the Web API, where it could be used in other applications.

Eventually, WIRM was packaged for distribution, and made available as a general purpose Web application server. A reference manual was written describing every API in

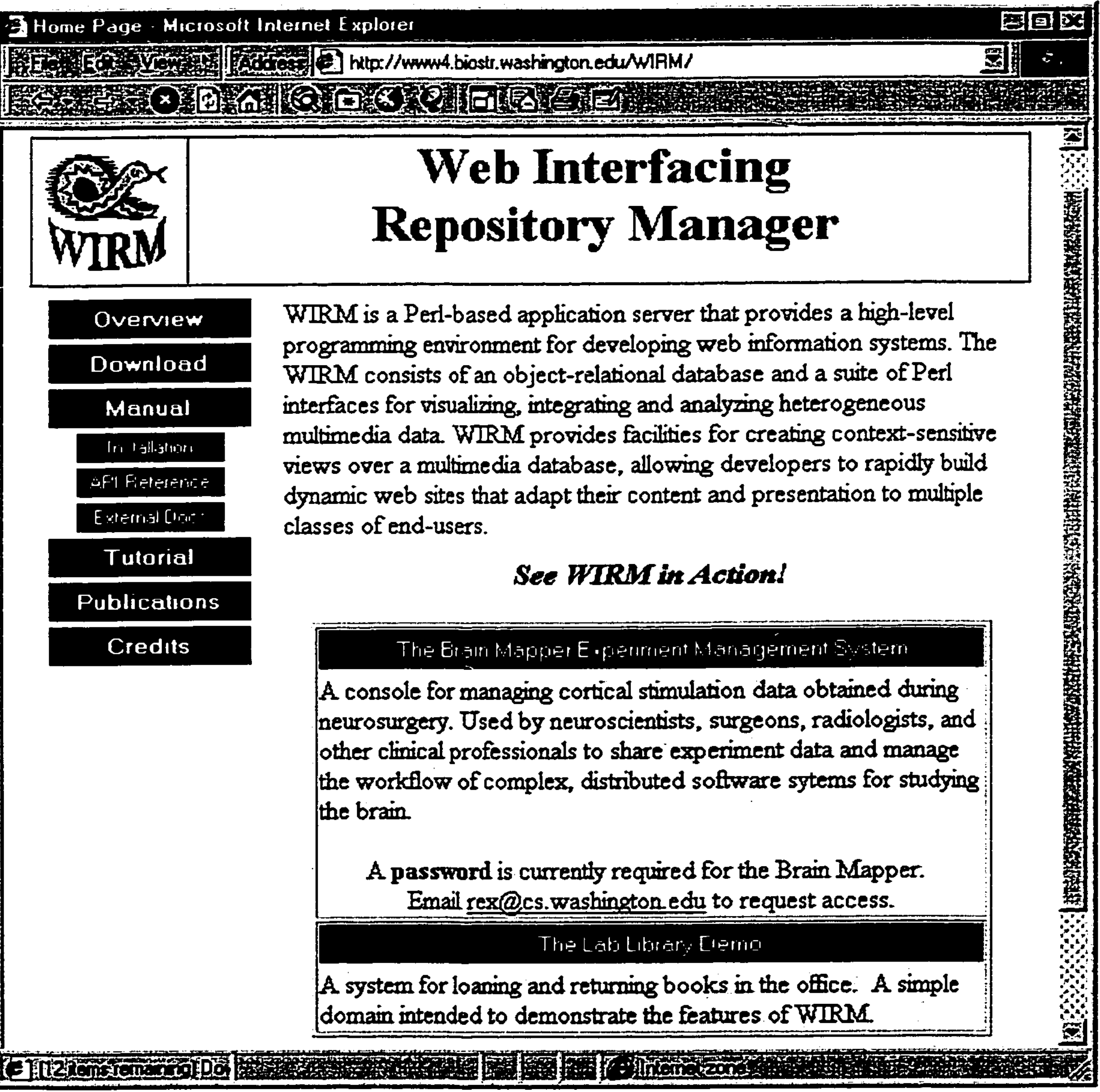


Figure 44: The WIRM Web Site

detail. A tutorial was developed and made available online. The WIRM code was ported to linux and tested on a variety of machines. An installation guide was written and tested, and the files were made available on a Web site under an open-source licensing agreement [Wir-URL]. Figure 43 shows the WIRM Web site.

7.3 The Brain Mapper Schema

The complexity of the Brain Mapper experiment data was well handled by the WIRM data model. I chose a patient-centric approach, in which the Patient schema was the root object, and all other objects belonged to a Patient object. There are currently twelve schemas, as depicted in Figure 44. The solid links represent the "part-of" relationship

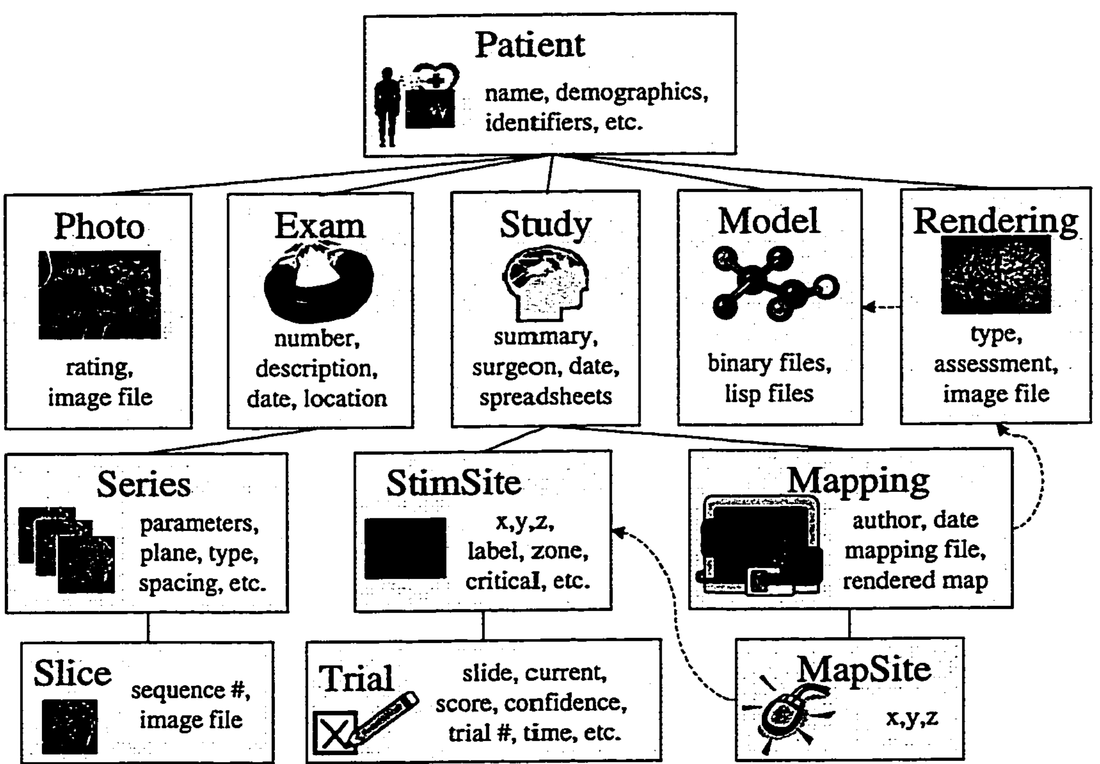


Figure 45: Brain Mapper Class Hierarchy

(e.g. a Study is *part-of* a Patient), which is implemented as an attribute of the child object (e.g. the Study schema includes a reference to a Patient object). The *part-of* links are many-to-one, e.g., a Patient may have multiple Photos. The dashed links represent secondary relationships, such as *refers-to*.

The hierarchical nature of the Brain Mapper classes directly results in a hierarchical navigation system, as each schema has a corresponding View Class that allows navigation to the sub-parts. The rest of this section now discusses each class in detail.

The Patient schema includes the name, age, sex, verbal IQ, and a research number assigned by the neurosurgeon. It also includes a repository patient number, as well as other internal information, such as the date the patient was registered with the repository, the location of the patient data files, and billing information.

Each Patient may have any number of Photos, Exams, Studies, Models, and Renderings. These sub-parts refer to the Patient in their schemas, but the Patient Schema doesn't refer to them explicitly, which would be redundant. However, they are accessible from the Patient view, as shown in Figure 45.

To illustrate Query-By-Context in action, the Patient View is shown twice: on the left as visualized for Privileged users (system designers and data producers), and on the right as visualized by Public users. Notice that the privileged view shows the patient's full name (altered for demo purposes), while the public view only shows the research number. A third user class, the Collaborator, is shown only the patient initials. This is implemented

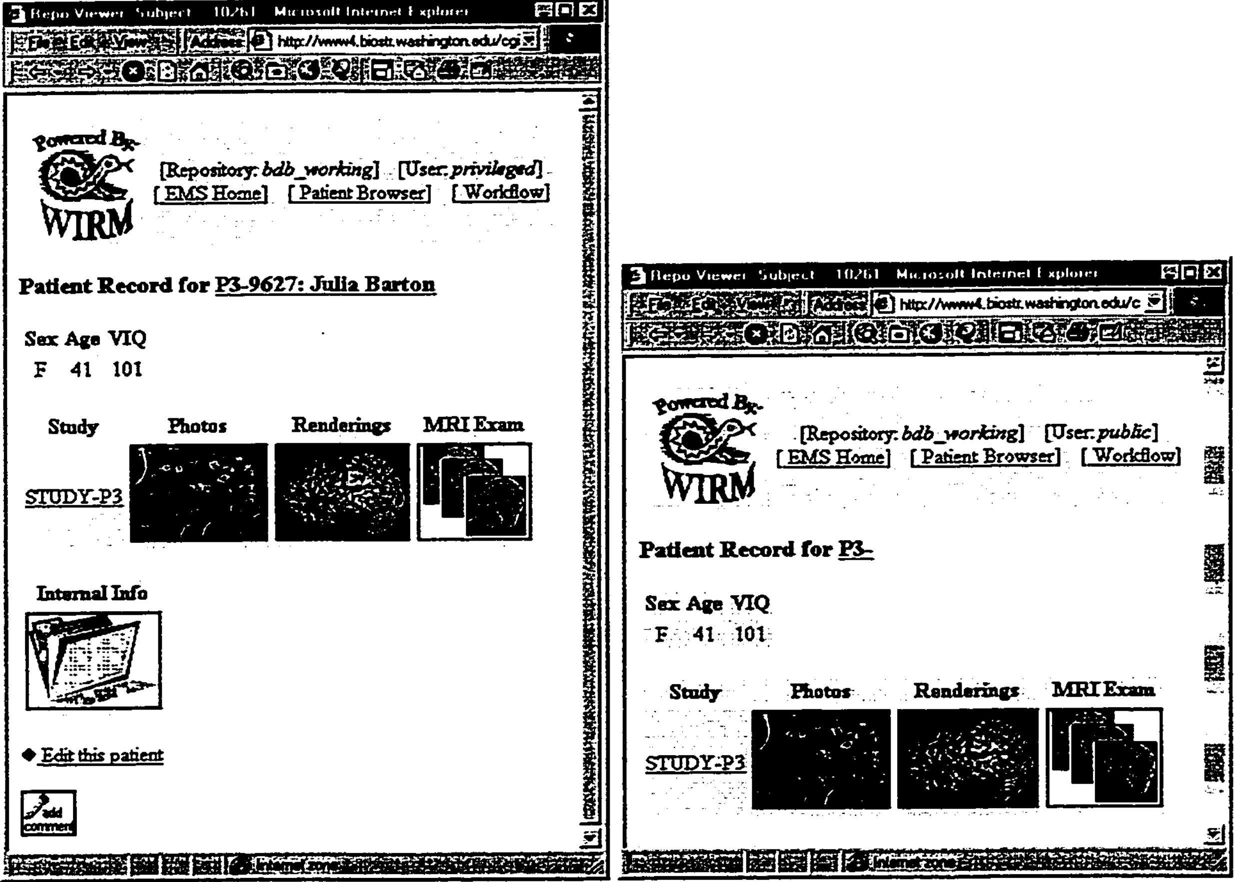


Figure 46: Two Views of a Patient

in the `Patient_view_label` method, using the User context as follows:

```
sub Patient_view_label {
    my($pat) = @_;
    if ($CONTEXT{user} eq 'privileged')
    {
        $lab = "P$pat->{pnum}-$pat->{research_num}:
                $pat->{first_name} $pat->{last_name}";
    }
    elsif ($CONTEXT{user} eq 'collaborator')
    {
        $lab = "P$pat->{pnum}-$pat{research_num}: $pat->{initials}";
    }
    else { # default = public
        $lab = "P$pat->{pnum}-$pat{research_num}";
    }
    return HT_href($lab, repo_vlink($pat));
}
```

By encapsulating the patient name in the Label function, the Query-By-Context affects every view that includes a Patient label, so a single change can affect thousands of possible views. Notice other differences between the Patient views: the public version does not include a link to the Internal Info, nor does it include the options of editing the patient or adding comments. In this way, access to an entire ViewClass is regulated, and workflow security is enforced.

When a Patient View is generated, the constituent sub-parts are retrieved via separate queries to the database, and iconic links are generated. For example, to generate the thumbnail of the Photo in the Patient View, a function `All_Photos_View_Icon` is called with the Patient's OID as a parameter:

```
sub All_Photos_view_icon {
    my($pat) = @_;
    my($photo) = repo_query_single("Photo",
                    "patient = $r->{oid} AND preference = 1"));
    if (!$photo) {
        return "[No photos found]";
    }
    return HT_href(HT_img(repo_thumbnail($photo->{image})),
```

```
            repo_vlink($pat, "AllPhotos"));
}
```

This function looks up the preferred photo of the patient using a Boolean SQL query filter, and then creates a thumbnail of the Photo's image which acts as a hyperlink to the virtual view *AllPhotos* for this patient.

The Photo schema contains a preference rating and a reference to a File containing the digitized image. Note that the image's location and metadata are managed by the File schema rather than the Photo schema. Figure 46 compares the Photo View and the File View. Note the "Add Comment" icon in the Photo view. Every object may have arbitrary annotations assigned to them that do not constitute part of the schema. In the Brain Mapper, photos are regularly annotated with comments describing the quality of the photo.

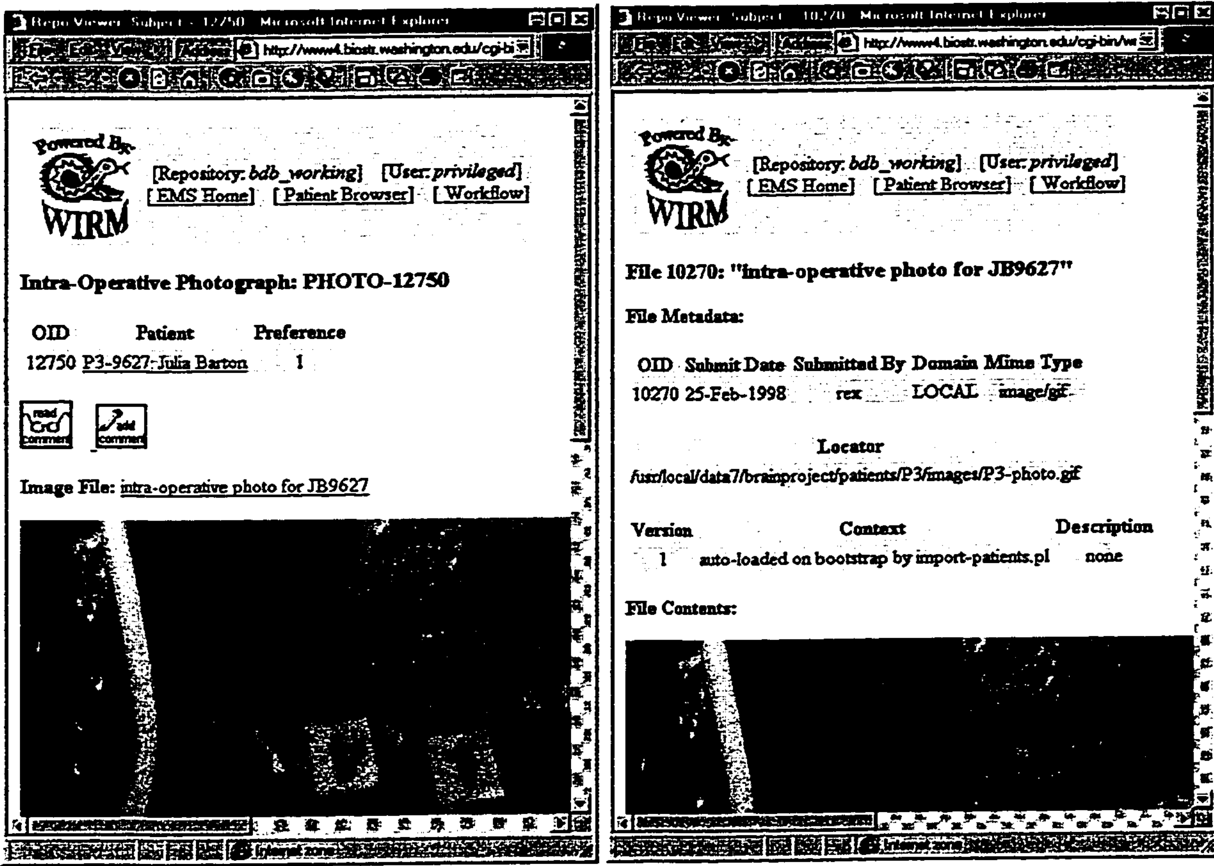


Figure 47: Photo View and File View

The Exam Schema includes the Exam Number assigned by the radiology department, the date the exam was imaged, the date it was imported into the repository, a high level description, the location of the image files, and the ID of a parameter file which was generated by the remote image server. Most of the data in the parameter file has been extracted and stored as values in the Exam and Series schemas, but the file is kept in the system for possible future reference, as suggested by the WIRM methodology. Figure 47 shows the Exam view. It contains a table of links to all the series that make up the exam. The table was created with the following code:

```
$s = repo_query("Series", "exam = $e->{oid}");
$out .= repo_view($s);
```

The first line queries the repository for all Series objects whose exam field refers to the current exam. The second line generates an HTML table using the repo_view function, which determines that the parameter is a set of Series objects and calls the Series_view_row method on each object to render the table.

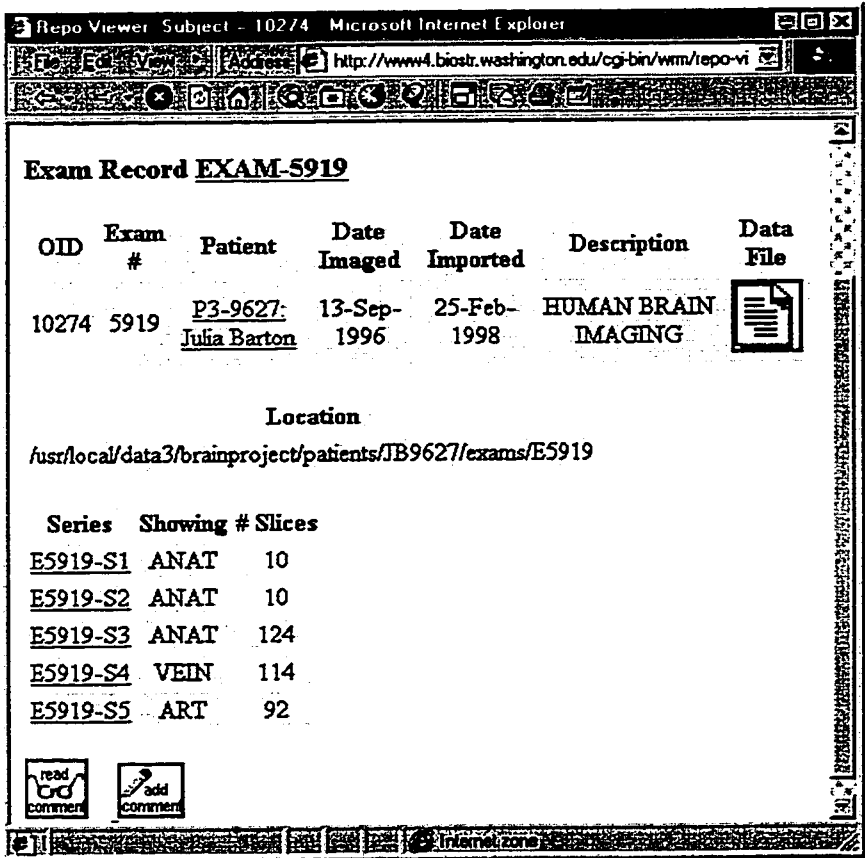


Figure 48: Exam View

Clicking on any of the Series links retrieves the full-page view for that Series, as shown in Figure 48. The Series schema consists of the Series Number, a pointer to the parent Exam, the location of the image files, a description of the type of anatomy that the series is optimized for (e.g. VEIN), the optical disk number from which the it was downloaded, and a large number of parameters extracted from the Exam data file, including the bits per pixel, field of view, dimensions, plane, scan index, spacing, and total number of images.

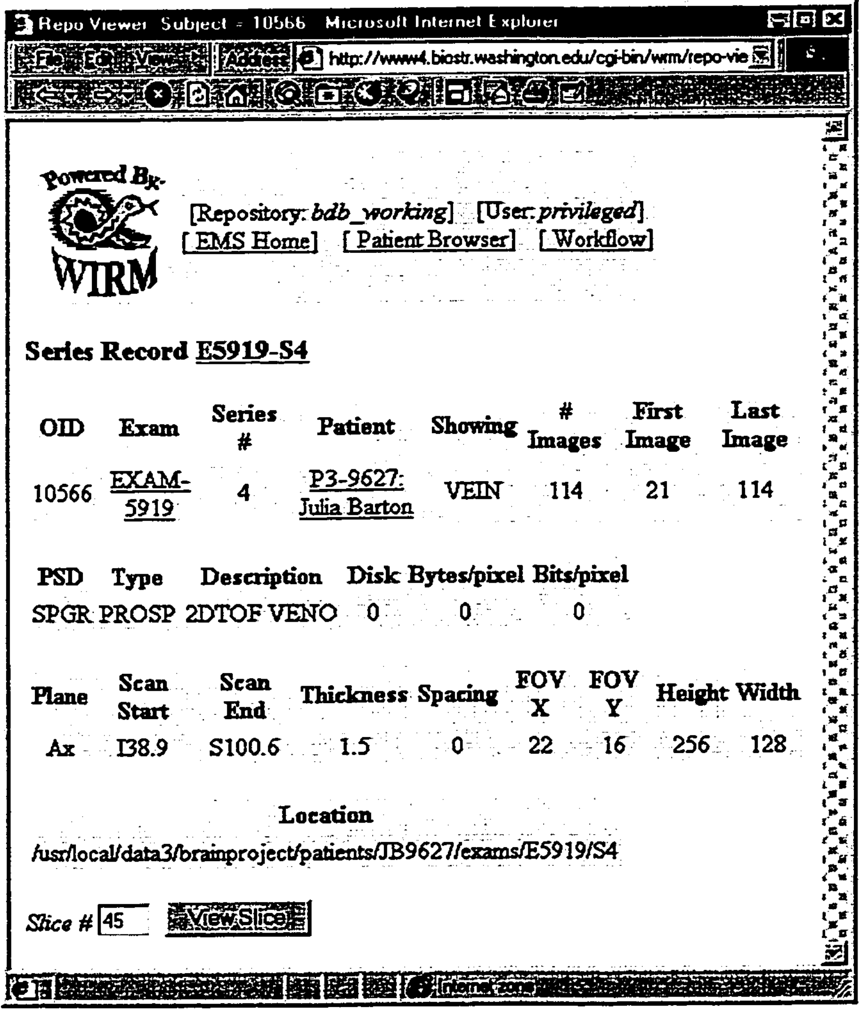


Figure 49: Series View

The Series view displays all the above information, plus a form element that allows the user to submit a slice number to view a particular slice in the series.

The following code from the top of the Series view page method is activated if a slice has been requested. It checks to determine that the slice number falls within the existing range for that series, queries the repository for the matching slice, and then replaces the Series view with the Slice view:

```
if (param('View Slice')) {
   $slice = param('slice');
   if ($slice >= $ser->{start_img} && $slice <= $ser->{stop_img})
{
        $sl = repo_query_single("Slice",
              "series = $ser->{oid} AND sequence_num = $slice");
        return Slice_view_page($sl);
   }
 }
```

The slice schema has three attributes: a pointer to the parent Series, a sequence number,

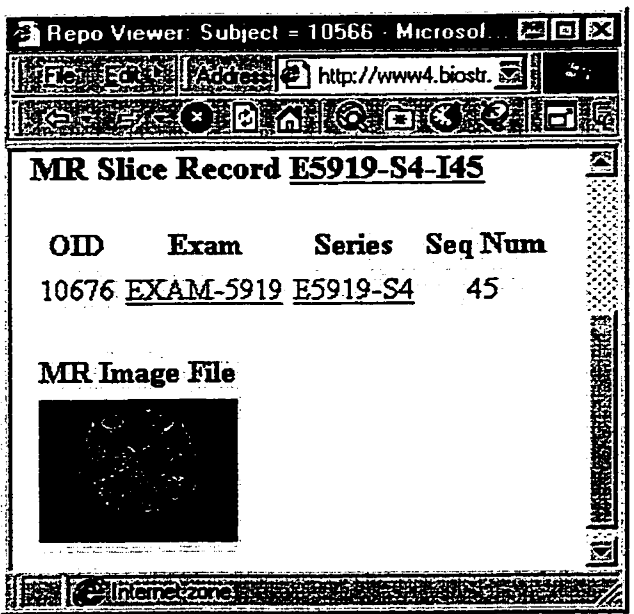


Figure 50: MR Slice View

and a reference to the slice file. The Slice View, shown in Figure 49, displays a thumbnail of the image which is generated by a special datablade for translating MR images to JPEG's in the Visualization Cache. The image can be retrieved in full by clicking on the thumbnail.

The slices are downloaded onto the repository server's file system by an interactive program that connects to the radiology department's image server. The retrieval program currently operates independently of the repository console, but I plan to build a front end to the application using WIRM's gateway API. The slices are not copied into the File Storage Area, but rather are copied into a separate directory and then registered with the repository as part of the workflow process. This method was chosen because it allows the retrieval of the files to be done independently of the repository, and does not require the volumes of images to be copied or moved. I expect that as the system continues to evolve, the acquisition process will become more tightly integrated with the repository, in which case the slices could be imported directly into the FSA when they are retrieved. This process of gradually evolving the workflow steps so that they become more integrated with the system is typical of an EMS, and parallels the feature extraction that occurs with the data objects themselves.

The slices are read as image volumes into an in-house graphics application, where they are cropped, aligned, and transformed into a 3D model. The 3D model (Figure 50) is stored as pairs of data files: a *Lisp* file and a *binary* file that contains the actual voxel data. There are three sets of model files: one for each of the vein, artery, and anatomy series. These files are registered with the repository and grouped under the Model schema, which has attributes referencing the six files.

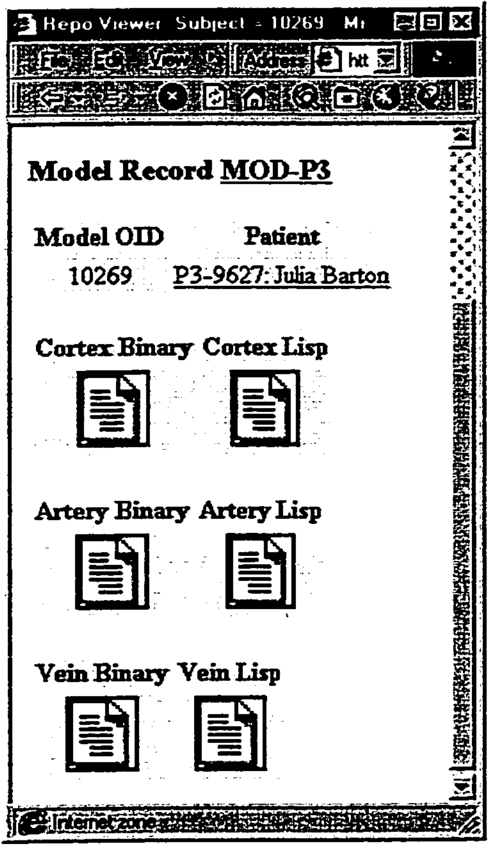


Figure 51: Model View

The Model files are used by the visualization software to generate 2D renderings, which are captured as screenshots and saved in the repository using the Rendering schema. There are various kinds of renderings: volume renderings, surface specular renderings, cutaways, rendered maps (showing the locations of stimulation sites), etc. The Rendering schema includes an attribute for describing the rendering type, a numerical rating based on a visual assessment of the rendering quality, and a preference ranking (1 is assigned to the rendering that should be used as the default view for a patient). The schema also includes a reference to the Patient object that the rendering belongs to, the Model from which the rendering was generated, and the File in which the actual image data is stored.

From the Patient view, the renderings icon takes the user to a table of all renderings belonging to this patient (see Figure 51). The Rendering's row view method includes a thumbnail image, allowing a group of renderings to be compared visually.

The stimulation data is stored as part of the Study class. The Study schema includes attributes that name the function of the study (e.g. "Language"), the surgeon who performed the study, and the surgery date. It also includes extensive summary information about the number of trials and errors in the frontal, parietal, and temporal lobes, as well as the total number of unstimulated trials and errors. Finally, it includes a

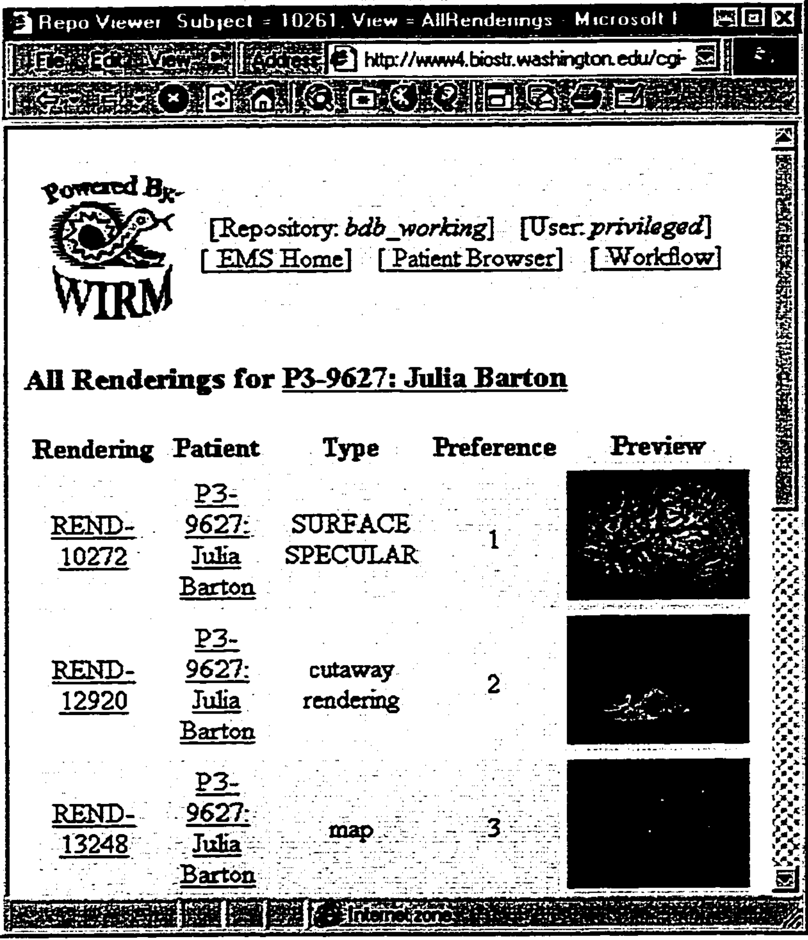


Figure 52: Multiple Renderings

reference to the preferred rendered map file for the study, as well as the spreadsheet file from which the trial data was extracted.

The Study view includes an image-map showing a screenshot of a brain rendering with the stimulation sites marked as digits overlaid on the image. When the viewer clicks on a site label, the corresponding site object is retrieved from the repository and displayed (see Figure 52). The Study view also includes a link to a specialized Summary view

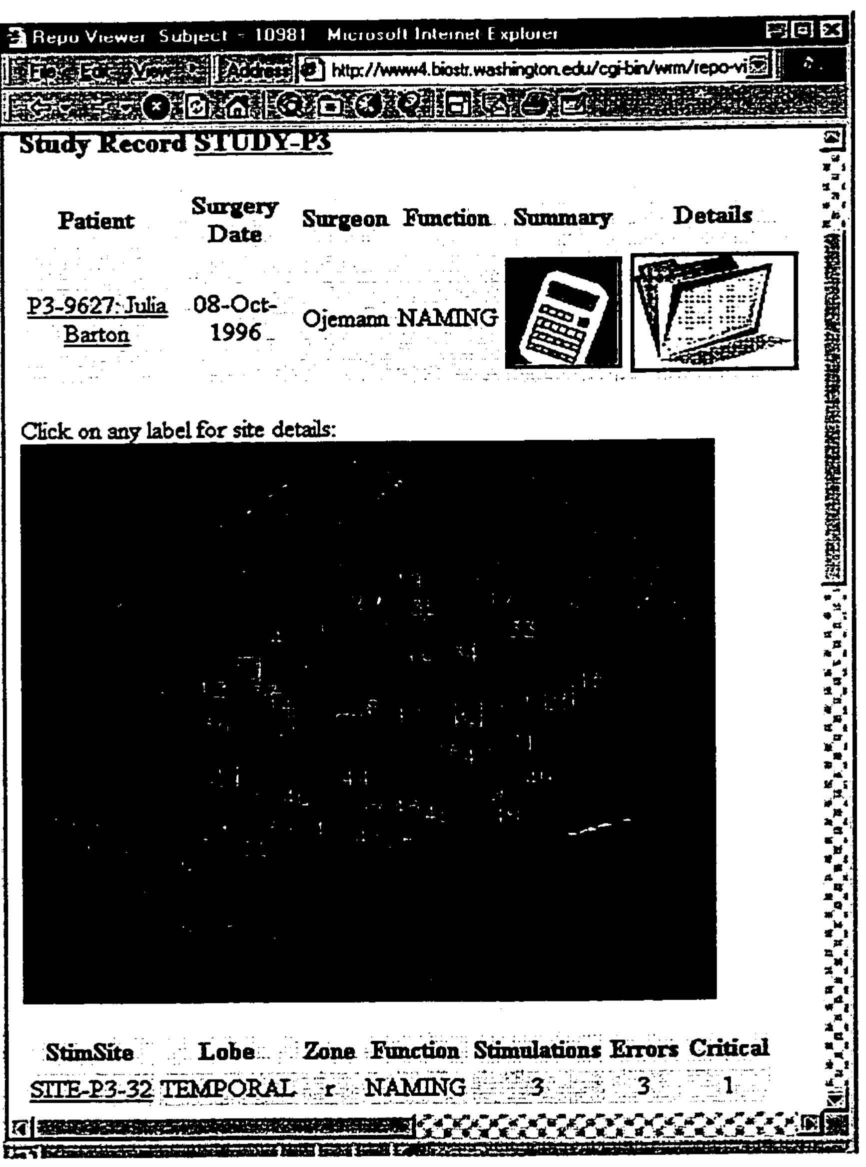


Figure 53: Study Record (Main View)

containing information about all the stimulation sites organized by lobe, and a Detailed view which includes a link to all the mappings performed on this patient, as well as a table showing all the sites (as shown in Figure 53). The Study Details is actually a *Virtual Class*, in that it has its own set of view methods but no underlying schema of its own.

The stimulation sites are represented by the StimSite schema, which includes the 3D coordinates for that site as computed from the Model (anterior, superior, and right coordinates) as well as the 2D pixel coordinates of the site on the rendered map file. It

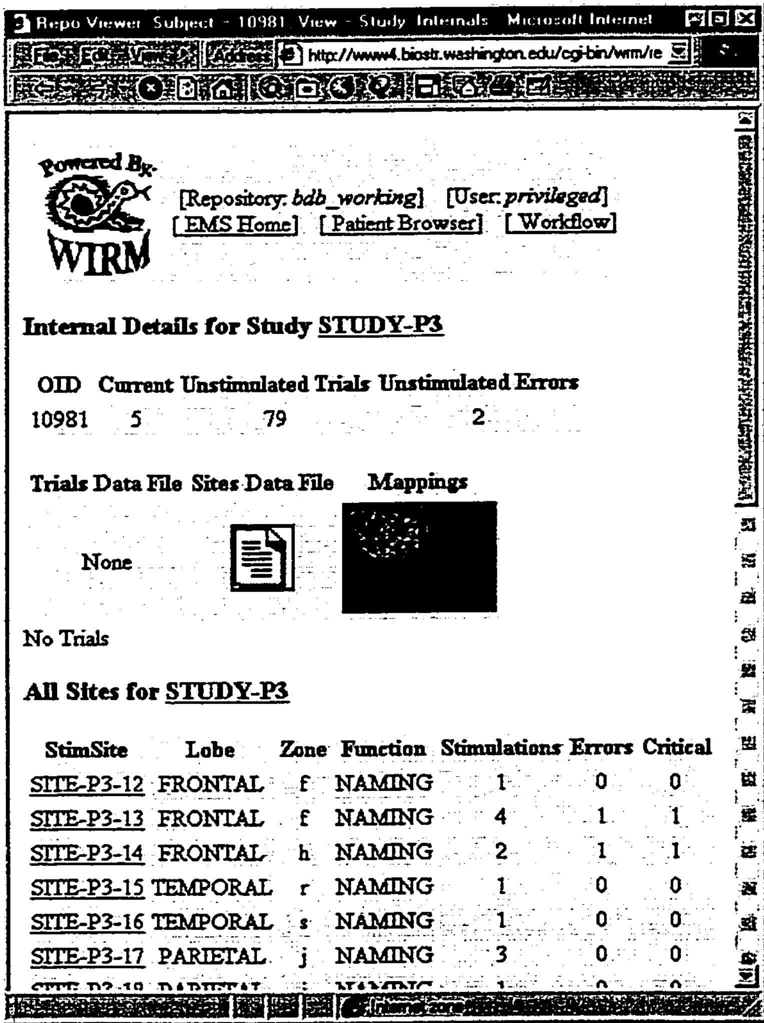


Figure 54: Study Record (Detailed View)

also records the number of stimulations and errors for the site, the lobe and zone of its location, an indication of whether or not the site is critical, and a set of labels used to identify the site under various circumstances.

StimSites should not be confused with MapSites, which are simply the 3D coordinates of the sites identified during a Mapping task. A Mapping schema consists of the author who performed the mapping, the date of the task, a screenshot of the mapped rendering, and a data file which was produced by the mapping software. The data file is processed to create the MapSites. The coordinates of the StimSite are derived from a selected Mapping.

Every StimSite has a number of corresponding Trials, one for each stimulation. Trial data is imported from a spreadsheet, which was transcribed from the audio recording. The Trial schema includes a reference to the corresponding site label, the trial number, the time of the trial, the slide that was to be named, the current applied, the score, and a confidence rating.

One issue that arises in defining the schemas is how much information should be left embedded in the multimedia object, and how much should be extracted as symbols in the schema. The answer usually depends on the capabilities of the available filters, and on the requirements of the analysis processor. The key point is that the schemas can evolve to handle gradual refinement as required by the experiment.

Often the best design for a schema is not known until after the designer has seen the system in use. Sometimes the designer will purposefully introduce schematic conflicts in the schema, because they best represent the current way that an attribute is used. For example, the Site label originally served both as the label in the Photograph and the label in the Mapping, which do not always correspond. Although they represented different concepts, it made sense to fold them into a single attribute for simplicity, as the

distinction was not important. Later, as we began to analyze trial data, it became important to separate the concept as two distinct attributes. The schema evolution feature allows us to model concepts "warts and all" until they become clearer to the designer.

7.4 The Brain Mapper Wirmlets

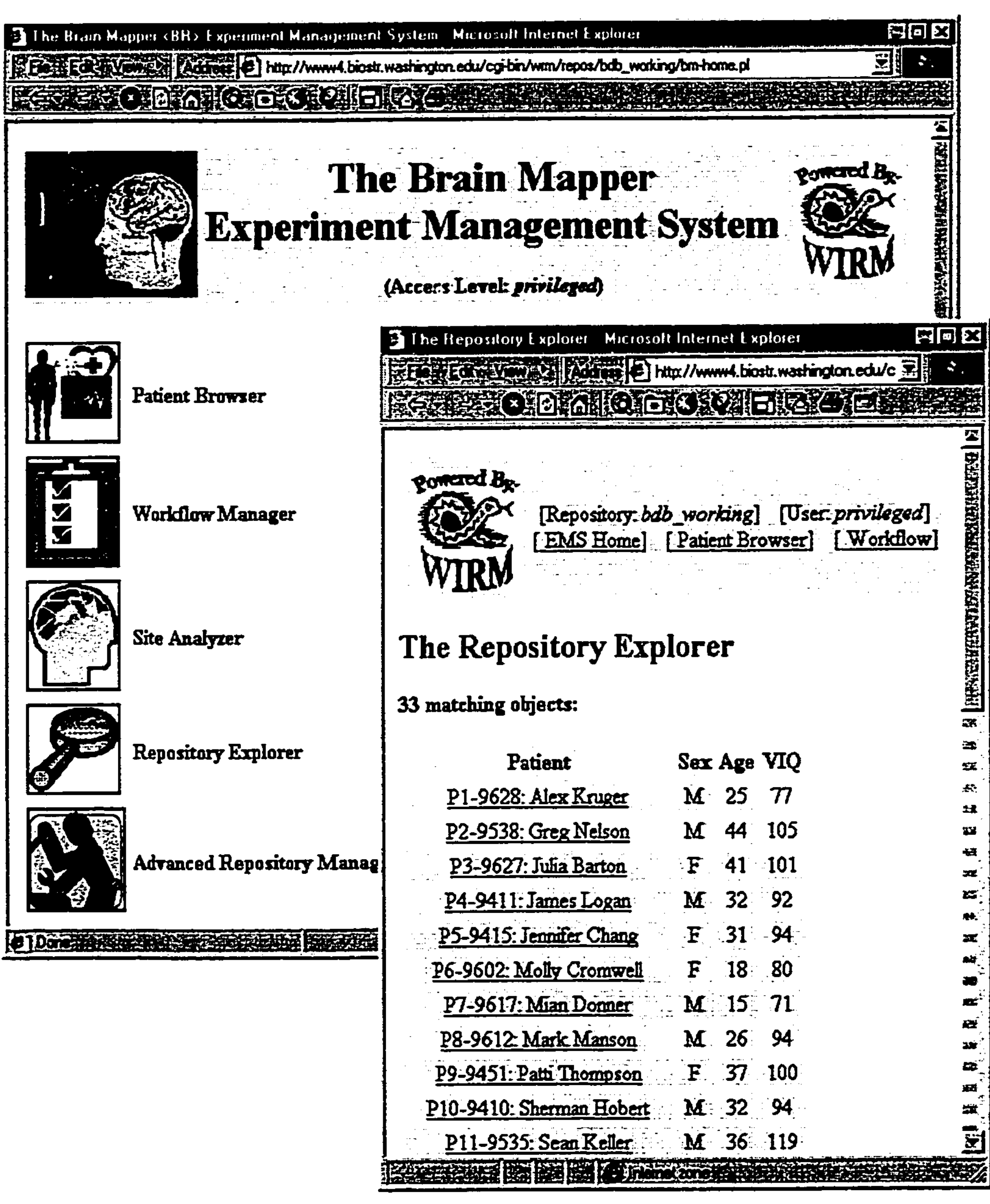


Figure 55: Brain Mapper Main Menu & Patient Browser

The main menu for the Brain Mapper EMS is shown in Figure 54. The available Wirmlets are the Patient Browser, the Workflow Manager, the Site Analyzer, the Repository Explorer, and the Advanced Repository Management console.
The Patient Browser is merely a link to the standard Repo Explorer Wirmlet, using the Patient class as the subject. As described in the Architecture chapter, when the Repo Explorer is activated with a class name as the subject, it retrieves all members of the class and displays them in table form. Thus, the Patient Browser provides the user with a top-down entry point from which to view all patients. As each Patient's row view includes a link to the page view for that patient, and each page view includes links to the sub-parts for that patient, a natural drill-down navigational hierarchy is formed. Figure 54 also shows the Patient Browser for the Privileged user (the Public user would not see names, of course).

The Workflow Manager is a menu Wirmlet for driving the workflow stages of data acquisition and processing. Like the Patient Browser, the Workflow Manager is patient-centric: all workflow stages are organized top-down from the perspective of acquiring the data for a single patient. The Brain Mapper EMS was designed prior to the generalized Make, Update, and Delete Wirmlets, so it does not yet conform to the model of encapsulating workflow within specially designated methods of class definitions. Instead, the make and update workflow stages are implemented as stand-alone Wirmlets. For example, the task of updating a patient is encoded in a custom-defined Patient Update Wirmlet, rather than as a `Patient_update` method within the class definition file. However, I expect that the process of transitioning to the method model will be straightforward, as the two approaches are conceptually isomorphic.

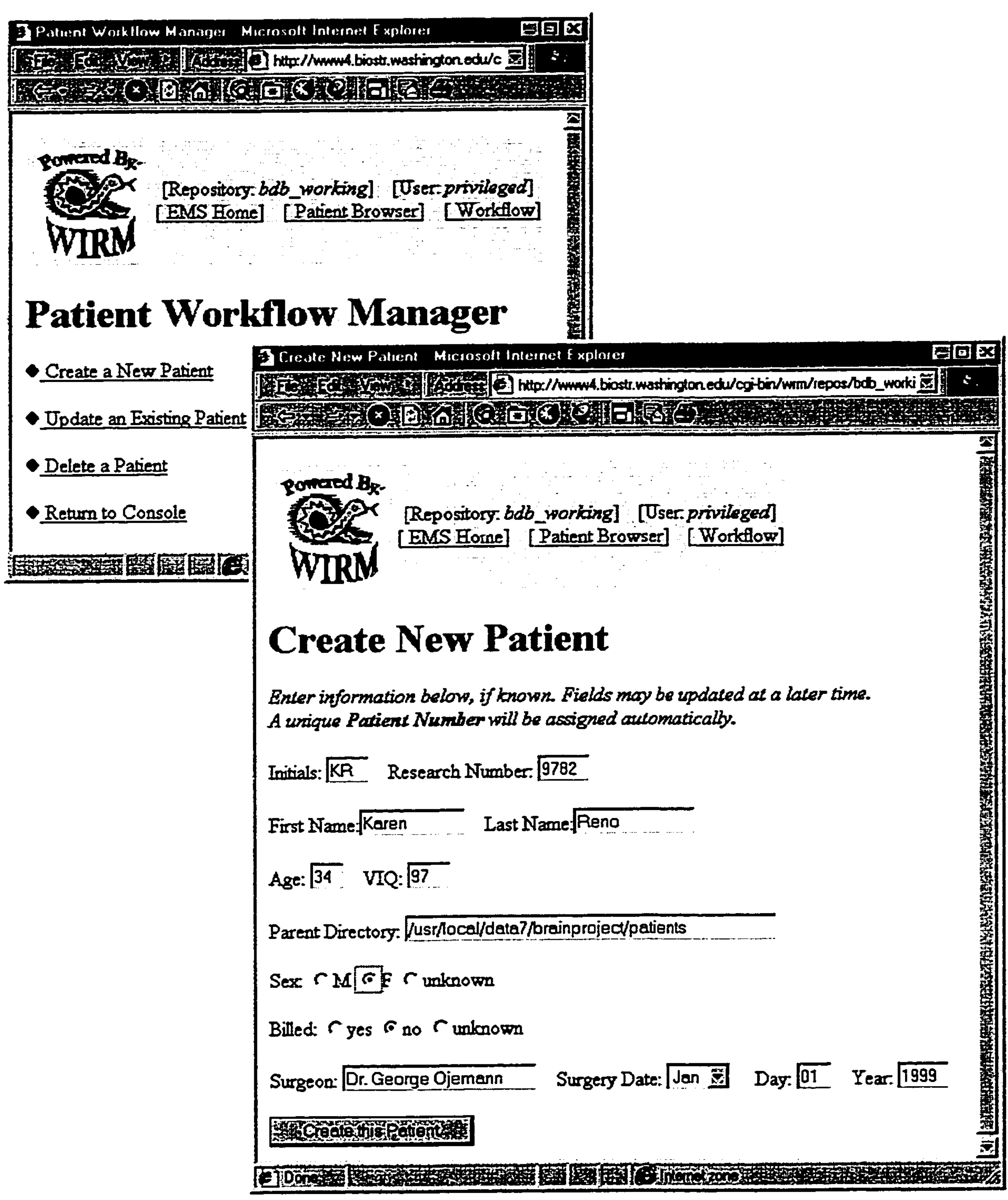


Figure 56: Patient Workflow Console and Make Patient Wirmlet

The main workflow Wirmlet provides options for creating new patients, updating existing patients, and deleting patients, each of which activates a custom Wirmlet. Those Wirmlets may direct the user to a number of other Wirmlets for performing workflow on parts of the Patient, such as updating a study.

As shown in Figure 55, the Patient Make Wirmlet provides a form for entering values for the attributes. The attributes that are prompted for do not necessarily correspond directly with the Patient Schema. Some attributes (such as the surgeon and surgery date) are part of the Study schema rather than the Patient itself, but Patient creation is a convenient time to request them. A Study object is created along with the Patient. Other attributes (such as the registration date and the patient number) are not prompted for because they are assigned automatically.

Some prompts use radio button groups to assign specific choices, rather than allowing free text entry. For example, the Sex choice is limited to M or F even though the schema accepts any character. This results in cleaner data that doesn't need to be normalized. Notice the Billed prompt, which can be "yes", "no", or "unknown". The billed attribute requires an integer, but the prompt is more readable. The result is translated by the Wirmlet into 1,0, or –1.

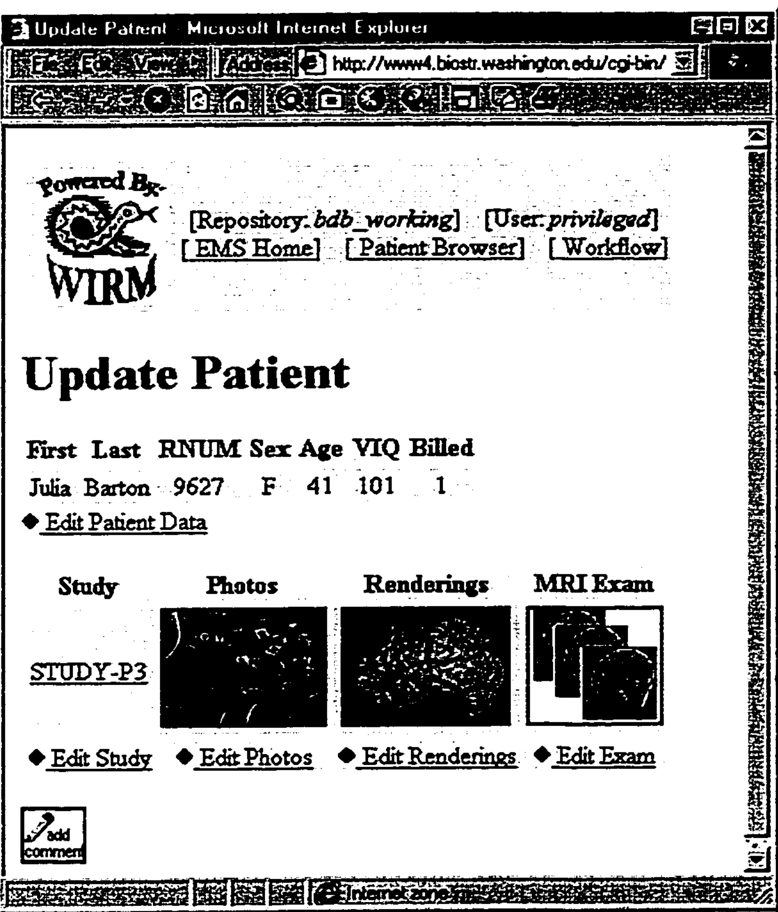


Figure 57: Patient Update

When the form is submitted, the attributes are extracted from the form elements and copied into a Repo Template, which is used to create a new Patient object through the `repo_new` function.

The Patient Update Wirmlet displays the various attributes of the Patient and provides links to the following Wirmlets: Patient Data Update, Photo Edit, Rendering Edit, Exam Edit, and Study Edit. The Patient Data Update Wirmlet (Figure 56) provides a form similar to the Create Patient Wirmlet, with the current values filled in. This allows users to register Patients with a minimal amount of data (e.g. name only) and update the information as it becomes available. The EMS users have found this extremely useful, as

this allows them to acquire data in a flexible order (e.g. the Exam can be imported before the demographic data is known).

The Study Edit Wirmlet (not shown) provides a form for editing the Study metadata directly (such as the surgeon, surgery date, and study function), as well as links to four Wirmlets that perform various pieces of workflow on the study: Upload Trial Data, Process Trials, Edit Trials by Hand, and Assign Rendered Map. Each of these will be discussed in turn.

The Upload Trial Data Wirmlet (Figure 57) allows the user to upload a Microsoft Excel spreadsheet containing the trial data, which was transcribed from the audio tape. The

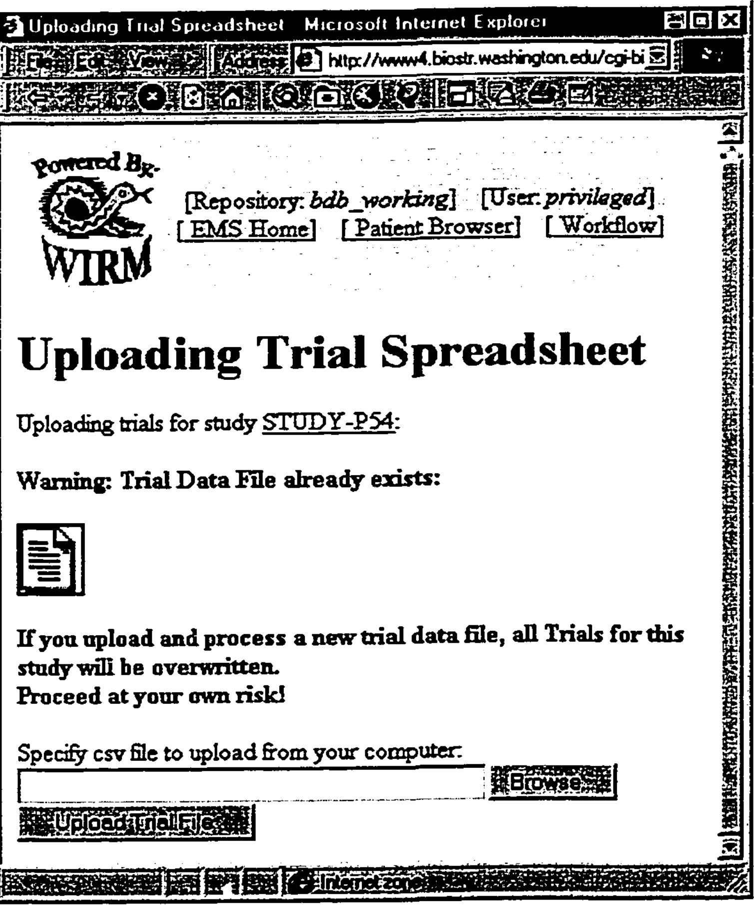


Figure 58: Uploading a Spreadsheet

data file can be imported from any machine connected to the Internet. This has been extremely useful for the Neuroscience researchers, who are working from various remote offices. Previously, they had to physically transfer the spreadsheet to the Biostructure lab on a diskette or use an FTP program with an unintuitive interface, and they had to know the internal directory structure of the Patient hierarchy in order to put it in the right place. By contrast, the Upload Trial Data Wirmlet provides a graphical browser for identifying the file on their own machine and then moves the file automatically into the repository, without requiring the researcher to know any information about the destination. If a spreadsheet already exists for that patient, the user is warned that the new file will replace the old one.

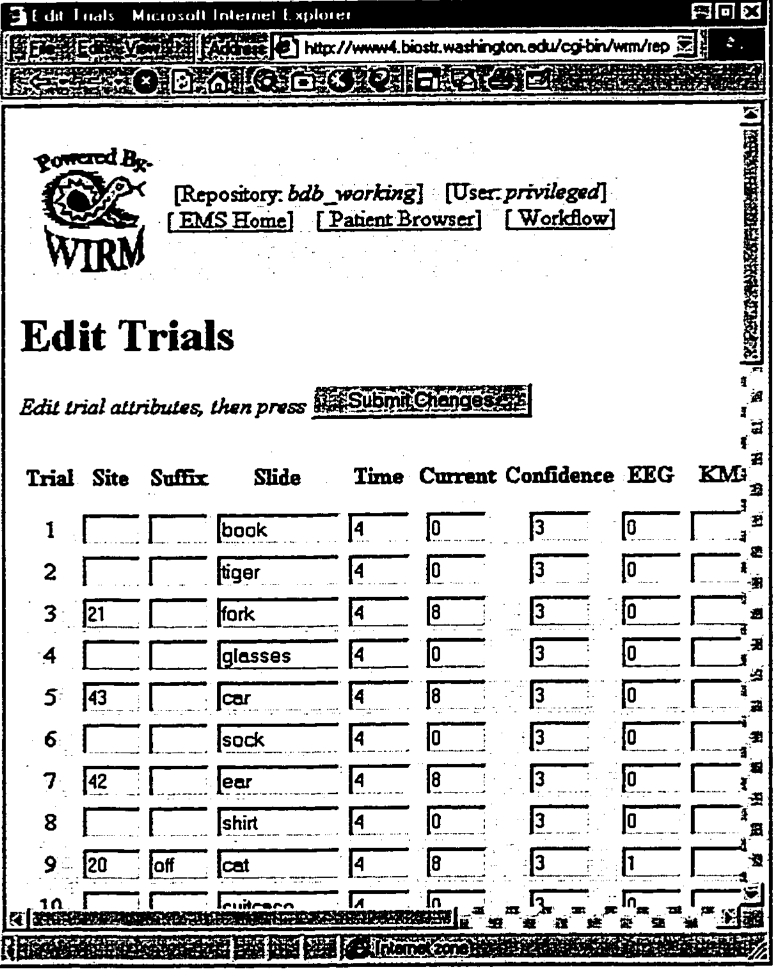


Figure 59: Editing Trials By Hand

The second stage of updating a Study is to process the spreadsheet. This is performed with the Process Trials Wirmlet, which parses the spreadsheet and creates a new Trial object from each row. In addition, the Wirmlet creates a new StimSite object for every trial, which names a previously unmentioned Stimulation Site. Once the Trial data is in the repository database, it is simple to query them to determine whether each StimSite should be considered critical, and to compute summary data for the Study. Furthermore, the Trial data may now be accessed directly from the Patient browser by clicking on a StimSite in the rendered image-map.

Occasionally, the neuroscientist wants to change the information in a specific trial. For

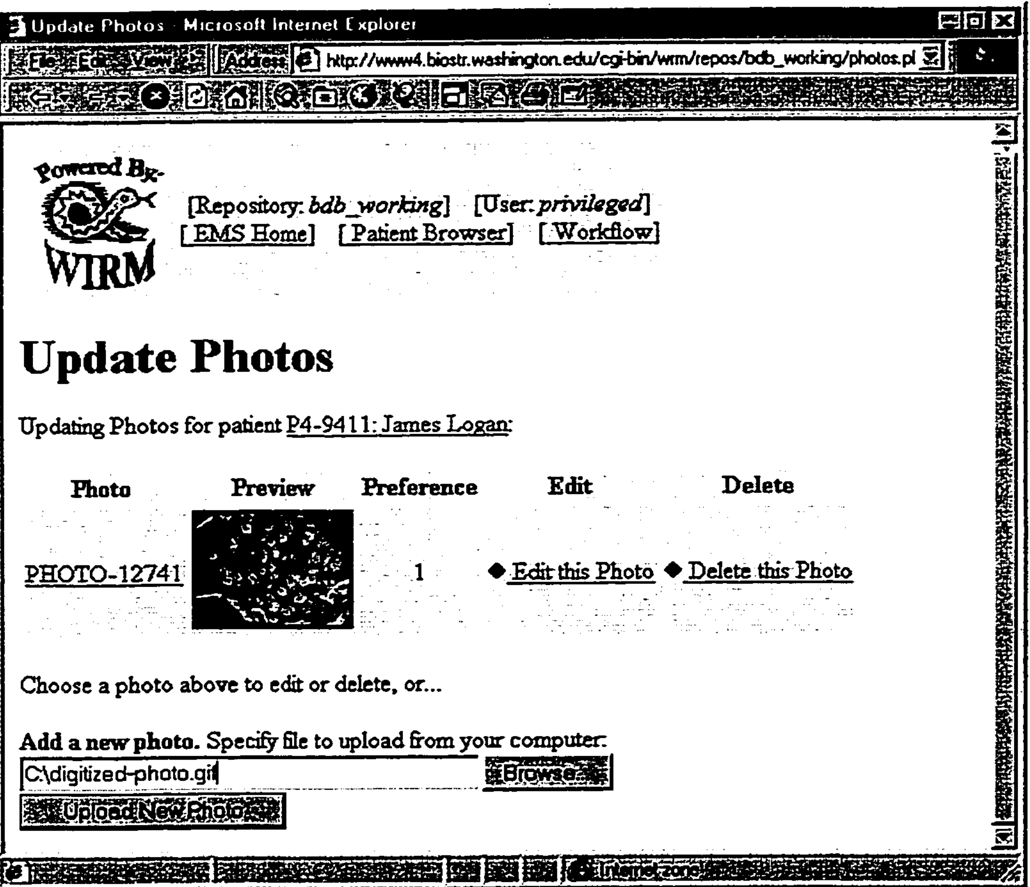
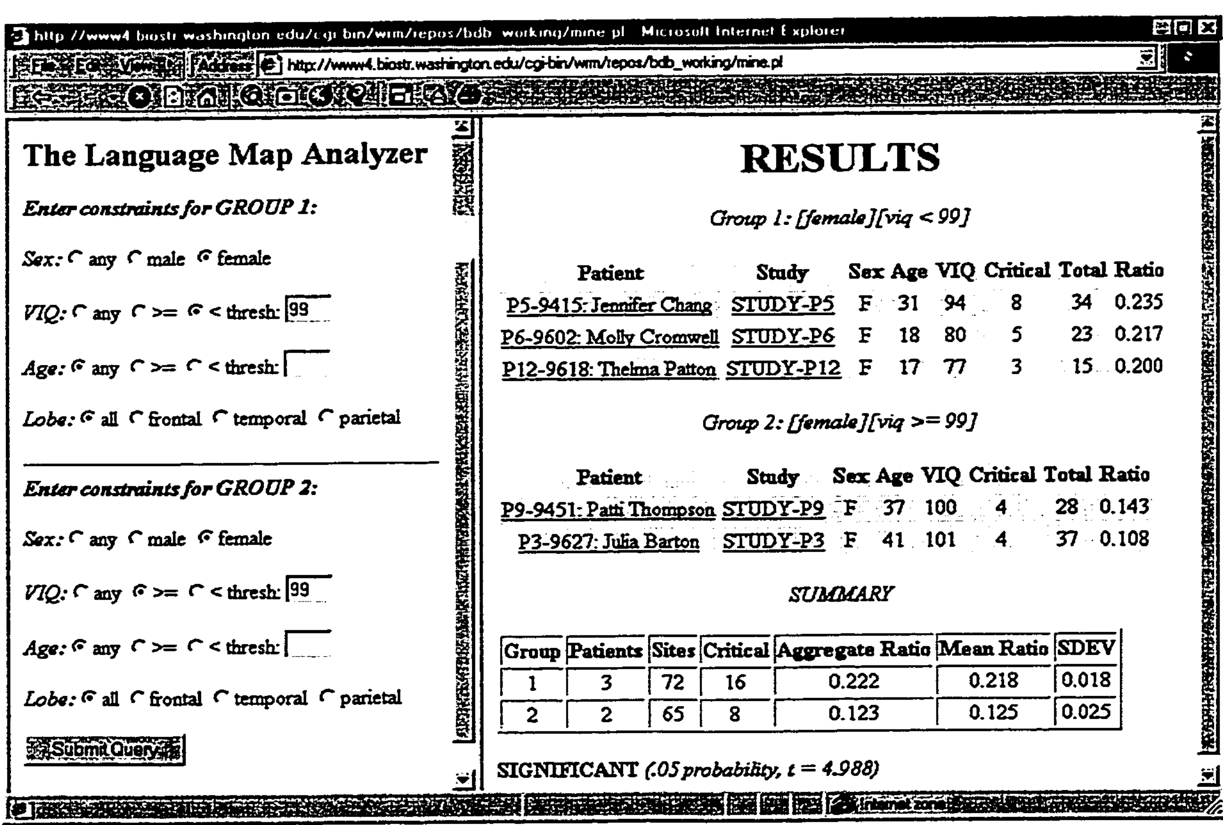


Figure 60: Photo Update Wirmlet

example, a further review of the audio tape may reveal that the patient did indeed name a slide correctly which was previously considered an error. Originally, this required the user to change the original spreadsheet, then upload the entire spreadsheet back into the repository and re-process all the trial data. To simplify this process, I created the Update Trials By Hand Wirmlet, which creates a form that allows the researcher to arbitrarily update Trials from any Web browser as shown in Figure 58, with the changes going directly to the database.

In addition to uploading Trial spreadsheets, users are able to upload digitized photographs and renderings using the Update Photos and Update Renderings Wirmlets respectively. Figure 59 shows the Update Photos Wirmlet, which allows the user to upload a new photo, edit the metadata of an existing photo, or delete a photo.



http://www4.biostr.washington.edu/cgi-bin/wrm/repos/bdb_working/mine.pl

The Language Map Analyzer

Enter constraints for GROUP 1:

Sex: any male female

VIQ: any >= < thresh: 99

Age: any >= < thresh:

Lobe: all frontal temporal parietal

Enter constraints for GROUP 2:

Sex: any male female

VIQ: any >= < thresh: 99

Age: any >= < thresh:

Lobe: all frontal temporal parietal

Submit Query

RESULTS

Group 1: [female][viq < 99]

Patient	Study	Sex	Age	VIQ	Critical	Total	Ratio
P5-9415: Jennifer Chang	STUDY-P5	F	31	94	8	34	0.235
P6-9602: Molly Cromwell	STUDY-P6	F	18	80	5	23	0.217
P12-9618: Thelma Patton	STUDY-P12	F	17	77	3	15	0.200

Group 2: [female][viq >= 99]

Patient	Study	Sex	Age	VIQ	Critical	Total	Ratio
P9-9451: Patti Thompson	STUDY-P9	F	37	100	4	28	0.143
P3-9627: Julia Barton	STUDY-P3	F	41	101	4	37	0.108

SUMMARY

Group	Patients	Sites	Critical	Aggregate Ratio	Mean Ratio	SDEV
1	3	72	16	0.222	0.218	0.018
2	2	65	8	0.123	0.125	0.025

SIGNIFICANT *(.05 probability, t = 4.988)*

Figure 61: Data Analysis Console

In addition to the Workflow modules, the EMS includes the *Data Analysis Wirmlet*, which provides a graphical interface for splitting patients into multiple groups based on their demographics and comparing their language sites. The left frame allows the researcher to filter patients into two groups based on their age, sex, and verbal IQ, and indicate which lobes should be evaluated. When the user presses submit, a T-test is performed on the ratio of critical to non-critical language sites for each group, and the results are displayed in the right side frame, including a summary view for each patient.

When new patients are entered into the system, they automatically become eligible for the Data Analysis. The neuroscience researchers found this feature extremely useful, and were able to compare new data to prior experiments and verify their findings. I plan to extend the Data Analysis console with multiple controls that allow more variables to be considered, and to work with statistical experts in devising new analysis techniques.

7.5 Evaluation of the Benefits of WIRM

From my experience with the Brain Mapper, I have identified a number of benefits afforded by WIRM which make the system extremely valuable. One of the most salient benefits is the ability for users to design their own schema online using forms in a Web browser which results in an instant Web-based drill-down navigation interface. I am not aware of any research system or commercial product that provides this functionality.

Another important benefit is the ability to dynamically evolve schemas from a visual interface and have the changes be propagated to existing data. To enable the repository to support queries that are more sophisticated over the multimedia data, the contents of the blobs must eventually be revealed to the query mechanism. This requires the designer to have an in-depth understanding of the inherent structure of the data to be modeled. My experience shows that this evolution is an ongoing process, and the degree

of schema refinement tends to increase over time as the system matures. The Schema Evolution facility makes this possible.

In terms of referencing attributes for building graphical views, the WIRM data model solved a major pitfall that was present in the relational model: dealing with the logical table structure. WIRM provided a more intuitive metaphor for dealing with data: sets of objects linked together by references, rather than cursor traversing a two-dimensional table. By abstracting away column positions and allowing the interface designers to reference attribute names directly as Perl hashes, the classical impedance mismatch problem was avoided [Jak93]. WIRM view definitions were intuitive and readable from a programmatic perspective.

As evidence of the high-level programming efficiency of WIRM's API's, here are the number of lines of Perl code for the class definitions and the Wirmlets. The Perl lines include liberal use of white space. The entire Brain Mapper EMS is implemented in fewer than 6000 sparse lines of code and uses approximately 100K of memory. This includes complex workflow and customized view facilities for over a dozen complex multimedia types, providing metadata management, document delivery, statistical data analysis, privacy and proprietary data regulation, and system integration. From my experience in building information systems, I believe that without the EMSBE, the Brain Mapper EMS would have taken an order of magnitude more lines of code and programming hours to implement.

Table 9 : Lines of Code in Brain Mapper EMS

MODULE	LINES
Class Definitions	4087
Main Console	57
Patient Workflow (General)	230
Study & Exam Workflow	384
Photo & Rendering Workflow	355
Data Analysis	227

This will be even further reduced when the latest versions of the system workflow Wirmlets are adopted. Because of the layered nature of the WIRM architecture, there is a high level of code reuse, and the API's themselves are extremely compact. The Web API and Tool API leverage highly off the Repository Object API, which in turn makes good use of the DB-API and the File-API. Together, the six WIRM API's are under 3700 lines of code. Table 10 shows the lines of code for each API.

Table 10: Lines of Code in WIRM API's

Interface	LINES
Table Manipulator	419
FSA Controller	251
Gateway	1270
HTML Generator	500
Repository Object API	854
Visualization Cache Manager	348

As can be seen from Table 11 below, WIRM code is extremely compact. The entire Brain Mapper console uses approximately 100KB, and the entire WIRM package (including system interfaces and generic Wirmlets) is under 130KB.

Table 11: Size of WIRM and Brain Mapper EMS

CODE	SIZE (KB)
System Interfaces	97
Generic Wirmlets	31
Brain Mapper Class Definitions	40
Brain Mapper Wirmlets	61

In addition to the ease of evolution and the elegance of expression, the WIRM allowed us to harness the power of Query-By-Context to create adaptive interfaces for each class of user. In addition to enforcing privacy and protecting proprietary data, I envision using WIRM to create multiple views for the various classes of privileged users. Radiologists, Surgeons, and Computer Scientists all want to use separate identifiers to refer to patients. Without context-sensitivity, one could try to force them to use a single identifier, or have a long identifier such as `P23-R9825-E3274-Joe-Smith.` However, Query-By-Context will allow us to distinguish between multiple dimensions of user classes, and update the patient Label View to use the preferred identifier for each user.

In summary, the high level rapid-prototyping ability of the Gateway API combined with the extremely intuitive modeling power of the REPO data model has allowed us to experiment with multiple interfaces without a large investment of programming effort. The flexibility of the Persistent Perl architecture allows us to implement any algorithm, unrestricted by the limitations of declarative templates. We were able to benefit immensely from the vast library of free Perl modules, and the WIRM's extensible architecture will allow the EMS to grow as the experiment evolves.

Chapter 8: Related Work

8.1 Existing Commercial Systems

In the past 10 years, a growing number of companies have developed products and services aimed at managing the data and workflow of clinical and research laboratories. These companies collectively are known as the Laboratory Information Management Systems Industry, and their solutions are designed for specific customer domains, such as waste management, food and beverage, oil and gas, and pharmaceutical industries. Some leading vendors in this market are LabSystems, Beckman, Perkin-Elmer, and LabVantage Solutions [Lsy-URL, Bec-URL, Pen-URL, Lab-URL]. Although the systems provided by these vendors are widely used, they do not offer the flexibility of view definition provided by WIRM, nor do they explicitly support context-based queries at the modeling level. While a growing number of them allow web-based interfaces to be added on over the existing interface, none support web view creation as a central aspect of the system. Furthermore, they do not handle arbitrary multimedia types, nor can they be dynamically extended with complex user-defined types.

Commercial LIMS tend to cost tens of thousands of dollars, and require a major investment in training and upkeep. They are geared towards large commercial laboratories, which deal with high volumes of relatively simple workflow and data types. WIRM, by contrast, is open source and freely available, and is aimed at smaller research groups which require more complex and customized workflow and data types.

In addition to the LIMS solutions, there are a number of industry-independent application servers that can be used to build web-based information systems. These tools are often packaged as additions to existing object-relational database servers (e.g. Oracle8 [Koc97] and Informix [Inf-URL]), or as third-party applications designed to work with any

relational database engine, e.g. Cold Fusion [For98]. These systems provide valuable services such as efficient transaction processing, state-based user sessions, encryption, and templates for web-database connectivity. However, none of them support the Query-By-Context view model, which means that the web developer is required to come up with their own scheme for defining context-sensitive hierarchical views, which greatly increases the complexity of building an EMS. One of my goals is to integrate WIRM's data and view models in a commercial system such as Cold Fusion [For98].

8.2 Research Systems for Web Site Management

There is a growing field of research focused on building Web site management systems. Like WIRM, these systems are concerned with creating Web views over multimedia content. Many of these systems do not focus on structured databases, but rather on integrating existing Web sites or exporting a collection of semi-structured documents. In

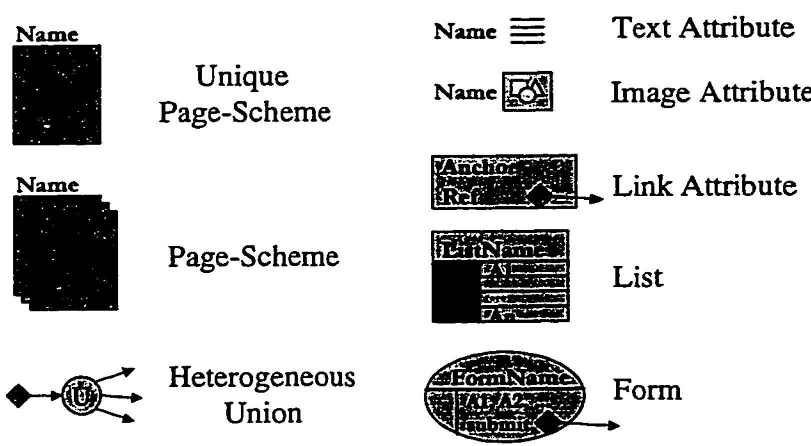


Figure 62: Araneus Data Model

this section, I discuss two such systems: Araneus and Strudel, and compare them to WIRM. Other related systems include Web-OQL [Aro98], Yat [Clu98], and TSIMMIS [Gar95].

Araneus [Atz97, Atz98] introduces a set of tools and languages for managing and restructuring data extracted from existing web sites, allowing multiple sites to be integrated in a new site. The Araneus Data Model (ADM), depicted in Figure 60, allows a site developer to describe the structure of an existing web site as a page-oriented schema. The model provides explicit constructs for describing the attributes of a hypertext document class, such as text, links, images, lists, and form elements. Using an ADM schema of a site, the Araneus system can generate wrappers that extract the content of the site into a relational database. Once the content has been extracted, it can be reorganized into different views using a navigational expression language called Ulixes.

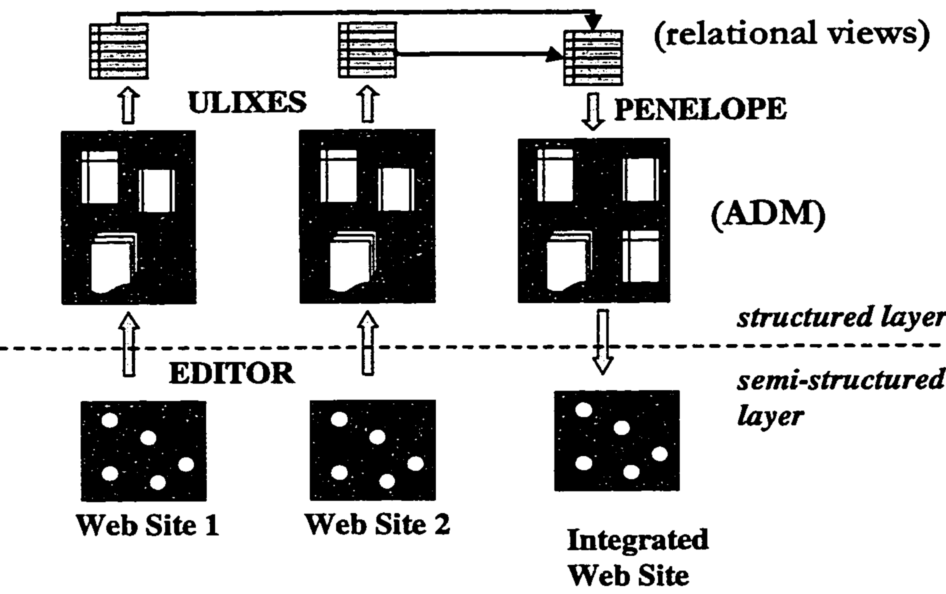


Figure 63: Restructuring a Web Site with Araneus

Finally, a new web site is derived from the views using a declarative language called Penelope. This entire process is illustrated in Figure 61.

While the ADM is related to the Query-By-Context View Model in that both address the problem of generating web pages from relational data, the approach taken is vastly different. With Araneus, the data already exists in web form, and the problem is focused on explicitly modeling the logical structure of the site for the purposes of reorganization, whereas in WIRM there is no pre-existing web site. The ADM takes a page-oriented approach, requiring the developer to explicitly declare the page structure of the resulting site in a declarative language, whereas the QBCx model takes an object-oriented approach, in which each class exports a procedural method which generates a self-describing web page.

Another system that applies database concepts to web site construction is **Strudel** [Fer98]. The key idea in Strudel is separating web site management into three orthogonal issues: content, structure, and presentation. In contrast to WIRM, Strudel encodes the site structure as a declarative *site graph*, allowing navigational integrity constraints to be explicitly expressed. WIRM does not support an explicit site graph, but navigational constraints can be implicitly enforced via the inclusion or exclusion of link emissions within View methods of the object classes. For example, adding a statement in the Patient's Page View that emits a link to the Exam is equivalent to declaring the constraint "every patient's exam is reachable from that patient's main page".

Like WIRM, Strudel allows web sites to be constructed from data residing in a database or file system. Unlike WIRM, which provides document management and image conversion routines, Strudel has no built-in support for handling multimedia types. Both systems support flexible data types: WIRM through its dynamic schema evolution, and Strudel through its semi-structured data model. Both systems require a view definition

stage, but WIRM view definition is procedural with embedded SQL declarations, whereas Strudel is declarative with embedded procedures.

In WIRM, web pages are generated at click-time by html-emitting procedural CGI scripts. By contrast, Strudel web pages are precomputed using declarative templates. Although template-based systems are highly intuitive and provide a clean foundation for theoretical analysis, they tend to be restricted in their practical use when compared to procedural systems that support general purpose computation for defining GUI behavior. This is evident in the fact that WIRM interfaces can be highly interactive, supporting arbitrarily complex forms and maintaining user session state, whereas Strudel supports only simple navigational interfaces.

As does Strudel, WIRM emphasizes the importance of separating content from structure. In WIRM, structure is encoded as calls to link generation functions in class definitions, whereas the actual content exists as object instances retrieved from relational tables. WIRM doesn't explicitly separate site *presentation* from site structure, but does provide mechanisms that allow the site designer to encapsulate presentation elements as variables and function calls. For example, the designer may define a Table Template, which describes the color, alignment, and format to be used when displaying a group of objects.

8.3 Research Systems for Laboratory Information Management

While Araneus and Strudel are representative of domain-independent efforts to build web site management systems, there are a number of projects focused specifically on experiment management systems. LabBase from MIT [Ste94, Goo94] is an example of a genomic laboratory database and workflow system built on an object-oriented database. OpenLabs [Ope-URL] is an international project aimed at developing modular, scalable components for clinical laboratory information systems. These systems propose domain-specific solutions for managing experiment data. However, the only project that

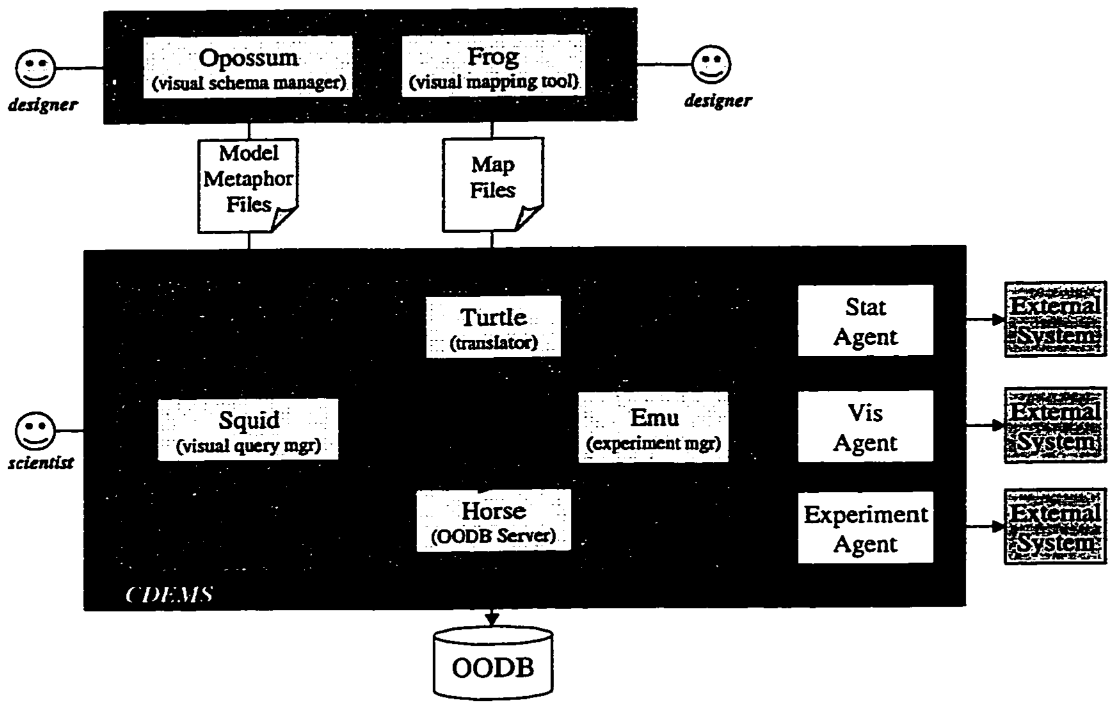


Figure 64: Zoo Architecture

proposes a general purpose environment for building experiment management systems is Zoo from the University Wisconsin [Ioa96].

Like WIRM, Zoo is a tool for building customized experiment management systems for laboratories. The Zoo architecture (Figure 63) resembles WIRM in that it provides a schema modeling tool (Opossum) [Hab95], a database server (Horse), a uniform GUI for issuing queries (Squid), and an API for interacting with external programs (Emu). While WIRM relies on the Perl pattern matcher to support translations between external systems and RSM objects, Zoo provides a visual tool (Frog) for specifying mappings between external data files and internal schemas.

The Zoo data model is an object-oriented model called Moose, which is similar to WIRM's object-relational Repository Schema Model in that it maps object classes to lists of primitives and tuples. Just as WIRM's API's provide an SQL interface for querying the repository, Zoo provides a declarative object-oriented query language called Fox.

Similar as they may seem from an architectural standpoint, WIRM and Zoo are very different in their approaches to building experiment management systems. WIRM is primarily aimed at Perl programmers, with some visual tools for common tasks, whereas Zoo attempts to remove most of the programming from the process of building an EMS. The resulting tradeoff is that Zoo systems are easier to build, whereas WIRM systems are more flexible and powerful in their data management capabilities. In WIRM, there is a clean separation between the EMSBE and a specific EMS, whereas in Zoo, a specific EMS is a customized installation of the Zoo system.

Perhaps the most salient difference between WIRM and Zoo is that WIRM is centered on building web-based interfaces to laboratory data, whereas Zoo uses its own proprietary desktop interface. Another important difference is that WIRM provides built-in multimedia support, whereas Zoo focuses on ASCII-based data. Further, Zoo provides no constructs for building context-sensitive web sites, which is an important benefit of WIRM.

Chapter 9: Future Work

In this section, I identify ten directions in which WIRM can be improved. Some of these approaches are already underway, while others will be left for future contributors. By releasing WIRM as an open-source application to the Perl community, I hope that the system will be extended by its users in the spirit of other open-source projects [Ray-URL].

9.1 Adopt a True Client-Server Architecture

As WIRM is currently implemented, the repository server is not a long-lived process handling requests from the web server, but rather a collection of interpreted Perl scripts which get re-executed on every web view. While this has been sufficient for handling a user community of a dozen project members, I expect that it would not scale well to a larger project, which may involve thousands of transactions per hour. The overhead of starting a new process on every web view can be eliminated by implementing a persistent process in which the WIRM API's are already compiled, which can field requests through a socket connection. Another possibility would be to use Apache's mod-perl facility [Apa-URL], which precompiles CGI scripts into the web server for efficient execution.

Although moving to a client-server architecture would result in more efficient web view generation, it would add an extra compilation stage in the edit-test cycle. This would not be desirable during periods of heavy development, in which the interpreted mode of the current version is extremely convenient for the programmer. It would be most useful to have two modes of operation, compiled and interpreted, which can be toggled according to preference.

9.2 Heighten Context Sensitivity

In the current Query-By-Context system, the EMS developer has access to three state-based context values: *who* (the identity of the current user), *what* (the type of view class being observed), and *which* (the identity of the subject being viewed). In addition, the following context values could be supported:

- *when* (the current stage of workflow being processed)
- *where* (the user's originating domain)
- *how* (the capabilities of the browser being used)
- *why* (predictions of the user's motives for requesting a web view)

These values could be useful for tailoring data access and organization beyond the current levels afforded by *who, what,* and *which.* For example, the Patient view might be presented differently depending on how many stages of workflow have been completed (*when*). Access to sensitive data could be restricted to browsers originating in internal facilities only (*where*). Alternate views could be defined for text-only browsers (*how*). Patient labels could be annotated with exam numbers if the user is known to be interested in radiology images (*why*).

As web technology evolves, users will expect more sophisticated navigational interfaces that adapt themselves to every situation. The Query-By-Context model provides a basis for designing next-generation data visualization systems that meet these demands.

9.3 Become Purely Object-Oriented

Although the current version of WIRM supports an object-oriented design style, it does not make use of Perl's built-in object-oriented features. WIRM was designed to work

with an older version of Perl that did not offer object-oriented support. Rather than defining each Repo class as a Perl module (in its own file package), Repo classes are actually implemented as groups of functions organized by name in a single class definition file. As a result, the syntax for accessing methods of an object is currently not of an object-oriented flavor, requiring the object to be explicitly passed to the function (e.g. *Patient_view($p)*) rather than using the more natural prefix syntax (e.g. *$p->view()*). Furthermore, WIRM does not make use of Perl's *reference blessing* facility, which allows an object's type to be automatically accessible from a reference to that object. Instead, type management is performed explicitly through an index of all instances.

Although the above issues are merely syntactic in their failure to conform to a true object-oriented style, there is a more fundamental feature lacking from WIRM classes: inheritance. All WIRM classes do inherit default methods from the top-level Repo class, but currently there is no way for methods to be inherited through intermediary class hierarchies. Schema designers may effectively inherit *attributes* from superclasses by basing new classes on existing classes, but methods must be explicitly copied in each sub class. By migrating to object-oriented Perl, WIRM would be able to overcome this serious limitation.

Another way in which the system could become more object-oriented is to make the WIRM API's into objects themselves. Then the Wirmlet designer could access facilities such as the Visualization Cache Manager by issuing requests to server objects rather than invoking a functional interface. I believe this would lead to code that is more intuitive. The process of objectifying WIRM has already begun, although it is not currently available for release.

9.4 Improve Support for Aggregate Types

The Repository Schema Model allows attributes to be atomic, composite, or aggregate types. Aggregate types can be ordered *lists* or unordered *sets,* and are useful for representing many-to-one relationships between classes. I chose not to implement aggregate types in the first version of WIRM, because aggregate relationships can be implemented using non-aggregate types in a standard relational table. For example, if a patient can have multiple exams, the exam table can include a column that references the patient, and a patient's set of exams can be retrieved by querying on the exam table. In other words, aggregate relationships must originate as an attribute in the contained class rather than the containing class. This has its limitations, however. For example, if an object can belong to more than one aggregation, a separate table must be defined to implement the containment relationships. Furthermore, it is often desirable semantically to express aggregates as attributes of the containing class.

This can be implemented using the Perl tie function, so that every time an aggregate attribute is referenced, the underlying members are retrieved from the database automatically and returned as a Perl hash (associative array). The programmer may iterate over the hash to access the list or set members as needed.

9.5 Integrate with more Applications and Data Types

As new scripts are written to interface WIRM with external applications, they can be incorporated into WIRM's API so that they are reusable by other EMS systems. Our goal in the Brain Mapper project is to provide tighter integration between WIRM and the applications that operate on the repository data. For example, the Skandha visualization software currently interfaces with WIRM at a coarse-grained file level, which must be processed by a customized script. Future iterations of system design will allow a finer grained interaction between Skandha and the repository, i.e. some Skandha operations

will be controllable from the web console, and each processing stage will be handled separately.

In addition to expanding the range of applications interfaced to WIRM, I also plan to provide methods for handling more data types. For example, the repository should be able to manage functional MRI data and other related formats. This may involve adding modules to the Visualization Cache Manager for converting these data types for web viewing, and possibly defining a new class to store metadata about fMRI images.

Another data type I plan to incorporate is audio files. We have digitized the audio recording of the patients performing their naming tasks, and we plan to build an interface that allows the user to click on a language site and hear the actual audio clip for that site. The user's browser will retrieve the sound file and activate a local sound player.

9.6 Provide Transparent Swizzling

In the current system, the programmer is required to explicitly translate between an object identification number (OID) and a pointer to the object. This is known as *swizzling* in database terminology. For example, if an Exam schema refers to a Patient object, the attribute type is an OID (integer) rather than an object reference. To access the actual Patient object, the programmer must make a separate query to retrieve the Patient whose OID matches the attribute in the Exam. Object-oriented databases perform this task automatically when the user attempts to access a referenced object, replacing the OID with a pointer to an instantiated object. This is known as *transparent swizzling*. The lack of transparent swizzling in the current version of WIRM leads to a large number of bugs while designing Wirmlets and class methods in the current system.

Consider this piece of code intended to retrieve a rendered map and assigning it to patient Smith's study:

```
$pid = repo_lookup_id("Patient", "last_name = 'Smith'");
$map = repo_query_single("Rendering",
      "patient = $pid AND rend_type = 'MAP' and preference = 1");
$study = repo_query_single("Study", "patient = $pid");
$study->{'preferred_map'} = $map;
```

The last line would cause a run-time error because the 'preferred map' field of the study expects an OID, which is an integer, but it is being handed a reference to a repository object, which is a pointer. Instead, the programmer is required to specify the OID field of the photo, which is less syntactically obvious:

```
$study->{'preferred_map'} = $map->{'oid'};
```

This seems like a minor inconvenience, but it occurs so often as to be a major headache for the programmer when dealing with complicated objects.

As another illustration of the benefit of transparent swizzling, consider the following scenario: a study refers to a rendered map, which in turn points to a file containing the image data. For documentation purposes, the description attribute of the file's metadata could be updated with a mention of the patient's research number. The following code would be required:

```
$patient = repo_get($study->{patient});
$rendered_map = repo_get($study->{rendered_map});
$file = repo_get($rendered_map->{image_file});
$file->{description} = "Rendered map for Patient $patient->{rnum}";
```

With swizzling, the above code could be replaced with this much cleaner statement:

```
$study->{rendered_map}->{image_file}->{description} =
```

```
"Rendered map for Patient $study->{patient}->{mum}";
```

Perl provides a feature called Tie that would make it possible to implement transparent swizzling, so that every time the dereference operator is performed on an OID, it is swizzled into an in-memory cache of objects (hashed by OID). Implementation of transparent swizzling has begun, and it will appear in a future version of WIRM.

9.7 Adopt a Better Transaction Model

Currently, every change to a repository object in memory must be committed to the database using the *repo_update* function. To commit a group of objects, they must each be updated separately. It would be preferable to support a transactional model that allows all changes to be written to the database on a "commit" action, rather than requiring the programmer to keep track of every change and explicitly invoke *repo_update* on each object. This could be implemented using an in-memory cache of all retrieved objects: whenever an object is brought into memory, store a pointer to it in the cache index. Objects will only be retrieved if they are not already residing in the cache. When a commit is executed, all dirty objects on the cache will be flushed back to the database. This will eliminate redundant retrievals, and will eliminate the potential integrity problems of having multiple copies of an object by guaranteeing that every time a program requests an object it will be given a pointer to the same instance in memory.

In addition to the lack of an object cache, WIRM currently has no mechanism for guaranteeing atomic updates, nor is there any facility for rolling back transactions. What is needed is the ability to lock objects while they are being manipulated, so that they can only be written to by one Wirmlet at a time.

9.8 Separate Content from Presentation

As demonstrated by Strudel [Fer98], there are many benefits to treating content, structure, and presentation as orthogonal issues in managing a web application. While WIRM provides facilities for separating content from structure, the process of defining the *presentation* of a web view is interleaved with the process of defining the structure. Although WIRM allows the view developer to encapsulate presentation elements as modular functions and variables (such as the Table Template), the fact remains that the presentation directives of a web site are embedded in the procedural view methods of the object classes. Consequently, the task of defining the "look and feel" of a web view requires the same programming tools that are used to define the structure in the view methods. The system would be more accessible if the two tasks could be delegated to separate users: *structure* designed by Perl developers, and *presentation* controlled by a graphic designer or other non-programmer.

I have begun designing a prototype version of WIRM in which presentation is completely separated from structure. In the prototype, the web views do not emit standard HTML documents, but rather XML entities. The web views will emit content (e.g. data attributes) wrapped in semantic markup describing what each piece of content is. In this manner, a view method is a procedural transformation between the attributes of a repository object (the current subject) and a unit of structured content. The presentation-free XML entity can then be visualized as an HTML document using style templates, which can be customized by a non-programmer via high level tools.

9.9 Refine Workflow Tools

Currently, WIRM lacks a high level mechanism for managing workflow. The design of workflow stages and monitors is completely up to the Wirmlet developer. The Query-By-Context methodology encourages the development of hierarchical workflow

interfaces, and the built-in Wirmlets (Object Maker, Object Editor, etc.) are useful tools in building a workflow management console. However, the system could benefit from explicit workflow support tools.

Currently, the Brain Mapper EMS provides a patient-centric workflow interface, which allows the end user to identify which stages of data acquisition and processing have been performed on a single patient. However, there is no top-down workflow management tool that allows the user to compare the workflow status of all patients at once. I envision a worksheet called the *TaskMaster,* which will summarize the workflow status of every patient. The user will be able to execute a desired workflow step directly by clicking on the appropriate cell in the worksheet. I expect that this kind of workflow interface could be generalized and offered as a standard facility of WIRM.

To implement the TaskMaster, one would create a new class to manage generic workflow stages, and provide a workflow definition console that allows the user to define new workflow requirements as methods of repository classes. Workflow steps should be hierarchical: that is, a given task can be made up of multiple sub-tasks, each of which can be assigned its own action. In addition, the system should support dependency specification, allowing the designer to declare which stages must be completed before a new stage can be started.

There is a large body of research on building workflow management systems for scientific computing [Ail98], and I would expect to leverage off of this work in designing WIRM's workflow facilities.

9.10 Support an ASP Approach

Consider the technical expertise required to maintain a WIRM-based EMS: the laboratory personnel must install, configure, and administrate a web server, cgi-bin directory,

database server, Perl interpreter, and a number of independent Perl modules (CGI.pm, database adaptor, Image Magick) as well as any external applications that play a role in the EMS workflow. In addition, they have to perform network administration and data backups. Compared to commercial LIMS products, WIRM's maintenance requirements are relatively low, but they still require a dedicated Unix administrator. Although some larger laboratories do have this kind of expertise on hand, many smaller research groups would be unable to afford such services.

In response to the high cost of technical personnel, many organizations are choosing to *outsource* their software administration tasks to external companies. With the rise of the web, a new industry of *application service providers (ASP's)* is being formed to fill this role in every domain. The ASP hosts the databases, web servers, and other back-end applications on their own machines for a monthly fee, and the customer is provided a simple web-based front end to the system. The end users are free from the technical complexities of system administration, and merely require a terminal with a web browser. I propose that WIRM can be easily adapted to the ASP model, allowing laboratories to reap the benefits of a web-based EMS while avoiding the administration costs.

In the current version of WIRM, the EMS developer can already define and evolve schemas using a web-based interface. However, the task of customizing class definitions and creating Wirmlets must be performed by editing Perl scripts on the WIRM server's machine. In an ASP model, the EMS developer would be able to access these scripts through a web form, and submit updates to the remote host without requiring direct access to the server's file system. This would be trivial to implement, but the problem of uploading a "Trojan horse" would need to be addressed.

In addition to providing web-based authoring of Wirmlets and class definitions, WIRM would need to be extended so that multiple repositories can be simultaneously served by

a single installation. This would require each repository to operate in its own "namespace" so that customers would be protected from each other.

WIRM would also need to be upgraded in terms of security, reliability, and scalability. The question remains: will laboratories be willing to entrust their precious data to an external company? If the appropriate legal and technical safeguards are placed in effect (encryption, redundant backups, etc.), this should not prove to be a serious barrier.

WIRM has already demonstrated its practical value as a tool for building laboratory information systems. If the ten improvements described in this chapter are implemented, WIRM could evolve into an important platform for both research and commercial endeavors alike. As new generations of laboratory systems inevitably emerge, giving rise to an ever-increasing proliferation of data, the need for customizable EMS's will continue to grow. WIRM will be there.

Bibliography

[Ahr96] J. Ahrens. Scientific experiment management with high-performance distributed computing. Ph.D. dissertation, University of Washington, 1996.

[Ail98] A. Ailamaki, Yannis E. Ioannidis, and M. Livny. Scientific workflow management by database management. SSDBM 1998: 190-199.

[Apa-URL] Apache/Perl Integration Project. http://perl.apache.org.

[Aro98] G. Arocena and A. Mendelzon. WebOQL: restructuring documents, databases, and webs. *ICDE 1998*: 24-33.

[Atk89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. *Proc. of the 1st int. conf. on distributed and object-oriented design,* 1989

[Atz97] P. Atzeni, G. Mecca and P. Merialdo. To weave the web. *VLDB 1997*: 206-215.

[Atz98] P. Atzeni, G. Mecca, and P. Merialdo. Design and maintenance of data-intensive web sites. *EDBT 1998:* 436-450

[Bec-URL] Beckman Coulter, Inc. http://www.beckman.com.

[Ber94] P.A. Bernstein and U. Dayal. An overview of repository technology. P*roceedings of the 20th VLDB Conference,* September 1994.

[Bri97] J.F. Brinkley and C. Rosse. The Digital Anatomist distributed framework and its applications to knowledge-based medical imaging. *J Am Med Assoc,* 4:165-83, 1997.

[Bri93] J. Brinkley, K. Eno, and J. W. Sundsten, Knowledge-based client-server approach to structural information retrieval: the Digital Anatomist Browser. *Computer Methods and Programs in Biomedicine* , vol. 40, pp. 131-145, 1993.

[Car90] M. Carey and L. Haas. Extensible database management systems, *ACM SIGMOD Record*, December 1990.

[Car95] M. Carey, L. Haas, P. Schwartz, M. Arya, W.F. Cody, R. Fagin, M. Flickner, A. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. Williams, and E. Wimmers. Towards heterogeneous multimedia information systems: the Garlic approach. *RIDE-DOM 1995,* Tapei, Taiwan, 1995.

[Cha98] S. Chaudhuri and V. Narasayya. AutoAdmin "what-if" index analysis utility. *SIGMOD 1998,* pp. 367-378.

[Chu94] W. Chu, A. Cardenas, and R. Taira. KMeD: A knowledge-based multimedia medical distributed database system. *Information Systems*, 20(2): 75-96, 1994.

[Clu98] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your Mediators Need Data Conversion!. *SIGMOD Conference 1998,* 177-188.

[Com90] Committee for Advanced DBMS Function. Third-generation database system manifesto. *SIGMOD Record,* 19(3): 31-44, 1990.

[Com91] Committee on a National Neural Circuitry Database. Mapping the brain and its functions: integrating enabling technologies into neuroscience research. National Academy Press, Washington D.C., 1991.

[Con90] D. Connelly. Embedding expert systems in laboratory information systems. *Amer. Journal of Clinical Pathology,* 94: (4) S7-S14, Oct. 1990.

[Cpa-URL] CPAN: Comprehensive Perl Archive Network. http://www.cpan.org.

[DA-URL] The digital anatomist home page. http://www1.biostr.washington.edu/DigitalAnatomist.html.

[Daya84] U. Dayal and H. Hwang. View definition and generalization for database integration in MULTIBASE: A system for heterogeneous distributed databases. *IEEE Trans. Software Engineering*, SE-10, 6, 628-644, 1984.

[Etz96] O. Etzioni. Moving up the information food chain: deploying softbots on the world wide web. *AAAI/IAAI, Vol. 2 1996*: 1322-1326

[Fer98] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: experiences with a web-site management system. *SIGMOD Conference 1998,* 414-425.

[For98] B. Forta, N. Weiss, and E. Crawford. The Cold Fusion 4.0 web application construction kit. Macmillian Publishing, 1998.

[Gar95] H. Garcia-Molina, D. Quass, Y. Papakonstantinou, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom: The TSIMMIS approach to mediation: data models and languages. *NGITS 1995.*

[Gei96] K. Geiger. Inside ODBC. Microsoft Press, 1996.

[Goo94] N. Goodman, S. Rozen and L. Stein. Building a laboratory information system around a C++-based object-oriented dbms. *VDLB, 1994.*

[Goy96] N. Goyal, C. Hoch, R. Krishnamurthy, B. Meckler, M. Suckow. Is GUI programming a database research problem? *SIGMOD Conference 1996,* pp. 517-528.

[Gre99] P. Greenspun. Philip and Alex's guide to web publishing. Morgan Kaufmann, 1999.

[Gro97] W. Grosky. Managing multimedia information in database systems. *CACM 40*(12): 72-80, 1997.

[Hab95] E. Haber, Y. Ioannidis, and M. Livny. OPOSSUM: Desk-Top Schema Management through Customizable Visualization. *VLDB,* pp. 527-538, 1995.

[Hin95] M. Hinton. Laboratory Information Management Systems. Marcel Dekker, Inc., New York, 1995.

[Hug-URL] D. Hugues. Mini SQL: A lightweight database server. http://bond.edu.au/People/bambi/mSQL/.

[Inf-URL] Informix. http://www.informix.com.

[Ioa96] Y. Ioannidis, M. Livny, S. Gupta, N. Ponnekanti. ZOO: a desktop experiment management environment. *VLDB,* pp. 274-285, 1996.

[Isa98] T. Isakowitz, M. Beber, and F. Vitali. Web information systems. *Communications of the ACM,* 41:7:78-80, July 1998.

[Jak93] R. Jakobovits. Persistent programming languages: the best of both worlds. *Univ. of Washington Technical Report,* UW-CSE-971203, Dec. 1993.

[Jak94] R. Jakobovits. The design and implementation of a database environment for vision research. *Univ. of Washington Technical Report,* UW-CSE-971204, Dec. 1994.

[Jak95] R. Jakobovits, L. Shapiro, and S. Tanimoto. Implementing multi-level queries in a database environment for vision research. *IS&T SPIE Symposium on Electronic Imaging: Science & Technology,* February 1995.

[Jak96a] R. Jakobovits, L. Lewis, J. Ahrens, L. Shapiro, S. Tanimoto, J. F. Brinkley. A visual database environment for scientific research. *Journal of Visual Languages and Computing,* Vol. 7, pp. 361-375, 1996.

[Jak96b] R. Jakobovits, B. Modayur, and J. F. Brinkley. A Web-Based Repository Manager for Brain Mapping Data. *Proceedings of AMIA Fall Symposium,* pages 309-313, Washington, D.C., Oct 28-30, 1996.

[Jak97a] R. Jakobovits. Integrating heterogeneous autonomous information sources. *Univ. of Washington Technical Report,* UW-CSE-971205, July 1997.

[Jak97b] R. Jakobovits and J. F. Brinkley. Managing medical research data with a web-interfacing repository manager. *Proceedings of AMIA Annual Fall Symposium,* pages 454-458, Nashville, Oct 25-29, 1997.

[Jak98] R. Jakobovits and J.F. Brinkley. Query-By-Context: A framework for modeling and navigating multimedia research data. *Proceedings of AMIA Fall Symposium,* Orlando, Nov. 7-11, 1998.

[Kim91] W. Kim and J. Seo. Classifying schematic and data heterogeneity in multidatabase systems, *IEEE Computer*, December 1991.

[Kim95] W. Kim. Modern database systems: the object model, interoperability,and beyond. ACM Press, New York, 1995.

[Koc97] G. Koch and K. Loney. Oracle 8: the complete reference. Oracle Press, 1997.

[Koe-URL] A. Koenig. The Msql perl adaptor. ftp://Bond.edu.au/pub/Minerva/msql/Contrib/MsqlPerl.README.

[Kos97] S. Koslow and M. Huerta. Neuroinformatics: an overview of the human brain project. Lawrence Erlbaum, Mahwah, NJ, 1997.

[Lab-URL] LabVantage Solutions. http://www.lims.com.

[Lit89] W. Litwin, et. al. MSQL: a multidatabase language. *Information Science*, 49, 59-101, 1989.

[Lit90] W. Litwin, L. Mark, N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22, 3, 267-293, 1990.

[Lev96] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. *Proceedings of the 22nd VLDB Conference*, 1996.

[Lim-URL] LIMSource. http://www.limsource.com/home.html

[Lsy-URL] LabSystems Online. http://www.labsystems.com.

[Mag-URL] Image Magick. http://www.wizards.dupont.com/cristy/.

[Mar96] R. Martin and D. Bowden. a stereotaxic template atlas of the macaque brain for digital imaging and quantitative neuroanatomy. *NeuroImage* 4:119-150, 1996.

[Mcd90] R. McDowall and D. Mattes. Architecture for a comprehensive laboratory information management system. *Analytical Chemistry*, 62: (20) A1069, Oct. 1990.

[Mod96] B.Modayur, J.Prothero, C.Rosse, R.Jakobovits, and J.F.Brinkley. Visualization and mapping of neurosurgical functional data onto a 3-D MR-based model of the brain surface. *Proceedings of AMIA Fall Symposium*, pages 304-308, Washington, D.C., Oct 28-30, 1996.

[Mod97a] B. Modayur, R. Jakobovits, K. Maravilla, G. Ojemann, and J. F. Brinkley. Evaluation of a visualization-based approach to functional brain mapping. *Proceedings of AMIA Fall Symposium*, pages 429-433, Nashville, Oct 25-29, 1997.

[Mod97b] B. Modayur, G. Ojemann, E. Lettich, R. Jakobovits, J. Sundsten, and J. F. Brinkley. Information system for mapping and studying cortical language organization. In *Abstracts, Society for Neuroscience Annual Meeting*, pages 2228, New Orleans, October 25-30, 1997.

[Oje89] G. A. Ojemann, J. Ojemann, E. Lettich, and M. Berger. Cortical language localization in left, dominant hemisphere. *J. Neurosurgery*, 71:316-326, 1989.

[Ope-URL] Openlabs: application of advanced informatics and telematics for optimization of clinical laboratory services. http://www.ics.uwe.ac.uk/projects/olab.html.

[Ore92] J. Orenstein, S. Haradhvala, B. Margulies, and D. Sakahara. Query processing in the ObjectStore database system. *SIGMOD Conference 1992:* 403-412.

[Pap95] Y. Papakonstatniou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. *11th International Conference on Data Engineering*, pp. 251-260, 1995.

[Pen-URL] Perkin-Elmer Informatics. http://www.peinformatics.com.

[Ray-URL] E. Raymond. The Cathedral and the Bazaar. http://www.tuxedo.org/~esr/writings/cathedral-bazaar/

[Ros98] C. Rosse, J. Mejino, B. Modayur, R. Jakobovits, K. Hinshaw and J. F. Brinkley. Motivation and Organizational Principles for Anatomical Knowledge Representation: the Digital Anatomist Symbolic Knowledge Base. *Journal of the American Medical Informatics Association,* vol. 5, no. 1, pp. 17-40, January 1998.

[Roz98] L. Rosenfeld and P. Morville. Information architecture for the world wide web. O'relly & Associates, Sebastopol, CA, 1998.

[Run96] E. Rundensteiner, H. Kuno, Y. Ra, V. Crestana-Taube, M. Jones, and P. J. Marron. The Multiview project: object-oriented view technology and applications. *SIGMOD, 1996*: 555.

[Set90] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous and autonomous databases. *ACM Computing Surveys,* 22, 3, pp. 183-236, 1990.

[Sig-URL] The University of Washington Structural Informatics Group.

http://sig.biostr.washington.edu

[Sil96] A. Silbershatz, M. Stonebraker, J.Ullman. Database research: achievements and opportunities into the 21st century. SIGMOD Record, 25(1):52-63, 1996.

[Sha94] L.Shapiro, S.Tanimoto, J.F.Brinkley, J.Ahrens, R.Jakobovits, and L.Lewis. A visual database system for data and experiment management in model-based computer vision. *Proceedings of the Second CAD-Based Vision Workshop,* pages 64-72, February 1994.

[She90] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous and autonomous databases, *ACM Computing Surveys,* 22, 3, pp. 183--236, 1990.

[Ste94] L. Stein, S. Rozen and N. Goodman. Managing laboratory workflow with LabBase. *Proceedings of the 1994 Conference on Computers in Medicine,* 1994.

[Ste-URL] L. Stein. CGI.pm - a perl5 CGI library. http://www.genome.wi.mit.edu/ftp/pub/software/www/cgi docs.html.

[Sto86] M. Stonebraker. Inclusion of new types in relational database systems. *Proc. of the International Conference on Data Engineering,* Feb. 1986, IEEE Computer Society Press.

[Sto94] M. Stonebraker. Readings in database systems, 2nd ed. Morgan Kaufmann Inc., New York, 1994.

[Sto96] M. Stonebraker. Object-relational DBMSs: the next great wave. Morgan Kaufmann, 1996.

[Sub94] V. Subrahmanian. Amalgamating Knowledge Bases. *ACM Transactions on Database Systems*, 19, 2, pp. 254-290, 1994.

[Tic85] W. F. Ticy. RCS - a system for version control. *IEEE Software Practice and Experience,* 15(7):637--654, 1985.

[Vin97] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2): 46-55. Feb. 1997.

[Wal91] L. Wall and R.L. Schwartz. Programming Perl. O'Reilly & Associates, Inc., Sebastopol, CA, 1991.

[Wid95] J. Widom, "Research Problems in Data Warehousing." *Proceedings of the 4th Int'l Conference on Information and Knowledge Management (CIKM)*, November 1995.

[Wie93] G. Wiederhold. Intelligent integration of information. *Proceedings of the ACM SIGMOD Conference* pp. 434-437, May 1993.

[Wil99] L. Will. SAP R/3 system administration: the official SAP guide. Sybix, 1999.

[Win93] J. Wiener and Y. Ioannidis. A Moose and a Fox Can Aid Scientists with Data Management Problems. *DBPL,* p.p. 376-398, 1993.

[Wir-URL] Web Interfacing Repository Manager Home Page. http://www4.biostr.washington.edu/WIRM.

Appendix A: WIRM API Reference

The functions exported by each of the six WIRM API's (FSA Controller, Visualization Cache Manager, are described in detail below.

A.1 FSA Controller

The FSA Controller uses a Perl DBM file called *WRM_FSA_INDEX* to associate File object IDs with file pathnames. The interface exports a function *file_path*, which encapsulates the index, allowing any file to be retrieved by ID.

Files can be imported into the FSA using either the *file_import_custom* function (which allows the user to specify a destination path within the FSA for readability), or the *file_import_default* function (which assigns a location automatically).

The functions are described in detail below.

`file_import_custom($fid, $fh, $custom_dest)`

Copies binary data from filehandle `$fh` into File Storage Area, using `$custom_dest` as the destination location, rooted in the FSA custom area. Saves the location in the WRM_FSA_INDEX dbm file. Returns FSA destination of imported file. Creates subdirectories in custom FSA area as needed. Returns 0 if import fails.
Parameter `$fid` will become the lookup key for retrieving the file contents, stored in WRM_FSA_INDEX. The `$fid` value should have been generated from repo_register_file (see repo.pl), which must be executed BEFORE calling file_import.

file_import_default($fid, $fh, $ext)

Same as file_import_custom, except it imports file into a default location within the FSA, based on hashing the `$id` value. The file extension ($ext) must be supplied.

file_path($fid)

Returns the full path (location + filename) to the file with the specified $fid. Note: this is currently just a lookup in the WRM_FSA_INDEX dbm file, but the file_path function will allow this method to change transparently if necessary.

file_default_dir($fid)

Specifies the default location for a file with the specified $fid. Note that this does not necessarily mean that the file actually resides there.

file_bucket($fid)

Encapsulates the hash function for computing the default location of a file. This is currently the file id divided by the value $FILE_BUCKET_SIZE. This allows files to be distributed across multiple directories, rather than having one single directory containing all files, which would become inefficient in some file systems.

file_mime_type($filename)
Returns the mime-type of the named file (based on the explicit file extension). First looks for recognized mime type in the repository's custom mime directory, then looks for it in the standard mime directory. Both are specified in the CONFIG file.

A.2 Visualization Cache Manager

The VCM is implemented as a Perl module called *viz.pl*, which uses a third-party module called *Image::Magick* [Mag-URL] to perform basic image manipulation. Image::Magick recognizes a wide range of image types, although users may extend the set of image types

by implementing their own conversion routines, such as was done with the GE MRI format. These extensions must be incorporated into the viz_convert function.
In addition to providing full-size JPEG replicas of images, viz.pl also provides functions for generating thumbnail versions.

Currently, the VizCache consists of three partitions: one for thumbnails, one for full-size images, and one for other file types. The VizCache location is specified in a configuration file, which allows the system to be customized according to the developer's requirements.

The location of a file in the VizCache is determined by the file ID (using a hash function), so that must be supplied when using the viz_get functions to request a file. Helper functions such as `viz_fullsize_dir()` compute the location. In addition, the virtual path can be computed with the `vdir()` functions (virtual paths are rooted at the hypertext document root, rather than the file system root, and are useful for generating Web Views).

Here are the details of the VCM API:

viz_get_fullsize($id, $source)

> Returns virtual path filename of full-size JPEG version of specified `$source` file in Web-viewable image cache. If the JPEG version does not yet exist, creates it using `viz_make_fullsize()`. `$id` parameter required for locating the converted file. `$source` must be a recognized image type (recognized either by Image::Magick or by one of the user-defined image types supported by this API).

viz_get_thumbnail($id, $source)

> Returns virtual path filename of JPEG thumbnail of specified `$source` file in Web-viewable image cache. If the thumbnail does not yet exist, creates it using `viz_make_thumbnail()`. Size of thumbnail determined by global variables

$THUMB_W and $THUMB_H. $id parameter required for locating the converted file. $source must be a recognized image type (recognized either by Image::Magick or by one of the user-defined image types supported by this API).

viz_make_copy($source)

Copies $source into Web-accessible VizCache "other" area. Returns virtual path filename of copy. If a copy exists in the VizCache with the same name, it will be overwritten. Note that unlike thumbnail and full-size areas, the "other" area does not maintain hashed locations based on file id.

viz_convert($source, $dest, $width, $height)

Converts $source file to $dest file. $source must be a recognized image type (see Image::Magick for types; also handles MR type). The extension of the $dest filename will determine the desired type to convert to. The converted file will be written to $dest. Optional $width and $height (in pixels) can be used to scale the result.

viz_mr_convert($source, $dest, $width, $height)

Converts $source file to $dest file. $source must be of type MR (GE Magnetic Resonance Image slice). The extension of the $dest filename will determine the desired type to convert to. The converted file will be written to $dest. Optional $width and $height (in pixels) can be used to scale the result.

viz_make_fullsize($id, $source)

Creates a full-size JPEG version of specified $source file in Web-viewable image cache using viz_convert(). Does not check to see if file already exists in cache (user should consider viz_get_fullsize to avoid overhead of converting files that already exist in cache). Returns virtual path name of resulting file. Directories along path will be created if they don't yet exist. $id parameter required for locating the converted file. $source must be a recognized image type (recognized either by

Image::Magick or by one of the user-defined image types supported by this API).

viz_make_thumbnail($id, $source)

Creates a JPEG thumbnail of specified `$source` file in Web-viewable image cache using `viz_convert()`. Does not check to see if file already exists in cache (user should consider viz_get_thumbnail to avoid overhead of converting files that already exist in cache). Returns virtual path name of resulting file. Directories along path will be created if they don't yet exist. `$id` parameter required for locating the converted file. `$source` must be a recognized image type (recognized either by Image::Magick or by one of the user-defined image types supported by this API).

viz_fullsize_dir($id)

Returns the real location (not including filename) of the Web-accessible cached full-size version of file specified by $id.

viz_fullsize_vdir($id)

Returns the virtual location (not including filename) of the Web-accessible cached full-size version of file specified by $id.

viz_thumbnail_dir($id)

Returns the real location (not including filename) of the Web-accessible cached thumbnail version of file specified by $id.

viz_thumbnail_vdir($id)

Returns the virtual location (not including filename) of the Web-accessible cached thumbnail version of file specified by $id.

viz_bucket($id)

Returns destination bucket number using hash function on supplied file $id.

A.3 Table Manipulator

The table manipulator is implemented in a file called *db.pl.* Since the RDBMS is currently MSQL [Hug-URL], the Table Manipulator is built on the *msql-perl adaptor* [Koe-URL] as the interface to the database server. Future version of db.pl will use the more generic DBI instead, which can interface with a wide range of database systems. Note that the db.pl buffers repo.pl from changes to the underlying database adaptor, so migrating to DBI should only minimally affect the exported interface.

As explained in the Architecture chapter, results of queries to the database are returned as *statement handles* by the Table Manipulator. A call to *db_select* executes the query, and the resulting rows may be retrieved using the *fetchrow* method. For example, the following code retrieves older male patients and prints out their names and ages:

```
$pats = db_select("Patient", "gender = 'M' AND age > 50");
while (@p = $pats->fetchrow()) {
  print "Name: $p[0], Age: $p[1]\n";
}
```

In the sections that follow, it will be shown how the Repository Object API improves on this interface by abstracting away the table structure.

Some of the Table Manipulator methods accept an optional "$filter" parameter. For these methods, the query is constrained by the filter string, which should have the following form:

```
column OPERATOR value
 [ AND | OR column OPERATOR value ]*
```

where OPERATOR can be `<, >, =, <=, >=, <>, LIKE, RLIKE` or `CLIKE`.

db_create_index($tablename, $columname)

Creates an index on the specified column that speeds up queries using that column.

db_select($table, $filter)

Returns statement handle reference to 2D array of selected rows from named table. If SQL-filter string present, applies it to where clause, otherwise defaults to all rows. Returns statement handle reference, from which each row can be accessed as an array of fields using $sth->fetchrow(). prints Msql error string if select fails.

db_select_row($table, $filter)

Returns array of field values. Selects single row from named table. Note distinction from ''db_select". If SQL-filter string present, applies it to where clause, otherwise defaults to all rows. If more than one row satisfies query, only the first is returned. prints Msql error string if select fails.

db_new_table($table_name, $column_string)

Creates new table named $table_name. `$column_string` specifies each column name followed by its type, which can be int, real, or `char(N)`. Spaces should delimit name and type, commas delimit name-type pairs. Prints Msql error string if creation fails (see Msql).

db_table_exists($table_name)

Returns 1 if table exists, 0 otherwise.

db_fields($table)

Returns reference to array of field names for specified table, ordered by column-order in table. Use db_types to get a corresponding array of the types of those fields. Returns 0 if table not found.

db_types($table)

Reference to array of type names for the columns of specified table. ordered by column-order in table. Each type will be one of: UNDEFINED INT CHAR REAL IDENT NULL TEXT DATE UINT MONEY TIME. Use db_fields to get a corresponding array of the names of those fields. Note: references to repos and files will appear as type INT at this level. Use the global hash `%WRM_SCHEMA_DEFS` to see the actual types at the repo level. Returns 0 if table not found.

db_fields_multi(@tables)

Returns reference to array of field names for list of tables, in the form ''table_name.field_name".

db_join_all($tables, $filter)

Create a join of all the fields of the specified tables. Return rows which satisfy the `$filter` clause.

db_join($tables, $cols, $filter)

Create a join of the specified columns from the specified tables. columns should be specified as ''table_name.column_name". Return rows which satisfy the `$filter` clause.

db_insert_row($tablename, $values)

Inserts a row into named table. `$values` should be a ref to a list of values whose types correspond to the columns of that table. The function adds the necessary quotes around string values, and translates empty numbers to be zero.

db_copy_into($dest, $source, $filter)

Queries `$source` table using $filter, then copies resulting rows into `$dest` table.

db_update($tablename, $setstr, $filter)

Updates record which matches $filter, using $setstr, which should be of the form:

column=value [, column=value]*

db_delete($tablename, $filter)

Deletes rows matching `$filter` from table.

db_drop($tablename)

Drops named table. All data will be destroyed.

db_connect($db, $host)

Connects to named database located on `$host` machine. Prints error message if connection fails.

db_date_now()

Gets current date in Msql date format, which is of the form DD-MON-YYYY, where MON is the common 3-letter abbreviation (e.g. "23-Jan-1999").

db_valid_type($typename)

Returns 1 if `$typename` is a valid database type (e.g. int, char, etc.), returns 0 otherwise.

A.4 Repository Object Interface

As explained in the Architecture chapter, the Repository Object API provides an object-relational data model to the CGI programmer by abstracting away the relational statement handles of the Table Manipulator and allowing the data to be viewed as collections of objects. The instances of an object type are maintained in a relational table, but the CGI programmer need not be concerned about the logical structure of the table representation, and instead operates on hashes whose values are bound to the row data.

The REPO API is implemented in the file *repo.pl,* which exports query functions that utilize the Table Manipulator to retrieve rows which satisfy an SQL filter clause, and then allocates an in-memory hash for each row. The CGI programmer may iterate over the results, performing arbitrary computations, and updates are propagated back to the table, where they persist between invocations.

Repo.pl connects to two Perl DBM files: *WRM_REPO_TYPES*, which maps every OID to a type name, and *WRM_SCHEMA_DEFS*, which records the attribute names and types for each object class.

The following section describes the major REPO API functions in detail. The functions are organized into five categories: object retrieval, object manipulation, schema definition, identity operators, and file access.

A.4.1 Object Retrieval

These methods are used to retrieve repository objects from the database into memory. As in the Table Manipulator, queries can be constrained by an optional filter string which will be applied as a WHERE clause in an SQL query. The filter which should have the following form:

```
column OPERATOR value
  [ AND | OR column OPERATOR value ]*
```

where OPERATOR can be <, >, =, <=, >=, <>, LIKE, RLIKE or CLIKE.

repo_get($oid)

Retrieves the object with the given OID from the appropriate database table and instantiates it as a hash whose keys are the attribute names and whose values contain the object's values. The hash values may be updated directly, but repo_update must be

invoked on the object to propagate the changes to the database. Returns a ref to the hash. If the object is not found, returns 0.

repo_lookup_id($type, $filter)

Returns the oid of a repo of the specified type whose values satisfy the filter clause. Returns 0 if there are no matches. Note: this does not instantiate the repo in memory, but only returns the OID. Use repo_query or repo_query_single to actually retrieve the objects. If there are multiple candidates, will return one of them at random.

repo_query($type, $filter, $order)

Repo_query returns a ref of list of refs to repos of the specified type whose values satisfy the filter clause. Returns 0 if there are no matches. The user may iterate over them using the standard foreach loop. See the synopsis for examples. The results may be ordered by specifying an attribute name in the optional `$order` clause. Comma-delimited multiple attributes may be specified for complete ordering, e.g.:

```
repo_query("Patient", "sex = 'M'", "last_name, first_name");
```

repo_query_single($type, $filter)

Repo_query_single returns a ref to a repo of the specified type whose values satisfy the filter clauses. Returns 0 if there are no matches. If there are multiple candidates, will return one of them at random. Use repo_query for getting multiple results. This function is provided as a convenience for the common case where you expect only one answer and don't want to be handed a ref to a list of refs.

A.4.2 Object Manipulation

The following methods are used to create, delete, and modify persistent repository objects.

repo_new($type, $template)

Creates a new persistent repository object of the type named by the string $type. A unique OID is assigned and returned. The optional `$template` parameter should be a ref to a hash whose keys are attribute names and whose values will be stored in the new object. Keys in `$template` which don't correspond to attributes of the schema will be ignored.

repo_delete($oid)

Permanently deletes the instance specified by the given OID. Removes it from the database and extracts it from the WRM_REPO_TYPES index.

repo_update($obj)

Takes a ref to an instantiated object and writes the object's values to the database. In-memory updates to a hash object are not made persistent until repo_update is explicitly called.

A.4.3 Schema Definition

The following functions are used to define and evolve repository object schemas. View definitions are handled separately.

repo_define($type, @atts)

Defines a new repo type named $type. `@atts` should be a list of strings defining the attributes, of the form (''name1:type1'', ''name2:type2'', ...). The type definition is stored in $WRM_SCHEMA_DEFS, and a table is created in the database. Note that attributes of type ''file'' and ''repo'' will be converted to ints in the database table (to contain OID's).

An ''oid'' column will be added to the table schema, and an index on oid will be created.

The `$type` parameter may designate a parent class for this object using the syntax

"parent:newtype". If a parent is designated, the child class will inherit all attribute names & types of the parent, in addition to the attributes named in the `@atts` list.

repo_evolve($type, @changes)

Use this function to add, drop, or rename the attributes of an existing schema. Instances of the class will be automatically updated to reflect the changes. `$type` should be the name of a schema, and `@changes` should be a list of strings describing the changes to be made. Changes can be:

add:name:type - add column with given name & type
drop:name - drop column with given name
rename:oldname:newname - change column's name

For example, the following invocation renames a Patient's "sex" attribute to "gender", and adds a new "age" attribute:

```
repo_evolve("Patient", ("rename:sex:gender", "add:age:int"));
```

Note: repo_evolve changes both the underlying table and the schema dbm index. The original table is copied into a temporary table then destroyed & recreated with the changed columns, and the data in the temporary table are then copied back into the new table.

repo_undefine($type)

Undefines the named class type by dropping the class's table and deleting its definition in the schema dbm.

repo_attributes($type)

Retrieves the attribute names & types of a given repo type. `$type` may be a typename or a repo object. Returns a reference to a list of strings, each with the form "attribute_name:attribute_type".

repo_attribute_names($type)

Retrieves the attribute names of a given repo type. `$type` may be a typename or a repo object. Returns a reference to a list of strings.

A.4.4 Identity Operators

The following functions can be used to translate between repos and their OID's, and to verify the type of a repo.

is_repo($v)

Takes a value of any type. returns TRUE if the value is a ref to a repository object, FALSE otherwise.

is_repo_group($v)

Takes a value of any type. returns TRUE if the value is a ref to a group of repository objects (e.g. the results of a repo_query), FALSE otherwise.

OID($obj)

Takes a ref to an object and returns its OID. If the parameter is already an OID, just returns the OID. This function is often used to make other repo functions more flexible, allowing them to accept both refs and OID's as parameters.

May also take a ref to a group of objects (e.g. the results of a repo_query). In this case, it returns a string of OID's separated by '+' signs. Useful for passing a group of OID's as a form parameter.

OBJ($oid)

Swizzles a repo or group of repos, as needed. Takes an OID and returns a ref to the associated object. Queries the database and instantiates a new hash in memory. If the parameter is already an object reference, just returns the reference. This function is often used to make other repo functions more flexible, allowing them to accept both

refs and OID's as parameters.

May also take a string of multiple OID's separated by '+'. In this case, it instantiates each OID and returns a repo group, which is a ref to an array of refs to repo objects. Useful for retrieving a group of repos from a form parameter.

repo_type($oid)

Returns the type name associated with the given oid.

A.4.5 File Access

The fields of the FDT are maintained in the relational database under a table called File. There are 3 classes of file that can be registered as repo objects: FSA, LOCAL, or REMOTE. FSA files are copied into the File Storage Area using the repo_import_file method, which has the advantage of assuring that the file will always be available for retrieval and that the file metadata will remain consistent with the file itself. LOCAL files are files accessible by the file system of the repository server, but are not copied into the FSA, so their metadata can become inconsistent if the file is moved or changed by an external agent. Finally, REMOTE files are designated by a URL. See gateway.pl for the implications of how these 3 classes of file are retrieved.

repo_register_file($fdt)

Use ``repo_register_file'' to create a repository object recording the metadata of a file. The file data itself is not copied into the repository. Use ``repo_import_file'' instead if data is to be copied. The $v parameter can be a reference to a File Description Template (FDT), or simply a string denoting the file's location. In the latter case, a default FDT will be generated. A new repository object is created, and the repo id is returned.

note: default assumption is that file to be registered is NOT in FSA. If it is in FSA, make sure $v->{domain} has value 'FSA'.

repo_import_file($fdt, $custom_dest)

repo_import_file should be used when the file data is to be copied into the repository's file storage area. The file description template ($fdt) must specify the file's source at a minimum (see repo_register_file for other attributes). An optional `$custom_dest` can be specified, in which case the given directory path will be created (if necessary) in the file storage area, otherwise a default location will be assigned. A new repository object is created, and the repo_id is returned. Returns 0 if the specified filename is unreadable. File is automatically registered. Do not preregister the file.

repo_import_file_from_handle($fh, $fdt, $custom_location)

Copies file data from specified file handle ($fh) into the file storage area, and registers the file metadata with the repository. Can be used with the CGI.pm file upload facility. An optional file description template ($FDT) can be specified (see repo_register_file), or a default will be generated. An optional `$custom_location` can be specified, in which case the given directory path will be created (if necessary) in the file storage area, otherwise a default location will be assigned. A new repository object is created, and the repo_id is returned. File is automatically registered. Do not preregister the file.

$v->{locator} or `$fh` must include name of originating file.

A.5 HTML Generator

The HTML Generator is a developer's interface that encapsulates some common user-interface constructs in high-level functions that emit HTML strings, allowing the rapid development of Web documents. The HTML Generator is based on the popular Perl module, *CGI.pm* [Ste-URL], extending it with some higher-level abstractions.

HT_date_choice($paramname, $default)

Generates a popup menu of the twelve month names (using 3-letter abbreviations), using `$paramname` as the parameter name, and having the optional `$default` value.

HT_diamond($label, $url)

Generates a hyperlink to `$url` using the specified label, bulleted by a red diamond-shaped graphic.

HT_end()

Emits the closing tags to end an HTML document.

HT_form_init($title, $action, $target)

Emits the header information for starting a form. The `$action` parameter should name a script to be executed when SUBMIT is pressed. The optional `$target` field names the target frame for the resulting action. Uses multipart form so file uploading will work. Makes background white. Uses `$title` parameter for document title.

Reads the current context from the 'context' CGI parameters, and assigns the context variables to the `%CONTEXT` hash.

HT_form_end()

Emits the closing tags to end an HTML form. Also emits `HT_carry_context()`, which carries the context CGI parameters to the next invocation so they don't have to be explicitly passed by the CGI programmer as hidden parameters.

HT_href($label, $url);

Generates a hyperlink to `$url` with specified label. Passes current context along URL, unless overridden by explicit mention of cx variables in URL.

HT_nav($label, $url, $space)

Generates a hyperlink to `$url` with specified label, surrounded by brackets, followed by a space delimiter `$space` pixels wide.

HT_img($src, $link, $width, $height)

Generates image tag using `$src` image. If optional `$link` is supplied, the image will be a hyperlink to that url. If optional `$height` and `$width` are specified, the image will be scaled (in pixels). Note: the image will not download faster if scaled in this manner.

HT_init($title)

Emits the header information for starting a non-form HTML page. Makes background white. Uses `$title` parameter for document title.

HT_list_diamond($items)

Generates a diamond-bulleted list of hyperlinks. The `$items` parameter should be a reference to a list of labels & urls of the form [$label1, $url1, $label2, $url2, ...].

HT_list_plain($items)

Generates a non-bulleted list of hyperlinks. The `$items` parameter should be a reference to a list of labels & urls of the form ($label1, $url1, $label2, $url2, ...).

HT_mail($address)

Emits a mailto: url to the specified address.

HT_nph()

Emits a non-parsed header. Scripts that use this header will send their output directly to the client browser's screen as it is generated, rather than buffering the output until the script completes. Useful for long-running scripts that generate periodic output.

HT_space($width, height)

Generates a space of the specified dimensions (in pixels). Useful for spacing form elements that would otherwise be scrunched together. Defaults to 15 pixels wide.

HT_table($header, $rows)

Generates an HTML table. `$header` should be a ref to an array of strings to be used as column headers. For example: [''Name", ''Age", ''Sex"].

`$rows` should be a ref to a list of refs to lists of cell values. For example: [[''James", ''44",''M"], [''Smith", ''28", ''F"]] For tables with a single row, a single ref to an array will suffice.

Default colors for rows of table will be alternating tan & gray. For control over the colors, alignment, and other table features, use the Table Template version described below.

HT_table($tt)

Generates an HTML table using a Table Template (TT). The TT is a hash ref with the following keys:

HEADER - A ref to an array of strings to be used as column headers. For example: [''Name", ''Age", ''Sex"]. If no HEADER value supplied, the table will have no headers.

ROWS - A ref to a list of refs to lists of cell values. For example: [[''James", ''44",''M"], [''Smith", ''28", ''F"]] For tables with a single row, a single ref to an array will suffice.

COLOR - a pair of color names (delimited by a hyphen) to indicate colors of alternating rows. Defaults to ''TAN-GREY". Also accepts ''NONE". Currently only supports GREY, TAN, and WHITE.

ALIGN - The alignment of values within their table cells. Can be ''LEFT", ''RIGHT", or ''CENTER" (default).

CELLPADDING - The number of pixels padding cell borders. Default is 2.

BORDER - The width of the cell borders. Default = 0 (no border).

Note that table templates may be re-used (i.e. store favorite template settings and then update the ROWS & HEADER fields).

HT_textify($string)

Removes all markup tags (bracketed values) from $string.

A.6 Gateway Interface

The Gateway Interface provides a high-level interface for displaying repository objects as Web pages, including methods for generating HTML tables of objects, and for generating form elements for choosing from a list of objects. There are methods for creating JPEG versions of image files, and for creating thumbnails and clickable image maps.

The Gateway interface provides methods for maintaining the Context-State as defined in the Query-By-Context view model. The context values are stored in a hash called %CONTEXT, using keys ''subject", and ''user", which are passed between Wirmlet invocations as parameters cx_subject and cx_user.

The Gateway functions are organized into four categories: Repo viewing, image handling, repo form processing, and system class definitions.

A.6.1 Repo Viewing

repo_view($subject, $function, $param)

repo_view is a type-independent function that emits an html string for visualizing repository objects. This can be used in place of the specific view functions for a given object type. `$subject` may be a reference to a single object or a group of objects to

be visualized. The optional `$function` specifies which view to use on the `subject(s)`. If `$function` is not specified, the default views are used for the given object's class (view_page for a single object, view_row for multiple objects). Third parameter ($parm) is an optional parameter to be passed to the view function if it takes one.

repo_view_label($obj)

repo_view_label is a type-independent function that emits a label for a given object. This can be used in place of the specific view_label function for a given object type.

repo_view_row($obj)

repo_view_row is a type-independent function that emits a row-view for a given object. This can be used in place of the specific view_row function for a given object type.

repo_table($obj_list, $view_class, $param)

Displays a group of objects in table form. `$obj_list` should be a ref to a list of objects (such as that returned by repo_query). The objects must all be of the same type. The optional `$view_class` parameter specifies which view function to use (the default will be repo_view_row applied to the given object type). The optional `$param` may be supplied to be passed to the `$view_class` function.

repo_vlink($subject, $view_class, $context)

repo_vlink generates a url to a view of the given $subject. Uses the WIRM repo-viewer facility. The optional `$view_class` parameter may be used to specify a non-default view of the $subject. The optional `$context` parameter may be used to pass a context to the view. Note: `$subject` may be a ref to a repo, or an OID, or the results of a query (a group of repos).

repo_mlink($class)

generates a url to the Make Object Wirmlet, using the given class.

repo_elink($subject)

generates a url to the Edit Object Wirmlet, using the given subject.

repo_login_link($subject)

generates a url to the User Login Wirmlet.

repo_explorer_link($class, $filter)

generates a url to the Explorer on the given class using the given filter.

A.6.2 Image Handling

repo_fullsize_jpeg($file)

Takes a ref to a File object or a File OID. Returns an IMG tag to a Web-visible full-size JPEG copy of the specified File. The image is converted to JPEG and copied into the visualization cache by the Viz facility.

repo_thumbnail($file)

Takes a ref to a File object or a File OID. Returns an IMG tag to a Web-visible thumbnail sized JPEG copy of the specified File. The image is converted to JPEG and copied into the visualization cache by the Viz facility.

repo_image_button($file, $button_name)

Creates a clickable image-button using the specified file as the source image, to be used for server-side image maps. The specified `$button_name` will be passed as a form variable when the button is clicked, and the coordinates of the cursor can be retrieved using $button_name.x and $button_name.y.

A.6.3 Repo Form Elements

repo_types_menu($param_name)

Generates a scrolling list of all possible repository object types, using the specified parameter name.

repo_choice($obj_list, $param, $type, $size, $multiple)

Given a list of repository objects ($repolist), generates a popup menu whose choices are labeled by the default labels of the objects. The OID's of the selected objects will be returned as results. The results will be named using the given parameter name. The form element can be changed from popup menu to scrollable list by passing the value ``SCROLL" as the third parameter. The viewable size may be set using the fourth parameter. By default, multiple choices are not allowed, but they may be allowed by passing the value ``MULTIPLE" in the fifth parameter.

repo_smart_prompt($paramname, $type)

Creates a form input element named `$paramname` appropriate for the attribute type given by $type.

repo_attributes_prompt($r)

`$r` can be either a classname or a ref to a repo or a repo template. Creates a table of form elements, one for each attribute of the class. If `$r` is a repo or repo template, the form elements will have default values using the attributes of $r. The form parameter names will be prefixed by ``att_". Use repo_attributes_parse to parse the form.

repo_attributes_parse($r)

`$r` can be either a classname or a ref to a repo or repo_template. Parses the CGI parameters created by repo_attributes_prompt. Stores the submitted results into the attributes of the repo and returns it. Note: calling environment is responsible for

committing the changes via repo_update or repo_new.

repo_banner()

Emits a header to appear at the top of every Web page. The banner should also provide a navigation bar and identify the repository being accessed. If the CGI programmer has created a function called "custom_banner" in the view definition file, it will be used by repo_banner. Otherwise, a default banner is emitted, which denotes the repository name and the user level.

A.6.4 System Class Definitions

The following standard class definition methods are defined for every built-in WIRM type (User, File, Annotation):

```
Class_view_label
Class_view_row_header
Class_view_row
Class_view_page
Class_make (and associated sub-methods)
```

In addition, the following methods are available:

File_view_icon($file)

Display an icon view of the given file, which serves as a button that leads to a page view of the file when clicked. If the file is a recognized image type, the icon will be a thumbnail version of the file's contents. Otherwise, a generic icon will be shown.

File_view_image($file)

Display a full-size image of the given file, which serves as a button that leads to a page view of the file when clicked. File must be an image type.

File_view_contents($file)

Displays the contents of a file. If the file is an image, displays the image using the Viz service (the image will be copied as a JPEG into a Web-visible image cache, whose URL will be specified in an IMG tag). If the file is a text type, the ASCII contents of the file will be emitted in place. For all other types, the file will be copied to a Web-visible location and a URL to that file will be provided.

Annotation_view_icon($subject)

If the given subject has an associated Annotation, returns a button that can be clicked to read an annotation. If the given subject has no annotation, a null string is returned.

Annotation_view_icon_make($subject)

Displays a button that can be clicked to generate an annotation. Uses the WIRM comment_add facility. The annotation will be associated with the given $subject.

Appendix B: Using WIRM to Build Portals

In addition to building experiment management systems for managing data within a single research group, WIRM can be used to build portals that manage data across multiple groups. In the same way that new technologies for collecting and evaluating scientific data have increased the complexities of laboratory information management, they are giving rise to an expanding legacy of internet-accessible research databases. As the range and scope of these resources threatens to become overwhelming for the individual user, there is an increasing need for standards that enable the integration of related but autonomous databases. Growing excitement over XML (eXtensible Markup Language) as a platform for "melding the Web into a seamless database" is evidence of this trend. However, the success of XML and other standards hinges on "the ability of professional societies and others to agree on what kind of information they want to share and how that information is to be structured". This process requires that collaborating investigators converge on a formalized shared vocabulary or global schema of their conceptual models, to which local data objects can be mapped. For geographically dispersed groups, this convergence is best achieved in the context of an active electronic community, in which researchers can compare their respective data models and discuss the semantic relationships inherent in their data. There exists an explosive commercial opportunity for providers of tools and services that facilitate electronic collaboration towards the development of domain-specific semantic standards for large-scale integration.

B.1 NeuroPortal: a Gateway to the Human Brain Project

The neuroscience community is a prime example of a field that could benefit from a collaborative environment for integrating informatics resources. Data about the brain exhibits an unparalleled degree of heterogeneity, as it ranges across the entire spectrum of

biological structure, from sub-cellular processes to neuronal systems to behavioral studies. Continued progress in understanding the brain is dependent on the integration of findings from all these levels. In an initiative to alleviate the information overload facing neuroscientists, the national Human Brain Project (HBP) was formed, supporting a wide range of independent projects to build computer systems for managing brain data. The Brain Project laboratories are supposed to integrate their systems and develop "querying approaches that will allow varied databases to be accessed with a single query, and retrieval of different types of data into a common information space" [Kos97]. A few sub-communities have formed, in which researchers have begun bottom-up efforts to integrate their systems with one or two closely related projects. However, as of yet, no comprehensive top-down integration strategy has been implemented, and the goal of a unified interface to the tools and data of the HBP is far from being realized. Many of the HBP-supported laboratories have proceeded in relative isolation, which is evident in the lack of interoperability between their tools.

Projects such as the Human Brain Project and The Human Genome Project are forerunners of the kind of multi-group collaborative environment that is becoming the norm in large science. In this section, it is shown how WIRM is well suited as a tool for performing Web-based systems integration for scientific consortiums.

As a prototype integration environment, WIRM could be used to develop a Web portal for the neuroscience community, which will provide structured access to the tools and databases of the 20+ Human Brain Project laboratories across the nation, and evolve as a hub for electronic collaboration. *NeuroPortal,* as it shall be called, would serve as a roadmap to the HBP's heterogeneous resources, and lay the foundation for their integration.

The three main objectives of NeuroPortal will be:

1. *Accessibility:* increase the accessibility of Neuroinformatics resources for HBP researchers and the neuroscience community as a whole, by serving as a structured, unified gateway to HBP software and databases.

2. *Integration:* accelerate the integration of HBP applications and databases by providing a central repository for comparing local data models, interfaces, and system architectures.

3. *Collaboration:* foster electronic collaboration between researchers by creating a central hub for hosting Web-based discussions and conferencing.

In terms of these objectives, I now describe the motivation for each of NeuroPortal's three components: Gateway, Knowledge Base, and Forum.

B.2 The Need for a Gateway

The past decade has seen a proliferation of techniques for studying the functioning of the brain, which has given rise to an exponential increase in the quantity and complexity of data collected. The models derived from this data are extremely heterogeneous in nature, and yet they are highly interrelated. For example, image data from functional magnetic resonance imaging can tie together behavioral studies with findings at a neuronal level. Tens of thousands of researchers are generating an ever-expanding legacy of data, which is impossible to keep track of by the individual scientist, whose scope is becoming narrower as specialization increases. This trend "threatens the ability of scientists to make conceptual links across different areas of study... which is precisely the fuel that has driven much of the progress in our understanding of the brain and behavior" [Kos97].

In 1989, at the request of NIMH and other federal agencies, a committee of experts in brain, behavioral, information and computer sciences was formed to address this problem

[Com91]. After two years of deliberation, they recommended an initiative to develop an international resource of network-accessible databases about the brain. In 1993, the Human Brain Project was announced, with the long-term goal of leveraging advances in information science to help manage the overload facing neuroscientists. The initiative promised to "alter fundamentally the ways in which newly acquired scientific data are made available to the scientific community, exchanged with collaborators, and used for purposes of education and clinical activities" [Com91]. Phase I supported the development of a wide range of independent projects for visualizing, simulating, and mapping knowledge about the brain. The HBP has now entered Phase II, in which these technologies are supposed to be woven together and made available to scientists, clinicians, and students. In response to this goal, a number of groups have begun to propose strategies for integrating their systems with one or two closely related projects. However, the lack of a central resource for comparing data models, formats, and tools has impeded progress towards large-scale integration.

The official Human Brain Project Web server contains a paragraph summary of each project, and a list of 21 links to the research groups' independent Web sites. However, each of the sites uses its own ad-hoc structure, making it difficult to compare applications across groups, or to find a specific resource without a priori knowledge about its location. Furthermore, there is no way to browse projects by category or view a complete listing of available tools and data sources. The maintainer of the official HBP site, Dr. Floyd Bloom, agrees with this assessment, and has agreed to serve as a consultant on this proposal to develop NeuroPortal.

Other gateways to neuroscience resources exist, such as the Neuroscience Virtual Library and Neuroguide, which provide alphabetical collections of links to neuroscience databases, journals, organizations, and laboratories. However, none of the existing gateways is knowledge-base driven, nor do they provide facilities for electronic collaboration. By initially focusing on the Human Brain Project, rather than the entire

scope of neuroscience resources, NeuroPortal will be able to provide a much deeper interface than these "catch-all" gateways. The structure of the Gateway will be encoded as classes in the Repository Schema Model, and the front end will be built using WIRM's Gateway interface.

B.3 The Need for a Knowledge Base

At the heart of the integration problem is the process of identifying common semantics across the systems to be integrated, and mapping them to a central representation. To integrate an existing collection of resources such as the HBP databases, the process of converging on a global schema must begin with a survey of the local database schemas, followed by a collaborative effort to classify them into a conceptual hierarchy. When an acceptable formalization is developed, the schema of each local database can be mapped to the global schema, allowing the construction of *wrappers* that translate data between sources. Once this has been achieved, it is possible to retrieve data from multiple sources using a single query interface.

The biggest challenge facing such an endeavor is the presence of *schematic conflicts* between the local schema of the independent databases, which naturally arise from uncoordinated representations of a given real world object. In addition to the obvious case of *naming conflicts* (in which semantically equivalent classes or attributes are assigned different names), there are a number of deeper issues which must be addressed during schematic integration or view definition. For example, *generalization conflicts* are a type of schematic conflict in which a class or attribute in one database subsumes multiple classes or attributes in another database.

Before schematic conflicts can be identified and resolved, the conceptual models of the target applications must be made explicit in a formalized symbolic representation. NeuroPortal's Knowledge Base will be a central depot for collecting and evaluating those

models, allowing researchers to identify semantic correspondences between projects, thereby facilitating the construction of wrappers and translators.
In the long term, the Knowledge Base will lay the groundwork for researchers to achieve a consensus of prototype classes, from which neuroscience applications can inherit common data structures and interfaces. A classic example of this "shared library" approach is the Image Understanding Environment, an ARPA project to develop a common object-oriented software environment for facilitating the exchange of research results within the Image Understanding community. Alternatively, the Knowledge Base could serve as the basis for the definition of a set of XML descriptors, allowing Web-based project data to be tagged for automatic integration via a Web robot.

WIRM's high level API's are well suited for building a Web based front end, which would allow researchers to update the knowledge base remotely.

B.4 The Need for a Forum

The integration of multiple heterogeneous applications can only occur in concert with the adoption of interoperability standards. The standards can take many forms: shared vocabularies, global schemas, common file formats, communication protocols, or a distributed object framework such as CORBA. For a loosely federated consortium such as the Human Brain Project, the standardization problem is characterized by the need to form a consensus while still maintaining a large degree of autonomy. For geographically dispersed groups, this convergence is best achieved in the context of an active electronic community, in which multimedia communication tools are available for sharing models and discussing semantic relationships inherent in the data. An electronic forum would promote cross-fertilization between geographically dispersed research groups, allowing the formation of special interest groups of researchers using similar software or working on related problems. It would open a new channel of communication for posing questions and broadcasting announcements. Unlike email-based collaboration, a Web

discussion facility would provide a more structured forum, allowing new users to join in on existing discussions at any time, and view entire archived discussion threads. WIRM's annotation facilities could be expanded to include a Web discussion facility.

Currently, there is a shortage of tools and services for facilitating collaborative data integration. There are no commercial enterprises devoted to the creation of domain-specific Web portals to enable knowledge-based integration of scientific resources. As an application server aimed at managing scientific research data over the Web, WIRM is the perfect tool to fill this niche.

Appendix C: Deblobification

A multimedia object that hasn't been filtered for symbolic content is considered a binary large object (BLOB) by the query processor. I have coined a term, *deblobification,* to describe the process of modeling the symbolic content that can be extracted from multimedia objects. Deblobification can be applied to any digitized object, regardless of type (e.g. images, laboratory instrument readings, spreadsheets, text documents, etc.). Note that the concept of deblobification refers to the *modeling process,* not the actual feature extraction that is performed on a particular object. A schema can be deblobified to support automatic feature extraction or user-specified classifications.

In practice, deblobification often occurs in stages of stepwise refinement. For example, an image database may originally begin with a collection of files and the standard metadata (file type, size, label, etc.). Eventually, users may wish to retrieve files based on keyword, so an extra attribute will be added to the schema for supporting keyword annotations. Later, the files will be classified into distinct categories, so the schema will again evolve to include a category identifier. At some point, the system may be extended with a feature extractor, in which case the file schemas could be augmented to store the results of automatic feature extraction for indexing purposes.

Multimedia data contain far more real-world information than traditional symbolic data (a picture contains a thousand words), and by preserving the multimedia content (rather than storing only extracted symbolic information), a system offers much richer prospects for integration and information reuse. In the traditional experiment process, a scientist extracts the information of interest and discards the extraneous multimedia. The resulting data elements are much less likely to be useful to another scientist, and can only be integrated with data that closely match the particular symbols that the original scientist was interested in. For example, consider a medical researcher's experiment that consists of analyzing MRI slices to count tumors. The experimenter may create a database of

specific details extracted from the data, such as statistics about tumor size and location. Such a database may be useful for integrating with other systems, but its value is limited by the subset of real world concepts that the experimenter chose to model. However, if the database includes the actual MRI slices, then external users have access to the full spectrum of information embedded in the multimedia, and can perform their own analysis (e.g. shape analysis of tumors).

As another example, consider the field of astronomy, in which terabytes of telescope data are stored as files in huge databanks. The files contain information whose need is unanticipated at the time of recording. Recently, an asteroid was detected whose trajectory seemed to put it on a collision course with earth. A larger set of data points was needed to refine the trajectory interpolation. The scientists were able to mine the historic telescope data to find earlier indications of the asteroid, and were therefore able to compute a more accurate trajectory (the earth was safe). The key point is that the information about the asteroid was embedded in the multimedia recordings, unknown to the scientists who had recorded it. The asteroid recording was a "diamond in the rough", embedded in the binary data. This type of "BLOB mining" will become more common as the ability to record and store multimedia events continues to grow.

The WIRM facilitates data integration, data discovery, and repeatability studies by providing support for storing multimedia objects in addition to the symbols extracted from them, as well as providing the ability to evolve schemas over time, to account for new attributes of interest. Rather than extracting symbolic information and discarding the blob, the WIRM methodology encourages the storing of blobs for future use, and the implementation of evolving filters which can mine symbols from blobs as they become of interest.

For each data element in a system, there exists a "home" source, in which it resides, a "parent" source, from which it came, a collection of one or more "parent objects" from which it was derived, and a set of "children objects" who derive their values from it. From the perspective of a data integrator, the most recent home of a data element is the source for that element, but it would be useful to be able to trace the genealogy of data objects to guide the process of further data evolution. By travelling backwards through the "information food chain" [Etz96], one eventually reaches a "conception point" where the data object (or some attribute of it) was first translated from a Real World Entity (RWE) into a recorded medium (e.g. hand-written observation, photograph, sound recording, MRI scan, etc.). For example, consider a patient's brain with language sites marked on it. A photograph captures the RWE in a recorded medium, after which it may go through a number of non-digital transformations before it is finally digitized, entered into the EMS, and then deblobified. The order in which these phases occur can be interleaved. For example, deblobification can occur by physical means before the multimedia object is entered into the system, so the system doesn't necessarily retain the multimedia.

Consider the information stored in a real world event, e.g. a surgery, which is infinite:

$$I(RWE) = \infty$$

Now consider the information captured in a recording of the event, e.g. an intraoperative

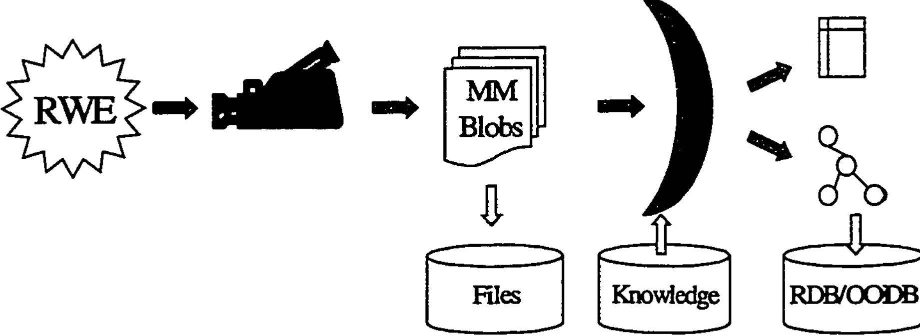


Figure 65: Knowledge-Based Deblobification

photograph, which is a subset of the information in the RWE, limited by the recording process (resolution, sampling rate, mode, viewpoint, field of view, proximity, etc.):

$I(RWE) \supset I(REC)$

The information in a BLOB as entered into a Multimedia system (e.g. a digitized photograph) may lose some of the information in the original recording, due to limitations in the digitization process (e.g. compression).

$I(REC) \supset I(BLOB)$

Information available to Query Processor is limited to the amount of deblobification that has been performed, or that the query processor is able to perform on the fly:

$I(BLOB) \supset I(QP)$.

The amount of refinement of a system's multimedia information can be expressed as the ratio of the query processor's information over the information contained in the blobs I(QP)/I(BLOB).

Vita

Rex Matthew Jakobovits was born on August 26, 1968 in Champaign, Illinois. His family moved to Honolulu, Hawaii in 1970, where he was raised. He attended the University of Hawaii and then transferred to Rutgers University in New Jersey, where he received his B.A. in English and Computer Science in 1991. He worked as a software developer in Pennsylvania for a year before entering graduate school at the University of Washington, where he completed his Ph.D. in Computer Science in 1999.