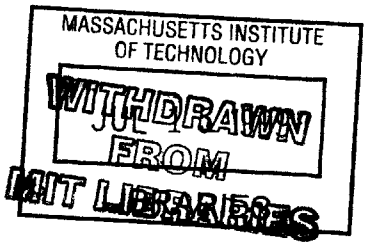
Designing Object-Oriented Interfaces for Medical Data Repositories

by

Patrick J. McCormick

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREES OF
BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING
AND
MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUNE 1999

ENG



Signature of Author: .....
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by: .....
C. Forbes Dewey, Jr.
Professor of Mechanical Engineering
Thesis Supervisor

Accepted by: .....
Arthur C. Smith
Professor of Electrical Engineering and Computer Science
Chairman, Department Committee on Graduate Theses

Designing Object-Oriented Interfaces for Medical Data Repositories

by

Patrick J. McCormick

Submitted to the Department of Electrical Engineering and Computer Science on May 21, 1999 in Partial Fulfillment of the Requirements for the Degree of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Repositories of medical data are predominately limited to crude interfaces that limit the flexibility of data mining applications and interactive query tools. This thesis explores different methods for providing an object-oriented query interface to a complex data source, with an emphasis on medical and genetic databases. This project designed a query interface to meet the goals of simplicity and extensibility. The query interface is intended to provide object-oriented access to any data source. Various methods and technologies were used in the design process, including CORBA and XML. A prototype server was implemented using the Java programming language to demonstrate generalized access to object-relational and relational databases. A prototype client was implemented as a Java Web Servlet to demonstrate the distributed nature of the interface. These prototypes were tested against a large database of magnetic resonance images using the DICOM information schema. We found that the system provides simple access to complex information schemas and is a useful tool for exploiting complex structured data, specifically in the medical and genetic domains.

Thesis Supervisor: C. Forbes Dewey
Title: Professor, MIT Mechanical Engineering Department

Table of Contents

List of Figures

List of Tables

Acknowledgements

No scientific research is done in a vacuum, and this thesis is no exception. Throughout this research I have received invaluable advice from several colleagues in my research group: Professor Forbes Dewey, for his continuing advice and support; Ngon Dao, for building the foundations of this project and providing patient advice; Matthieu Ferrant for his assistance and insight; and Donna Wilker, for her administrative support and kind ear.

I would also like to thank the members of the Brigham and Women's Surgical Planning Lab for their technical assistance, discussions, and support throughout this project. Thanks to Dr. Charles Guttmann, Simon Warfield, Mark Anderson, Ted Zlatanov, and David Gering.

Thanks to Informix Software, Inc., in particular Bob Hedges of the Chief Technology Office. Informix provided support for this project in the form of software, technical support, and early entry into the JDBC driver beta test program.

This project was supported by the National Institutes of Health Training Grant in Genomic Sciences #HG00039.

Finally, I thank my family for their support over the past five years. Thanks to Mom, Ralph, Annie, and Dad.

Introduction

The medical industry processes an enormous amount of information every day. Lab reports, prescriptions, x-ray films, ultrasound videotapes, and many more types of information fill every patient's medical record. It seems obvious that modern computer systems should be integrating into hospitals to relieve this information burden. Yet computer-based patient records (CPRs) are still a far-off goal for many hospital information systems. In the 1998 HIMSS Leadership Survey, information technology executives listed their top three priorities as recruiting staff, integrating heterogeneous systems, and deploying a CPR, respectively.[1]

The academic medical informatics community has worked for several decades to deal with the issues involved in creating, deploying, and maintaining CPRs. In this time, many innovations such as object-oriented methods, faster hardware, and better networking have eased some of the difficulties in medical informatics. However, technology alone cannot solve the core difficulties which keep reappearing again and again.

Medical informatics research shows that the root cause of many CPR difficulties lies in the inability for different medical record systems to communicate with each other,

a sentiment echoed in the literature[2, 3]. This problem occurs despite years of work invested in industry-wide data exchange standards. The theory of this thesis is that if well-defined interfaces are developed using existing standards, integration will be much easier.

To test this, my research group embarked on a project with Dr. Charles Guttmann's group at the Brigham and Women's Hospital to convert an existing pool of MRI files into a database system capable of interacting with users and other programs on a very high level. The promise is that this kind of interface can be used to implement data mining, metadata, and other advanced features.

The true value of this work is in fully describing the problems from a general informatics viewpoint, enumerating potential solutions, and analyzing the advantages and disadvantages of each through implementation and deployment.

Chapter 1: Background

Medical informatics covers many fields, including biology, medicine, information theory, mathematics, computer science, and software engineering. The major theme that runs through medical informatics research in all of these fields is cooperation. In order for different systems to interoperate, they must agree on communications standards for all levels. These standards cover everything from the language used by physicians to describe day-to-day encounters to the bytecodes exchanged between computer servers and clients. The research for this project focused on the efforts of the medical informatics community to reach agreement on these standards. We begin with a look at the attempts to codify the unique and precise vocabulary used by physicians.

1.1. Semantic Encodings

In order for different individuals in a profession to communicate in a given language, they must agree on a common vocabulary. The long history of medicine has generated a wide variety of different ways to describe similar concepts. These concepts include metrics, diagnoses, diseases, pharmaceuticals, and many other ideas that require precise description.

Using different vocabularies to describe the same semantics is known as a "semantic collision." A semantic collision makes it difficult for physicians to communicate, and makes it nearly impossible for computer systems to exchange information. To keep these collisions to a minimum, several standards are available that can be used to encode semantic concepts in an unambiguous format.

One well-known collection of descriptive terms, with more than 10,000 defined elements, is the SNOMED lexical system. In the United Kingdom, the popular Read system has more than 30,000 entries. Still another parallel classification system is called ICD-9. Each of these systems reduces the potential confusion caused by using different terms for the same idea[4].

For the domain of diagnostic medical images, an important milestone was achieved in 1993 with the establishment of the Digital Communications in Medicine (DICOM) standard for medical images[5]. This standard has evolved to cover magnetic resonance, computed tomography, x-rays, x-ray angiography, ultrasound, nuclear medicine, waveforms such as electrocardiograms, and visible light images such as laparoscopic and pathology images[6]. In addition, the DICOM standard has been extended to include structured text and semantic references[7].

There is also a standard information model for the general clinical information domain. The Health Level Seven (HL7) organization was formed about ten years ago to establish interchange standards in the clinical information domain. The initial focus was on billing and administrative functions. More recently, the group has developed a much more comprehensive data schema that includes clinical information as well. This new

effort, HL7 Version 3, has many attractive features including some very advanced object-oriented data exchange concepts. This standard allows systems to easily exchange information with clinical information providers without unnecessary ambiguity.

There are other heterogeneous information domains that have been developed to serve a variety of different medical and biological specialties. The National Library of Medicine has established the Uniform Medical Language System (UMLS) to identify the different data elements that are used by all of these systems. Where possible, they have found cognates between the elements in different semantic domains. Even with this effort to identify semantic collisions, the total number of unique information objects in the UMLS is over 350,000.

Research on the human genome has produced a very active informatics domain that deals with storing and analyzing genetic information[8]. Geneticists need to access and share data between many different databases. They also need tools that will help organize these data, and identify sequences from fragments obtained in different laboratories.

These standards allow researchers to create information architectures that are prepared to evolve gracefully, remaining unambiguous as new concepts enter the physiological domain. These standards ensure that these architectures can exchange basic concepts with other databases that have otherwise incompatible information models.[9].

1.2. Data Exchange

Once the content can be encoded, the next step is to exchange this content across various types of computer systems. When a system specification is written, it is in many

cases clear what data is expected as input and what data will be returned as output. There are instances, however, where a given data type is not sufficient to capture the semantics of the information being exchanged. For example, as interfaces become more complicated, users will need to query the models for information about themselves. This information, known as metadata, allows the model user to learn the model's context and the assumptions upon which it rests.

Many operations return data that is more complex than a simple number or a small string. Complex data cannot be interpreted unambiguously without being accompanied by contextual information. To develop an effective system for exchanging complex data, the appropriate industry standards can be used. The proper use of standards is to produce a common denominator for an abstract concept such as images or structured text[3]. Standards should not impede intellectual progress, but enable the creation of new knowledge[10, 11].

1.2.1. Structured Text as an Exchange Medium

One means of exchanging data between components is structured text. There are two major challenges in exchanging structured text. First, there needs to be a standard framework for describing the structure of the text. This problem is best addressed with the Extensible Markup Language (XML) standard[12].

XML allows users to structure their data in a hierarchical format. Using a Document Type Definition (DTD), a structure can be defined for a given domain. Documents in a domain can be verified for correctness against the DTD of the domain. Object-oriented structures have been shown to map directly to the structure of XML[13],

which is necessary for XML to be used easily with object-oriented systems. XML is easier to use than its predecessor, the Structured General Markup Language (SGML), which has been used as the basis for other structured reporting projects[14]. XML has the potential to become the common language spoken between components, with object-based DTDs providing message context.

The clear role for XML is as a transfer syntax for any type of data. XML's potential comes from its nearly infinite flexibility and portability.

A framework for exchanging textual information is necessary but not sufficient for data exchange. The textual information needs to be semantically consistent. Towards this goal, data exchange systems can leverage the semantic domain standards described above.

1.2.2. Metadata Exchange Standards

Sometimes components do not require raw data from a medical repository, but only wish metadata about the object. Similar problems arise on the World Wide Web when clients do not wish an entire document, but only keywords and index information. To address the need for a document metadata exchange standard, the World Wide Web Consortium (W3C) developed the Resource Data Framework (RDF).[15] RDF is expressed in XML, which resolves the low-level processing issues. RDF is designed to express metadata as a graph of resources linked by directed assertions. The model that RDF uses for expressing relationships is a useful reference for developing ways for components to describe the relationships between different data objects.

1.3. Encoding Relationships

Establishing a structure for data exchange merely allows computers to agree on exchanging information that maps to semantic concepts. Systems need to also exchange information on the relationships between these concepts.

Current electronic medical record (EMR) systems are capable of storing and recalling large amounts of original medical data. However, they are often unable to properly store additional data derived from the original data. Derived data is often stored with no apparent link to the original data. This implies that the derived data is original data, which is not the case. This distinction is significant for physicians reviewing medical records. Physicians need to be able to refer back to the original data to properly evaluate the derived data.

An example of "derived data" is a text report written about an X-ray. The report is based on the X-ray film, and has references to particular features of the image. While the film is objective data from a machine, the report is a subjective analysis performed by a single physician. If the patient wishes a second opinion, the next physician will want to see the original X-ray, the analysis, and the relationships between the two. Without all three of these components, the medical record is incomplete.

The difficulty for standard EMR systems is that it is hard to store the original and derived data in such a way that retains the relationship. Also, there is no convenient way to store references from one document to the other. What is needed is an EMR system that preserves these relationships in such a way that they are easy to search through. Also,

these arbitrary relationships between otherwise unrelated classes should be treated similarly to established relationships which are a part of the information model.

One of the problems with current EMR standards is that relationships between objects in the model are often poorly illustrated.[11] This means that if a "patient" object from one standard is converted to another standard, the relationships between the "patient" object to other objects are lost. A system for making it explicitly clear how one object is related to another is necessary in this case.

Relationship management is crucial for enabling components to fully exchange medical record data.

1.4. Databases in Medicine

Once systems can correspond with each other using standard semantics and communication protocols, they need to find a persistent location to store the data. The solution to this problem is the database.

Medicine has used databases to store clinical information since the earliest mainframe systems became available. This has always been an ill-fitting proposition, however. Database systems are never flexible enough to include all the data generated by clinical encounters. Even modern database systems fail to meet the complete needs of the medical community.

To appreciate the difference between database types, the concept of a data model must be understood. Fundamentally, data is an undifferentiated mass of bits. An abstract model is needed to organize data views and optimize access. However, the process of

fitting data to a particular abstract model restricts the flexibility of the data. More flexible data models are unfortunately also more complex. As computer power and information theory have advanced, more complex data models have become possible.

Data models support information models, which define the way that data is stored for a particular domain. For example, a human resources database for a company would include the concept of "employee" and "manager" in its information model. Information models are very domain-specific, and often vary widely within domains.

1.4.1. The Hierarchical Data Model

One of the earliest data models is the hierarchical data model. A hierarchical database stores data in a tree-like structure. Each node of the tree contains a set of fields which contain data. Children can only be accessed through their parent nodes, which limits the ways in which users can access the data. One of the first widespread medical data systems, the Massachusetts General Hospital Utility Multiprogramming System (MUMPS), used the hierarchical data model to store information.[16]

1.4.2. The Relational Data Model

The relational data model was proposed originally by Codd[17] in 1970, and is the prevailing data model in use today. It views data as a set of tables composed of rows. Each row (also called a "tuple") contains a set of columns. Each column has one of a limited set of data types. Columns in different tables can be "joined" together to form powerful queries.

Before the relational data model, databases used "flat files" which were basically single tables. The crudest form of flat file databases contain only two columns; a "key" and a "value." This concept provided the basis for Codd's relational database concepts.

While the relational data model is mature and reliable, it has shortcomings when the data are complex[18]. The most important shortcomings are the lack of nested information entities and the limited set of supported data types. These two limitations preclude the storage of unstructured data such as images and audio.

1.4.3. Object-Oriented Data Models

Object oriented databases (OODBs) use a data model that incorporates unique object identifiers, data encapsulation, and inheritance. There are many advantages of the object-oriented data model over that of the relational model, but most commercial OODBs suffer from non-standardization. This lack of standardization has resulted in the development of separate and proprietary data models and interfaces. The various object-oriented data models lag behind the relational data model in transaction management, replication support, and development environments.

There are many instances of object-oriented databases being used in the life sciences. At MIT, the Whitehead Institute designed an object-oriented system for storing data related to the Human Genome Project.[8] This database required frequent schema changes and an interface that scientists were comfortable with.

In the field of genetics, one of the most popular databases is ACEDB. This database uses an object-oriented data model and is specifically designed for storing

biological data about genomes. The object-oriented structure allows for a flexible information model and efficient storage.[19]

Peter Gray describes an object oriented database for storing protein models [20]. This was based on a semantic data model that separated the query interface from the actual implementation, a concept that we will revisit in the next section.

1.4.4. The Object-Relational Data Model

The object-relational data model (ORDM) was developed by Stonebraker and Kim, independently, in the 1990's.[21, 22] The ORDM combines the best of both the relational model and the object-oriented model. In an ORDM, columns can now contain many different types of data instead of the limited set offered by the relational model. Columns can contain entire rows or even user-defined types.

The ORDM exploits relational data model technologies and standards because the model is based on relations. Algorithms developed for concurrency control, transaction management, and query optimization are all valid within the ORDM framework because the notion of relations and tuples are preserved. Another advantage of the ORDM is that it supports dynamic data typing, which allows users to generate new data types and structures on demand.

A key advantage of the ORDM is that the industry-standard query interfaces can be used with it. The Standard Query Language (SQL) was designed for relational databases. While SQL has been extended by the various object-relational database vendors to accommodate user-defined data types and inheritance, it still remains tightly bound to the concept of rows with columns. The industry-standard Open Database

Connectivity (ODBC) and Java Database Connectivity (JDBC) interfaces have been extended to deal with the different results that object-relational SQL may return.

Object-oriented databases are centered around the ability to manipulate persistent objects inside a database. Object-relational databases are unfortunately forced to manipulate objects through the SQL interface that views all objects in the database as simple tables. Programmers are forced to write "glue" instructions for packing and unpacking classes from a database, attribute by attribute. Besides the code overhead, there is an impedance mismatch between the data types used by SQL and the data types used in the programming language used to create the object which only serves to make the process more difficult.

The new SQL-3 standard[23] promises to solve some of these problems by allowing a tighter mapping between SQL row types to user-defined classes.[24] Still, these improvements do not solve the problem of creating a transparent object interface to a database.

1.5. Modern Software Engineering

A complicated system of software must be designed and tested to tie together the data, the databases, and the underlying semantics. For this need, medical informatics turns to software engineering.

A cornerstone of modern software engineering is object-oriented (OO) theory. When object-oriented ideas first began to appear in the 1960's, its proponents promised sharable code, modular programs, and faster development time. Due to the gap between

theory and implementation, we are only just beginning to see these fruits of OO software development. Medical informatics has been a staunch supporter of OO from the start, because medical informatics problems are so much more complex than traditional software challenges.

1.5.1. Object-Oriented Component Frameworks

Component frameworks have long been a topic of research and discussion in software engineering[25-27]. Software components are practical implementations of object-oriented software theory. While it is easy on an abstract level to describe objects communicating with each other, in actual practice this requires a great deal of thought and agreement, especially if the objects in question are written by different designers.

There are currently several different standards for component frameworks. The widest industry support is given to the Common Object Request Broker Architecture (CORBA), developed by the Object Management Group. Microsoft supports the Component Object Method (COM) standard. The Java language has also incorporated a component architecture, in which Java components are known as "JavaBeans." Each of these standards has in common the concept of an "interface". Writing the actual action of a component is easy. The problem is letting other components know how to use this action. By publishing an interface, it is easy for other programs to call on a component and ask it to perform actions and return results. Also, different components can support the same interface, which means that the calling program does not need to be updated to be used with a different implementation of the same interface.

In a large system of components, a certain amount of code is devoted to "business logic." Business logic is the set of rules that tell the system how to perform business functions. This is in contrast with the code that tells the system how to access data and perform other low-level tasks.

For CPRs, the business is medicine, so the business logic is embodied in the medical data schema and related functions. A strong, extensible system allows for a distinct separation between application and business logic. Software component standards, using interfaces as a central tool, allow developers to separate the business logic from the application logic.

1.5.2. Interface Hierarchies

Interfaces can be arranged into an interface hierarchy, which presents a standard means for using real-world concepts in an application. Components are implementations of one or more of these interfaces. Using components, programs can be written not for specific implementations, but for an interface to an abstract concept. For example, a program for displaying patient data would be written to use a patient object interface. This program could run with patient information database as a component that supports the general patient interface, instead of having to be rewritten to deal with the idiosyncrasies of individual information systems.

These interfaces are object-oriented, giving our framework the advantages of inheritance. Past projects have successfully built complex object-oriented schemas, as long as the overall view remains consistent and the development protocol is clear[28].

Using components with interfaces allows researchers to choose from a variety of data views. It is easy to have a single component support multiple interfaces. A program written to gather data from two different interfaces does not care if the interfaces are two different components or a single *über*-component. Using multiple interfaces is similar to using multiple database views, where raw data in a database can be browsed in multiple contexts.

Comparing interfaces to a database view leads us to ask, why not apply interfaces to the database environment directly? We can treat databases as a set of components

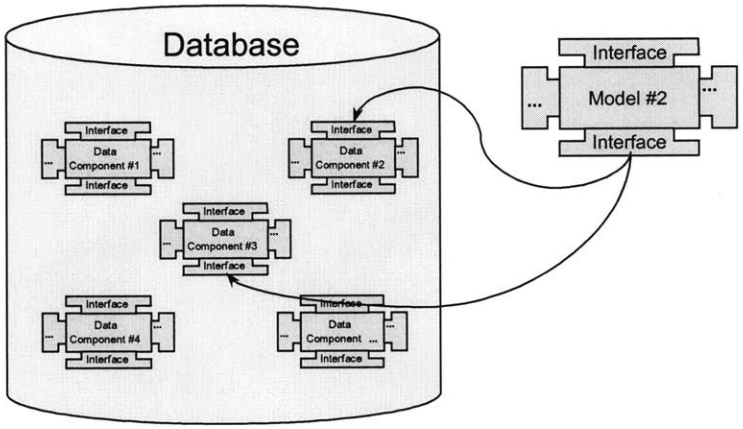


Figure 1-1: A database interface tree

supporting the same interface as the models.

Figure 1-1 illustrates an example of a database with component interfaces.

1.5.3. Building Interfaces with Design Patterns

A hierarchy of interfaces can be constantly evolving. The framework should support different versions of interfaces, and be able to define interfaces on demand. However, we need to rely on the interface designers to support a graceful evolution of interfaces through careful initial designs. Interfaces that age well are a product of careful

design according to preset principles. Software engineers have been working on the issues behind component interface design for some time, and have codified the elements of good design as "design patterns." The canonical catalog of fundamental design patterns was authored by a group known as the "Gang of Four", also known as the GoF.[29]

Design patterns are derived from pattern languages developed to express the architecture of buildings. Instead of having to build each component system from scratch, component architects refer to listings of design patterns that implement basic object-oriented concepts. Design patterns are not tied to any specific language, but can be used with any kind of object-oriented system.

1.5.4. Explaining Interfaces with the Unified Modeling Language

Software architects need a common language to express the abstract object-oriented concepts they are attempting to implement in data and functional models. Starting in the 1980s, object methodologists created a variety of different methods for analyzing and designing systems. By the late 1980s, there were several different object methods, each with strengths and weaknesses. The three major methods were unified by Grady Booch, Ivar Jacobsen, and James Rumbaugh to form the Unified Modeling Language, known as UML in 1996.[30]

The UML provides a common language and syntax for describing five different views of a system:

- Use Cases – exposes requirements
- Design View – reveals the problem and solution spaces
- Process View – models the processes and threads

- Implementation View – describes the physical layout
- Deployment View – models system engineering issues

The UML authors liken these views to the different layouts used in construction; a floor plan, electrical layout, plumbing, HVAC, and so on.[26] While the entire project was not modeled in the UML, we used UML notation and concepts to describe the goals, design, and implementation of the project.

The UML has been strongly embraced by medical standards committees. The HL7 standards committee in particular has used the UML from the start of their version 3.0 protocol redesign. They began by creating a "meta-framework" known as the Message Development Framework (MDF). The MDF is used to create the information model for the protocol, known as the Reference Information Model. Finally, an Interaction Model is designed that defines the actions of systems exchanging data via HL7. In each of these cases, Use Case diagrams were drawn to identify the key concepts behind the models.[28]

The UML has more to do with structured thinking that software. It provides a framework to model a system, and if desired, can be used to create software that mirrors the models. The authors see the need for the UML because in their words, "Software development organizations start out wanting to build high rises but approach the problem as if they were knocking out a dog house."[26] While modern software engineering tries to build high rises, healthcare is demanding skyscrapers. Only research and industry cooperation will lead to systems that will help healthcare professionals more than hindering them.

1.6. Challenges in Medical Informatics

Dr. Nathan Goodman and the other designers of the Human Genome Project information systems saw in 1995 a significant barrier to informatics research: "There is no middle ground in which a designer could choose to build some parts of the system, while adopting existing components for the remainder. ... Genome informatics cannot flourish as a scientific discipline until it becomes possible for significant contributions to be made by individual scientists. This cannot happen until the 'unit of contribution' becomes the component, rather than the complete system."[31] Despite the fact that Goodman's article was written almost five years ago, his main points are unfortunately still true.

The Apache open source web server provides an example of how this component approach can reap incredible dividends. When the Apache project began, one developer took on the Herculean task of splitting the sprawling source code into a set of interlocking modules.[32] This gave contributors the ability to work on one module without having to worry about breaking the entire system. The module system reduced the "unit of contribution" from a system-wide patch to a module-level revision, and allowed many more eager developers to contribute their ideas and energy to the Apache project. The module framework is a significant reason why the Apache web server is today the most popular web server on the Internet.[33]

It is clear that the "unit of contribution" must become the software component. In order for this to become a reality, we need a framework for plugging together biomedical

research components. The background above describes the multitude of tools available. The remainder of this thesis attempts to discover what design decisions and implementation steps are necessary to make this framework a reality.

Chapter 2: Design Goals

2.1. Background

Dr. Charles Guttmann of the Department of Radiology at the Brigham and Women's Hospital (BWH), in collaboration with Dr. Howard Weiner of the Department of Neurology, leads a group studying the long-term effects of multiple sclerosis under different treatments. The study tracks these effects using a magnetic resonance imaging (MRI) scanner. The project began over nine years ago. Over the course of the project, nearly 2,000 MR studies have been performed. Each MRI study contains several series of images, and each study requires up to 30 MB of storage.

The images are processed by researchers affiliated with Dr. Guttmann at the Surgical Planning Laboratory (SPL) at the BWH. These researchers have developed methods for segmenting the MRI data into different types of brain matter, and can also derive 3-D models that describe the neural structures.

2.1.1. Current Storage Method

Currently, the SPL stores the data by placing it in a filesystem directory hierarchy, described in Figure 2-1.

doe/92928399/02378/001/I.001

Patient Last Name	Patient ID	Study Number	Series Number	Slice Number

Figure 2-1: Components of the directory structure currently used to file the MR images.

Both patient name and ID are specified in the directory structure since in some studies, patients have erroneous or incomplete IDs. The images themselves are stored as Genesis or Signa medical image files.

Metadata for the medical images are stored in parallel directory structures maintained by the metadata creator. UNIX shell scripts are used to performed automated processing.

Using a filesystem to store data has one significant advantage in that it is very easy to use. Also, using a filesystem to store data is reasonably scalable through the use of symbolic links and the Network File System (NFS). All computers in the SPL laboratory have full access to the entire repository.

The clear disadvantage of a filesystem is in query capability and speed. Fast queries are limited to the directory structure; anything further requires a custom script and iteration across all images in the

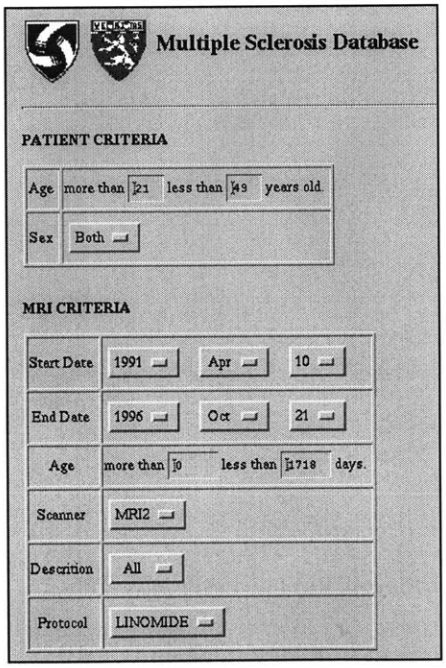


Figure 2-2: Partial screenshot of a web browser user interface for a relational database that stored SPL data. [34]

search set.

2.1.2. Previous Database Efforts

Matthieu Chabanas, a student at the SPL, previously designed a relational database for accessing the SPL image data.[34] The database used file pointers to access image data. This system stored information from the image file headers in a set of relational tables.

The user interface for this system was provided through a web browser. The system used a set of Common Gateway Interface (CGI) scripts to communicate with the database and return results to the user. A partial screenshot of this interface is shown in Figure 2-2. Researchers at the SPL commented that they felt that this interface was easy to use and should be emulated in future projects.

The CGI scripts were programmed in the C language. They communicated with the database using a proprietary interface to the relational database.

Chabanas felt that implementation of the database was not difficult. He commented that using a standard image format such as DICOM would make it easier to import different image types into the database. He found that the most difficulty was in dealing with the limitations posed by using a web browser for the user interface. Finally, he noted that future database projects would need to develop a strategy for accommodating the massive storage demands of a fully populated image database.

2.1.3. The SPL Slicer

The most complex application used to view data on the filesystem is an MRI series viewing program known as the "Slicer." A screenshot of the Slicer in action is shown in Figure 2-3.

The Slicer is an interactive viewer for three-dimensional medical images derived from multiple cranial data sets. The user moves a plane to "slice" the skull at any angle and view the structures at that level.

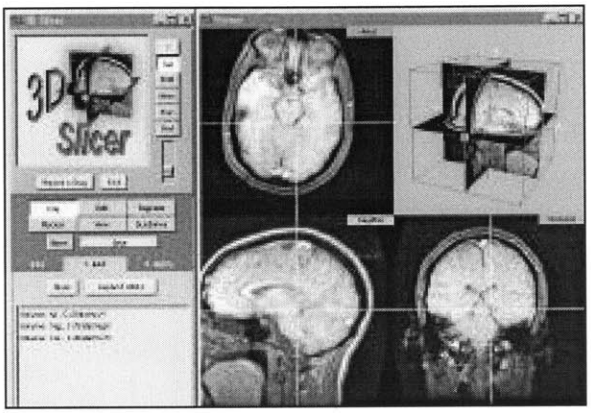


Figure 2-3: Screenshot of the Slicer program.[35]

The Slicer can display MRI series data, derived structural data, and other high-level medical objects. It is a very powerful tool that can be used to analyze neurological data in ways not possible through traditional methods.[36]

A framework file is used to describe a set of medical data viewed in the Slicer and their relationships. This file is written in a homegrown file format known as the Medical Reality Modeling Language (MRML).[35]

2.2. Initial Goals

Using the information gathered through interviews with SPL staff and by reviewing the reports of past projects, I developed a list of initial goals:

- DICOM compliance

- Extensibility
- Platform and language neutrality, emphasis on Java
- Easy to use query interface to all data
- Creation and management of relationships
- Basic user interface
- High security

Each one of these goals stemmed from specific needs expressed by the SPL researchers.

2.2.1. DICOM Compliance

To exchange data with all equipment, the new system needed to read industry-standard DICOM images. The SPL MRI image data are in a proprietary format, but the SPL has a set of tools to help convert the proprietary files into industry-standard DICOM images.

My research group previously created a database schema for storing DICOM medical images[37], and we quickly decided that this new project would be an excellent application of the schema. This immediately satisfies the "DICOM compliance" goal. However, while the problem of the database structure was well addressed in the schema effort, the issue of actually interfacing with the database and retrieving information was largely left up to the existing industry-standard database protocols.

A major part of the system will involve establishing a query interface to the DICOM database schema. By leveraging the DICOM standard, the system will not only

be compatible with DICOM files, but also other programs that are based on the DICOM information model.

In order for the query interface to provide views into the DICOM database schema, it will need to reflect an object-oriented data model. This data model will include inheritance, aggregation, and user-defined data types.

The design of the data model for the query interface is discussed in Chapter 3.

2.2.2. Extensibility

The extensibility requirement is the most difficult, because it is very easy to create a system that is inflexible bordering on useless, and also easy to create a system that is so flexible it requires months of education to modify.

We considered developing software to directly access the database tables using an industry standard interface. As the database interfaces are largely language and platform neutral, this appeared at first to be a good solution that would meet the goals. However, we realized that this solution, while quick, would result in applications tied to the database information model. Dr. Guttmann wanted to ensure that applications could be easily revised to use different kinds of databases.

To maximize extensibility, we decided to create an intermediary between user applications and the database. This intermediary would present a relatively constant

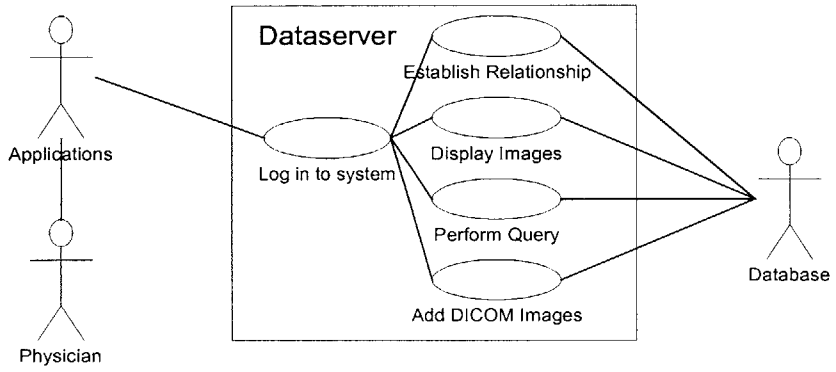


Figure 2-4: An initial use case diagram

information model to clients, even if the database information model were to be revised. Through this data model, it would serve data from the database in response to client requests. We named this intermediary system as the "Dataserver."

The above user case diagram (Figure 2-4) explores the ways in which physicians might use the system.

The use case illustrates the major outside actors on the Dataserver. The actors are the end user (a physician in most cases,) application programs, and the database. The use case also shows some of the actions that the Dataserver needs to provide. These actions reflect the initial goals listed above.

The concept of the Dataserver allows for very powerful implementations. Because the Dataserver "wraps" a data source, it can be used to provide a uniform interface to flat files, relational databases, object-relational databases, live data feeds, and even other Dataservers.

2.2.2.1. Aggregate Dataservers

One of the major extensibility goals of the SPL researchers is to integrate the large amounts of metadata that are produced by their analysis procedures into the database. As discussed above, the only link to the origin of analysis results is often the name of the directory containing the file.

The current scheme is crude, but it illustrates two points that we need to keep in mind when accommodating metadata. First, the data has a simple, implicit association to a single object in the original data set. In this case, the MRI series are associated with metadata describing 3-D models of the neural structures displayed in the MRI scan. Second, the metadata should be kept separate from the original data. The metadata for a particular piece of information may require an entirely different data model, and certainly requires a different information model. It often makes little sense to merge the information models for metadata and data, since they stem from totally different data domains.

While using a separate database for different kinds of metadata may make sense, it poses serious implementation difficulties. It is hard to efficiently merge data across database boundaries. To perform database joins, it is necessary to break a query into sub-queries, perform the sub-queries on each domain, and then copy the results into a third database, where the final correlations are computed. While this process offers excellent possibilities for parallelism and scalability, in the simple case it presents major speed and space issues.

Another problem with using multiple databases is that the complexity of the system is greatly increased. There are many possible combinations of database types, vendors, and models. To address this problem, a layer of abstraction would be useful. The Dataserver abstraction discussed above is a good starting point.

We ended the last section with the comment that a Dataserver can provide a uniform interface to other Dataservers. If we create Dataserver abstractions for our metadata and original data sources, then it makes sense to create an "aggregate" Dataserver that provides a unified interface to a set of Dataservers. Users of this Dataserver see one set of data classes and are able to perform queries across the entire set of data. The unified interface hides the fact that some queries are performed using multiple databases across several data domains.

The aggregate Dataserver model is very extensible, since it allows individual researchers to "plug in" their own Dataserver in the future. This is a good step toward the component "unit of contribution" discussed by Goodman.[31]

Imagine that we have a Dataserver for genomic data, and another Dataserver for imaging data. Since genomics and imaging have few links to each other, and may even use different data models, they need to be maintained in separate databases. Figure 2-5 illustrates an example of this example aggregate Dataserver.

While this separation makes sense from an implementation point of view, it is difficult for researchers to correlate data across both databases. To present researchers with a unified interface for imaging and genomics, an aggregate Dataserver is created that hosts all of the classes from both Dataservers.

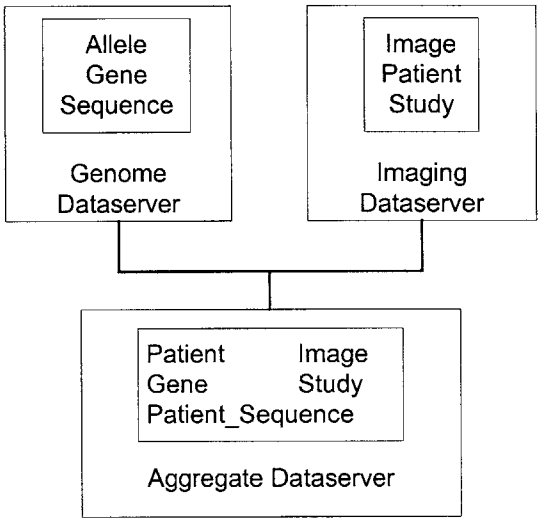


Figure 2-5: Illustration of an aggregate Dataserver.

An aggregate Dataserver is not limited to merely presenting a merged set of data classes. Entirely new classes can be synthesized from classes in the original Dataservers. As a somewhat contrived example, imagine that we present a single interface unifying the concept of a gene sequence and a patient record into a single patient record correlated with the patient's genome. This extra class helps the user compare the genetic sequences of patients against a large database of known anomalies, for example.

This feature is especially convenient if the synthesized class comprises elements of several classes that are not clearly related to each other. This kind of synthesized class gives developers the opportunity to optimize queries and joins on the synthesized union of data classes.

2.2.3. Platform and Language Neutrality

To ensure maximum portability on the client side, the SPL requires that applications can be written for the system in either the C++ or Java programming

languages. Specifically, the Slicer is seen as the most important client for the database, because it is interactive and processes a high volume of information.

Dr. Guttmann also wanted portability on the server side. Applications should be able to use databases with different data models by accessing them through different server applications that implement the same query interface. Specifically, Dr. Guttmann was considering building a purely relational-model database based on the Chabanas database, and wanted the query interface to be able to communicate with that data model.

	Distributed?	Languages supported	Platforms supported	Year of inception
CORBA	Yes	Java, C, C++, others	Windows NT, all Unix systems	1994 (Version 2)
Enterprise JavaBeans	Yes	Java	Windows NT, Solaris	1998
Microsoft DCOM	Yes	Java, C++, Visual Basic	Windows NT	1997

Table 2-1: Summary of component language standards

The association lines in the use case (Figure 2-6) show the cases where the Dataserver must interact with the database and with the application. While all component standards (as discussed in Chapter 1) are extensible, we needed to evaluate the standards based on platform and language neutrality as well as general viability.

Table 2-1 is a brief summary of the available software component standards, based on the major needs highlighted by the SPL.[27, 38] Of the three prevailing standards available, only CORBA is both language-neutral and platform-neutral. Also, CORBA is much more mature than Enterprise JavaBeans.

After a survey of the available software component standards, we decided that the CORBA object standard best met the goal of "platform and language neutrality" in the interface. CORBA allows developers to define an interface in a special Interface

Definition Language (IDL). IDL files can be mapped to most popular programming languages. Writing in IDL helps with the overall design of the system since it forces a division between the interface and the implementation of the system. Any CORBA client written to a particular IDL can be used with any CORBA server written to the same IDL, using any combination of programming languages and platforms.

CORBA's major disadvantages stem from its complexity. Most major implementations are commercial, and some implementations have difficulty interoperating with other implementations. CORBA can be difficult to use in practice, especially since the IDL-generated code can be more complex that would be otherwise necessary. However, there is no other standard that is being widely embraced and has full implementations immediately available.

2.2.4. Easy To Use Query Interface

In general, the structure of a query to a database reflects the data model, while the content of the query reflects the information model. For example, a SQL query expression contains slots for requesting particular tables and columns. To actually write the query, the user needs to know the names and content of the tables and columns in the database.

The data model used by the query interface may have nothing to do with the underlying database. It is potentially more efficient to have the underlying data model be similar to the interface data model, but not necessary. Therefore, the data model used by the query interface should be primarily influenced by the "ease of use" goal, and secondarily by "ease of implementation."

The query interface must also have an easy method for exploring the information model. This way, users can browse the information model in a dynamic fashion. Traditional SQL-based interfaces usually have "system catalogs" which are tables that contain metadata about all aspects of the database, including the names of tables and columns. For a user to browse an SQL database, they need to key in queries on these system tables and know how to interpret the rows that are returned.

Finally, the information model that the query interface interacts with must be consistent and easy to understand. Using the DICOM database schema as a basis for the information model is a significant step forward in that direction.

The actual design of the data and information models used by the query interface is discussed in the next chapter.

2.2.5. Creation and Management of Relationships

One of the features that the SPL is interested in exploring is a way of establishing "relationships" between different objects. These relationships would merely alert the user that there was a non-obvious association between one object in the database and another. For example, a user could link together two different studies that showed similar findings using such a relationship.

Generally, if a relationship cannot be made between two objects in the same domain, such a relationship is simply not suitable. The information model should support all meaningful relationships between objects. However, there are some relationships that cannot be anticipated when an information model is initially constructed. In this case, the user would need to create user-defined relationships that lie outside of the information

model. Relationships between information models fall into this category, leading us to believe that this technology could be a powerful aid to aggregating many data sources under the same interface.

The DICOM medical imaging standard is currently reviewing support for the ability to add references to information that lies outside the DICOM information model.[39, 40] This support, known as the Structured Reporting (SR) Supplement, allows DICOM objects to include external references to other data sources, or references to other DICOM objects. The supplement is designed to allow DICOM files to be used in domains not fully covered by the DICOM information model. SR can be seen as a "metadata" object in the DICOM information model, because it allows users to work on a level above data by providing a means to describe objects and the relationships between them.

The term "edge" is another term for user-defined relationships, since they can be conceptualized as directed arcs drawn between objects.

Consider treating edges on the same level as the model-defined relationships. That is, the query "What objects of class A are related to the objects in class B?" would return not only objects related through model-defined relationships, but also objects related through edges. This could quickly become extremely confusing since the user is unable to discern which objects in the result set are present because of the model-defined relationships versus the edges.

Due to these limitations, the query interface must treat edges separately from model-defined relationships.

The benefits of allowing users to create relationships outside of the information model are not clear. If the database contains multiple, non-integrated information models, this ability could be a boon to users wishing to link data between the different models. However, even in this case, edges may merely be a way to delay the proper solution to the above problem, a refactoring of the information models to create a unified model.

2.2.6. User Interface

The overall goal for this project is to provide an advanced system for performing medical data research. While we have discussed goals relating to the "back office" elements of the system at length, no attention has yet been paid to the end-user experience. In order for the underlying aspects of the project to be fully appreciated, a user interface needs to be supplied that demonstrates as many features as possible. It also needs to be extensible to support modifications that the users may desire in the future.

The user interface can be seen as a direct interface to the abstract query interface described above. Thus, a detailed feature listing cannot be composed until the query interface is in the late stages of design. However, we can set down some basic goals along the lines of keeping the application extensible and easy to use.

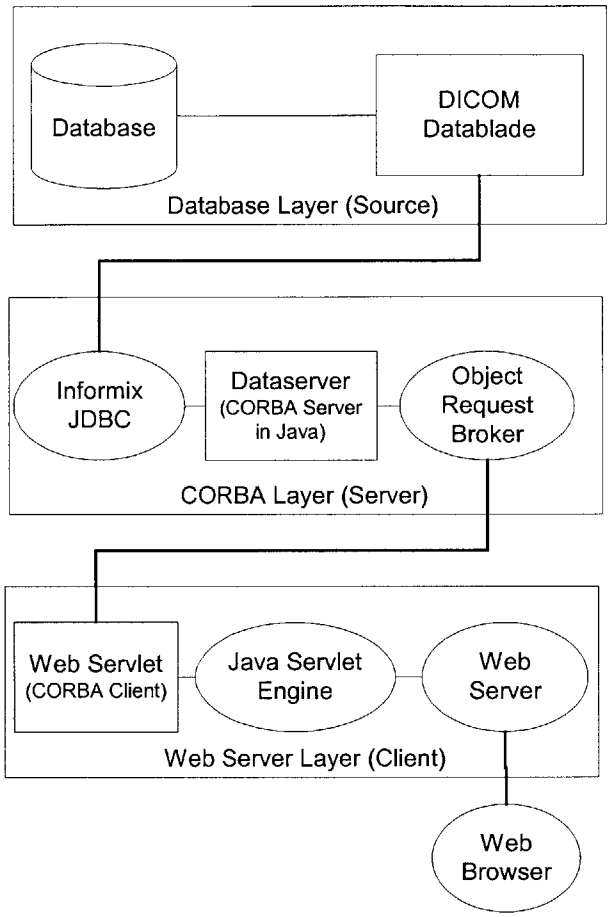


Figure 2-6: A layered approach to the design of the query system

First of all, the user interface should be as cross platform as possible in order to let the widest number of people use it. Based on the current literature, this immediately suggests a Hypertext Transfer Protocol (HTTP) based World Wide Web solution.[41-43]

A Web solution would require a server capable of interacting with the query interface through CORBA and with the client browser using the Hypertext Markup Language (HTML). An excellent up-and-coming technology for this is the Java Servlet, a special type of Java program that runs continuously inside a web server and fields user requests. Java Servlets allow developers to create Java programs that interact through a

web browser instead of a graphical user interface. Figure 2-6 shows how this type of client fits into the framework already described.

Java Servlets are superior to traditional Common Gateway Interface (CGI) based server programs because they run continuously instead of being loaded and unloaded for each user request. This allows for many optimizations and special behavior that is not otherwise possible. Also, since Java Servlets are written in Java, they benefit from the language's development speed and general robustness.

A limitation of Java Servlets is that the technology is relatively immature, and that it is difficult to use conventional libraries written in C with Java programs. Also, because Java Servlets work in a highly multithreaded environment, they are generally more complex than equivalent CGI programs.

Using HTML as the user interface limits the interactivity of system. Some of these limitations can be overcome with client-side technology such as JavaScript and Java applets, but not all.

The primary goal of the end-user interface is to provide an adequate conduit to the query interface. Also, the end-user interface should be as extensible and portable as possible to maximize its value. Ease-of-use and robustness concerns should be left for later revisions of the user interface, as a serious effort to create an excellent end-user experience is worth an entire thesis project in itself.

2.2.7. Security

Security is a very important topic to hospital administrators[1]. The security of Web-based interface systems in particular have been studied in detail.[44]

Good security protocols involve using proven, off the shelf encryption and authentication technology while sealing off all but a few "ports of entry" in application software. For this project, the query interface must provide support for authenticating users before allowing them access to the system. The underlying database must be configured to restrict access to sensitive data. Finally, encrypted channels must be used when working over insecure networks.

This project does not need any major innovations in secure system design, but does require careful attention to standard security concerns.

2.3. Conclusions

The above goals and their sub-goals form a comprehensive requirements list for a medical data query and retrieval system. The major focus of this project will be on the query interface component, as decisions made at this level affect all of the goals.

The following chapters detail the actual design and implementation of the system. Throughout the project, the above requirements were kept in mind as goals to be met and exceeded.

Chapter 3: Data Model Interface Design

The most fundamental part of the project is the query interface, and the basis for the query interface is the data model. Without a powerful data model, large segments of the information model would be inaccessible. Reaching these conclusions was a long process that required much research and several cycles of development.

3.1. Background

The initial piece of the project was the DICOM database schema developed by Dao[37] that was discussed in the previous chapter. This information model uses the object-relational data model discussed in Chapter 1.

The object-relational data model and the DICOM database information model provide a rich basis for a query interface. Several designs were necessary before the an interface was developed that allows unfettered access to the large number of data elements and relationships in the DICOM database.

The data model used by the interface is derived from, but not a duplicate of, the object-relational data model used by the DICOM database. The overall difference is that the new data model attempts to be more object-oriented and easier to use. The new data

model can be seen as a simpler interface for accessing an object-relational data model as well as any other data model that can be mapped to an object-oriented structure.

Creating a new interface for an existing system is the basis for the Adapter pattern in the Gang of Four design patterns book[29]. The Adapter pattern is used to match existing interfaces to domain-specific interfaces. The protein model data schema by Gray[20] (discussed in Chapter 1) also interfaced a generic query system to a specialized data model.

The data model and interface design feature sets are tightly interconnected, so I developed them in tandem. New elements in the data model demanded additional functionality in the interface, and vice versa. In order for users to see a consistent interface, the interface must represent a coherent data model.

The design of the modified data model and its interface proceeded over a long period of time, punctuated by several cycles of implementation and reflection. There were three major attempts to develop query interface. The first attempt created a direct interface to the database's object-relational data model. The next attempt, implemented a query interface based on a simple flat-file data model. The third attempt developed an object-oriented system that borrows heavily from the object-relational data model.

After deciding on the object-oriented system, two different methods were tried to develop the interface. The first method used the CORBA IDL to present the data and information model. The second method used an XML-based representation of the data and information model.

3.2. *Directly Accessing the Object-Relational Data Model*

The first option is to directly access the object-relational data model used for DICOM database schema using SQL-3.

3.2.1. SQL-3 As The Interface

The simplest query interface allows the application to send queries in the SQL-3 language and retrieve the results. However, this approach has several drawbacks.

First, using SQL-3 limits extensibility by tying the application to the database data model. While the focus is on using the DICOM database schema, the extensibility goal requires that implementers can map the query interface to other data models.

Next, this approach makes it difficult to discover the information model. Unless the user is given a map of the tables and the relationships between them, database vendor-specific methods must be used to find the list of tables, the columns within the tables, and the relationships between the tables. For large information schemas, this task requires automated assistance, unless the user is exceptionally patient.

Finally, this interface is very difficult to use. SQL-3 is very flexible and expressive, which makes it hard to quickly write queries that are effective. For object-relational schemas that make extensive use of distinct types, row types, and array types, using SQL-3 becomes extremely unwieldy for the casual user.

Since our DICOM information schema uses the object-relational model extensively, SQL-3 queries against this database were quickly littered with difficult-to-understand syntax and convoluted join clauses. While the resulting queries were very

useful, it became clear through experimentation that a simpler interface will broaden the audience of prospective users and streamline the process of formulating a query.

3.2.2. "Precooked" SQL

The problems with using bare SQL as a query interface clearly show that some sort of abstraction layer is needed. A first cut abstraction is to provide some sort of layer in the SQL language itself.

The DICOM database schema is implemented as an Informix Universal Server Datablade, a shared module that is compiled and loaded into the database's memory area. This Datablade is an extension of the database. The functions in the Datablade are treated as SQL functions, even though they are written in C. Datablade User-Defined Routines (UDRs) offer much more flexibility than SQL stored procedures because UDRs can call libraries and execute external functions.

Using Datablade UDRs, we can "precook" the SQL for the user. Instead of having

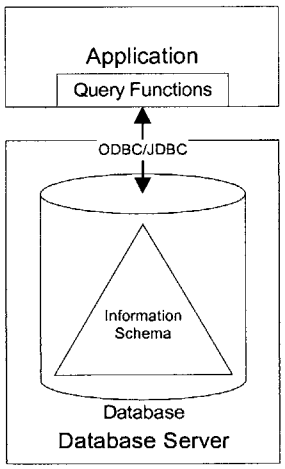


Figure 3-1: Example of a thick database application.

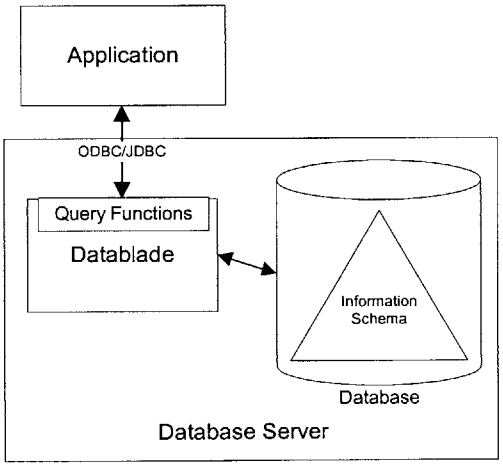


Figure 3-2: Example of a thin database application.

to assemble a complicated SQL query, we can treat a query as a function call with a set number of parameters and a return value. This function call is then translated into SQL, and the results are returned as a standard SQL result set.

Functionally, Datablade UDRs act similarly to external programs that execute SQL queries via an ODBC or JDBC interface. A key advantage of UDRs over external programs is that they allow an "thin" interface to the database by moving all program logic inside the database. The thin interface allows for easier interoperation, since instead of having to make the query functions work with whatever language and platform the application is written in, they merely need to be integrated with the Datablade once. After this point, any application can call the functions using the industry-standard ODBC and JDBC APIs. In a sense, UDRs are really Remote Procedure Call servers.

Figure 3-1 illustrates a traditional thick database application. In this application the query logic is in the application. The query functions are restricted to the resources and speed of the application machine. If the application is ported to a different platform, the query functions must be ported also. Figure 3-2 shows a thin database application that calls the query functions through a ODBC or JDBC interface. The query functions can now use the full capabilities of the database server platform. Moving the application to a different platform is quick because the query functions do not need to be ported.

Note, however, that communication with the query functions are restricted to the ODBC/JDBC connection, which allows much less flexibility than an actual function call. This includes limitations on data types and an inability to pass by reference. Specifically, it is difficult to trade structured data types with UDRs. This is the result of attempting to

use an interface designed for communicating database rows and simple results for generalized remote function calls.

Using any function call based solution as a query interface solution is limited in general. SQL is implemented as a language instead of a set of function calls because SQL needs the flexibility of language to be extensible and powerful. Any reduction of this query language into a limited set of function calls results in a crippled query interface that is unable to easily express complex queries. However, a set of function calls is easier to use because they offer a clear set of options to the user in contrast to the open-ended nature of a language. This tradeoff between an open-ended language and a set of limited function calls will reappear throughout the design of the query interface.

The limitations of UDRs discussed above, plus the fact that they are by nature tied to a particular database data model and vendor, make them unsuitable as a query interface.

3.3. The Flat-File Approach

After deciding that direct database interfaces were too constrictive, we looked to distributed component software systems for an interface solution. We decided to develop our interface in CORBA.

The first interface was developed by Ted Zlatanov of the SPL. The data model used by this interface is very simple, and closely resembles a flat-file database. The flat-

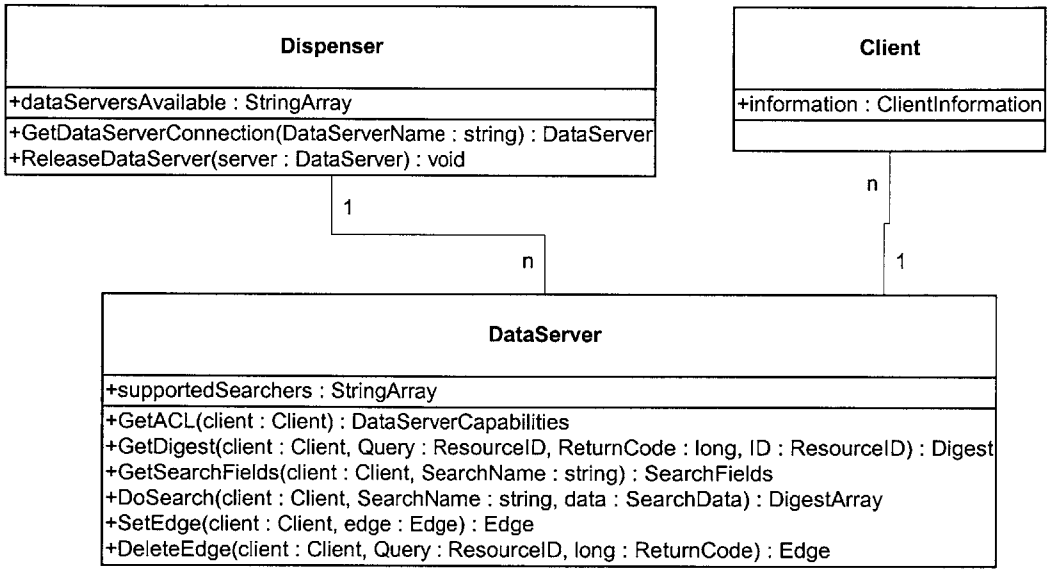


Figure 3-3: Simplified UML diagram of the interface classes in the flat-file IDL

file interface sees a database as a single gigantic table. The table has a single primary key that is used to index the data. Figure 3-3 shows a UML class diagram of the primary classes provided by the interface.

This query interface contains the concepts of a DataServer (as discussed in Chapter 2), a Dispenser, and a Client. The user begins by querying the Dispenser for the list of available data servers, and then gets a handle to a DataServer object. Each query operation in the DataServer requires a Client object to provide authentication information to the data source.

The Dispenser object is based on the Dispenser design pattern by Orfali and Harkey[45]. The Dispenser design pattern allows the server to maintain all connections to the database, and eliminates the need to wait for a new database connection on each request.

The primary method offered by the Dataserver is DoSearch. The user provides DoSearch with a list of search criteria, and DoSearch returns a list of all the rows (called "digests") that match the criteria.

Criteria consists of a domain, an attribute, a comparison operator, a value, and finally a logic operator that specified whether the criteria should refine or expand the current result set.

This interface also provides support for arbitrary relationships. Each Digest contains information about any edges associated with the row, and the user can set or remove an edge between any two rows.

A data type called "ResourceID" uniquely identifies objects in the database. ResourceID is used to establish relationships between different objects.

It was difficult to map the object-relational data model to this flat-file data model. For the trial implementation, we used only a handful of tables from the DICOM information model, and hand-tuned the SQL generator to properly query these tables. (Further details are in Chapter 4.)

However, the interface was simple and easy to use. We liked the simple method of expressing criteria. After the trial implementation, our goal became to create a query

interface that maintained the original interface's simplicity while being flexible enough to map to multiple data models.

3.4. *The Object-Oriented Approach*

As a starting point for developing a general object-oriented interface to data, I experimented with mapping an object-relational data model to a more pure object-oriented model. Table 3-1 is a table derived from Kim[22] showing the high-level relationships between object-oriented concepts and object-relational concepts.

Object-Relational Concept	Object-Oriented Concept
row type	class template
table	class
row	object
column	attribute
row reference	object reference

Table 3-1: Object-relational to object-oriented mapping

The design approach used by Dao[37] in constructing the DICOM information schema offers some insight into the limitations of the Informix object-relational model implementation. Dao was not able to use all of the concepts in the object-relational data model due to limitations in the Informix Universal Server. Dao's schema does not use object references to other objects, but instead relies on traditional relational link tables to establish relationships between classes. Also, Dao's schema is forced to explicitly declare primary keys on inherited tables, even though the database is supposed to propagate that information on its own.

For the best compatibility with modern programming languages, the query interface needs to express concepts in a purely object-oriented manner, even when the implementation is forced to use workarounds.

3.4.1. *CORBA IDL as an Information Modeling Language*

Since the project was already using CORBA, the first data model tried was the CORBA object-oriented data model. This attempt experimented with expressing the full information model in terms of a set of CORBA interfaces. The information schema for each data source would be represented in the CORBA interface definition language (IDL). The IDL is a cross-platform and language-neutral method for describing distributed objects.

Using the MRML information model, as derived from David Gering's work[35], a set of CORBA interfaces were built that directly mirror the concepts in MRML. The

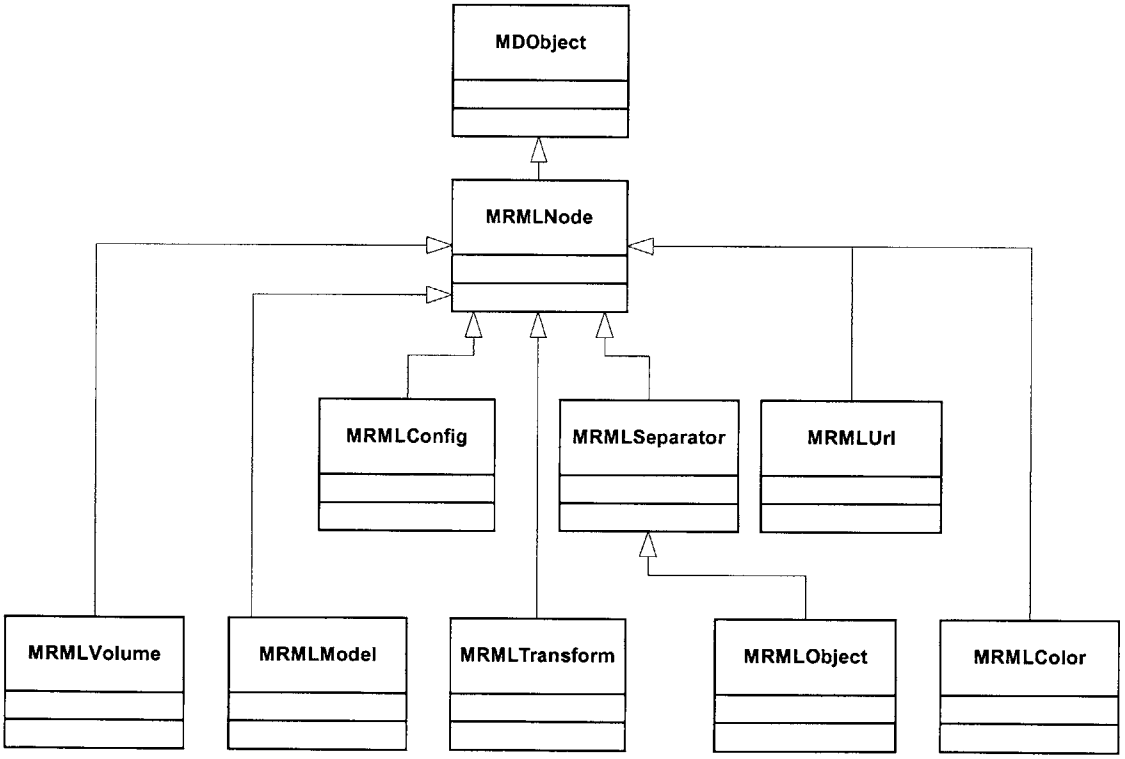


Figure 3-4: The MRML information model expressed as a set of CORBA interfaces

resulting interface hierarchy is shown in Figure 3-4. (The operations and attributes have been removed for clarity.)

To satisfy the extensibility goal, CORBA offers a set of methods for dynamically exploring and using an interface. The traditional way to use CORBA is to compile a single IDL file into server stubs and client stubs. Then, clients and servers are written to work with these stubs. A problem that arises is that any change in the interfaces requires a significant amount of work. As a remedy for this problem, CORBA provides a dynamic interface system that allows applications to use objects without knowing the exact interface specification.

In practice, the dynamic interface system is overly complex and very difficult to use. Even using the dynamic interface system, changes in the information schema would result in extensive and time-consuming changes to the underlying source code.

Also, the CORBA object-oriented data model is clumsy to use. Since CORBA was designed to be used with many different non-object-oriented languages, there are many design compromises made as a result of the needs of one vendor or one language. For example, Michi Henning, an expert on CORBA, reports that the entire concept of a CORBA attribute "was added as syntactic sugar on the insistence of one particular vendor."[46] A further discussion of the distance between the promises of CORBA and the reality of the current implementations is in Chapter 4.

While the CORBA data model is not rich enough to use as the basis for a query language, CORBA is still sufficient to be the vehicle for a more abstract data model. To

create this more abstract data model, we experimented with building several different data models in the Extensible Markup Language (XML).

3.4.2. XML as a Data Modeling Language

Having abandoned SQL return values, flat-file, and CORBA as potential data models, a data model was needed that could be easily mapped to relational, object-relational, and object-oriented data repositories. However, it is difficult to define a data model from scratch.

To define the data model, we experimented by recasting the existing DICOM information model from the object-relational data model to various simpler data models. The resulting information model was expressed in XML.

An ironic part of designing the data model as an XML Document Type Definition (DTD) is that we are never promising that the DTD will actually be used. Instead, we are using the data model of DTDs as an aid to structure the concepts of our independent data model. In this way, the DTD, the XML documents, and the XML parsing software act as scaffolding for our data model. Once the data model is complete, we can wipe away the XML, and we will be left with an abstract data model that can be expressed in any structured language.

The first attempt at an XML information schema document looked something like this:

```
<datadefs>
 <object name="Patient">
  <attrib name="Name" type="string"/>
  <attrib name="ID" type="string"/>
 </object>
</datadefs>
```

This document defines an object of type Patient, with two attributes, both of type string. From this document, a DTD can be derived which looks like this:

```
<!ELEMENT datadefs (object*)>

<!ELEMENT object (attrib*)>
<!ATTLIST object name CDATA #REQUIRED>

<!ELEMENT attrib EMPTY>
<!ATTLIST attrib
 name CDATA #REQUIRED
 type (string|char|short|long|float|date|datetime) #REQUIRED>
```

The XML DTD is somewhat difficult to read, but can be understood by comparing it to the XML document above. The top-level element is a "datadefs" element, which contains many "object" elements. Each "object" element contains a set of "attrib" elements, and has a single attribute named "name" that contains character data and is required. Each "attrib" element contains no other elements, and has two attributes. The first attribute is called "name", and is required. The second attribute is called "type", and requires a value in the set of datatypes shown above.

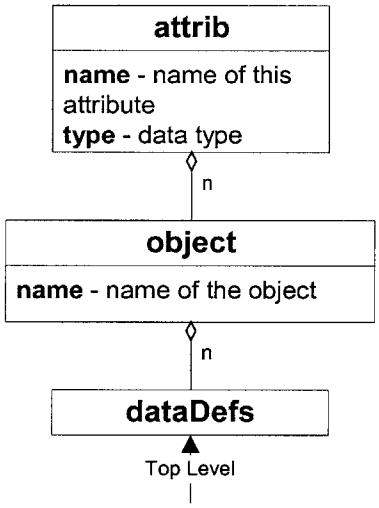


Figure 3-5: Diagram of a simple DTD.

Figure 3-5 shows a diagram of the above DTD. The lines ending diamonds are UML aggregation symbols to indicate that "n" parts may be considered part of the whole. That is, several attributes may be part of one object, and several objects may be part of a single dataDefs element.

XML parsing software uses the DTD to ensure that a document follows a particular set of rules. This also makes life much easier for software engineers, who can rely on the XML parser to catch all of the low-level document structure mistakes.

DTDs are limited, however. Higher level rules, such as "attribute A is required if attribute B has value X, but not otherwise" cannot be expressed in a DTD and must be explicitly enforced by the parser writers.

Even so, the DTD is an excellent way to express basic rules for the structure of a document expressing an information model. These basic rules act as the data model.

I went through several iterations of the data model, adding more object-oriented concepts from Table 3-1 each time. Each iteration was expressed as a DTD.

Several additions expand the initial data model above. A "classdef" element acts as a template for classes and stores lists of attribute definitions. The "object" concept is renamed to be a "class." The concept of a "relationship" is added to handle the idea of an object reference.

A relationship is a link between two classes. A relationship can be one-to-one, one-to-many, or many-to-many. An example of this kind of relationship can be had from the DICOM standard. In DICOM, a patient object is connected to several study objects. This means that there is a one to many relationship between the patient and study classes.

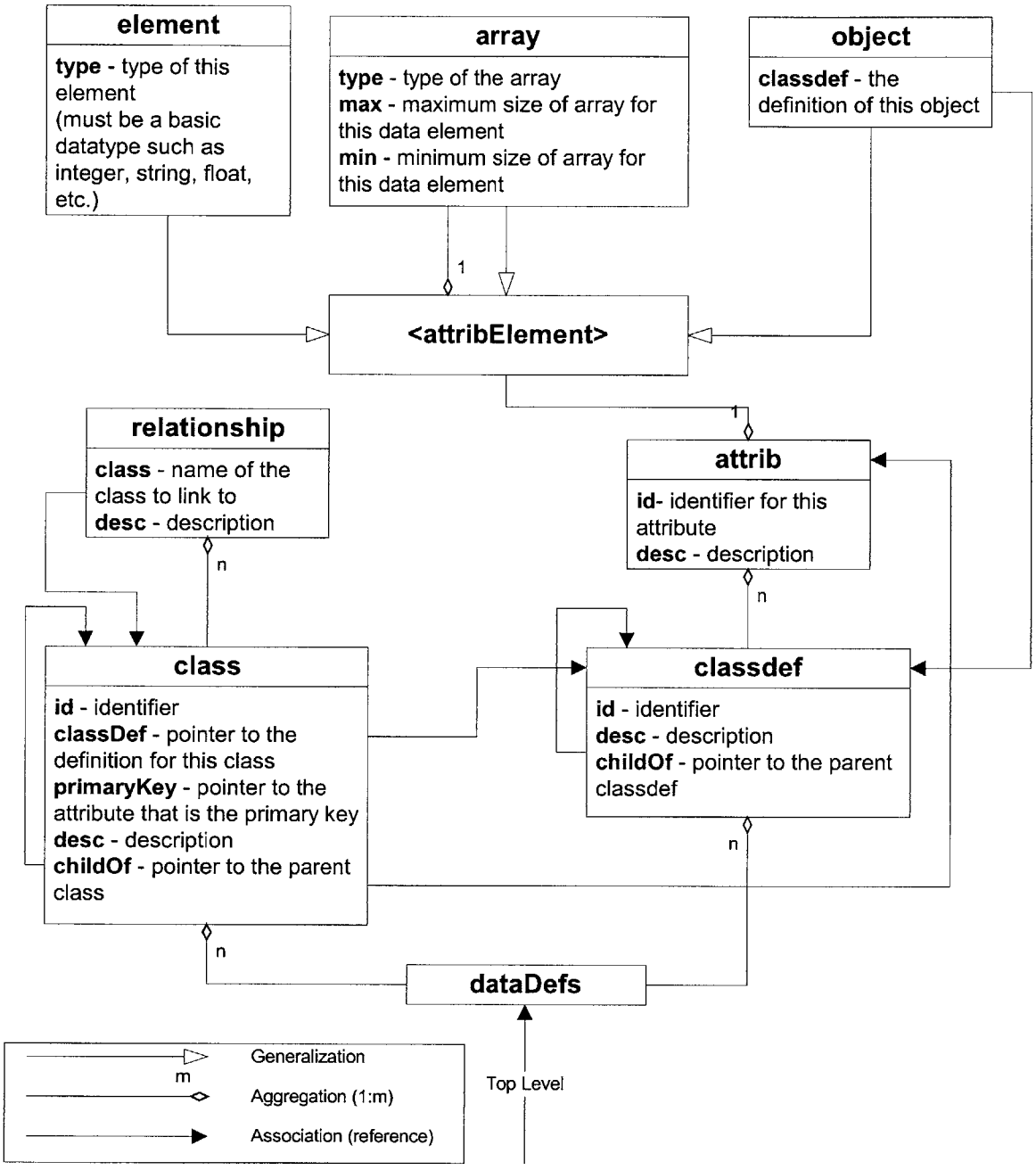


Figure 3-6: UML diagram of data model classes

The concept of a relationship is handled through joins and link tables in relational databases, through row references in object-relational databases, and through object references in object-oriented databases. The idea behind the relationship element is to

abstract these particular implementations away so that the user could simply perform a query asking for all objects in class B related to a particular object in class A, where A has a one to many relationship to B.

The "type" attribute of the "attrib" element above was too restrictive to be able to adequately express the full range of possible data types. This attribute was replaced with an "AttribElement." There are three types of AttribElements, an "element" (basic data type), an "array", and an "object".

Figure 3-6 displays the diagram of the data model with these added concepts. The dark arrows are "association" arrow which note where attributes of elements refer to other elements. For example, "Class" and "Classdef" both point to themselves since their "childOf" attribute, when present, refers to the parent instance of the element.

XML documents do not have a truly object-oriented structure. Strictly speaking, XML elements are hierarchical. However, it is very easy to express object-oriented concepts in the XML. For example, the "AttribElement" concept discussed above is an abstract class. In Figure 3-6, AttribElement is noted with angle brackets because it does not really exist in the DTD, but acts as a placeholder to remind us that "element", "array", and "object" are all different subtypes of the common AttribElement concept.

Even though it is easy to map object-oriented concepts to an XML DTD, the object-oriented design is not always clear to future users. To this end, the World Wide Web Consortium is working on different object-oriented schemas for XML, and has received suggestions for object-oriented schemas in XML.[47] There is even a strawman

object-oriented XML proposal aimed at CPRs.[48] I considered experimenting with one of these suggestions, but decided that a full object-oriented XML schema was not necessary.

Here is an example of a portion of the DICOM information model expressed in the data model described by Figure 3-6:

```
<datadefs>
  <classdef id='dcm_physician_t' desc=''>
    <dbType type='dcm_physician_t'/>
    <attrib id='tag_mitx_phyx' desc=''>
      <element type='string'>
      </element>
    </attrib>
    <attrib id='PhysiciansPhone' desc='Physicians Phone Number'>
      <array type='set'>
        <element type='string'>
        </element>
      </array>
    </attrib>
    <attrib id='PhysiciansName' desc='Physicians Name'>
<element type='string'>
      </element>
    </attrib>


  <class id='dcm_physician' classdef='dcm_physician_t'
         primaryKey='tag_mitx_phyx' desc=''>
    <relationship class='dcm_visit' attrib='SOPInstanceUID'
                  type='m:n' desc=''>
    </relationship>
  </class>
</datadefs>
```

Now, the flexibility of the model needs to be proven by mapping it to the object-relational data model.

3.4.3. Mapping the Data Model to the Object-Relational Data Model

The above experiment with XML to encode the attributes used for different objects is an elegant solution. This usage of XML demonstrates the core philosophy of

markup languages, to separate the format from the data. The data model DTD is the format, and the XML information schema document is the data.

The "data" in this case is not actual patient records, but the data formats for those records. This demonstrates another strength of XML, the ability to encode metadata. Thus, it seems a logical extension that we can encode the mapping between an object-relational database and the higher-level data model in XML as well.

The elements prefixed with "db" to indicate the mapping from elements to actual database concepts including rows, types, columns, tables, and link tables. Figure 3-7 shows the data model above extended with these mapping elements.

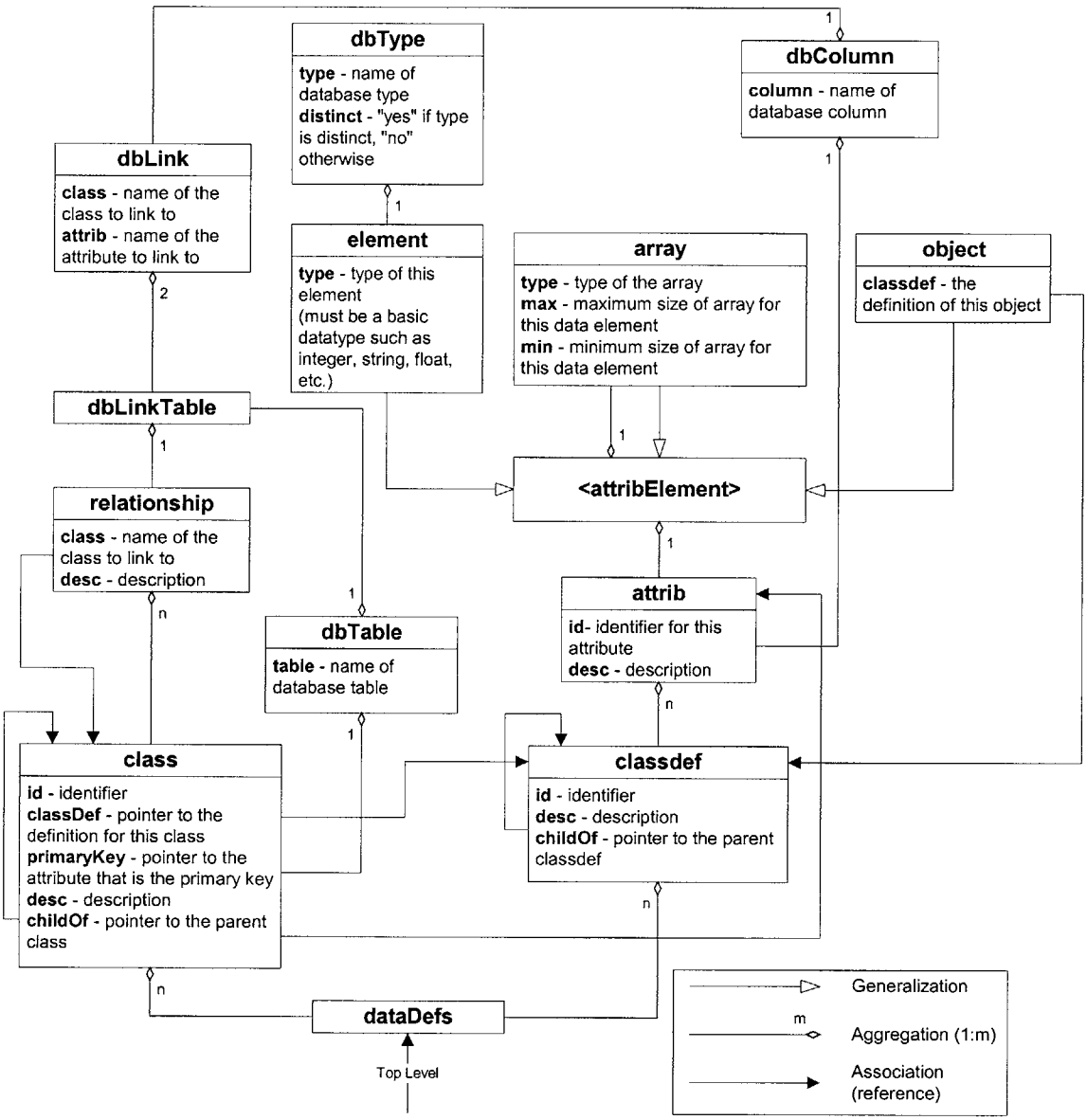


Figure 3-7: Simplified UML class diagram of the full XML DTD that maps the query data model to the object-relational data model

An example mapping of an attribute looks like this in the XML information schema document:

```
<attrib id='tag_mitx_phyx' desc=''>
  <dbColumn column='tag_mitx_phyx'/>
  <element type='string'>
    <dbType distinct='yes' type='dcm_ui'/>
```

```
  </element>
</attrib>
```

The Document Type Definition that is described by Figure 3-7 appears in Appendix B.

The next task is to develop a query interface that allows access to this data model.

3.4.4. Designing the Query Interface for the Data Model

The original query interface described in section 3.3 provided a good starting point for the final query interface. However, there were several deficiencies that needed to be addressed.

One problem is that the result sets are returned as structures, not interfaces. Also, if different result sets (Digests) are returned, they must be combined by the client. This means that the server cannot perform efficient techniques such as lazy result evaluation or table joins. A solution to this problem is to return results as interfaces, from which the user requests different values.

The original IDL is not modularized. To aid user understanding of the interface, the data types and interfaces are segregated by roles. These roles form four packages:

- **types** – includes basic data types such as byte arrays, date structures, attribute information.
- **util** – includes utility classes, such as InputStreamI for transferring large amounts of data.
- **core** – included the core concepts, such as Object and Query.
- **dataserver** – included the DataServer and DataServerDispenser

These packages are diagrammed with their dependencies in Figure 3-8.

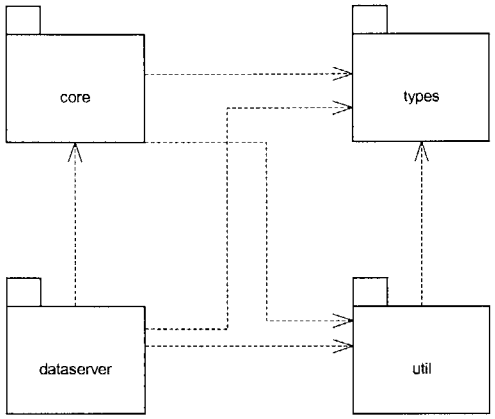


Figure 3-8: UML package diagram for the query interface.

Another deficiency is the lack of error handling. The original interface uses integer return values to indicate success or failure. For the new interface, exceptions indicate error conditions. The package system helps in this regard because it keeps the exceptions closely bound to the functions that use them.

The previous interface returns all data as strings or as byte streams. There is no support for data types such as nested arrays or nested objects. The new interface uses dynamic data typing to enable almost any type of object to be returned. A disadvantage of this is that it increases complexity for the user, since they must introspect each data value and interpret it accordingly.

The criteria format is similar to the original attempt since it was sufficient to create the queries requested by the SPL researchers. The basic Edge functionality worked well, so those functions also remained largely the same.

3.4.5. Lessons Learned from Designing the Query Interface

Through the design process, some useful maxims came to light as guidelines for building distributed interfaces.

1. **Passing an object as an interface does not mean a function can treat the interface as an object.**

This lesson is best explained by example. At one point in development, a "DoSearch(QueryI)" function was specified. The idea was to have the DoSearch function find the Query implementation's database connection and use it to perform the search.

However, when this function was implemented, it quickly became apparent that it is not possible for DoSearch to ask for the Query implementation's database connection, since DoSearch is limited to the functions in the Query interface. To separate interface from implementation, there are no functions that access the Query's database connection. Future implementations of the DataServer interface might not use a database at all.

The mistake here is in treating the Query interface as the Query implementation. It is important to remember the distinction between the two.

2. Users must be able to introspect all interfaces

The original QueryI interface did not have a "getCriteria" method to return the list of criteria used in the query. At the time, it was thought that the the user, would always know what criteria were being used, since the user built the query in the first place.

However, this is not always the case. In one trial, an function used a QueryI as input, and it became clear that the only way for the function to know the contents of the Query would be to separately pass the list of criteria to the function. This is not very efficient.

In general, all interfaces must have introspection functions. Imagine if someone hands you an object with a particular interface – what can you tell them about the object? If the answer is "not much," more introspection functions are needed.

3. Use case pseudocode is a useful tool for finding problems with interfaces

It is difficult to test interfaces. It is possible to write an interface, then write a full implementation, test, and iterate, but this is time consuming. Also, it results in a lot of wasted effort, especially if partway through implementing an interface the designers find a fundamental flaw.

A good tool for testing the query interface was pseudocode scenarios. A list of usage scenarios was composed, and for each interface a short program was written in psuedocode to solve the problem at hand. Through these examples, certain design flaws emerged that would have resulted in significant lost time if the design was implemented.

These lessons proved to be very useful in designing a robust query interface. When the interface was implemented, there were very few unexpected surprises.

3.4.6. A Quick Tour of the Query Interface Features

Figure 3-9 is a UML class diagram of the final query interface design. The association lines indicate the relationships between different interfaces. For example, a DataServer has many Queries, a Query has many Objects, an Object has one ClassDefMetaData, and so on.

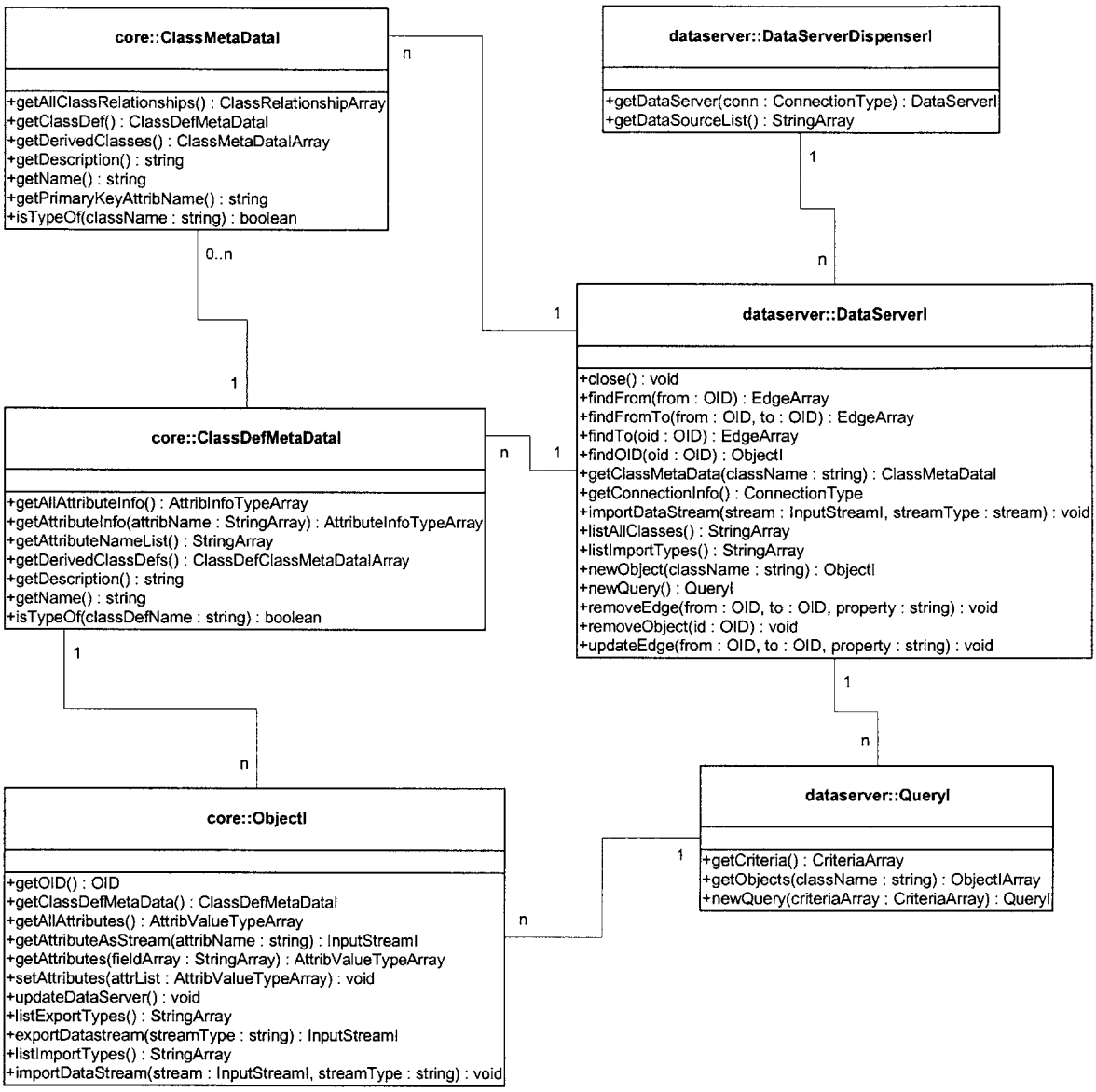


Figure 3-9: UML class diagram of the query interface.

The MetaData classes express the information exposed in the data model diagram (Figure 3-6). Using the MetaData classes, the user can quickly learn about the entire data schema.

The DataServerDispenser is a modified form of the Orfali and Harkey[45] Dispenser design pattern discussed above. The general idea is that CORBA objects should not maintain database connections on their own, but should instead request a connection from a pool, perform a query, and then return the connection. This interface pushes the management of connections down to the implementation level. To the user, it appears that they need only log in once to the DataServer, which then maintains a persistent connection to the database. In practice, good implementations will use connection pooling to achieve high efficiency.

The DataServer provides a number of methods to investigate the information schema, and also allows the user to begin building a Query.

A Query object is immutable. Once created, the criteria for a query cannot be changed. Part of the reason for this is that immutability gives implementations more options. Implementations can choose to begin query processing at the moment of construction, since they can be assured that the criteria will not change. There are two query generation methods; the newQuery method in the DataServer creates a query with no criteria, which automatically includes the entire dataset. The newQuery method in the Query interface itself creates a copy of the existing Query with a given set of additional criteria.

The chain of queries created by a series of newQuery calls acts as a history, allowing users to quickly back up to previous queries without needing to repeat the actual query against the database.

Each criterion includes the class, attribute, value, comparison operator, and the logic type. As in the original interface, the logic type indicates whether to refine or extend the existing result set. The initial result set is the universal set.

The actual method that performs a query against the database is "getObjects", which takes the name of a class as a parameter. This class is the "target" class, and instructs the query generator to find all objects in the target class that match the given criteria.

The getObjects method returns an array of Object interfaces. Each Object interface provides access to the attributes inside the Object. Each Attribute value can be an array, a basic type, or even another Object interface.

Each Object also has a unique identifier that is valid across the entire DataServer. This identifier is used by the object removal, update, and Edge functions.

The Edge functions are functionally unchanged from the first interface. They provide a method of recording arbitrary associations between individual objects.

Taken together, the methods within the query interface provide a powerful and simple method for querying any data source.

3.5. Conclusions

The data model and query interface design phase of the project took more time and effort than initially anticipated, because many aspects of the problem are very subtle. The result is well worth the time spent, however, as it meets all of the design goals described in Chapter 2. This query system is domain-nonspecific, and can be used with

any data source that can be mapped to the data model. Chapter 4 will illustrate the construction of a trial implementation for an object-relational database using a medical image information model, but there is no reason why this system cannot be applied to other domains and data sources.

Chapter 4:
Server Design and Implementation

Designing the data model and developing the information model was only the first part of the project. The prototype server implementation needs to implement these plans while being robust, extensible, and easy to use.

There were two servers constructed for this project. The first project was a trial implementation of the original "flat-file" IDL diagrammed in Figure 3-3. The first server provided a good learning experience into how CORBA works, and exercised the then-early beta Informix JDBC driver. Also, I developed the DICOM tag mapping interface described in section 4.2.

However, this first implementation was largely abandoned when the interface was revised. This chapter deals with the experiences developing the second server, as it is substantially more interesting from a software engineering perspective.

4.1. Background

The following software was used to develop the DataServer:

- Sun Java Development Kit version 1.2
- Sun Solaris version 2.5.1

- Informix JDBC Driver version 2 (beta version)
- Inprise Visibroker for Java version 3.3 (later version 3.4)
- IBM XML for Java parser version 2.0.4

Java is used as the development language because in previous projects it has been shown to be very easy to write programs quickly. Multithreading and networking is very easy to do in Java. Java also works very well with CORBA.

Informix was gracious enough to allow us to beta test their JDBC version 2 driver, which was necessary to use the object-relational features of the Informix Universal Server.

The JDBC version 2 specification mandates the use of version 1.2 of the Java Development Kit. While version 1.2 is the latest version available, and has many time-saving features, some CORBA implementations do not support it very well.

As our existing Informix database was running under Solaris 2.5.1, I decided to develop the prototype DataServer on the same platform.

Based on the industry press, Visibroker for Java appeared to be the best CORBA implementation available. It also has excellent Java support. We found in preliminary tests that with the proper configuration, it would work well with Java version 1.2.

4.2. Mapping out the Information Schema

Before the DataServer could begin implementing the query interface, it needed to learn about the underlying information schema. The original idea was to write out the information schema in XML by hand, but it quickly became apparent that the DICOM

database schema was far too complex to manually express in the new data model. For the server to meet the goal of flexibility, it should be able to adapt to any information schema.

4.2.1. Development of the ClassMapper

To accomplish this goal, a tool called the "ClassMapper" was created. The ClassMapper is given a list of tables in the database and derives the information schema for all of the tables, including the relationships between tables. The progression of the tool is quite illustrative of the roles that XML can play in a software project.

The ClassMapper tool was originally a separate program that simply printed out an XML file for the server. The tool was later modified to read the information schema into a set of data structures derived from the XML DTD. The data structures were then serialized to create the XML file. Finally, the tool was integrated into the server itself. It was easier to use the ClassMapper in the initialization stage of the server instead of going to the trouble of writing out an XML file and then reading it back in.

To summarize, the XML was first used as a data modeling language, then used for creating data structures, then as an object serialization language, and finally not used at all. The XML was knocked away as if it were so much scaffolding, leaving the data model and data structures to stand on their own.

This demonstrates the versatility of the XML for describing a wide variety of structured data, and the utility it can provide throughout a software project.

As an aside, the XML parser was so easy to use that the server configuration file was changed from a tab-delimited format to an XML format.

4.2.2. The ClassMapper Algorithm

The algorithm for the ClassMapper is briefly expressed in the following pseudocode:

```
for each table i from provided list
  readTable(i)
map relationships between all known tables
```

The readTable function builds a Class and ClassDef definition from a table and its rowtype. It also finds all the columns in the table and derives Attribute definitions for each one of them. The attribute mapping function supports distinct types, row types, arrays, as well as the basic SQL datatypes.

The relationship mapping phase is where ClassMapper looks for relationships between tables, specifically primary key to foreign key relationships.

If ClassMapper sees two tables with a primary key to foreign key link, it creates a one to one relationship between them.

One to many relationships are more complex, as ClassMapper must look for relationships made to link tables. A "link table" is a table used to establish a one to many (or many to many) relationship between two tables in a relational database. The list of tables initially passed to ClassMapper is a list of all the non-link tables in the database. The user must pass in the names of these tables since there is no easy heuristic to determine if a table is merely a link table or something more.

This list of "known" tables makes the one-to-many relationship process straightforward. If the ClassMapper sees two known tables related to each other via an

unknown third table, it assumes that the unknown table is a link table and creates a one to many relationship between the first two tables.

The ClassMapper copies the relationships of parent classes to their children, ensuring that relationships are properly inherited.

Once ClassMapper has finished processing, the server has a complete information schema for each database.

4.3. Query Generation

The next challenge in the implementation of the DataServer is query generation. As discussed in section 3.4.6, the query mechanism works by specifying a set of criteria, and then specifying a target class. Performing a query asks the DataServer, "Please give me the list of all objects in the target class that meet these criteria." While this question is simple for the user to pose, it is complex for the DataServer to answer.

The DataServer needs to join each of the classes used in the criteria with the target class. To figure out if a join is even possible, the DataServer finds a chain of relationships between each of the criteria classes and the target class. If no relationship chain exists, then the criteria class has no relationship to the target class, and can be ignored. If an unrelated criteria expands the result set, then the result set stays the same. If the unrelated criteria refines the result set, then the result set immediately becomes null and processing ends.

4.3.1. *Translation of a Query into SQL*

For object-relational databases in particular, a query results in a single SQL statement being assembled and executed. The process is summarized in Figure 4-1.

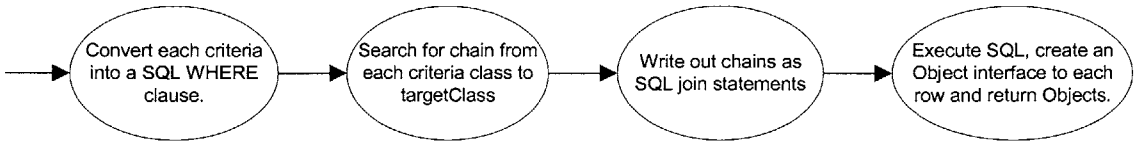


Figure 4-1: State diagram of the query generator

The generated SQL "SELECT" statement consists of three major parts; the list of columns, the "where" clause, and the "join" clause. The where clause of the SQL statement instructs the database what conditions the returned data must meet. The join clause is the latter part of the where clause, and tells the database to "join" multiple tables to each other. The "from" clause, which lists the tables involved in the query, is derived from the where and join clauses.

The only column selected is the primary key of the target table. The target table name comes from the target class. If any results are returned, this primary key value will be used to retrieve additional fields.

Each criterion is translated into a single where clause, which are connected with AND or OR depending on whether the criteria expands or refines the result set. The class, attribute, and comparison operator are translated into the database equivalents that were discovered by the ClassMapper. If the datatype is a distinct type, the query generator applies the correct type cast.

The most complex part is assembling the where clause. As discussed above, the DataServer needs to find the chains of relationships linking criteria classes and the target class. This is done using a breadth-first search algorithm. Once a relationship chain is found, the proper SQL is assembled from the information recorded by the ClassMapper.

Once this is done, the SQL statement is finalized and submitted to the database. The results are communicated to the user as a series of Object interfaces.

4.4. Implementing the Object

The Object implementation performs the difficult work of translating the database query results into data types that the client can understand.

When the Object is created, it does nothing; this way, if the user decides to not retrieve attributes from an Object, no time is wasted. When a user performs a query for some or all attributes, the Object queries the database and retrieves the values

The process of generating the SQL statement in this case is very easy. The names of the desired columns are added to a where clause specifying the primary key of the object.

To improve performance, the Object caches data results from the database, so repeated queries will not result in a database query each time. If the user needs to ensure that the object is always up-to-date, it is easy to request a "fresh" Object using the getOID() function in the DataServer interface.

After the database query is completed, the data value is retrieved with a JDBC getObject() call, which returns a Java Object. This function call is quite special, as it

automatically converts the SQL data type into a Java data type. This function call effectively eliminates a large part of the "impedance mismatch" between Java and SQL.

With the Java Object in hand, a procedure uses the Java "instanceof" operator to find out what the type of the Object is, and converts it to a CORBA Any type. This CORBA Any type is then returned to the user.

The same process happens in reverse when attributes are set and the Object is updated.

4.5. *Handling Database Connections*

In order to provide efficient service, the server uses database connections only as necessary. When a DataServer is created, a ConnectionPool is found (created, if necessary) and given to the DataServer. This process is illustrated in Figure 4-2.

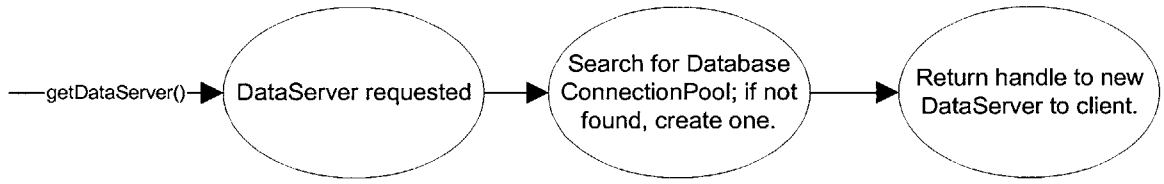


Figure 4-2: State diagram for DataServer object creation

The ConnectionPool is a class that holds a set of JDBC database connections. When a connection is needed, a function takes a connection, uses it, and then puts it back. This is much more efficient than creating a connection from scratch and then destroying it.

The server uses the database ConnectionPool class written by Jason Hunter for his book, Java Servlet Programming.[49]

This system proved to be efficient and easy to use. Best of all, the connection pooling was hidden entirely below the interface layer. From the interface point of view, it appears that a single database connection is always active.

4.6. *CORBA Development Issues*

Under Visibroker, developing for CORBA was not difficult. The process for

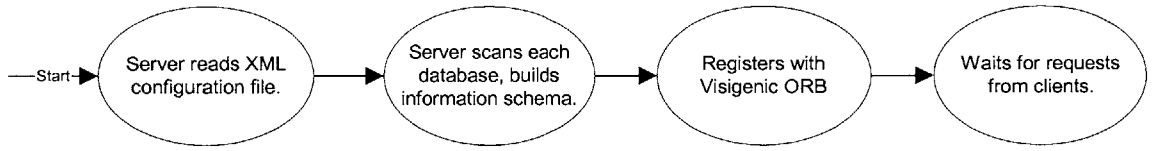


Figure 4-3: State diagram for initializing the server.

making a server available is diagrammed in Figure 4-3.

First, the server reads in the configuration file using the XML parser. Then, the server scans each database listed in the configuration file with ClassMapper and builds the information schemas. The server could be optimized by caching the information schema, since this data is unlikely to change often.

After this setup is completed, the server is ready. It registers with the Visibroker ORB, and then goes into an event loop to await client connections.

In general, there were few difficulties working with the Visibroker toolkit. Visibroker was highly sensitive to the versions of other software being used. For example, on a Solaris 2.6 machine running Java 1.2.1.02, Visibroker regularly failed to return byte array data by throwing spurious exceptions. Upgrading to 1.2..1.03_prerelease solved the problem.

It remains unclear whether the fault is with Visibroker or Sun. The clear lesson is that the more interdependent software components involved in a system, the more possibility there is for failure.

4.6.1. Testing CORBA Compatibility

As a test of the expandability and flexibility of the server implementation, the server was ported to the Iona Technologies OrbixWeb product, version 3.1.

First, the products differ in the way that servers interact with the ORB.

Visibroker has a program called "osagent" that allows the developer to use the Basic Object Adapter to register objects with the agent.

Orbix appears to have a more traditional approach. Orbix supplies an ORB agent called "orbixd" that acts much like Visibroker's osagent. However, servers cannot register themselves with the Orbix ORB. Instead, developers must run an external program to perform registration.

This means that both ORBs require different, incompatible code to set up a server or client. This is difficult to deal with using Java since the Java language has no support for conditional compilation.

Orbix lacks some CORBA libraries that Visigenic supports. The most glaring omission is the lack of the "DynAny" classes, which allow CORBA programs to dynamically create data types on the fly. Without this functionality, programs are limited to creating types explicitly described in the IDL.

The server could not run under Orbix. Unfortunately, the Orbix ORB has no support for running CORBA servers that require Java version 1.2.

This brief experience was not sufficient to fully evaluate the Orbix software, but it is clear that there can be wide variations between CORBA implementations.

4.7. Conclusions

The server was an enjoyable engineering challenge. It was built from scratch in the span of only a few weeks due to the well-defined interface specification. Minor tweaks were made to the interface as the server was developed, but were related more to improving usability than fixing design flaws.

An especially notable point is that all of the software worked well together despite the "bleeding edge" status of most components.

Chapter 5:
Client Design and Implementation

5.1. Choosing the Client Components

The client design had to be simple enough to write in a short timeframe, flexible enough to handle last-minute changes in the interface, portable across many platforms, secure, and easy to use.

For development speed, extensibility, and flexibility, the Java programming language was chosen. Since CORBA was used, we could have chosen C++, but Java's development speed and CORBA integration made it the clear choice.

Using Java, there are several choices for the client architecture. The client can be a Java application, a Java applet, or a Java web servlet.

5.1.1. The Client as a Java Application

The first choice is to write a Java application. Java applications are just like native code applications. They are manually downloaded by the user, installed, and run in a standalone Java Virtual Machine. The user needs to separately download and install the proper version of a Java Virtual Machine in order to run a Java application. Java applications are not to be confused with Java applets, which are discussed shortly.

Running the client as an application on the host machine adds many unknown variables since it is not known what the client environment will be like. While Java is nominally cross-platform, different implementations behave in subtly different ways. For non-interactive programs, this is not a problem since most of the functions used are well-defined. However, many of Java's graphic user interface functionality is not as well defined, which means that a GUI on one platform can look radically different from a GUI on another platform.

Not only does the client code have to run on the client machine, but the CORBA Java classes must also run on the host machine in order to connect to the CORBA server. Visibroker has been known to act erratically in a controlled development environment, so there is definitely a chance that it will malfunction in an unknown client environment.

In addition, securing the connection between the client and server requires an additional module for Visibroker which takes time to install and configure on both the client and server.

These reasons make it clear that it is not a good idea to build the client as a Java application.

5.1.2. The Client as a Java Applet

The second choice is to write a Java applet. An applet differs from an application in that it runs inside a web browser's Java Virtual Machine when a user visits an applet-enabled web page. It is not installed, and does not run standalone. The applet shares the Java Virtual Machine with any other applets that are running.

While applets do not have to be installed, they suffer from a number of limitations that do not affect applications. First of all, an applet cannot write or read files from the local disk. Second, applets are forced to run in whatever Java Virtual Machine is bundled with the web browser. There is no way for a user to download a newer Java Virtual Machine. Since most major web browsers do not have a Java version 1.2 Virtual Machine, Java applets are restricted to using the old Java version 1.1 GUI classes and data structures..

All of the CORBA problems noted with applications also affect applets. Since applets are only allowed to create network connections back to the server they were downloaded from, this means that the CORBA DataServer would need to run on the same machine as the web server.

Because of these limitations, Java applet is not a viable solution either.

5.1.3. The Client as a Java Web Servlet

The third choice is to write the client as a Java web servlet. A Java web servlet is a Java class that runs inside a web server and communicates via the Hypertext Transfer Protocol (HTTP). Like an applet, a servlet runs inside a common Java Virtual Machine with all of the other servlets. When a web request is received for a servlet, the web server first looks for an existing instance of the servlet. If none is found, a new instance is created and a thread made to answer the web request. The servlet stays running in the web server until a set idle time elapses. After the idle time elapses, the servlet is destroyed by the web server.

A Java servlet is much better than a traditional Common Gateway Interface (CGI) program. A CGI program spawns a new process for each request. There is no interprocess communication, and significant wasted overhead is created. A Java servlet runs continuously as a single process with many answering threads. Since there is only one instance of the servlet, the threads can collaborate with each other and share information.

Since the Java servlet runs inside the web server, the environment can be carefully set and tuned. The web server can use version 1.2 of the Java Virtual Machine, and the CORBA libraries will work with the servlet as if it were any other local client.

The tradeoff is a lack of control. While the Java GUI is clunky, it does offer fully controlled interactivity. The Java servlet can only communicate via static web pages. This is not as bad at it used to be, thanks to client-side JavaScript and Dynamic HTML.

JavaScript is a client-side web browser scripting language that allows a developer to program a web page to react in certain ways to user input. For example, if a user moves the mouse pointer over a particular icon, the icon can be changed to a different image. JavaScript's name is unfortunate because it has absolutely nothing to do with the Java language.

Dynamic HTML is a set of extensions to HTML that introduces a Document Object Model (DOM) for HTML which gives JavaScript users a standard set of objects to manipulate in a web page. JavaScript is well supported by most major web browsers, while Dynamic HTML is supported in different ways by each browser.

Finally, a Java servlet can be easily secured by sending pages over Secure HTTP (HTTPS). All major web browsers support HTTPS.

For these reasons, the Java web servlet architecture bests meet the needs of the client application.

5.1.4. Software Infrastructure

For the supporting software for the Java Servlet, the best option is the Apache Web Server version 1.3.3 with mod_ssl and the Apache Java Servlet Engine. The Apache Web Server is the most popular web server available.[33] The mod_ssl component for Apache allows the Apache web server to perform secure HTTP. Finally, the Java Servlet Engine adds servlet capabilities to the Apache web server. All of these products are open source and free of charge.

5.2. Client Design

The state diagram for the servlet is illustrated in Figure 5-1. The two large boxes symbolize the two types of pages offered by the servlet. The round circles symbolize actions taken during user requests.

The Login screen is the first screen shown a new user, and is shown in Figure 5-2. If the user presents the appropriate username and password, they are sent to the Query/Results screen, shown in Figure 5-3. This screen allows the user to build a query and at any point look at the results from the query.

If a user stops using the servlet for a certain period of time, the user is not allowed to use the servlet anymore and must log in again.

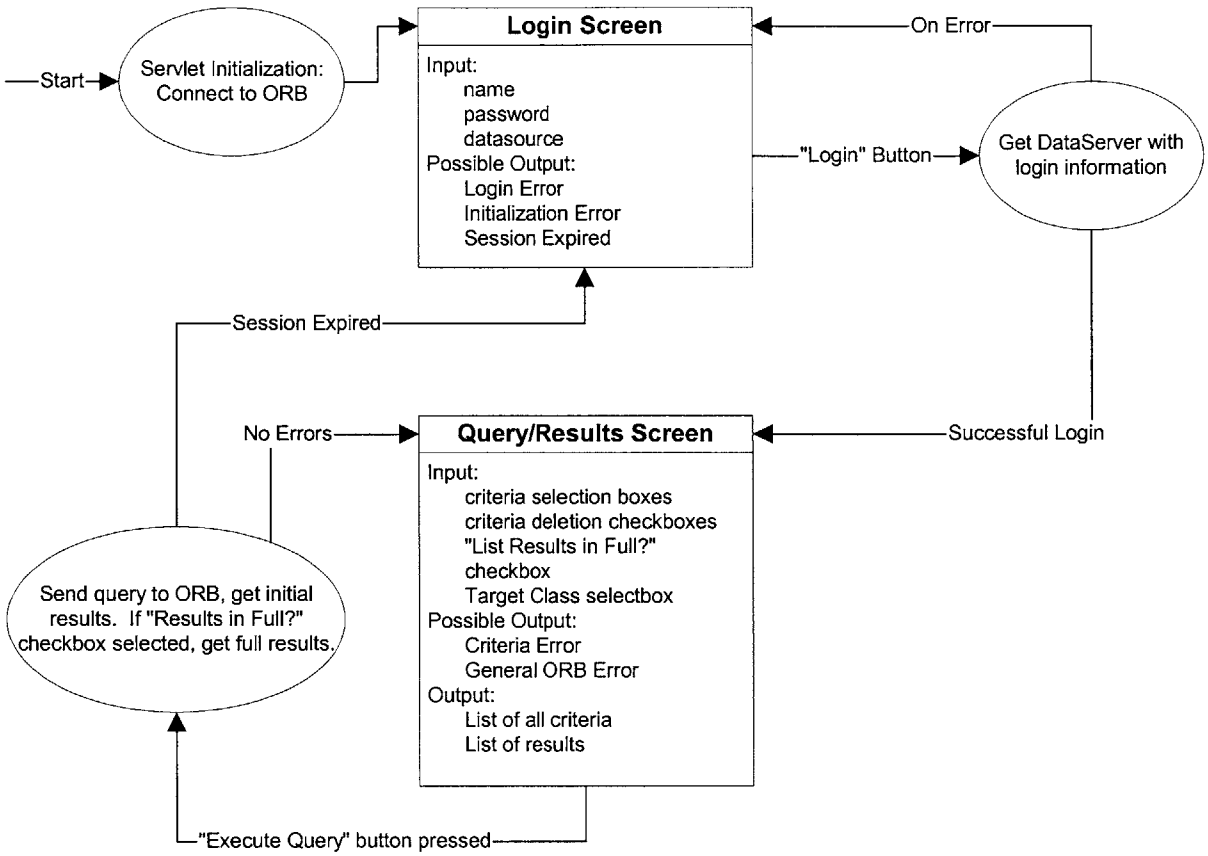


Figure 5-1: State diagram for web servlet client

5.3. *Java Servlet Issues*

Writing the servlet was very straightforward. There were no CORBA integration problems, and no web server integration problems.

The Java Servlet Session API made it very easy to track users. Session API wraps user tracking behavior in an abstract interface. The implementation uses HTTP cookies or URL rewriting to track the user's progress.

The integration with the Apache Server was good. The servlet can query the server about the type and security of the connection. This is used in the Login screen (Figure 5-2) to print the exact security status.

5.4. HTML and JavaScript Issues

In order to provide an interactive user experience, the servlet uses JavaScript to enable a set of selection list boxes with the possible choices for the user.

When a query class is selected, the JavaScript automatically loads the class attributes into the the attribute selection list. When a fields is selected, the appropriate operators are shown in the comparison selection list. While this results in a good experience for the user, it requires that the page have a list of all the classes, their attributes, and the attribute data types (to discern with comparisons are available.) In effect, the entire data schema is transmitted each time to the web browser as part of the embedded JavaScript.

It is possible to have the appropriate fields initialized through queries to the servlet, but the time it takes to connect to the web server, request an entirely new page, and display it takes much longer than the instantaneous response the user expects.

Some sort of capability to cache large amounts of data on a web client in a persistent fashion would certainly make it easier to provide access to a large store of data such as the information schema.

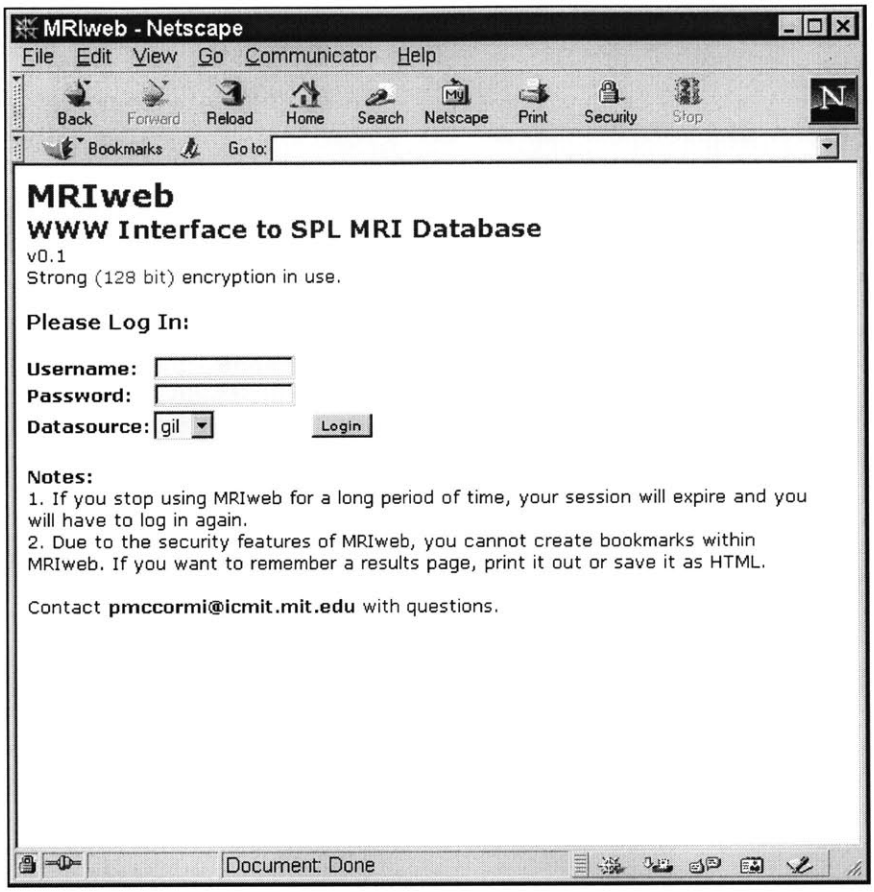


Figure 5-2: Screen capture of the client login screen.

One possible answer to this problem is to write a Java "mini-applet" that performs this task. This tiny Java applet would not have to connect to the ORB, or use the advanced Java features in version 1.2, but could offer a small GUI to browse the information schema. This option was not fully explored, but may be implemented in the future.

Uploading files to the web server is difficult but possible with the seldom-used HTTP PUT option. We found that this method worked well for individual files, but not forlarge sets of files since most browsers' implementation of PUT requires that the user select one file at a time.

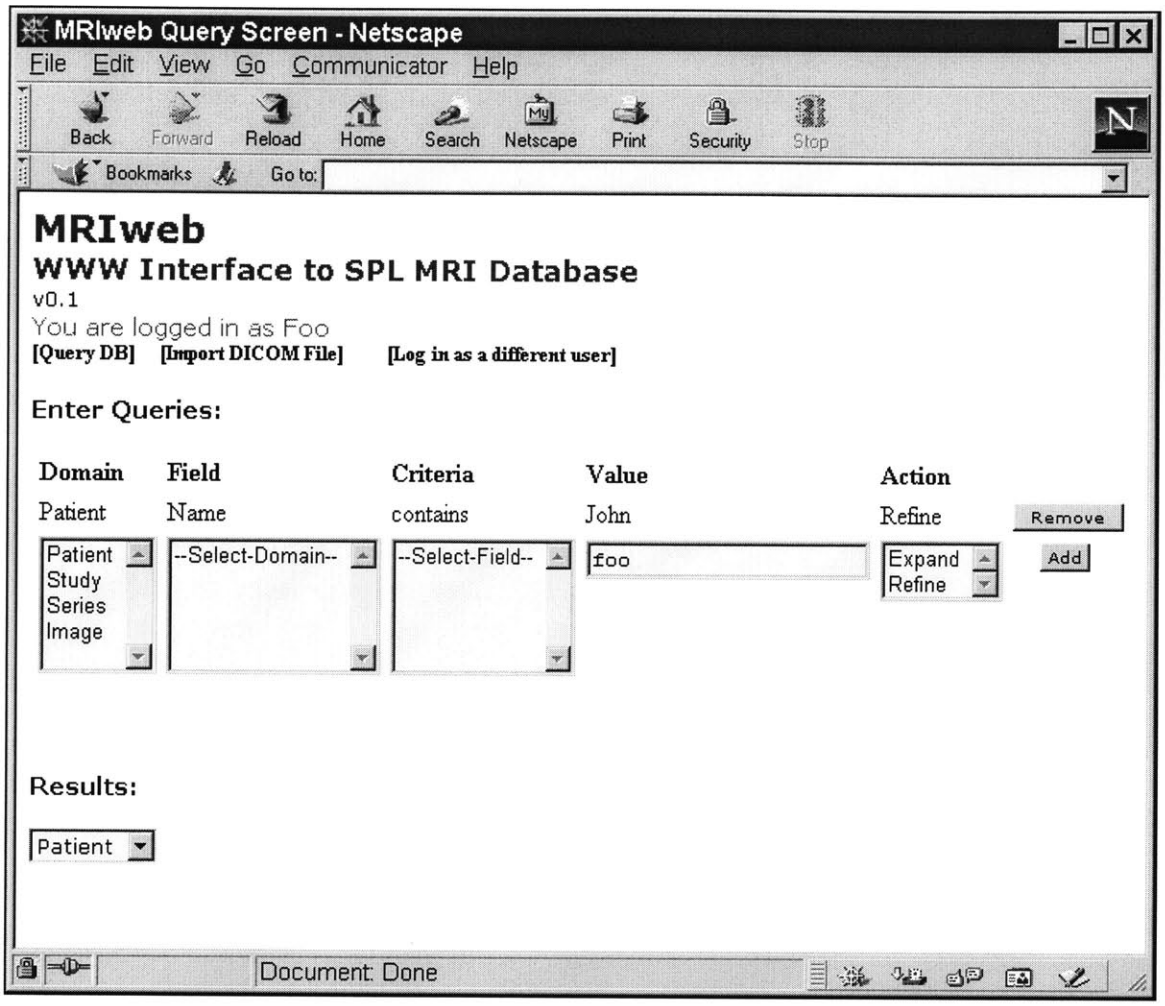


Figure 5-3: Screenshot of client query screen.

5.5. Conclusions

Writing the client as a Java servlet was a good choice because of the fast development time, easy CORBA integration, and wide portability. However, because of the request/response nature of HTTP, it is difficult to provide constant interactivity. Future advances in HTML and HTTP promise to make web browsing more interactive. Also, while writing the entire application as a Java applet is not a good idea, small portions of client functionality can be provided by Java mini-applets.

Chapter 6: Results

6.1. Deploying the DataServer

The software on the deployment machine at the Brigham and Women's Hospital was intended to be identical to the development machine. The sole exception was the operating system version. The development machine runs Solaris 2.5.1, while the deployment machine runs Solaris 2.6.

This exception proved to be crucial, as the latest versions of the Java Development Kit are only available for Solaris 2.6 and 7. Also, prior to upgrading to the latest JDK pre-release, certain features of Visibroker would not function on the deployment machine (as discussed in section 4.6) while Visibroker worked fine on the development machine.

Special care was taken during development to make the source code of the DataServer as machine-portable as possible. Variables such as file locations and network addresses were set in configuration files instead of being hard-coded into the program. The user experience involves unpacking the source code and configuring two files. At this point, the user can build and run the DataServer.

6.1.1. Loading the Database

In order to load the database with a set of test records, a program written in the Perl scripting language was built to navigate the SPL file hierarchy (illustrated in Figure 2-1) and load the images into the database with the proper identification numbers.

This was a moderately difficult task due to the different kinds of images available and the variable quality of the image header data.

The files are stored as Genesis and Signa images. These files can be converted into DICOM Part 10 files using David Clunie's "dicom3tools" software. The conversion software converts the image header tags into DICOM tags. However, on some of the images, the image header contains incorrect data. This can lead to misplacing images. For example, an image that belongs in series 5 may have an image header that says it should go into series 1.

To correct this problem, the series, image, study, and patient numbers are derived from the file hierarchy. With these numbers, a set of DICOM unique identifiers (UIDs) are created for each image.

To preserve backward compatibility, the original filename of the image is stored in the database. This provides an easy way to convert programs from using the filesystem to using the database.

The Perl programming language proved to be an excellent choice because of its easy string manipulation capability, and the ability to easily work with files and external programs.

Once the DICOM file has been converted, a short Java JDBC program loads it into the database. The query interface could be used, but is unnecessary overhead in this case.

6.1.2. Performance and Usability

A database was created with nearly 800 records for testing the user interface and the DataServer. The implementation worked well overall, providing correct results to difficult queries.

We installed the database, the DataServer, and the CORBA ORB on the same machine to eliminate network traffic as a factor in performance testing.

Performance of basic query and retrieval functions appeared to be reasonably fast, but bottlenecks were apparent in image handling. In order to learn more about the nature of these bottlenecks, a specially instrumented build of the DataServer was created. This DataServer reports the elapsed time for the getAttribute functions. A special client was created to track the elapsed time of receiving values from these functions.

The test client retrieves a single 152KB image from the database. The client uses two different modes of image transfer. First, the client uses a streamed transfer (via a custom data streaming interface) with 10KB blocks and 50KB blocks. Second, the client simply requests the pixel data as a single large return value.

Task	Time (secs)
Retrieve data from database	1.7
Create Any value to store data	0.8
Retrieve data value as return value	1.4
Retrieve data value via stream	0.1
Total time to retrieve image	3.9

Table 6-1: Performance timings

Three stages were identified to be potential bottlenecks:

- Database column value retrieval

- Creation of the CORBA Any value.
- Transmission to client.

Database connections and disconnections are not included because the connection pooling eliminates this overhead.

Table 6-1 has a list of the average time it took the test client and test server used to accomplish each of the bottleneck tasks.

The bottleneck appears to be primarily in the database retrieval stage. It is not immediately clear what can be done to reduce the time it takes to retrieve the image data. The database query is very simple, consisting of a single column lookup in a single table. The result fetch is performed through the JDBC "getObject" call; there may be a better way to use this function call, or a better way to use getObject.

The creation of the Any value takes a significant amount of time, and points to the overhead added by using CORBA. This overhead may be endemic to all component software systems.

It is clear that retrieving the data as a stream versus as a return value is optimal. The reasons for this probably have to do with the specifics of how the CORBA implementations are designed, and are outside the scope of this performance review.

Overall, the DataServer shows that there are definitely performance areas that can be improved upon at the interconnections between the different components.

6.2. *Extensibility*

The query interface is very extensible due to its construction, and because it is not dependent on CORBA. The query interface can be expressed as a set of Java Beans, COM components, or as an Application Programming Interface. This would take some time, because the implementation would have to be revised to fit the new interface. However, since the basic concepts remain the same, this revision would largely concern how values are passed and returned, and would not affect the core code.

The server implementation currently works with Informix object-relational and relational databases. Since the code uses JDBC and SQL, it would not be difficult to make the implementation work with a different database.

The client implementation can be moved to any other Java Servlet web server, with some slight modifications. Of course, any revisions in the query interface would require revisions in the client.

Since Java is a cross-platform language, the server and client implementations will run with any platform that supports Java version 1.2 and the Visibroker ORB. Recompiling is not necessary since Java object files are portable from machine to machine.

Also, Java's pure object oriented nature makes it easy to subclass from the existing server and client implementations to produce more capable implementations.

Together, the Java platform and a component-agnostic design philosophy meet and exceed the goal of extensibility.

6.3. Managing Multiple Information Domains with User-Defined Relationships

We experimented briefly with the use of user-managed relationships, and found that they were not as useful as we had hoped. Part of the reason is that the DataServer was never used to access different information schemas.

Within a well-designed information schema, there should be no reason to establish arbitrary links between objects. Even with a badly designed information schema, these arbitrary object-to-object relationships become merely "band-aids" to make up for the lack of a proper class-to-class relationships. It is better to fix the schema than to apply arbitrary relationships to "fix" the problem.

Cross-schema relationships are a different matter. There is often a need to associate objects in one domain with objects in a different domain, but this is not possible because the domains are completely different, often occupying separate databases. The DataServer shows that databases can be brought together through the use of abstraction layers, but the only way the abstraction can be useful is if data in multiple databases across multiple domains can be correlated properly.

For the multiple domain problem, arbitrary relationships are a useful start. However, they are by no means solution. For awhile, arbitrary relationships will serve as a useful glue to bind together data that needs to be associated. But as the number of relationships begins to soar, performance will grind to a halt because individual object-to-object relationships do not scale up to replace class-to-class relationships.

This means that the only way multiple databases can be bound together is with yet another information schema, a meta-information schema that is able to create a loose confederacy of data. The "loose" part is emphasized because if a tight, ordered federation is desired, it is best to simply refactor the domains into a single, common domain. Of course, this is often not possible, especially with data sources that are very old and cannot be easily rearranged.

The user-defined relationships are a good idea. In a way, they give users a method for experimenting with their own data schemas by building their own relationships. They act as a temporary solution for shortsighted information schemas, and as a suggestion box for schema improvements.

6.4. Conclusions

The first results of the design and implementation of the query interface for generalized object-oriented data sources are the many ideas and concepts discussed and experimented with in the preceding pages.

Another set of results are the DataServer and Client implementations which demonstrate that the principles discussed earlier result in good software that can be extended to work with new systems and embrace new functionality.

The most exciting results of the DataServer work are the avenues of research that the DataServer opens up. If we can conceptually abstract a single database, why not abstract a set of databases, each with its own particular information domain?

Chapter 7: Conclusions

7.1. Software Design

Software engineering is a relatively young field, so there is still much to be learned in the areas of software design and software project management. This project used concepts including design patterns, interface abstraction, and rapid development cycles. While these concepts are not new, this project explored effective ways to apply them.

The common thread through modern software engineering techniques is to enhance extensibility while maintaining functionality. Most projects deliver the required functionality, but are impossible to extend. Some projects are very extensible, but deliver limited functionality or poor performance. In the future, hopefully it will become easier to deliver both extensibility and functionality, instead of trading more of one for less of the other.

7.1.1. Application Servers

This project showed how difficult it can be to use a distributed object system, and gave a glimpse of the potential payoffs. The software world is moving toward a time when most software will be written in the form of components, so that modern

applications consist of nothing more than several objects strung together. A useful step forward for this project would be to turn it into a set of components served from an application server. An application server is a relatively new type of product that handles the management of components and maintains a separation between business logic and system logic. Something like this could provide benefits to academic research, where new software is always being developed and tested. "Rapid development" techniques can be translated into "rapid deployment" techniques with application servers.

Many of the same hopes were held for the Java language and Java applets in particular, but slow implementation, an excessively stringent hold on the language by Sun, and the need to remain truly cross-platform convinced the software community to look to broader solutions. Hopefully, the multi-vendor application server offerings will make it easier to concentrate on building software.

As for the component standards, Enterprise JavaBeans will probably be the platform of choice for future advancements of this project. CORBA is unfortunately too deeply rooted in the old, procedural languages to be flexible enough for the future. Enterprise JavaBeans are reaching maturity, and provide a more cross-platform alternative than Microsoft COM.

7.1.2. Industry Standards

This project would not have been able to achieve the goals of extensibility and portability without the use of several industry standards. Industry standards, including protocol standards, data exchange standards, and semantic standards, are all crucial to

moving software forward. The XML, UML, and HTML are just some excellent examples of how industry agreement can benefit everyone.

However, standards need to be nurtured. Without active development of implementation and cooperation with standards-making bodies, the software industry will be forced to convey complex data through poorly defined and implemented standards. This is equivalent to pushing a watermelon through a straw, and about as effective.

7.2. Future Directions

As discussed above, in the future this project could be moved from a CORBA architecture to a JavaBeans architecture. As JavaBeans will be using the CORBA wire protocol in the future, a JavaBean implementation would still be compatible with the existing CORBA clients that have been written.

Another project would be to develop more mappings between the data model used by this project and other data sources, such as object-oriented databases and real time data sources.

An important path to explore in the future is the integration of multiple information domains as represented by multiple data sources. Information domains are artificial barriers that need to be overcome in order to integrate the vast amounts of data available in scientific research and other areas. A good example of this project is a system that integrates genetic patient data with human genome mapping data. Data-mining here could mean the identification of latent traits, or new insights into the functionality of a specific gene.

It is hoped that this project will be useful for searching many kinds of medical and academic databases in the short term, and an impetus to integrating every kind of research database in the future.

Appendix A: Project Infrastructure

This project was large by academic standards. It covered two geographically separate sites, over ten computers running three different operating systems, and relied heavily on the DICOM database schema project which was still active as this project was moving forward. To keep the project under control, I experimented with several systems that promised to make it easier to get the job done.

In most cases we limited our search to open source products to capitalize on the increased reliability, extensive support, and easy customizability. The disadvantage of open source products is that they generally have less features than their commercial counterparts and are slow to embrace new standards.

Throughout this section, references to URLs are indicated as footnotes, not to entries in the bibliography.

A.1. File Sharing

Our primary client platform is Microsoft Windows NT Workstation 4.0[1]. Our primary server platforms are Compaq Digital Unix 4.0D[2] and Sun Solaris 2.5.1[3]. The

---

[1] http://www.microsoft.com/ntworkstation/default.asp

Digital Unix machine is the primary data store with a 16 GB StorageWorks Redundant Array of Inexpensive Drives (RAID) array. The RAID array is regularly backed up to the M.I.T. campus mainframe.

To share files between the Solaris and Digital Unix platforms we used the Network File System (NFS) protocol[4]. Once the initial system administration issues were resolved, we had no problems with this solution.

To transparently share files between the NT and Unix platforms, we initially experimented with PC-NFS for Windows NT from Intergraph Solutions[5]. Unfortunately, we did not get the excellent results we saw between the Unix platforms. The NFS protocol is based on Unix filesystem semantics, and the NT filesystem semantics are sufficiently different that we experienced performance and usability problems. All stages of use from authentication to actual file transfer resulted in poor end-user experiences.

The solution to this problem is a product called Samba[6] which provides NT file sharing services on Unix. Samba maps the NT filesystem semantics to Unix, which proves to be much easier than the reverse case. Samba performance meets or exceeds NT Server performance in empirical tests, and the end-user experience is excellent at all stages of operation.

---

[2] http://www.unix.digital.com/

[3] http://www.sun.com/solaris/

[4] http://www.cis.ohio-state.edu/htbin/rfc/rfc1813.html

[5] http://www.intergraph.com/nfs/

[6] http://samba.gorski.net/samba/samba.html

Samba allowed us to expose the Web, FTP, and other important shared directories on group file server. This changed the traditionally laborious task of updating FTP and web server files into a point-and-click proposition from Windows NT workstations.

For file sharing with other institutions, we used the Apache Group web server[7] and the NCFTP Software FTP server[8]. The Apache web server is very modular and has many extensions such as secure HTTP, Java servlets, and in-process Perl CGI execution that are necessary for our research. The NCFTP Software FTP server is a very robust and scalable FTP server that allows for "virtual" users and restricted accounts. These features were necessary so we could set up FTP accounts for other institutions without worrying that someone would steal the login password and break into the system.

The Apache web server is open-source and free, while the NCFTP Software FTP server is proprietary and free for educational institutions.

A.2. Electronic Mail Services

Due to our research group's active nature and involvement with several external sites, our electronic mail needs went beyond simple message exchange.

Our group needed:

- On-server mail storage, since we need to access our personal mail archives from a variety of sites and platforms
- Flexible, automatic mailing lists

---

[7] `http://www.apache.org/`

[8] `http://www.ncftp.com/`

- Message archiving system for industry mailing lists
- Ability to handle all mail protocols with Secure Sockets Layer
- Ability to handle mixed-media documents
- 100% reliability

The standard M.I.T. Information Systems mail system, while very reliable, did not meet any of the other needs listed above. We decided to experiment with assembling our own mail server, potentially sacrificing reliability for features. Despite this risk, our reliability remained near 100%. Over two years of constant use there has been no major service outages and no loss of data.

For on-server mail storage, we examined several mail servers that use the Internal Message Access Protocol (IMAP)[9], a standard protocol for server-side mail storage. IMAP borrows heavily from the news server paradigm, where messages are stored in hierarchical trees of folders. IMAP adds strict security to this model, ensuring that mail is only read by the intended recipients. By storing mail on the server, IMAP allows users to use a multitude of mail clients, and to have full access to their mail archives from anywhere on the Internet. This is in contrast with the Post Office Protocol (POP)[10] which uses a store-and-forward paradigm for mail. POP users store their mail archives on the local machine, and are unable to access their archives remotely or use different email clients.

---

[9] `http://www.cis.ohio-state.edu/htbin/rfc/rfc2060.html`

[10] `http://www.cis.ohio-state.edu/htbin/rfc/rfc1939.html`

We chose the Carnegie-Mellon University Cyrus IMAP[11] server for our laboratory. It was easy to install on the Digital UNIX platform, and requires little to no maintenance. For backward compatibility, the Cyrus server supports POP as well as IMAP. We used Sendmail 8[12] as our mail transfer agent.

Users of the IMAP server are pleased with the ability to read their electronic mail on multiple remote machines using any IMAP client. The most popular IMAP clients are Microsoft Outlook 98[13], Netscape Communicator 4.5.1[14], and the University of Washington's Pine 3.96[15].

All of these electronic mail clients support the MIME standard, making it easy to trade mixed media files with other researchers and within our group.

Several national mailing lists that are relevant to our research are archived on the IMAP server in special folders that all users can examine. This allows newsgroup-style browsing of these lists, and eliminates the need for everyone to individually subscribe and filter the mailing lists.

For running our own mailing lists, we installed the Majordomo list server[16]. This program is written entirely in the Perl[17] programming language. It provides list

---

[11] http://andrew2.andrew.cmu.edu/cyrus/imapd/

[12] http://www.sendmail.org/

[13] http://www.microsoft.com/outlook/

[14] http://home.netscape.com/communicator/v4.5/index.html

[15] http://www.washington.edu/pine/

[16] http://www.greatcircle.com/majordomo/

moderation, list archiving, and silently handles user subscribe and unsubscribe requests. We use Majordomo for lists where members were added and removed on a regular basis. One of the moderated lists hosted by our server has over 1,600 subscribers and a turnover rate of roughly ten to twenty subscription changes each week.

To provide security for our email connections, we use the Secure Sockets Layer proxy (described below) to wrap our IMAP and POP connections. Of the email clients listed above, both Outlook and Communicator support SSL mail connections.

This network of products satisfies our electronic mail needs in a long-term fashion.

A.3. Remote Access Computing

Given the wide variety of platforms that our group uses on a daily basis, it is often necessary for a single user to be using two or three computers at once, where each machine runs a different operating system. We searched for a way to remotely access other machines in a way that offered speed, graphical interaction, and security.

For secure remote access, we could use a variety of secure terminal programs, some of which are described in the next section. However, none of these terminal programs allowed graphical interaction with the remote machines.

For a long period of time, we have been using the X Window System to remotely access Unix machines. The X Window System allows programs, known as "X clients," to run on remote desktops known as "X Servers". Note that the traditional concept of server

---

[17] `http://www.perl.com/`

and client is reversed here. The communications protocol used by X Windows is a "thin client" protocol. Typical messages include "mouse moved to this position", "this image was drawn at this location", and "create a new window."

While this system works well, there are some important disadvantages. First, the X Window System works only for remotely accessing Unix machines. Windows programs cannot act as X clients.

Second, most X servers on the Windows platform require a license on each machine which we did not want to pay for.

Third, because of the inverted client/server model used by X Windows, it is not possible to remotely access machines from behind a firewall. A terminal program can operate without problems from behind a firewall because it initiates a connection to the remote server. As long as the outgoing port is not blocked by the firewall, the remote server is free to communicate over the client-constructed channel.

In X Windows, a user behind a firewall must request an X client on a remote machine to connect to their X server. Since the X client is often unable to penetrate the firewall, it cannot make a connection to the user's X server. This need is important to our group since this researcher would often work from a remote location protected by a firewall. This problem also causes performance difficulties, since each X client is forced to maintain an individual connection to the X server. While this works well in a local area network environment, performance degradation occurs over a wide area network.

Finally, display state is stored on the user machine with the X server. This means that if the user turns off their machine, all of X clients terminate. This is frustrating, especially if the user wishes to run lengthy processes on remote machines.

After using the X Windows System for some time, we discovered a newer technology known as Virtual Network Computing[18] (VNC). This system was developed at the Olivetti Research Laboratory in Cambridge, England. VNC was originally developed as a thin-client access method for network computers. VNC addresses all of the deficiencies noted above with the X Windows system.

First, the VNC model is client/server in the traditional sense. A server runs on the remote computer, and clients can connect to it. This immediately addresses the multiple-connections issue and the firewall issue. Also, this means that application state remains entirely in the hands of the host computer, making it easy for a user to stop what they are doing, move to another terminal, and immediately resume working without any interruption.

Second, VNC has been ported to a wide variety of platforms. Servers are available for many varieties of UNIX and Windows. The UNIX servers are actually X servers that use a VNC network connection instead of a video display. This allows for total compatibility with existing X window systems.

Finally, VNC is open-source and free, so there are no licensing fees to pay.

A.4. Security

Security is always a priority for any computer network, but in our case the need was particularly acute. First, our research often involves sensitive medical data. Second, MIT's visibility makes it a constant target of sustained network attacks, often in the form of "packet sniffing," where computers are compromised and used as listening posts to gather passwords and data passed in the clear across the campus network.

The services we desired were:

- Secure login
- Secure email
- Secure web service
- Secure ports for applications such as VNC

For secure login, we implemented two different options. First, we installed the MIT Kerberos network authentication system[19]. Kerberos provides secure authentication services for FTP, telnet, printing, IMAP and POP mail, along with several other common network services. Kerberos also provides some basic encryption for the telnet application; all other services transmit data unencrypted.

The primary problem with Kerberos is that it has very poor application support. Only one Windows NT mail client, Eudora Pro, supports "Kerberized" POP mail. Eudora

---

[18] http://www.uk.research.att.com/vnc/

[19] http://web.mit.edu/kerberos/www/

does not support Kerberized IMAP mail. There are no FTP servers except for M.I.T.'s own implementation that support Kerberized FTP.

Also, since Kerberos was designed around the concept of securing a single network domain, it does not work well across domains. Cross-domain security is a priority for our projects since we work with many different institutions.

The next option we experimented with was the Secure Shell (SSH) protocol.[20] SSH is designed as a drop-in replacement for the Remote Shell (rsh) functions provided by most Unix operating systems. SSH uses strong cryptography to secure all transmissions. For authentication, SSH uses either a standard challenge/response system (encrypted) or an RSA certificate.

SSH also provides secure file transfer with the "scp" program, and can enable encrypted port-forwarding during SSH sessions. "Port-forwarding" means that ports on the remote host can be "forwarded" to the local machine over an encrypted tunnel. Applications on the local machine merely point to the local end of this tunnel, and communicate with the remote server in a secure fashion without being aware of the encryption.

This enables the use of otherwise insecure programs in a secure fashion. For example, when I use VNC from my home computer, I first start an SSH session to the server. The server forwards the VNC port to the port of the same number on my local machine. Then, I start the VNC client and tell it that the server is at "localhost". The

---

[20] `http://www.ssh.org/`

client sends requests to my local port, and SSH sends the request and receives the response over the encrypted channel.

In this way, SSH met our needs for secure login and secure applications. However, we still had the problem of secure email and web service. While we could use port-forwarding for these issues, we wanted a solution that was more flexible and scalable.

Many modern email and web clients have support for encrypted sessions using the Secure Socket Layer (SSL) protocol[21], including Microsoft Outlook, Microsoft Internet Explorer, and Netscape Communicator.

I compiled the SSLeay toolkit[22] to add SSL capability to my server applications. For web service, I used the mod_ssl Apache web server patch[23] with the latest version of the Apache web server. For email service, I used the sslproxy program[24] from Objective Development to wrap the POP, IMAP, and SMTP ports. A solution that integrated with the Cyrus mail server software would have been preferable, but that was not possible. All of this software is open-source and free.

In addition to these software packages, we maintain a highly aware security posture on our critical systems. Log files are checked regularly for possible intrusion

---

[21] http://www.netscape.com/info/security-doc.html

[22] http://psych.psy.uq.oz.au/~ftp/Crypto/

[23] http://www.modssl.org/

[24] http://www.obdev.at/Products/sslproxy.html

attempts, and we stay up-to-date on operating system security holes and patches through the BUGTRAQ mailing list.[25]

A.5. Revision Control

In our research group, each researcher generally works on a separate project. Each project is strongly related to each other, but actual source code level collaboration is relatively rare. However, when I began working on this project I found that I needed to make several changes to a separate project owned by another researcher. I also found that in my personal projects, I needed a way to easily revert to older designs. I looked for a revision control system that would meet both of these needs.

I tested the Revision Control System[26] (RCS), but found that I did not like the file locking semantics. In RCS, a single user locks a file while they are working on it, preventing other users from checking in any changes, however minor.

We had great success meeting these goals with the Concurrent Versioning System[27] (CVS), an open source revision control system currently maintained by Cyclic Software. CVS is a client/server system for recording revisions to software projects. It can be used for maintaining any set of text files, such as web pages. CVS has clients for Windows, Unix, and even the Emacs text editor.

---

[25] `http://www.netspace.org/lsv-archive/bugtraq.html`

[26] `http://www.cyclic.com/cyclic-pages/rcs.html`

[27] `http://www.cyclic.com/`

The primary difference between CVS and RCS from a usage point of view is that CVS allows developers to work concurrently on the same files. Whenever a developer checks out copies of files from the repository, these copies become his "working branch" from the original files in the repository. Then, the developer can check in changes from his branch to the repository. If one developer checks in a file after another developer has already begun to modify it, CVS will update the second developer's copy through a special merge operation. This feature of CVS frightens many developers who fear that the merge could lead to data loss or confusion, but in practice it works very well.

We found great success using CVS in our collaborative projects, and I personally found CVS to be an excellent way for me to monitor progress in my solo projects.

A.6. Issue Tracking

At the same time that we began using CVS to coordinate our source code efforts, we found that we needed some way to track the status of our "to-do" items. We found that email reminders tended to get lost in the clutter, and it was difficult to merge our personal task lists. I looked at several bug tracking systems for our use, but many of them were difficult to set up and in general hard to use. Finally, we experimented with the "Bugzilla" bug tracking system[28] from Netscape and found that it met our needs very well.

---

[28] `http://www.mozilla.org/bugs/source.html`

Bugzilla is a set of Perl scripts that use a MySQL[29] relational database to track bugs from inception to verification and closure. We used it more for general issue tracking than actual bugs, but we found that the easy to use web pages, simple administration, and email alerts satisfied our project tracking needs.

A.7. Conclusions

Finding the right software infrastructure to support our research took a lot of time. The above software decisions were reached piecemeal over a three year period. Most of this time was spent experimenting with different products, and determining what kinds of software would best fit our work patterns. The choices made significantly increased our overall performance and the quality of our work.

---

[29] `http://www.mysql.com/`

Appendix B: Data Model Document Type Definition

```
<!-- ..................................................................... -->
<!-- BWH SPL DataDefinitions specification DTD ..... -->
<!-- ..................................................................... -->

<!-- $Id: datadefs.dtd,v 1.4 1999/05/14 22:07:29 pat Exp $ -->

<!--
#
# TYPICAL INVOCATION:
# <!DOCTYPE datadefs PUBLIC
#       "-//ICMIT//SPL DataServer Configuration::19990417//EN"
#       "http://icmit.mit.edu/dtd/datadefs.dtd">
#
-->

<!-- ORDM-specific mapping elements -->

<!ELEMENT dbTable EMPTY>
<!ATTLIST dbTable
  table CDATA #REQUIRED>

<!ELEMENT dbColumn EMPTY>
<!ATTLIST dbColumn
  column CDATA #REQUIRED>

<!ELEMENT dbType EMPTY>
<!ATTLIST dbType
  distinct CDATA "no"
  type CDATA #REQUIRED>

<!ELEMENT dbLinkTable (dbTable, dbLink, dbLink)>

<!ELEMENT dbLink (dbColumn)>
<!ATTLIST dbLink
  class ID #REQUIRED
  attrib ID #REQUIRED>

<!-- domain-nonspecific data model elements -->

<!ELEMENT datadefs (classdef*, class*)>
```

```
<!ELEMENT classdef (dbType?, attrib*)>
<!ATTLIST classdef
  id ID #REQUIRED
  desc CDATA #IMPLIED
  childOf IDREF #IMPLIED>

<!ELEMENT attrib (dbColumn, (array | element | object))>
<!ATTLIST attrib
  id ID #REQUIRED
  desc CDATA #IMPLIED>

<!ELEMENT array (dbType, (array | element | object))>
<!ATTLIST array
  type (set | multiset | list) #REQUIRED
  max CDATA '0'
  min CDATA '0'>

<!ELEMENT element (dbType)>
<!ATTLIST element
  type (string | char | short | long | float | date | datetime | blob)
#REQUIRED>

<!ELEMENT object EMPTY>
<!ATTLIST object
   classdef IDREF #REQUIRED>

<!ELEMENT class (dbTable, relationship*)>
<!ATTLIST class
  id ID #REQUIRED
  classdef IDREF #REQUIRED
  primaryKey IDREF #REQUIRED
  desc CDATA #IMPLIED
  childOf IDREF #IMPLIED>

<!ELEMENT relationship (dbLinkTable)>
<!ATTLIST relationship
  class ID #REQUIRED
  desc CDATA #IMPLIED>
```

Bibliography

1. Ninth Annual HIMSS Leadership Survey. Chicago, Illinois: Healthcare Information and Management Systems Society; 1998:30.

2. McDonald CJ. The barriers to electronic medical record systems and how to overcome them. *Journal of the American Medical Informatics Association*. 1997;4:213-221.

3. Korpman RA, Dickinson GL. Critical assessment of healthcare informatics standards. *Int J Med Inf*. 1998;48:125-32.

4. Cimino JJ. Review paper: Coding systems in health care. *Methods of Information in Medicine*. 1996;35:273-84.

5. Digital Imaging and Communications in Medicine (DICOM). Rosslyn, VA: The National Electrical Manufacturers Association; 1998.

6. Korman LY, Delvaux M, Bidgood D. Structured reporting in gastrointestinal endoscopy: integration with DICOM and minimal standard terminology. *Int J Med Inf*. 1998;48:201-6.

7. Delvaux M, Korman LY, Armengol-Miro JR, et al. The minimal standard terminology for digestive endoscopy: introduction to structured reporting. *International Journal of Medical Informatics*. 1998;48:217-25.

8. Goodman N, Rozen S, Stein L. Building a Laboratory Information System around a C++-Based Object Oriented DBMS. *20th VLDB Conference*: Santiago, Chile; 1994.

9. Sideman S, Beyar R. Integrative and interactive studies of the cardiac system: Deciphering the cardionome. In: Sideman S, Beyar R, eds. *Analytical and Quantitative Cardiology*. New York: Plenum Press; 1997:341-352.

10. Benson T. Why industry is not embracing standards. *Int J Med Inf*. 1998;48:133-6.

11. Dudeck J. Aspects of implementing and harmonizing healthcare communication standards. *Int J Med Inf.* 1998;48:163-71.

12. Bray T, Paoli J, Sperberg-McQueen CM. Extensible Markup Language (XML) 1.0 Specification. . http://www.w3.org/TR/1998/REC-xml-19980210: World Wide Web Consortium; 1998.

13. Harding L. Merging OO Design and SGML/XML Architectures. . http://www.infoauto.com/articles/arch/rim/oo-sgml.htm: Information Assembly Automation, Inc.; 1998.

14. Kahn CJ. A generalized language for platform-independent structured reporting. *Methods of Information in Medicine.* 1997;36:163-71.

15. Lassila O, Swick RR. Resource Data Framework (RDF) - Model & Syntax Specification. http://www.w3.org/TR/REC-rdf-syntax/: World Wide Web Consortium; 1999.

16. Bloise MS, Shortliffe EH. The computer meets medicine: Emergence of a discipline. In: Shortliffe EH, Perreault LE, eds. *Medical Informatics: Computer Applications in Health Care.* Reading, MA: Addison-Wesley; 1990.

17. Codd EF. A relational model of data for large shared data banks. *Communications of the ACM.* 1970;13:377-87.

18. Turner P, Keller AM. Reflections on object-relational applications. *OOPSLA workshop on object and relational databases*: Austin, TX; 1995.

19. Matthews DE, Sherman BK. ACEDB Genome Database Software FAQ. http://probe.nalusda.gov:8000/acedocs/acedbfaq.html. 1993-1998.

20. Gray PMD, Kulkarni KG, Paton NW. Object-Oriented Databases: A Semantic Data Model Approach. In: Hoare CAR, ed. *Prentice Hall International Series in Computer Science.* New York: Prentice Hall; 1992:237.

21. Stonebraker M, Brown P, Moore D. Object-Relational DBMSs: Tracking The Next Great Wave. In: Gray J, ed. *Series in Data Management Systems.* 2 ed. San Francisco, CA: Morgan Kaufmann; 1999:297.

22. Kim W. Introduction to Object Oriented Databases. In: Schwetman H, ed. *Computer Systems.* Cambridge, MA: The MIT Press; 1990:234.

23. Kulkarni K, Carey M, DeMichiel L, et al. "Introducing Reference Types and Cleaning up SQL3's Object Model." SQL3 Change Proposal X3H2-95-456 R2, December 18, 1995.

24. Kulkarni K. Personal Communication. "Re: ORDBMS." 1999.

25. McConnell S. Rapid Development. Microsoft Press; 1998.

26. Booch G, Rumbaugh J, Jacobson I. The Unified Modeling Language User Guide. In: Booch G, ed. *Object Technology Series*. Reading, MA: Addison-Wesley; 1998.

27. Rogerson D. Inside COM. Redmond, WA: Microsoft Press; 1997.

28. Beeler GW. HL7 version 3--an object-oriented methodology for collaborative standards development. *Int J Med Inf.* 1998;48:151-61.

29. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. In: Kernighan BW, ed. *Professional Computing Series*. Reading, MA: Addison-Wesley; 1995.

30. Fowler M, Scott K. UML Distilled. In: Grady Booch, James Rumbaugh, ed. *Object Technology Series*. Reading, MA: Addison Wesley Longman; 1997:183.

31. Goodman N, Rozen S, Stein L. The Case for Componentry in Genome Information Systems. *Meeting on Interconnection of Molecular Biology Databases*: Stanford University; 1994.

32. Fielding R, Kaiser G. The Apache HTTP Server Project. *IEEE Internet Computing*. 1997;1:88-90.

33. Netcraft Inc. Netcraft Web Server Survey. http://www.netcraft.com/survey/; 1999.

34. Chabanas M. Database for Multiple Sclerosis. . *Applied Mathematics*. Grenoble, France: Institut Universitaire Professionnalise; 1997:40.

35. Gering D. MRMLAid (Documentation for the Medical Reality Modeling Language.) 1999.

36. Gering D, Nabavi A, Kikinis R, et al. Integrated Visualization System for Surgical Planning and Guidance using Image Fusion and Interventional Imaging. *MICCAI '99*; (in review).

37. Dao N. Design and Prototype of an Object-Relational Database for Medical Images. . *Mechanical Engineering*. Cambridge, MA: Massachusetts Institute of Technology; 1998.

38. Hoque R. CORBA 3 Developer's Guide. Foster City, CA: IDG Books; 1998:617.

39. Bidgood WD, Jr., Horii SC, Prior FW, Van Syckle DE. Understanding and using DICOM, the data interchange standard for biomedical imaging. *Journal of the American Medical Informatics Association*. 1997;4:199-212.

40. Digital Imaging and Communications in Medicine (DICOM). NEMA PS 3 (Suppl) 23: Structured Reporting. Rosslyn, VA: The National Electrical Manufacturers Association; 1997.

41. Pierik FH, van Ginneken AM, Timmers T, Stam H, Weber RFA. Restructuring routinely collected patient data: ORCA applied to andrology. *Methods of Information in Medicine*. 1997;36:184-90.

42. Goldberg HI, Tarczy-Hornoch P, Stephens K, Larson EB, LoGerfo JP. Internet access to patients' records [letter]. *Lancet*. 1998;351:1811.

43. Cimino JJ. Beyond the superhighway: exploiting the Internet with medical informatics. *Journal of the American Informatics Association*. 1997;4:279-84.

44. Halamka JD, Szolovits P, Rind D, Safran C. A WWW implementation of national recommendations for protecting electronic patient information. *Journal of the American Medical Informatics Association*. 1997;4:458-464.

45. Orfali R, Harkey D. Client/Server Programming with Java and CORBA. . 2nd ed: John Wiley & Sons; 1998.

46. Henning M. "Re: newbie: Why do I use attributes in IDL?" http://www.deja.com/[ST_rn=ps]/getdoc.xp?AN=456643468&fmt=text; 1999.

47. Fuchs M, Maloney M, Milowski A. Schema for Object-oriented XML. . http://www.w3.org/TR/NOTE-SOX/: World Wide Web Consortium; 1998.

48. Silberzahn N. Dealing with the Electronic Patient Record variability: Object Oriented XML. *GCA SGML/XML Europe '98 Conference*: Paris, France: http://www.digitalairways.com/NiS/ParisXML98/; 1998.

49. Hunter J, Crawford W. Java Servlet Programming. . *The Java Series*. Cambridge: O'Reilly; 1998.