NISTIR 6926

The NIST Design Repository Project:
Project Overview and Implementational Design

Simon Szykman and Ram D. Sriram
Manufacturing Systems Integration Division
National Institute of Standards and Technology

Contributors
Khanh Trieu, Jean-Francois Heintz,
Guilhem Assant and Sylvain Archiambault

National Institute of Standards and Technology
Technology Administration, U.S. Department of Commerce

NISTIR 6926

The NIST Design Repository Project:
Project Overview and Implementational Design

Simon Szykman and Ram D. Sriram
Manufacturing Systems Integration Division
National Institute of Standards and Technology

Contributors
Khanh Trieu, Jean-Francois Heintz,
Guilhem Assant and Sylvain Archiambault

April 2002

U.S. DEPARTMENT OF COMMERCE
Donald L. Evans, Secretary
TECHNOLOGY ADMINISTRATION
Michelle O'Neill, Acting Under Secretary of Commerce for Technology
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
Arden Bement, Director

TABLE OF CONTENTS

LIST OF FIGURES

AUTHORS' NOTE

Although this report includes a brief project overview to give the reader some background context, this document does not provide a functional description of the Design Repository System (i.e., what the different types of data objects (knowledge structures) are, how these objects and links between them are used to create product models, etc. This report is not meant to be an introduction to the project or the system implementation. It is intended to serve as a technical reference describing the design, rationale, and implementation of the software system developed as part of the NIST Design Repository Project. The document assumes that the reader has some degree of familiarity with the project goals, as well as with the purpose and functionality of the system. This document will explain how things are implemented, and why they are implemented in a particular way, but generally does not include the background information of why something was being done in the first place.

Readers who are unfamiliar with the project itself and who wish to acquaint themselves with the project before reading this report can refer to Section 1.3 for a list of various publications related to this project. In particular, reading reference [5] and browsing reference [10] should provide a reasonable starting point from which one can delve deeper into the details of the system implementation.

1 INTRODUCTION

1.1 Motivation

In the past, product development was often done within a single company by co-located design teams. In more recent years, there has been a shift in product development paradigms. Product development is more often done by geographically and temporally distributed design teams. There is a high level of outsourcing, not only of manufacturing but also of actual product development efforts. Product development across companies, and even within a single company, is often done within a heterogeneous software tool environment. The Internet and intranets are supplanting paper and telephones as a means of exchanging product development information. As a result of this new product development paradigm, there is a greater need for software tools to effectively support the formal representation, capture, and exchange of product development information.

The existing generation of computer-aided engineering (CAE) software tools has undeniably revolutionized product development in contrast to methods used before the advent of these technologies. Nevertheless, the current generation of product development software tools addresses the needs of traditional product development processes, and does not adequately support the needs of industry's new paradigm described above. People are exchanging information across distributed design teams and corporate boundaries earlier, and reusing information to a greater extent. But because existing software tools do not capture a broad spectrum of product development information, these exchanges occur informally (face-to-face across a table, by phone, by paper). It is a lack of formal representations for product development information that creates a significant barrier to its effective capture and exchange.

The CAD/CAM/CAE software industry is ultimately a customer-driven one. It is therefore expected that as needs mount, a new generation of tools will emerge to address these needs. As the complexity of products increases and product development becomes more distributed, newly emerging software tools will begin to cover a broader spectrum of product development activities than do the traditional mechanical CAD systems. Accordingly, the ability to effectively and formally capture additional types of information will become a critical issue.

This research involves the development of a vision of next-generation product development systems. This work does not attempt to promote specific tools, but rather seeks to provide an information modeling infrastructure and implementation framework after which new systems can be modeled. The high costs of poor interoperability among today's computer-aided design tools are likely to be significantly compounded in the future if the problem remains unaddressed. This technical thrust addresses a fundamental problem whose solution can impact literally billions of dollars of costs to industry. It is hoped that sufficient diffusion of these concepts into industry will provide a foundation for improved interoperability among software tools in the future.

The economic benefits notwithstanding, the effort of developing a generic knowledge infrastructure for the next generation of tools is one that neither industry nor the CAD/CAM/CAE vendor community is likely to undertake alone. The National Institute of Standards and Technology (NIST) has U.S. industry as its primary customer and works to address problems that have significance to industry but that companies are not likely to solve on their own for one reason or another. NIST's emphasis is on economic impact to industry and society on a broad level rather than a corporate bottom line. Furthermore, NIST is not biased toward any particular class of problems, company, or industry sector, focusing instead on generic solutions that have broad-based applicability in industry. As a result, NIST is uniquely situated to invest in an effort to anticipate and address infrastructural needs in next-generation product development systems.

The NIST Design Repository Project involves research toward providing a technical foundation for the creation of design repositories—repositories of heterogeneous knowledge and data that are designed to support representation, capture, sharing, and reuse of corporate and general design knowledge. The infrastructure being developed consists of formal representations for design artifact knowledge, web-based interfaces for creating and browsing design repositories, and facilities to allow searching of repositories using concepts that have engineering design relevance, such as product function.

Through the course of this project, a variety of research issues have arisen that will in the long term affect the way in which design repositories are implemented and used. These issues include:

- Development of an information-modeling framework to support modeling of engineering artifacts to provide a more comprehensive knowledge representation than traditional CAD systems.
- Implementation of interfaces for creating, editing, and browsing design repositories that are easy to use and effective in conveying information that is desired.
- The use of standard representations, when possible, and contribution to long-term standards development where standards currently do not exist.
- Development of taxonomies of standardized terminology to help provide consistency in, and across, design repositories, as well as to facilitate indexing, search, and retrieval of information from them.

1.2 Project Background

The NIST Design Repository Project was initiated in 1996, in order to develop a prototype to demonstrate the use of a knowledge representation language developed as part of CONGEN (CONcept GENeration) [1], a project which aimed to provide partially automated support for engineering design. The initial Design Repository implementation was done during the summer of 1996. This initial implementation was developed as a stand-alone application using a number of languages including C++, Flex, Andrew Bison, and Tcl/Tk,, and using ObjectStore (a commercial object oriented database from Object Design Inc.) as a database back end.

The second Design Repository Project prototype system was designed to be a web-based system to allow distributed access to stored knowledge from a variety of hardware/software platforms. This prototype was implemented mainly in C++, and enabled access to knowledge (again stored in an ObjectStore database back end) through a web server via common gateway interface (CGI) scripts. In addition to moving from a stand-alone application to a web-based client-server architecture, the main knowledge browser interface was redesigned with significantly enhanced functionality, a design repository editor was developed, and the underlying knowledge representation evolved significantly.

This Design Document focuses on the third Design Repository Project prototype. The general concept behind the Design Repository System remained largely unchanged between the second and third implementations. However, based on insights gained from the second implementation, fundamental implementational changes were made. The most significant of these changes were (1) a change in the type of client-server architecture used, (2) a change from C++ to Java as the main development language platform, (3) a change from an object-oriented database management system to a relational database management system for the back end, and (4) extensive redesigns to the interfaces based on usability testing. In addition, the underlying knowledge representation continued to evolve into a Core Product Model during this period, in part as a result of a collaborative effort that was broader in scope than just the NIST Design Repository Project.

1.3 Related Publications

The first NIST Design Repository Project implementation is described in [2]. After the initial prototype was developed in the Summer of 1996, a workshop was held to identify industry needs related to this area of research in order to guide future development. The presentations and discussion summaries from this workshop can be found in [3].

A general description of the second prototype implementation as is stood in 1999 was published in [4]. This description is somewhat dated. More comprehensive information about the second prototype can be found in [5] and [6], the former being a magazine article that provided a high-level overview of the project, its objectives and system development, and the latter being a more technical description of the web-based architecture and interface implementation.

A more detailed description of the representation of engineering function that was developed for the NIST Design Repository Project can be found in [7]. This paper includes a discussion of the role of terminology and taxonomies in design knowledge representation. A related paper ([8]) addresses practical implementational issues rather than fundamental representational issues, and discusses the use of XML (eXtensible Markup Language) as a mechanism for knowledge exchange in the context of engineering function. Although the general discussions in [7] remain valid, the content of the taxonomies for engineering function and associated flows discussed in that paper have since been superceded by more recent work done in collaboration with researchers at University of Missouri-Rolla and University of Texas, Austin. That collaborative effort is summarized in [9].

The development and content of the NIST Core Product Model, the representational infrastructure that underlies the current design repository implementation is summarized in [10] and [11].

Two case studies involving the product modeling and development of design repositories for two artifacts that were designed at NIST (the Artifact Transport System and the Charters of Freedom Encasements) are summarized in [12] and [13].

1.4 Technical Overview

This report does not include detailed source code for the Design Repository System implementation. Source code is commented and documented following appropriate conventions that will allow automatic generation of API documentation using Sun Microsystems' Javadoc tool. The source files for the implementation are stored in a CVS (Concurrent Versioning System) repository that is accessible via the web at:

`http://thrym.msid.cme.nist.gov:8080/Devpt/DRP/`

The Design Repository Project web interface can be accessed at:

`http://thrym.msid.cme.nist.gov:8080/jsp/`

On the server side, system requirements for installing and testing the Design Repository web interfaces are as follows:

- Sun Microsystems' Java Web Server 2.0
- Oracle 7 relational database management system (release 7.3.3.0.0, using the query language PL/SQL release 2.3.3.0.0)[1]

---

[1] For more details regarding Oracle 7 as a "requirement," please see the Database paragraph in Section 2.1.

- Java 1.2 with Java Servlet API (Application Programming Interface) and JDBC (Java Database Connectivity) API
- Web browser (Netscape Navigator/Communicator 4.0 or later, or Internet Explorer 4.0 or later)
- JavaScript 1.1 enabled

On the client side, system requirements for using the Design Repository web interfaces are as follows:

- Web Browser (Netscape Navigator/Communicator 4.0 or later, or Microsoft Internet Explorer 4.0 or later)
- JavaScript 1.1 enabled

Due to NIST firewall restrictions, the URLs for the interfaces and source code CVS repository are only accessible from within the NIST network.

The remaining portions of this document provides detailed information regarding the design, architecture, and implementation of the main components of the current Design Repository Project tool suite. Included are the project specifications, and descriptions of the architecture and the implementation illustrated with numerous diagrams and schemata. Section 2 describes the general architectural design of the Design Repository System. Sections 3, 4, and 5 describe the Design Repository Browser and Editor interfaces, the Design Repository Search Tool, and the User Management System, respectively. Each of those three sections cover the functionality, specifications, and implementation for these main components of the system.

2 DESIGN REPOSITORY SYSTEM ARCHITECTURE DESIGN

2.1 Overview of Functional Components

This section describes the different functional components of the Design Repository System architecture. The main functional components are as follows:

- Request Handler: The Request Handler handles the various requests from the user. This component is the starting point for processing any kind of request from the user.
- Database Exchange Manager: The Database Exchange Manager is the interface between the web server and the database server, and handles the exchange of data between the two servers. This component uses an object model to store data in memory when retrieving information from the database. The Database Exchange Manager is used by the Request Handler to retrieve data to satisfy user requests.
- Database Connection Manager: The process of spawning a new connection to the database is time consuming enough to cause perceptible delays in response to user requests. The Database Connection Manager is used to dynamically maintain a pool of open connections to the database, so that user requests can be handled without having to open new connections in response to these requests. When a request gets passed to the Database Exchange Manager from the Request Handler, the Database Exchange Manager requests an (already open) connection from the Database Connection Manager.
- Database Management System: The database used for the Design Repository System is Oracle 7, a relational database management system. Because the underlying representation used for modeling product knowledge is conceptually an object-oriented representation (see Appendix A and [11]),

one might argue that an object-oriented database would be better suited to this application. In general, the choice of a database management system may be driven by numerous factors in addition to the obvious issue of the nature of the information being modeled. In the case of this project, several other important drivers led to the choice of a relational database system for the back-end database.

One factor is the fact that standardized Java-based application programming interfaces (APIs) exist for creating generic (non-vendor-specific) interfaces to SQL (Structured Query Language) compliant relational databases, whereas no such APIs yet exist for object-oriented databases. Because databases from many different vendors are used by companies in industry, we considered it important for this project to not be tied to any one particular vendor's database management system, to show relevance to as large a cross section of industry as possible. The heterogeneity of database systems outside of NIST also could potentially serve as a barrier to collaboration with other groups within the scope of this project. The ability to construct generic interfaces that could communicate with databases from different vendors would make external collaborations easier. Lastly, if the Design Repository System, were developed using an object-oriented database, there might be a perception that this type of database is a requirement for implementing such a system in industry. Because relational databases are much more prevalent in industry than in object-oriented databases, this perception might inhibit diffusion of this technology into industry. The choice of a relational database system for the back end demonstrates that implementations do not require object-oriented database, and are possible with relational databases as well.

Although the current implementation uses Oracle 7 as the database back end, interactions with the database are accomplished using the JDBC (Java Database Connectivity), a standardized Java-based SQL API. Because database communication is not Oracle-specific, it should be possible to substitute any SQL-compliant relational database management system as the back end. It should be noted that not all relational database management systems support the triggers and sequences used for database consistency maintenance in the current Design Repository System implementation. Thus, while a substitution of the database back end should not impact the ability to store and retrieve information from the database, in some cases a change of back end may require that certain consistency constraints either be sacrificed, or implemented in software instead of maintained through the database itself.

- User Interfaces: The user makes use of various user interfaces to interact with the Design Repository System. The Design Repository System interfaces serve as the end user's access point to product information that is stored in design repositories, enabling the end user to author, edit, retrieve and view this information. Information supplied by the user is processed and stored in the Design Repository database. Information retrieved by the user is formatted using HTML (Hypertext Markup Language) and JavaScript, to provide a client-side presentation of information that is readily interpretable by a human user.

- User Management System: The User Management System allows administrators to manage access to product information by allowing the creation of groups, the addition and removal of users from groups, and the granting of permissions associated with information access. Based on the groups a user belongs to and the permissions a user has been granted, a user may be denied access to information, a user may be permitted to only view information (read-only privileges), or a user can also be allowed to edit information (read/write privileges). Information for each product is stored in a design repository (with all of the repositories residing in a single database). Each repository belongs to a group, so that each user can have different permissions (no access, read-only, read/write)

for each repository based on the permissions that have been granted to that user by the administrators of each group the user belongs to.

- Web Server: The web server is used to handle communications between the user at the client side and the main part of the Design Repository System on the server side using HTTP (Hypertext Transfer Protocol), the communication infrastructure on which the world wide web is built. The web server used in the current implementation is Sun Microsystems' Java Web Server 2.0.

The interaction between the various functional components is shown in Figure 1. Regardless of the activity users are involved with (browsing/editing information, searching a repository with the search tool, or performing user management activities), users interface with the Design Repository System using a web browser as a client. Thus, although these activities are functionally different and are implemented as different components, the interfaces that the user interacts with to accomplish these activities are not individually shown in Figure 1. The interface output logic box that appears in Figure 1 is not, strictly speaking, a separate component, but rather is used to illustrate the fact that information is formatted (using HTML and JavaScript) for web-based viewing before being sent to the user.

Section 2.2 discusses the design of the client-server architecture. Sections 2.3 – 2.5 provide more detailed descriptions of the Request Handler, the Database Connection Manager, and the Database Exchange Manager, respectively. Because the design and implementation of the Browse/Editor interfaces, the Search Tool, and the User Management System went well beyond basic software architectural decisions, they are addressed in greater detail in separate sections of this report.

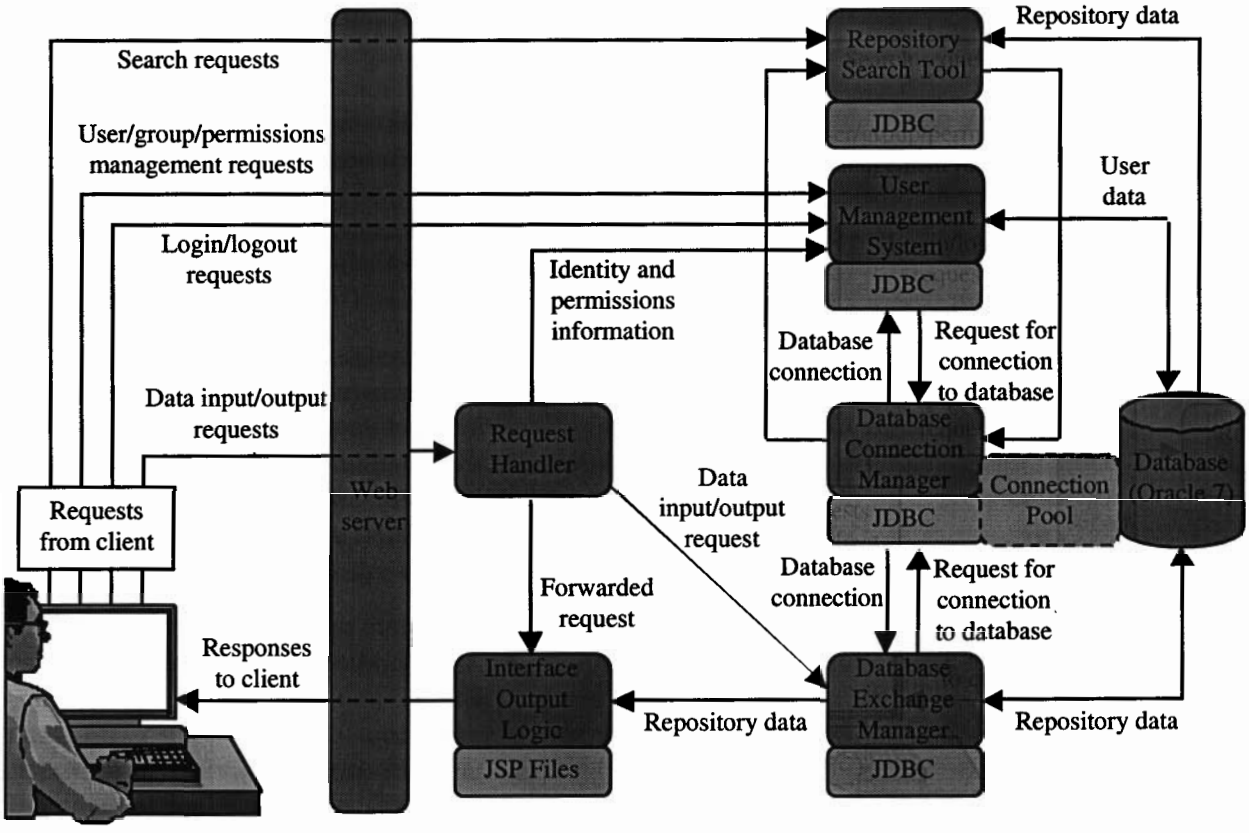

Figure 1. Interaction between functional components

Because the database and the web server used for the NIST Design Repository Project are commercial systems rather than tools developed as part of the project, this report does not include technical or implementational descriptions of these systems. Technical information about Oracle 7 can be found in [14] or at `http://docs.oracle.com/`. Technical information regarding Sun Microsystems' Java Web Server 2.0 can be found at `http://docs.sun.com/`.

2.2 Client-Server Architecture Design

The Design Repository System is based on a three-tier architecture. In general, three-tier architectures emerged to overcome the limitations of two-tier architectures, which include poor performance with large numbers of users, and limited flexibility. In a two-tier architecture, one tier is the client and the other tier is the server-side database management system. In a three-tier architecture, the third tier is an application that sits between the client and the database server. There are a variety of ways of implementing this middle tier, such as transaction processing monitors, message servers, or web/application servers. The middle-tier in a three-tier architecture can be used to perform such functions as queuing, application execution, database staging, and others. Three-tier client/server architectures have been shown to improve performance for groups with a large number of users (even as high as thousands) and provides greater flexibility than a two-tiered approach. Today, most sophisticated web-based applications that involve data exchange are based on three-tier client server architecture.

Figure 2 is a simplified view of Figure 1 that illustrates the three tiers in the Design Repository System architecture. In the Design Repository System, the client is the web browser that runs on the user's computer. The third tier is the Oracle 7 database management system. All of the other components described in Section 2.1 form the middle tier. From the user's perspective, the web browser is the client-side application and the rest of the system functions as a server-side application.[2] In practice, various components may reside on different machines. The web server may be on one machine, the rest of the middle tier application on another, the database server on yet another, and depending on the size of the databases, the data itself may be distributed among additional machines. Standard protocols such

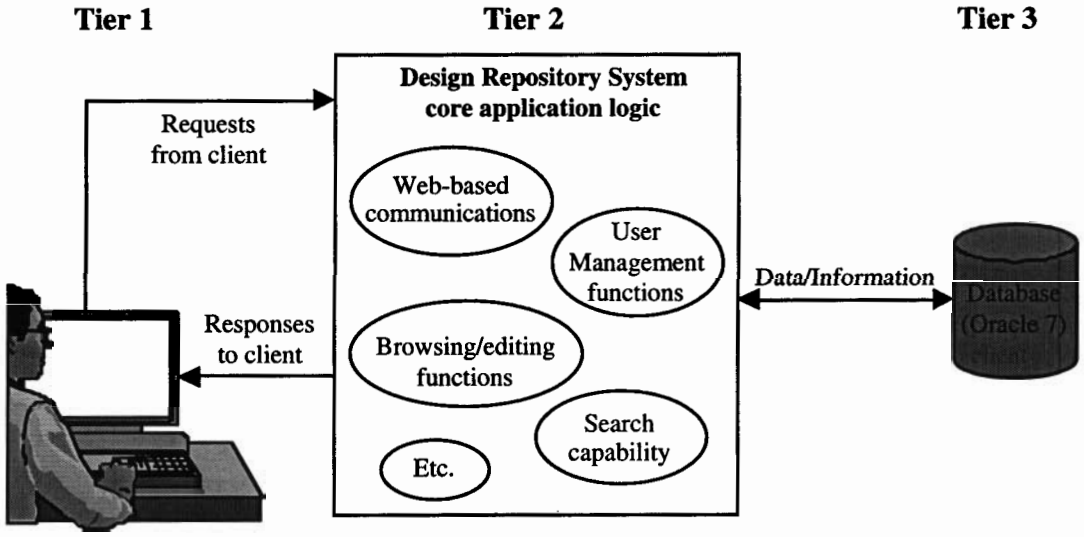

Figure 2. Three-tier architecture for the Design Repository System

[2] This statement is true only from the user's perspective. From the perspective of the DBMS, *it* (the database) is the server, while the middle tier application that is making database queries appears to be the client.

as TCP/IP (Transmission Control Protocol/Internet Protocol) and HTTP exist for handling communications among distributed components, so dealing with distributed components does not greatly add to the difficulty of implementation if the application development is done using techniques that conform to these standards.

The desire for a three-tier architecture, the need to conform to standard protocols, and the intended usage of the application motivate a number of architectural choices, including which type of three-tier architecture to use, which development language to use, which technologies to incorporate into the system implementation, etc.

The three classes of client/server models are a fat client, a thin client, and an ultra-thin client, which differ in how much of the core application logic[3] resides on the client side vs. the server side. Fat clients, which have a significant portion of the core application logic on the client side, were more common prior to today's prevalence of web-based applications. A client can easily be "fat" when the client application with the built-in application logic is installed on a local machine. For web-based applications, web browsers provide a standardized client and any application logic must be delivered to that client separately. Fat clients are not commonly used for web-based applications due to delivery issues, such as bandwidth limitations that can lead to impractical download times for fat clients, and the frequency at which updates are made to application logic in a fast-moving software development world.

The difference between a thin and an ultra-thin client is again the degree to which a portion of the core application logic is done on the client side. Examples of thin clients might be applications that require the downloading of plug-ins or Java Applets, allowing some of the "work," be it computing, data processing, etc. to be done on the client machine. Ultra-thin clients attempt to minimize the processing done on the client machine. The dividing line between thin clients and ultra-thin clients changes as technology evolves. For example, the use of JavaScript (a scripting language supported in most web browsers) in web pages allows browsers to easily and quickly perform certain types of functions that in the past would have required an Applet to be downloaded in order to accomplish on the client side.

The Design Repository System is implemented using an ultra-thin client/server model. One of the main motivations for selecting an ultra-thin client over a thin client was the recognition that many of the intended users of the type of technology developed in this project would be small and medium sized businesses that would not necessarily have the latest (i.e. powerful) computer hardware available to them. The ultra-thin client minimizes the burden on the client machine. The second motivation was a desire to support a broad base of potential users in a heterogeneous software environment. For example, one approach to developing a thin (rather than an ultra-thin) client would have been to create a Java Applet-based interface that would be downloaded to the client machine, allowing some of the data processing to be handled on the client machine. However, the Java language is an evolving one. Web browser support for Java 2, the latest version, may be constrained by one's choice of web browser, the version of the web browser being used, as well as the operating system on the client machine. An Applet-based interface would therefore not be accessible by as broad a user base as one which uses only HTML and JavaScript, which is more uniformly supported by web browsers than Java 2.

Among development languages, C++ and Java are the most widely used languages. Although earlier versions of the Design Repository System made extensive use of C++, a decision was made to shift to Java for the latest implementation. This decision was made based on the availability of existing Java-based technologies to support web-based application development, including Java Servlets, JavaServer Page (JSP), and JavaBeans.

---

[3] The application logic is also commonly referred to as *business logic* in some software development communities.

A Java Servlet is a Java program that runs on the server side, as contrasted with the more familiar Java Applets, which are Java programs that run on the client side. The advantage Servlets have over Applets is that they can enable complex programming, but being on the server side they avoid the download times that would be needed to download large Applets to a web browser. An alternate traditional method for supporting server-side processing is the use of CGI (Common Gateway Interface) scripts. CGI scripts spawn processes of short duration for each data access. These processes are terminated once a script execution has completed; once a process terminates no portion of the computation remains in active memory. In contrast, Servlets are applications that run continuously, allowing information to be cached in active memory significantly reducing the time expended by making repeated accesses of the database.

Servlets, being implemented in Java, provide a component-based, platform-independent method for development of web-based applications. Servlets are also more portable than some of the proprietary web server extensions that might be used for application development (e.g., Netscape Server API), as they are not limited to use with a single vendor's web server.

JavaServer Page (JSP) is an open specification developed by an industry-wide effort led by Sun Microsystems, which provides a simplified method for rapidly creating web pages that display dynamically-generated content. The JSP specification defines interactions between the web server and JSP pages, and allows the format and syntax of pages to be defined. JSP pages are compiled into Servlets in order to be used. It is possible to accomplish with Servlets alone that which is done with JSP. The advantage to using JSP is that the JSP separates the form of web pages—the templates that define how they appear—from their content. In applications such as the Design Repository System, where web page templates are static but content is dynamically generated (from a database in our case), using JSP provides a much-simplified development approach to generating web pages.

JavaBeans is a portable, platform-independent component model written in Java. The component model allows developers to author reusable platform-independent software components that can be reused for building larger applications. Being a complete component model, JavaBeans provides features such as properties, events, methods, and persistence. Although the concept of reusable software components is not new to JavaBeans, JavaBeans has been developed to enable automated analysis of components, to simplify customization of components, and to provide an industry-wide foundation for component-based software development using Java.

Both Java Servlets and JavaServer Pages allow server-side programming using the Java language. Based on the roles that these technologies play in software development, three alternatives were considered for the overall system architecture approach: (1) using only Java Servlets, (2) using JavaServer Pages in conjunction with JavaBeans, or (3) using Java Servlets, JavaServer Pages and JavaBeans.

With the first alternative, the middle tier would consist of the web server and Java Servlets, which would handle all of the processing in the middle tier (see Figure 3). This processing would include processing HTTP requests coming from the client via the web server, exchanging data with the database server, dynamic creation of web pages containing HTML and JavaScript, and sending them to the client via the web server using HTTP. The advantage of this approach is that it is easy to implement relative to the other alternatives. However, the disadvantage is that because the web page generation code is built into the code for the Java Servlets, subsequent maintenance (e.g. updating of web page templates) can become a burden.

For the second alternative, the middle tier would consist of the web server and a set of JavaServer Page-JavaBean pairs (see Figure 4). Each type of object stored in a product repository would be handled by a separate JavaServer Page. HTTP requests would go to the appropriate JavaServer Page via

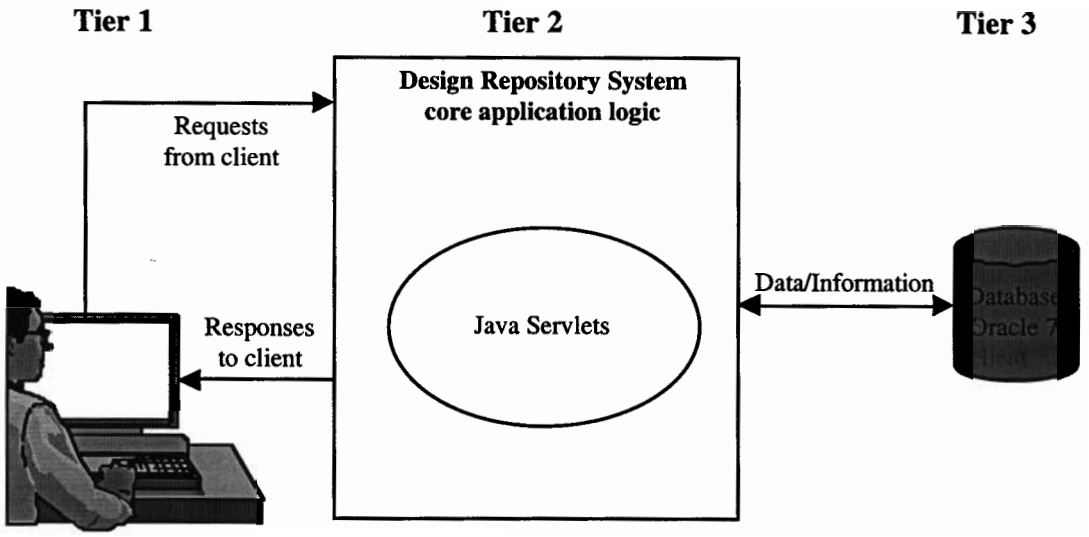

Figure 3. Three-tier architecture with Java Servlets

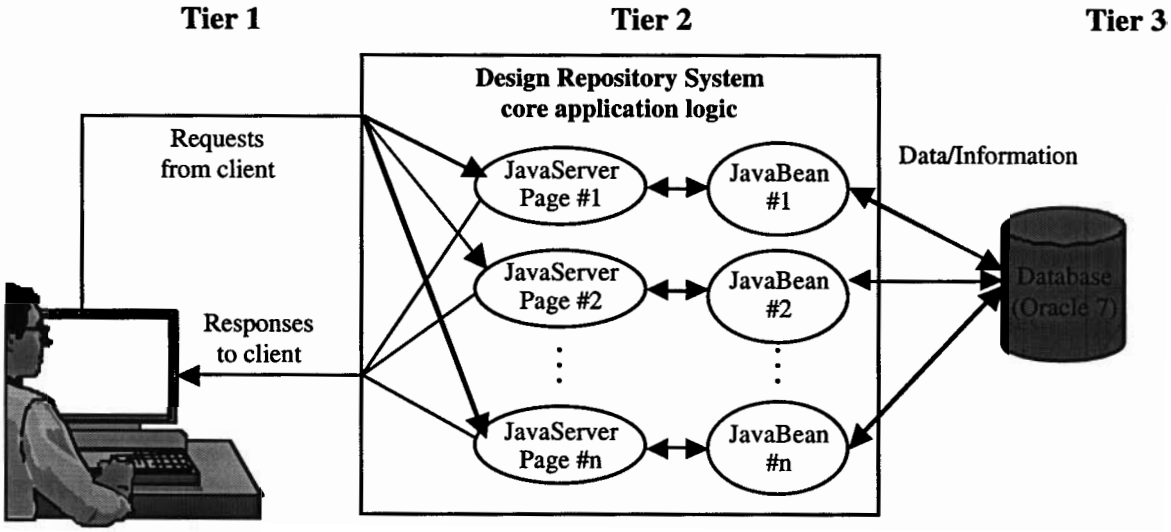

Figure 4. Three-tier architecture with JavaServer Pages and JavaBeans

the web server, depending on which type of information is being processed. For instance, a request for product function information would go to one JavaServer page while a request for a product form would go to another. Each JavaServer page would form the basis for a web page, and would use an associated JavaBean to retrieve data from the database, providing the dynamic content for the web page containing the requested information.

Since JavaServer Pages are compiled into Java Servlets in order to be used, this architecture is physically nearly identical to the previous one. The use of JavaServer pages simplifies the task of writing the HTML generation code, making implementation easier than for the first alternative. This approach is not without drawbacks, however. Because the JavaServer Pages include Java code, this approach is

not well suited to systems that have sophisticated core application logic (i.e., complex functionality). Too much Java in the JavaServer Pages can make it very difficult to debug software during development, and longer-term maintenance can still be problematic.

The third alternative is to use Java Servlets, JavaServer Pages, and JavaBeans (see Figure 5). With this architecture, Java Servlets are used to handle incoming requests, processing, and other core application logic.

For a given request, an appropriate JavaBean is instantiated to handle data exchange with the database. Information about the request is passed on to the appropriate JavaServer Page, which then receives the query results from the JavaBean and sends them out to the client in the appropriate format.

This approach provides the intended benefits of Java Servlets, as well as of the JavaServer Pages and JavaBeans combination. Specifically, this architecture effectively separates the core application logic (the core functionality of the application) from the graphical user interface definition (the code that relates to formats or templates for viewing and displaying information). Because of this decoupling, these two layers can be developed almost entirely independently from one another. More importantly, once deployed, they can be maintained and updated separately. The application logic can be extended without having to modify the code associated with displaying information, and similarly, the interface can be revised without digging into the core application code.

The only disadvantage of this architecture in comparison to the earlier ones is that the implementation itself is more complex as a result of incorporating all of the technologies described in the previous alternatives. Because of the increased complexity, a more thorough understanding of the various technologies is necessary in order to achieve a successful implementation. Nevertheless, because of the separation of layers, such an architecture can be easier (though not necessarily faster) to implement. More importantly, the burdens associated with extending the system after initial deployment, and longer-term system maintenance, are considerably reduced.

The choice of architecture was done with the intent of producing an application that would be fast, robust, and easy to extend and maintain. This decision was made with some insight into what would be required to achieve the system functionality that was desired. A system with very modest interface re-

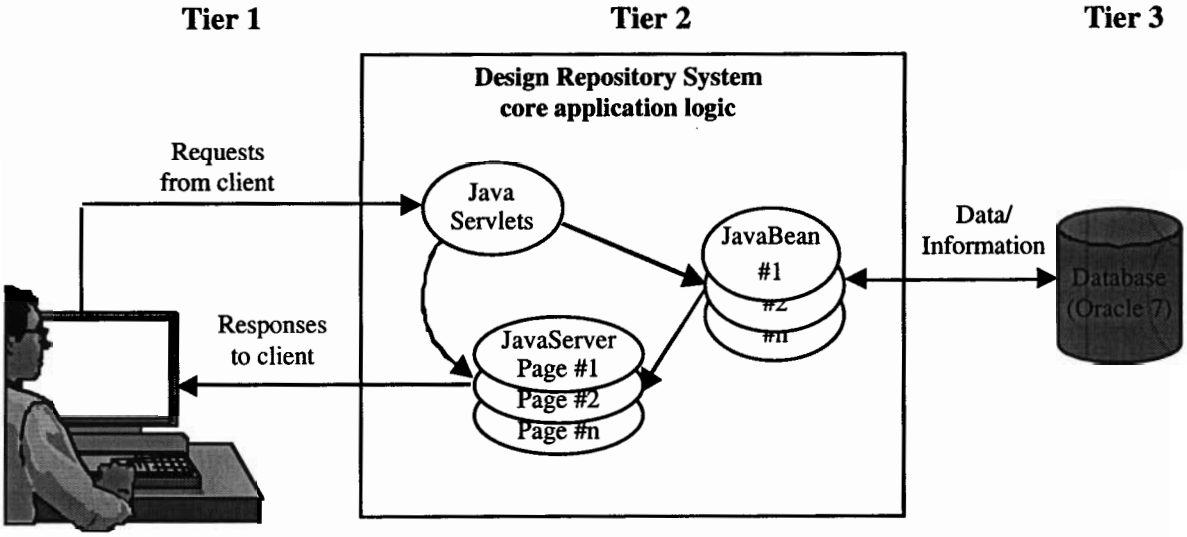

Figure 5. Three-tier architecture with Java Servlets, JavaServer Pages and JavaBeans

quirements might best be implemented using the first architecture. A system with comparatively simple core application logic might be best implemented with the second alternative. Such a system would be easier to implement, and could still be easy to maintain if significantly less core code were needed to provide the desired application functionality. Based on the requirements of the Design Repository System, the third alternative was selected for this project. Among the three architectures considered, all could provide a fast and robust application, but the last alternative would be significantly easier to extend and maintain. Comparing Figures 1 and 5, one can see that the generic architecture illustrated in Figure 5 is simply a higher-level view of the more detailed Design Repository System architecture shown in Figure 1.

2.3 Request Handler

The Request Handler is the entry point to the middle tier of the three-tier architecture. The Request Handler consists of a set of Java Servlets that receive HTTP requests from the client through the web server, and execute Java code to process the requests. The Request Handler first verifies that a request that arrives is a valid one. Assuming it is, it then dispatches commands to the Database Exchange Manager and passes information regarding the request to the code associated with the user interface definition so that the results of the query can be constructed, appropriately formatted, and sent back to the user.

The input to, actions by, and output from the Request Handler are as follows:

Input:

Information request, such as name and class of requested object, information about user mode (browsing or editing), user information (groups, access privileges), etc.

Action:

If the request is incorrect, return an error page to the client. Otherwise, processes the request. If necessary, give orders to the Database Exchange Manager to retrieve or update data as per the user request. Pass on information to the appropriate JavaServer Page for the response to the user.

Output:

Orders to the Database Exchange Manager.

Responses to requests and/or confirmations of transactions are returned to the user as web pages (formatted in HTML and JavaScript) sent through the web server.

Figure 6 shows several different types of requests that a user can make, along with corresponding actions that the Request Handler may take through the use of Java Servlets (specific Servlet and file names are included for the benefit of developers/implementers involved with the NIST Design Repository Project)

2.4 Database Exchange Manager

The Database Exchange Manager is the interface between the middle tier and the third tier of the system architecture (see Figure 2). The Database Exchange Manager sends SQL requests to the database in order to exchange data using JDBC (Java Database Connectivity), a standardized Java-based SQL API (application programming interface). This data exchange is done via an open connection to the database, which the Database Exchange Manager obtains from the Database Connection Manager. When data is retrieved from the database, these data are kept in memory using a Java object model that mirrors the Core Product Model used in the NIST Design Repository Project. Once stored in the object

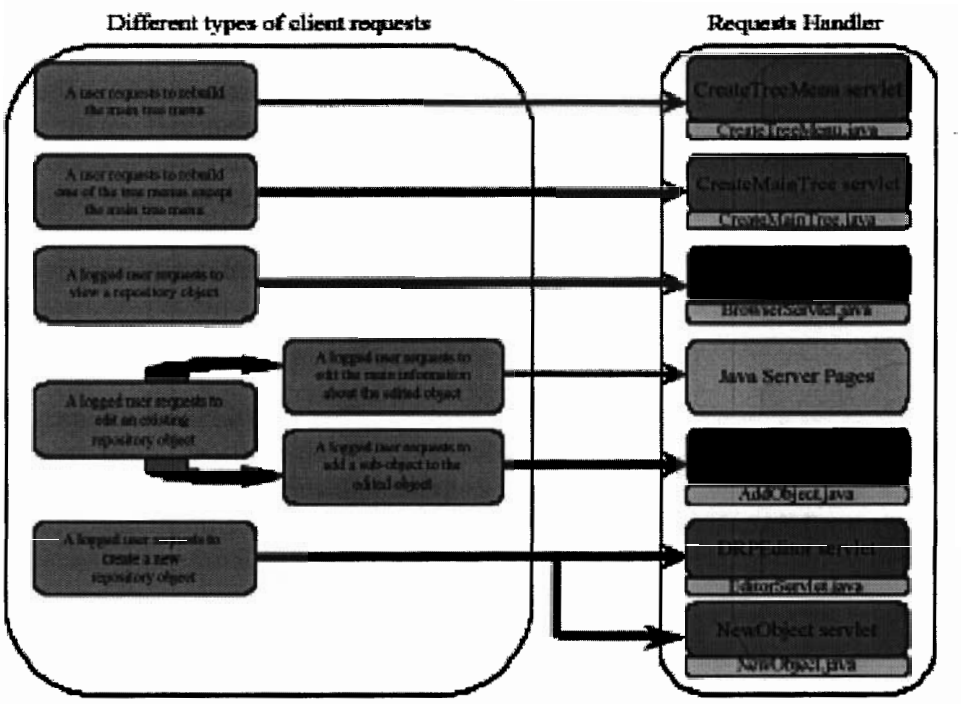

Figure 6. Client requests and Request Handler actions

model, these data can be accessed by the other components of the Design Repository System. The Database Exchange Manager is implemented using JavaBeans. As JavaBeans are Java classes, they are well suited to capturing the necessary object model. Moreover, the user interface output logic is implemented using JavaServer Pages (as will be discussed further in Section 3), which is a technology that is designed to be used in conjunction with JavaBeans.

The input to, actions by, and output from the Database Exchange Manager are as follows:

Input:

Step 1. Information about the data to retrieve or update

Step 2. An open connection to the database obtained from the Database Connection Manager

Action:

Step 1. Request open connection to the database from the Database Connection Manager

Step 2. Retrieve or update data as per user request and then free the database connection

Output:

Step 1. None

Step 2. Data retrieved from the database.

For the benefit of developers/implementers involved with the NIST Design Repository Project, Figure 7 illustrates the hierarchy and JavaBean names for the object model incorporated in the drp.beans package, which mirrors the Core Product Model.

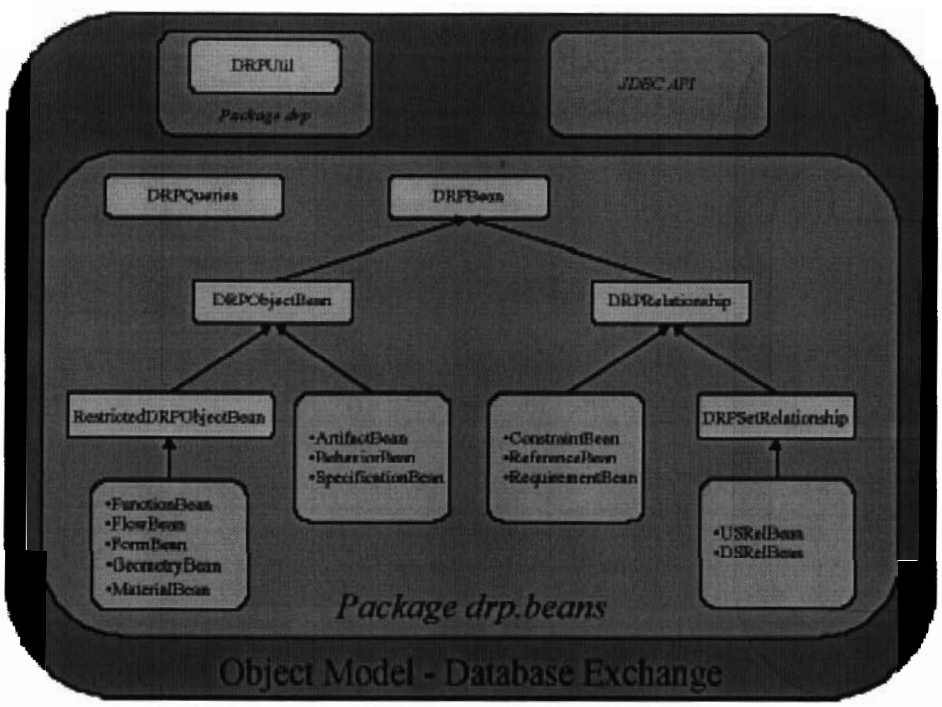

Figure 7. Object model for the Database Exchange Manager

2.5 Database Connection Manager

The Design Repository System provides an interface for creating, retrieving and editing product information stored in a database. These activities involve the exchange of information to or from a database. Many of the actions available to a user through the interface result in communication with the database. Establishing a connection to a database is a time consuming function because the database must allocate resources (such as memory), authenticate the user, set up the corresponding security context, etc. Calling the establishment of a connection "time consuming" is, of course, a relative characterization. Given the network architecture at NIST, with the web server and database servers being run on separate machines, establishing a connection to the database typically takes one second or longer. References on interface design commonly cite 200 milliseconds as being the threshold below which response appears to be instantaneous and above which humans perceive a delay. Given that, it becomes apparent that it does not take much of a delay to create unappealing response lags in a human-computer interface. Indeed, earlier implementations of the Design Repository System did suffer from excessive delays in response to user input, and the time for opening connections to the database server for each query was experimentally found to be one of the most significant contributors to the time lag. The Database Connection Manager was developed to address this issue.

The Database Connection Manager is designed to implement a pooling technique that allows multiple database connections to be established and maintained in a connection pool, so that they can be shared transparently among multiple incoming requests. The Database Connection Manager creates the pool of database connections when the main application of the Design Repository System is initially started up. Consuming the time required to establish connections at startup significantly reduces the overhead on database queries made later in response to user requests. Once a user request is satisfied, the connection is released back into the pool without closing it.

The connection pool starts up with a default number of connections, but is implemented to optimize resource usage by altering the pool size based on demand. The number of connections in the connection pool can be dynamically increased as the number of free connections in the pool drops below some limit, or reduced when the number of unused connections exceeds a specified number. When it is necessary to add connections, establishing new connections takes as much time as it would without a connection manager in place. However, this time usage does not delay the system response to user queries because new connections are added while some unused connections are still available in the connection pool. Thus the connection management is done invisibly to the user without causing unwanted response delays.

The input to, actions by, and output from the Database Connection Manager are as follows:

Input:

Alternative 1. A request for a database connection from the Request Handler, along with information about the user

Alternative 2. The number of available connections decreases below some specified amount

Alternative 3. The number of available connections increases above some specified amount

Action:

Alternative 1. Pass an available connection to the Request Handler if the user has appropriate authorization

Alternative 2. Create a batch of new connections and add them to the pool

Alternative 3. Close a batch of excess connections in the pool

Output:

Alternative 1. Available connection returned to the Request Handler

Alternative 2. None

Alternative 3. None

The following implementational details are given for the benefit of developers/implementers involved with the NIST Design Repository Project, The Database Connection Manager consists of a manager class that provides an interface to multiple connection pool objects. Each connection pool object manages a set of JDBC connection objects that can be shared by any number of Servlets. The database connection pool class, DBConnectionPool, provides methods to:

- get an open connection from the pool,
- return a connection to the pool,
- release all resources and close all connections at shutdown.

It also handles connection failures, such as time-outs, and can limit the number of connections in the pool to a predefined max value, so as not to overload the database management system.

The manager class, DBConnectionManager, is a wrapper around the DBConnectionPool class that manages multiple connection pools. It:

- loads and registers all JDBC drivers,

- creates DBConnectionPool objects based on properties defined in a properties file,
- maps connection pool names to DBConnectionPool instances,
- keeps track of connection pool clients to shut down all pools gracefully when the last client is done.

Those two classes are fully commented and documented in the Javadoc format. This documentation is available in the ConnectionManager directory of the CVS repository of the Design Repository Project. Also available is an example of a Java program using the connection manager. Refer to those for further details about the implementation.

With the overall design of the Design Repository System architecture having now been described, the remaining sections of this report describe the functionality, specifications, and implementation for the system's main user interfaces. The following section discusses the Design Repository Browser and Editor interfaces. The Design Repository Search Tool and the User Management System are discussed in Sections 4 and 5, respectively.

3 DESIGN REPOSITORY BROSWER AND EDITOR INTERFACES

3.1 Functionality Summary

The various user interfaces serve as links between the first tier (the web browser client) and the middle tier of the system architecture. The interfaces accept requests from the user, and deliver the results of the request via dynamically generated web pages. Web pages include content in HTML, and in most cases, JavaScript as well. The use of JavaScript allows some processing of page content to be done on the client side. For instance, the dynamically expandable and collapsible hierarchical product decomposition (illustrated in Section 4) is accomplished using JavaScript.

The Design Repository Browser provides an interface that allows a user to navigate through the body of product knowledge contained in a design repository. The user can navigate through this information along a variety of paths. The user can move up and down the physical hierarchical product decomposition to reach a design repository object[4] of interest. The user can also move from one design repository object to another along links between the objects. As illustrative examples, when viewing a given artifact (an object that represents a system, an assembly, a component, etc.), the user can move "down" in the hierarchy to one of its constituent parts, or "up" in the hierarchy to a higher-level artifact which has the current artifact as one of its parts. The user can also move along links that do not correspond to the physical hierarchy. For instance, the user can move from an artifact object to an object representing its function, or to other objects representing energy flows of which the current artifact is a source or destination.

The Design Repository Editor allows the user to create product models by authoring new design repository information, as well as modifying existing information in a repository. When viewing a product model from the browser interface, a user that has the appropriate privileges can click on an edit button to enter the editing mode. In edit mode, the user can modify information on the current page. While in this mode, the user also has the ability to create new data entities (i.e., new artifacts, func-

[4] Although the Design Repository System is implemented using Oracle 7, a relational database, the core product model used for representing product knowledge is conceptually an object-oriented model. Thus, although information is physically stored in tables in a relational database, the term "design repository object" (or simply "object") will be used to refer to the data entities that are used to represent a product.

tions, flows, etc.), or to add, change, or remove links between entities. Clicking on a button to return to browsing mode commits the changes to the database.

3.2 Interface Specifications

3.2.1 Use Cases

Use case diagrams provide an external view of the system, graphically illustrating its interactions with predefined *actors*, i.e., external users. Use case diagrams identify a high-level set of activities that a system allows, the actor that interact with the system, and shows which actors can perform which activities. Use case diagrams do not, however, attempt to characterize transitions between activities. That level of information is conveyed by activity diagrams, which will be discussed in the next section.

The diagram shown in Figure 8 illustrates the use case schema for the Design Repository Browser and Editor interfaces. Although the two interfaces require differing portions of code at the implementation level, they are designed to appear very similar to one another. The intent is to avoid giving the user the impression that these are two distinct interfaces, but instead to have them appear as a single interface that allows the user to move back and forth between a BROWSER mode and an EDITOR mode.

The use case are two kinds of actors, the Real User and the Logged User.

Real User: This is the human user of the design repository interfaces.

Logged User: A Real User who has successfully logged into the design repository system. Logging in provides the application with access to information about the user, such as identify information (e.g., real name, email address, etc.) and permissions information (e.g., which groups a user belongs to, which groups a user has administrative priviledges over, etc.).

A Real User is limited to viewing the physical decomposition hierarchy for products in the design repository (subject to constraints resulting from user/group permissions that may limit access), or searching for objects without being able to view the detailed object models themselves. The Real User can log into the system and become a Logged User. This allows viewing and editing of design reposi-

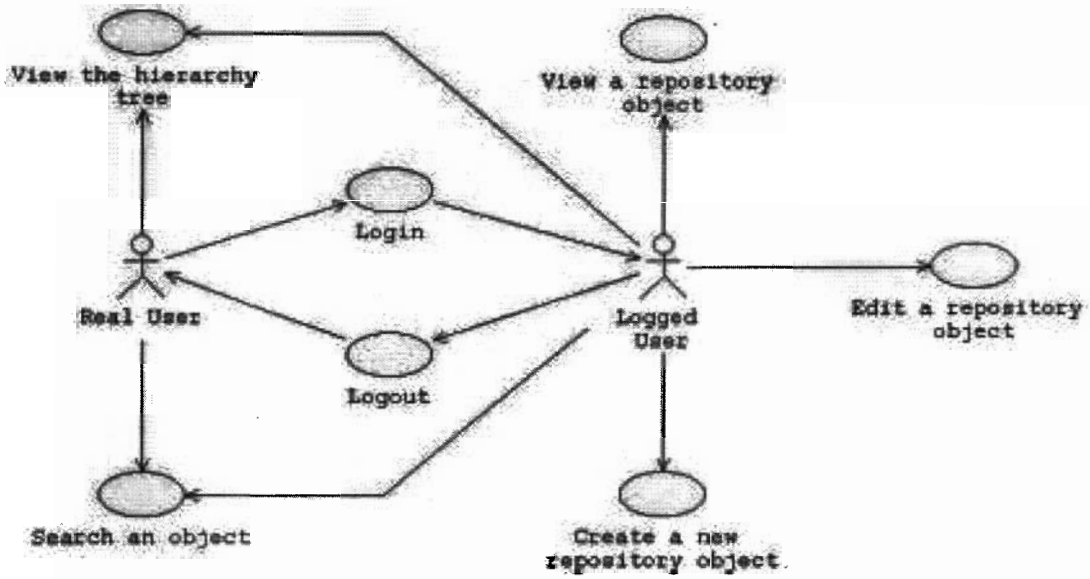

Figure 8. Use case schema for the Design Repository Browser and Editor

tory objects, as well as authoring of new objects. These activities are again subject to limitations in user/group permissions. The Logged User can log out and become a Real User.

The descriptions of the various activities represented by nodes in the use case schema are as follows:

`View the hierarchy tree`

Actor: Real User or Logged User

Description: The user navigates into the design repository through a hierarchy tree. From this tree menu, the Logged User can select a design repository object to view (subject to restrictions due to access privileges).

`Search for objects`

Actor: Real User or Logged User

Description: The user can search for objects in the design repository using different search criteria (details regarding the search tool are documented in a separate section of this document).

`Login`

Actor: Real User

Description: The Real User logs in the system using the login form. This use case won't be documented here because the documentation for the user management already exists.

`Logout`

Actor: Logged User

Description: The Logged User logs out of the system

This use case won't be documented here because the documentation for the user management already exists.

`View a repository object`

Actor: Logged User

Description: Displays information about a design repository object. The user needs to have the permission to view the object (read permission). The object view also contains links to other objects that are related to the object being viewed (e.g., there is a link between an artifact and its function).

`Edit a repository object`

Actor: Logged User

Description: Displays information about a design repository object and allows editing these data. The user should have read/write permissions for the object being edited.

`Create a new repository object`

Actor: Logged User

Description: The user creates a new design repository object. The user must belong at least to one Group in order to be allowed to create an object.

3.2.2 Activity Diagrams

Activity diagrams are diagrams that look like flow charts. These diagrams encompass a set of activiities which may be the same activities as shown in a use case diagram, or as is the case here, may be a more detailed set of activities that take place as part of the higher-level activities shown in a use case diagram. In contrast to use case diagrams, which show interactions between actors and activities, activity diagrams graphically show the transitions between activities—which activity nodes can be reached from a given state in the system. Activity definitions describe in greater detail the preconditions and postconditions for each node in the activity diagram, thereby defining under which circumstances a given activity can take place, and what the effect of an action is on the state of the system.

Activity diagrams can be created at several levels of detail. The main activity diagram for the Design Repository Browser and Editor interfaces is shown in Figure 9. The solid black dot indicates the starting point of the diagram; there is no ending point because the user can continue to transtition among activities indifinetly. Several of activities represented by one node in Figure 9 will be described by separate, more detailed activity diagrams in the next section.

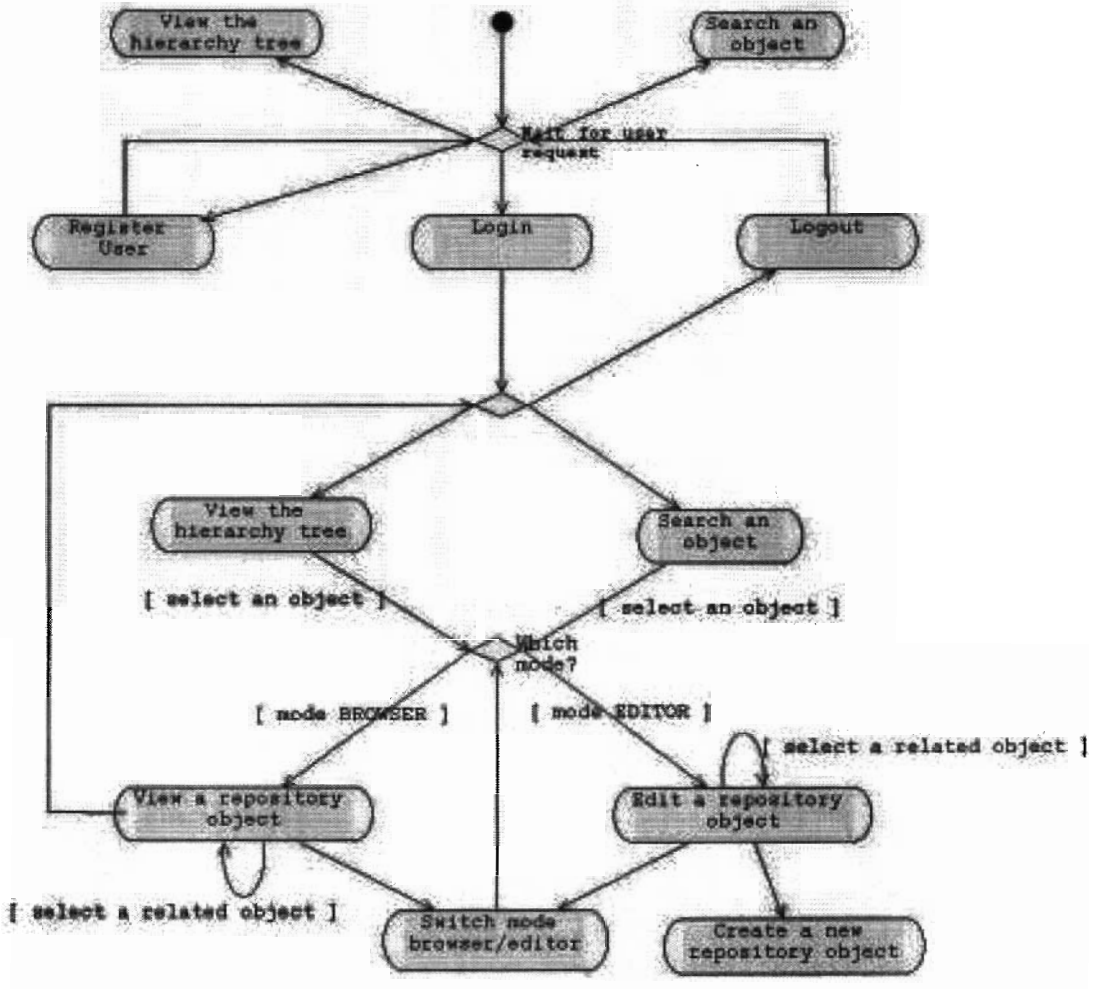

Figure 9. Main activity diagram for the Design Repository Browser and Editor

The activity descriptions for these activities are as follows:

`Register User`

Precondition(s): The user is not registered.

Action: Register a new user. (The User Management System is described in greater detail, including additional use cases and activity diagrams, in Section 5.)

Post-condition(s): The user is registered and receives a login and a password.

`Login`

Precondition(s): The user is registered.

Action: Checks the login and password of the user. If correct, the user is logged in.

Post-condition(s): User logged if login and password are correct.

`Logout`

Precondition(s): The user is logged in.

Action: Log out the user.

Post-condition(s): The user is logged out.

`View the hierarchy tree`

Precondition(s): None.

Action: The user can view the whole physical product decomposition hierarchy tree, expand and collapse the tree, and switch between artifact object tree and function object tree.

Post-condition(s): none

`Search an object`

Precondition(s): None.

Action: The user launches a search for design repository objects using various search criteria.

Post-condition(s): The user is looking for an object. The Design Repository Search Tool is described in greater detail, including additional use cases and activity diagrams, in Section 4. This post-condition *is the starting point for the activity diagram shown in* Figure 27 (Section 4.2.2).

`View a repository object`

Precondition(s): The object ID and class are known, the object exists in the repository, the user has the appropriate permission to view it, and the interface mode is BROWSER.

Action: Displays information about the object.

Post-condition(s): None.

`Edit a repository object`

Precondition(s): The object ID and class are known, the object exists in the repository, the user has the appropriate permissions to view and edit it, and the interface mode is EDITOR.

Action: Displays information about the object and allows the user to edit the object.

Post-condition(s): If the user has edited any data, the repository updates these changes.

`Create a new repository object`

Precondition(s): The user has the appropriate permissions to create a new object, an object with the name being given to the new object does not already exist in the repository, and the interface mode is EDITOR.

Action: The user creates a new object by entering data needed through a sequence of HTML forms.

Post-condition(s): If the user has successfully entered the data needed, the object is created in the repository.

`Switch mode browser/editor`

Precondition(s): The current interface mode is either BROWSER or EDITOR.

Action: Switch the current mode from the current mode (BROWSER and EDITOR) to the other.

Post-condition(s): If the interface mode was BROWSER, it is now EDITOR. If the interface mode was EDITOR, it is now BROWSER.

3.2.3 Detailed Activity Diagrams

Because several of the nodes appearing in Figure 9 represent complex activities, this section provides another level of detail regarding these activities. More detailed descriptions are provided for the main activities available to a Logged User: *View a repository object*, *Edit a repository object*, and *Create a new repository object*. These more detailed descriptions consist of activity diagrams, activity descriptions, and sequence diagrams. Where activity diagrams illustrate the transitions between activities, sequence diagrams provide an abstract view of the sequence of interactions (communications, message passing, etc.) between distinct components of the software. An activity diagram provides insight into how a system is intended works irrespective of implementation, whereas a sequence diagram gives an indication of the actual structure of an implementation.

Figure 10 shows the detailed activity diagram for the activity *View a repository object*. The solid black dot indicates the starting point of the diagram; the black dot with a circle around it indicates the end of the diagram. The middle node is present because the Design Repository System must verify that the Logged User has the appropriate privileges to view an object before displaying information about the object. The activity descriptions for the activities shown in Figure 10 are as follows:

`Retrieve information about user`

Precondition(s): The user is logged in and the interface mode is BROWSER.

Action: Retrieve user information needed to verify read permission.

Post-condition(s): All information about the user needed to verify read permission is in memory.

`Retrieve information about the object to view`

Precondition(s): The object ID and class are known, and the interface mode is BROWSER.

Action: Retrieve all information about the object to view.

Post-condition(s): All information about the object is in memory.

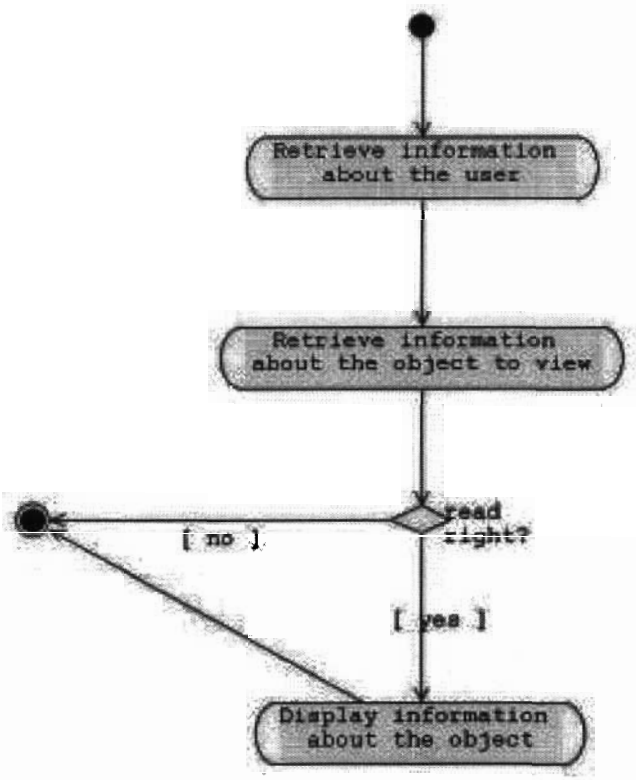

Figure 10. Detailed activity diagram for the activity *View a repository object*

`Display information about the object`

Precondition(s): All information about the object is in memory and the user has the permission to view the object. The interface mode is BROWSER

Action: Displays in a web page the information about the object.

Post-condition(s): None.

Figure 11 shows the sequence diagram for the activity *View an Artifact*. This is the sequence diagram for the *View a repository object* activity when the type of design repository object being viewed is an artifact.

Figure 12 shows the detailed activity diagram for the activity *Edit a repository object*. As with the *View a repository object* activity, the Design Repository System first needs to verify that the Logged User has the appropriate privileges to perform the activity (read/write permission). Once the user is approved, information is displayed in a forms-based interface that allows the user to edit the information. Buttons on the page may also redirect the user to a new web page to execute more complex editing activities such as adding new objects rather than editing information for the current object.

The activity descriptions for the activities shown in Figure 12 are as follows:

`Retrieve information about user`

Precondition(s): The user is logged in and the interface mode is EDITOR.

Action: Retrieve user information needed to verify read permission.

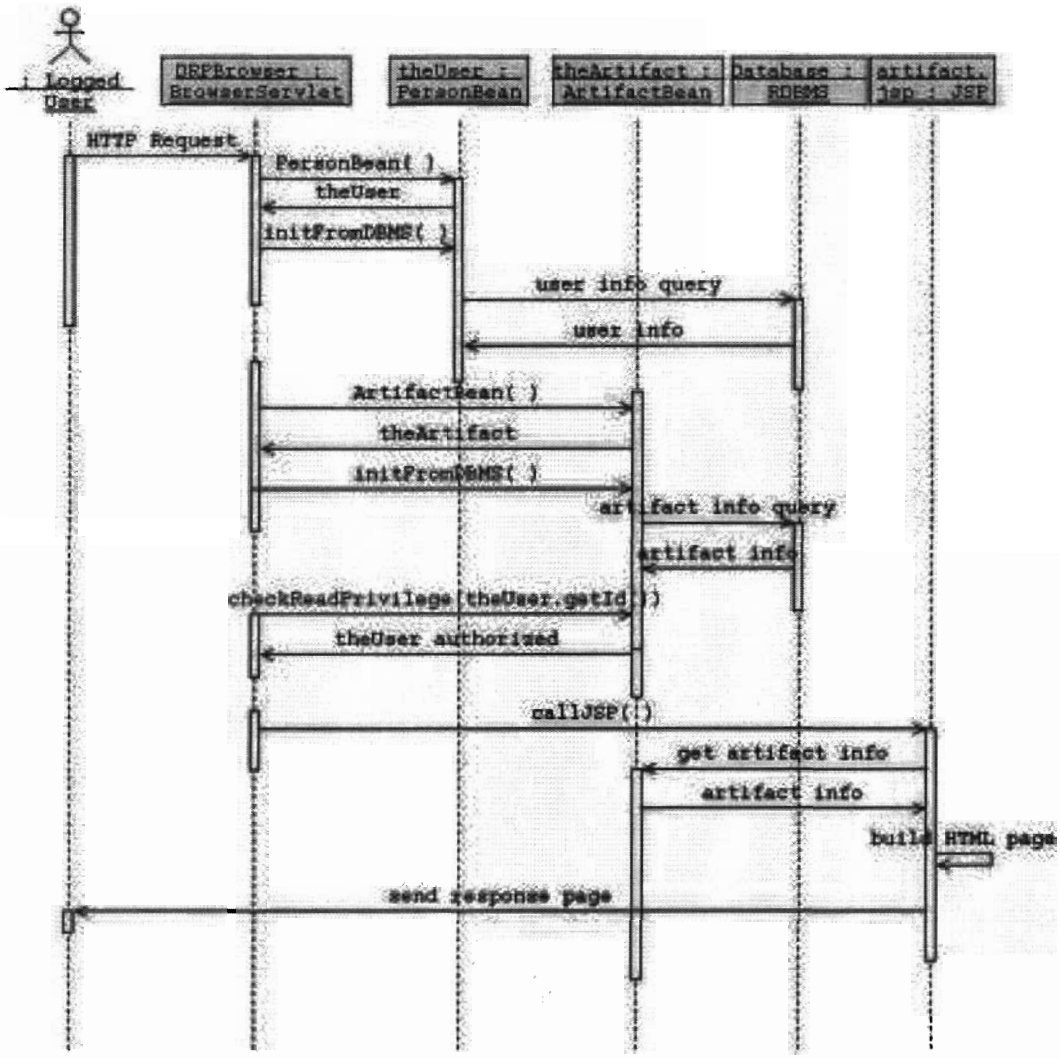

Figure 11. Sequence diagram for the activity *View an artifact*

Post-condition(s): All information about the user needed to verify read/write permission is in memory.

`Retrieve information about the object to edit`

Precondition(s): The object ID and class are known, and the interface mode is EDITOR.

Action: Retrieve all information about the object to view.

Post-condition(s): All information about the object is in memory.

`Edit the main information about the current object`

Precondition(s): The interface mode is EDITOR and the user has the permission to edit the object.

Action: The user modifies the information about the object through an HTML form.

Post-condition(s): If the user has made any changes, the relevant data is updated in the repository.

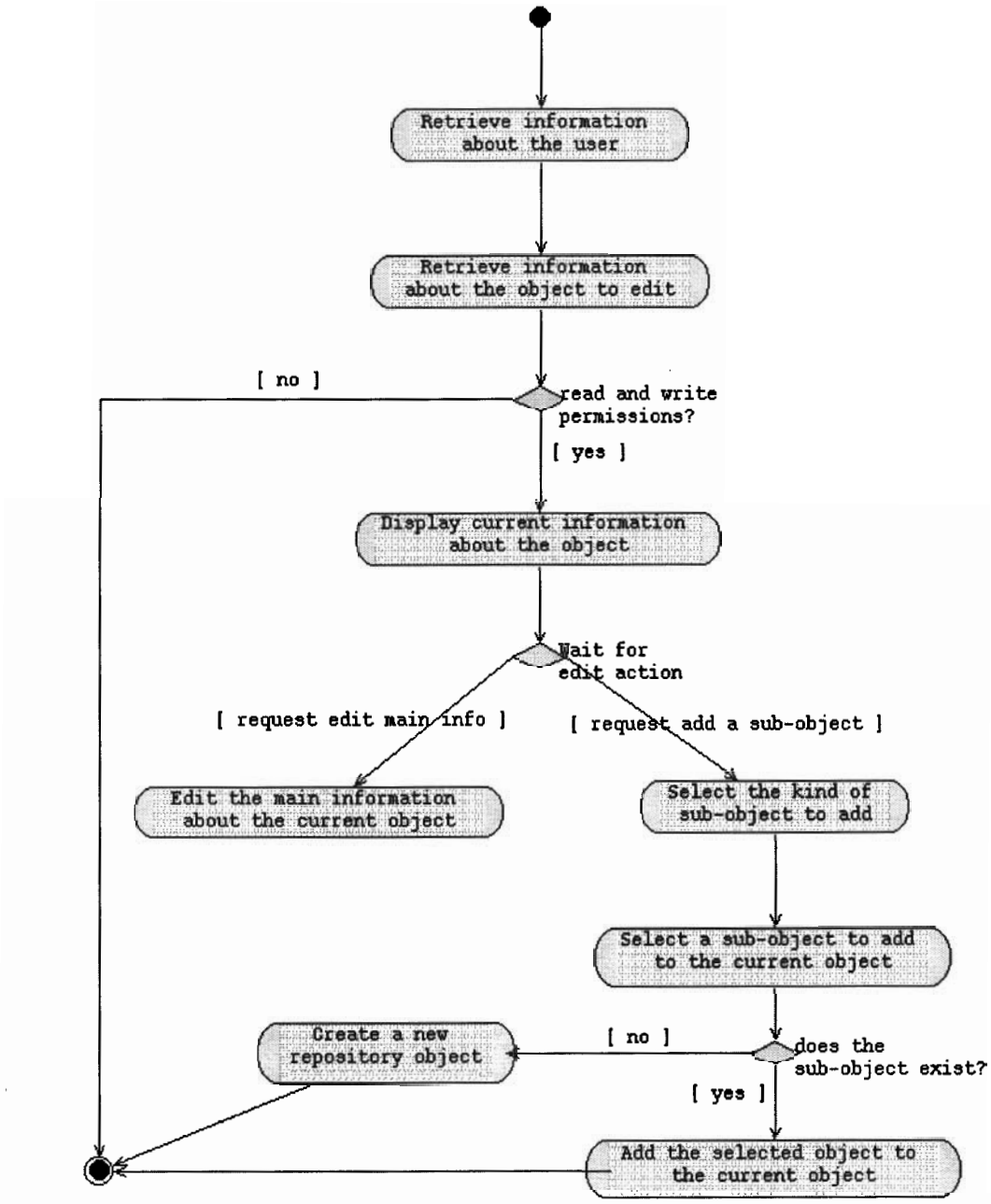

Figure 12. Detailed activity diagram for the activity *Edit a repository object*

Select the kind of sub-object[5] to add

Precondition(s): The interface mode is EDITOR.

Action: The user selects the kind of sub-object to add to the object currently being viewed.

[5] The term *sub-object* is used here to refer to an object that is to be linked to the current object. An assembly (artifact) may be linked to several artifacts representing components of the assembly, but in this context a function object that is created to be linked to the current artifact object would also be considered a sub-object.

Post-condition(s): The kind of sub-object to be added is known. More specifically, the class of the sub-object and the type of relationship between the current object and the sub-object are known.

`Select a sub-object to add to the current object`

Precondition(s): The interface mode is EDITOR and the kind of sub-object to add is known.

Action: The user selects the sub-object to add. This can be done in one of two ways: (1) the user can open a hierarchical tree menu and select the sub-object from a list, or (2) the user can enter the sub-object name into the HTML form.

Post-condition(s): The sub-object class, the type of relationship between the current object and the sub-object, and the ID of the sub-object are known.

`Add the selected object to the current object`

Precondition(s): The interface mode is EDITOR, the current object class and ID are known, the selected object class and ID are known, the type of relationship between the current object and the selected object is known, and the user has the appropriate permissions to edit the current object.

Action: The selected object is added as a sub-object of the current object.

Post-condition(s): The modifications are done and committed to the repository.

Figure 13 shows the sequence diagram for the activity *Edit an artifact*. This is the sequence diagram for the *Edit a repository object* activity when the type of design repository object being edited is an artifact.

Figure 14 shows the detailed activity diagram for the activity *Create a new repository object*. As with the two preceding activities, the design repository system must verify that the Logged User has the appropriate permissions to create a new design repository object. This time, the server only needs information about the user since the object to be created does not yet exist. Once the permissions are checked, the system switches into AUTHOR mode.[6] The user then enters information about the new object through a sequence of HTML forms, the contents of which will depend on the type of object being created. At the end of this sequence, a summary page is displayed allowing the user to check the data before confirming the creation of the new object.

The activity descriptions for the activities shown in Figure 14 are as follows:

`Select the new object class`

Precondition(s): The interface mode is EDITOR.

Action: The user selects the class of the object to be created.

Post-condition(s): The new object class is known.

`Retrieve information about user`

Precondition(s): The user is logged in and the interface mode is EDITOR.

---

[6] Because any user that has read/write privileges can both edit objects and add new objects, there is no reason for the user to be aware that an AUTHOR mode exists. The AUTHOR mode exists for implementational convenience, but the change in mode is invisible to the user. As far as the user is concerned, editing objects and authoring new objects are both done from the Design Repository Editor interface.

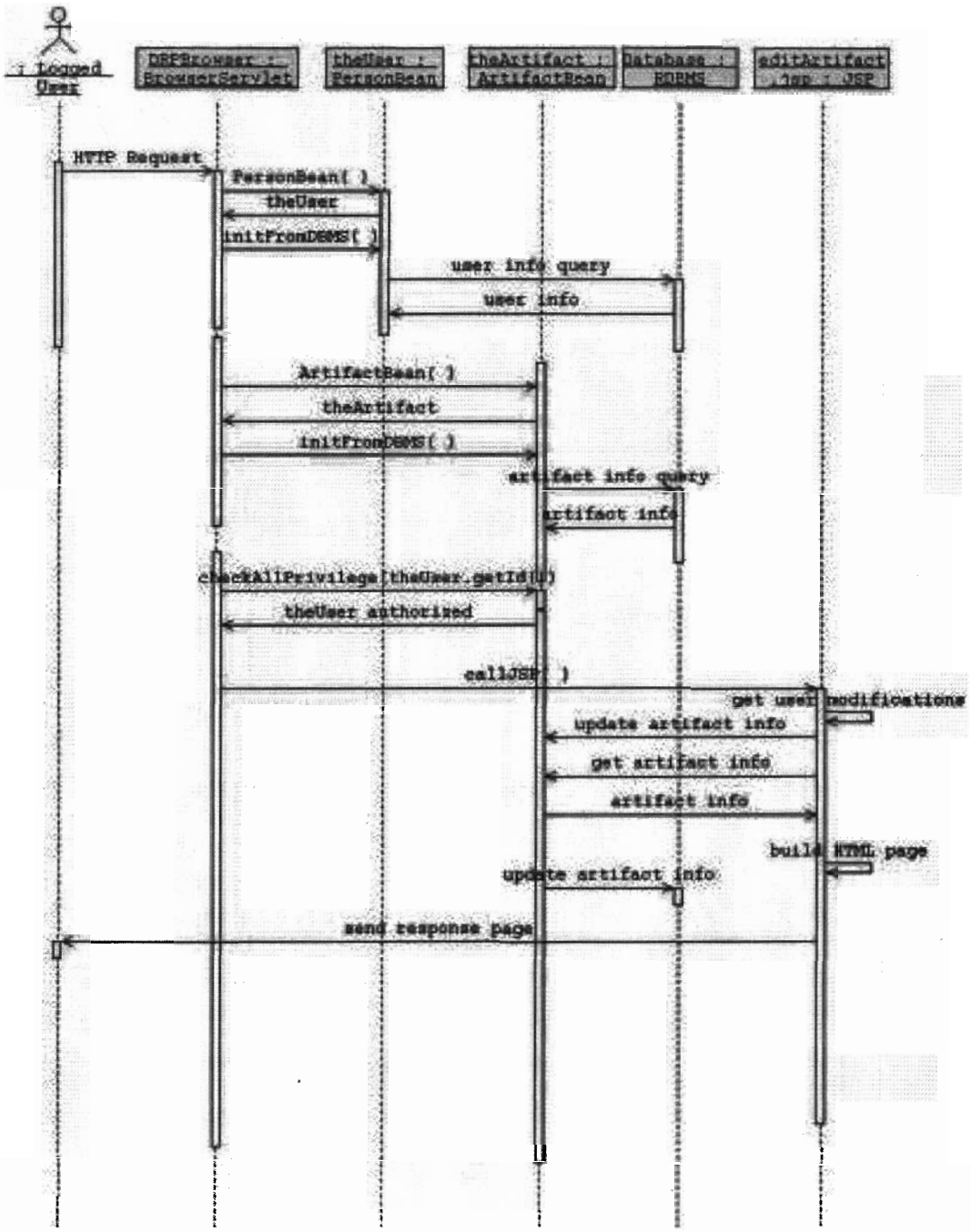

Figure 13. Sequence diagram for the activity *Edit an artifact*

Action: Retrieve user information needed to verify read permission.

Post-condition(s): All information about the user needed to verify read/write permission is in memory.

Switch to author mode

Precondition(s): The interface mode is EDITOR.

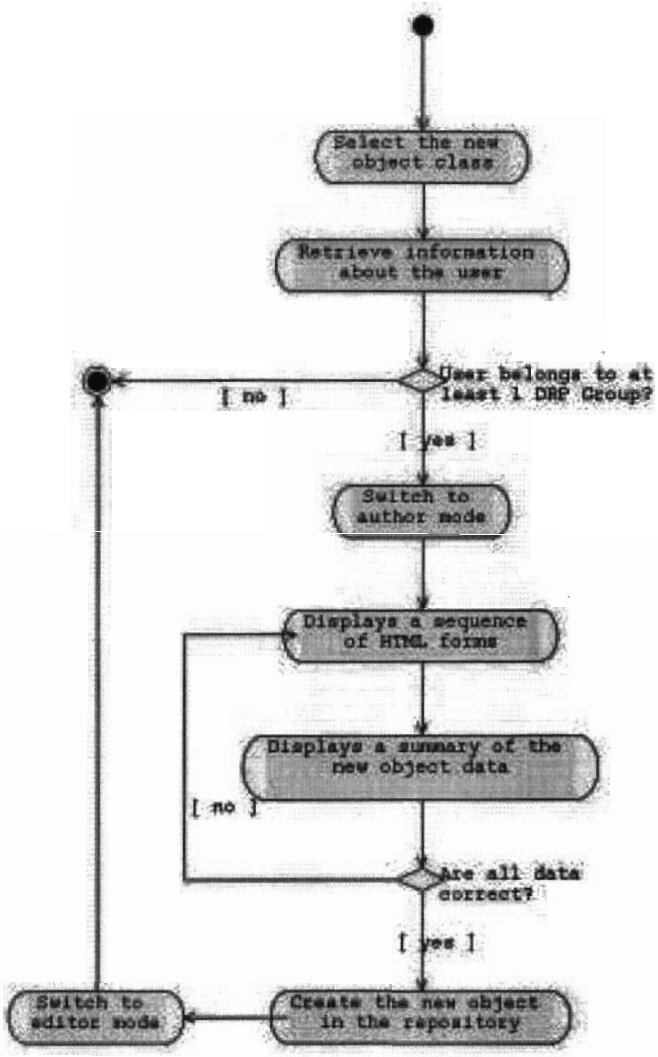

Figure 14. Detailed activity diagram for the activity *Create a new repository object*

Action: Switch the interface mode to AUTHOR. This mode is a special mode only used when creating a new object.

Post-condition(s): The interface mode is AUTHOR.

`Display a sequence of HTML forms`

Precondition(s): The interface mode is AUTHOR.

Action: The web server displays a sequence of HTML forms that allows the user to enter data about the new object. This sequence depends on the new object class.

Post-condition(s): All data entered by the user is in memory.

`Display a summary of the new object data`

Precondition(s): All needed data about the new object is known.

Action: Display all information previously entered by the user and wait for a confirmation.

Post-condition(s): None.

Create the new object in the repository

Precondition(s): All minimum data about the new object is known and the user has confirmed that the data are correct.

Action: Create the new object in the repository.

Post-condition(s): The new object exists in the repository.

Switch to editor mode

Precondition(s): The interface mode is AUTHOR.

Action: Switch the interface mode to EDITOR.

Post-condition(s): The interface mode is EDITOR.

Figure 15 shows the sequence diagram for the activity *Create a new artifact*. This is the sequence diagram for the *Create a new object* activity when the type of design repository object being created is an artifact.

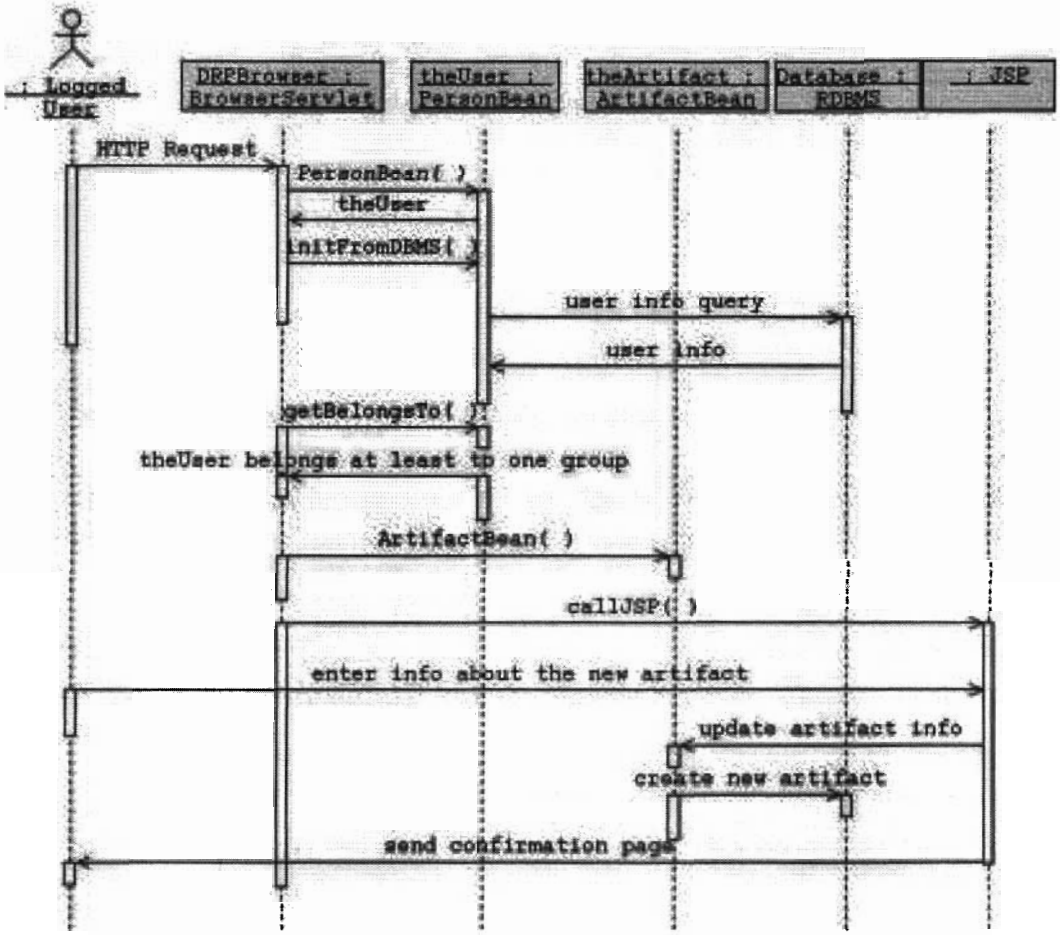

Figure 15. Sequence diagram for the activity *Create a new artifact*

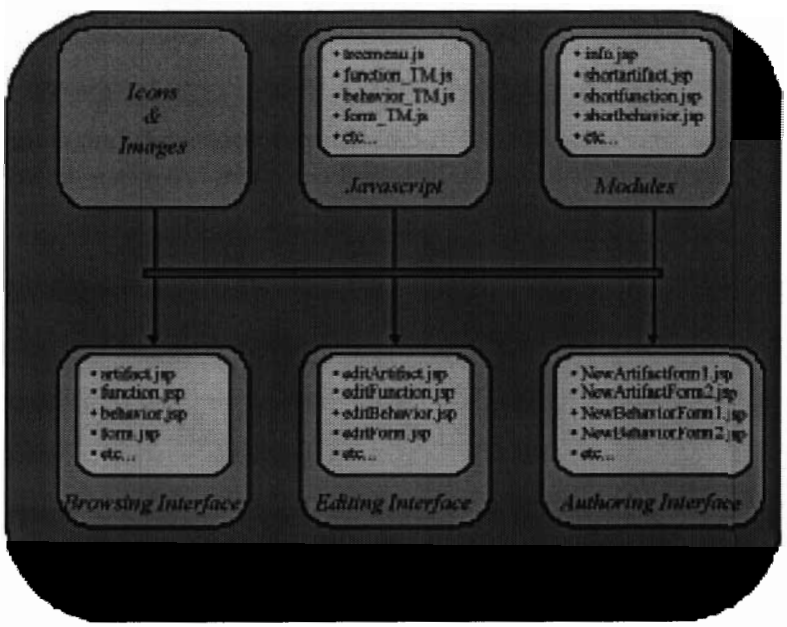

Figure 16. User interface implementation

3.3 Implementation

3.3.1 Interface Descriptions

The user interface logic is implemented mainly using JavaServer Page (JSP). Information is sent from the middle tier to the client interface via the web server using HTTP. This information consists of a combination of HTML (to implement frames and static web pages) and JavaScript (to implement some interactive features such as the tree menu or the expand/collapse feature). JSP syntax provides what is called the *include* directive, which causes a file or block of text to be inserted into a JSP file at compile time. This allows the interface content to be separated into smaller, more manageable, elements. Because many of the interface pages share the same portions of content, these common parts are implemented as separate modules which can be reused as needed using the *include* directive.

Figure 16 illustrates the main blocks into which the Design Repository Browser and Editor interface code is decomposed. Examples of names of JavaScripts and JavaServer Pages are provided for the benefit of anyone who is browsing the Design Repository System code, to make it easier to identify what one is looking at.

The Design Repository Browser and Editor interfaces have a frame-based structure where each of several different frames is used to display certain kinds of information. A schematic of this frame-based structure is shown in Figure 17.

The overall layout for the frame set shown in Figure 17 is contained in a file called index.html. This is a static HTML file that specifies the geometry for the various frames. This file also specifies which additional files supply the content for each of the frames in the frame set. Below are descriptions for each of the frames shown in Figure 17. Names of files corresponding to the content of a frame are given in parentheses; file names that end in "`.html`" are static HTML files, while file names that end in "`.jsp`" are JavaServer Pages that include dynamic content.

- Top Frame (`topFrame.html`): This static frame displays the Design Repository Project banner.

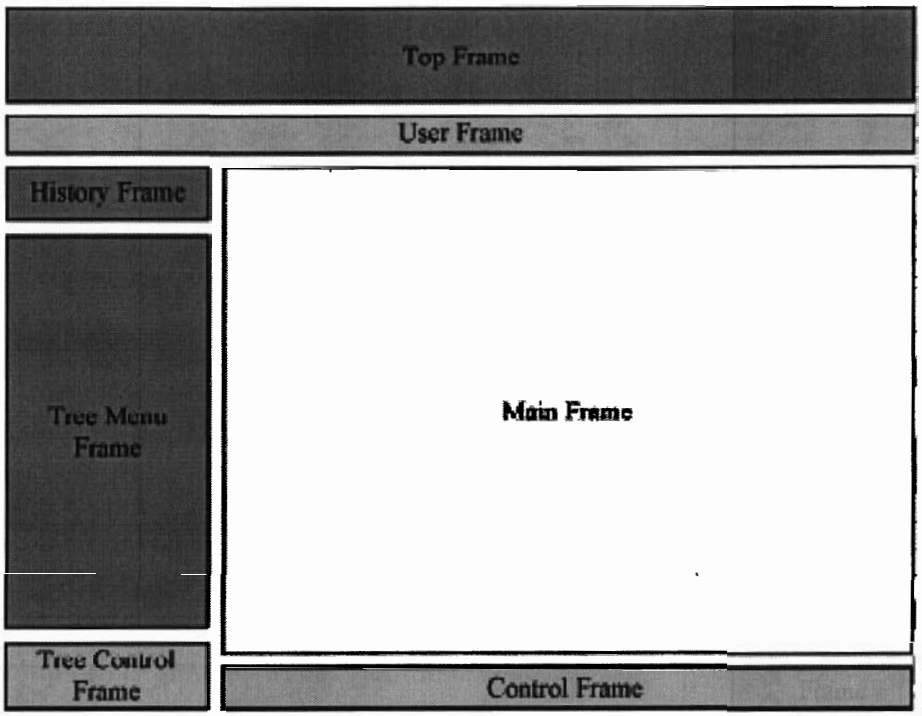

Figure 17. Frame-based structure of the Design Repository Browser and Editor interfaces

- User Frame (`userFrame.jsp`): Displays a welcome message including the name of a user when a user is logged in. The User Frame also contains a link to the search tool and the login/logout page.
- History Frame (`HistoryFrame.jsp`): Displays the list of the objects viewed by the user in the current session and allows quick access to these objects.
- Tree Menu Frame (`treeFrame.jsp`): This frame displays the hierarchical decomposition tree for artifacts (by default) or functions (if selected) for products in the design repository database. The dynamically expandable and contractable hierarchical product structure tree makes use of a third-party script called FolderTree, which is written in JavaScript, and makes use of Dynamic HTML (DHTML) capabilities supported in recent versions of most web browsers. FolderTree is freely available at <`http://www.geocities.com/Paris/LeftBank/2178/foldertree.html`>
- Tree Control Frame (`ctrlTree.jsp`): Displays controls for the Tree Menu Frame, such as switching between an artifact tree and a function tree or rebuilding these trees.
- Main Frame (varies): The main frame is empty in its initial state (using the file `blank.html`). Different JavaServer Pages are used to display content about various objects in a design repository as the user interacts with the system. Editing of objects and authoring of new objects is also done in the Main Frame using JavaServer Pages.
- Control Frame (`ctrlBrowse.jsp` in browser mode, or `ctrlEdit.jsp` in editor mode): Contains controls that affect the content of the Main Frame, such as switching between browser and editor mode, or selection of new objects to create in author mode.

Figure 18 shows the welcome page of the Design Repository System interface. The figure shows the various frames and their default content when a user makes an initial access to the system. At this stage

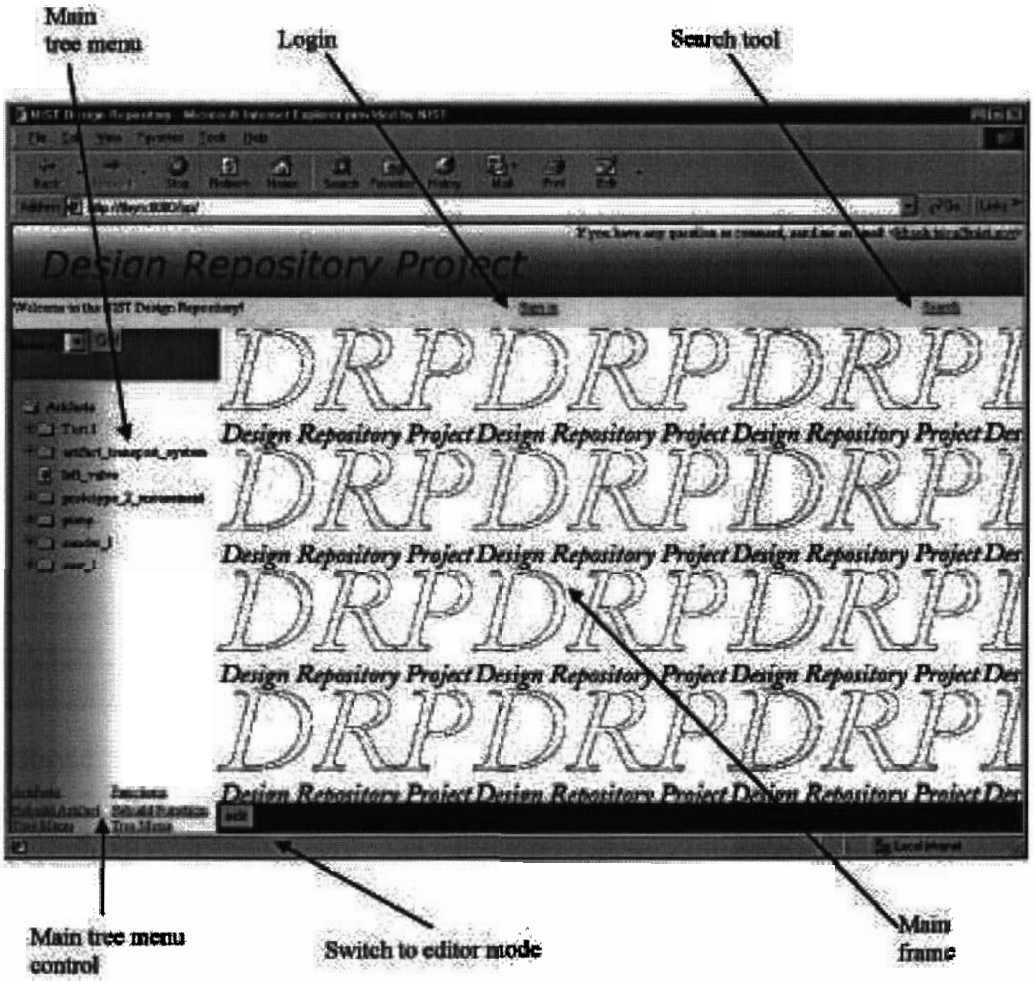

Figure 18. Screenshot of the welcome page

the user has not yet logged in. The user can click on the login link in the User Frame to access a login screen and log in with a user name and password.

Without logging in, a user can view the hierarchical product decomposition tree in the Tree Menu Frame, or can perform a search using the Search Tool. However, a user who is not logged in does not have read privileges to view additional information. Any attempt to view details of the objects that appear in the Tree Menu Frame, or objects resulting from searches, will automatically bring the user to a login screen requiring a login before information may be displayed. The "edit" button in the control frame is used to switch to editor mode, but as with viewing of object details, a user who is not logged in will be presented with a login screen before being able to use the editor.

Figure 19 shows a screenshot of the Design Repository Browser interface when the user is viewing an artifact. The hierarchical product decomposition tree (shown partially expanded in the Tree Menu Frame at the left) allows a user to expand and view in detail one or more portions of a physical hierarchy, while leaving the rest of the tree collapsed to hide the product structure detail that is not currently of interest to the user.

When a design repository object is viewed, all of the information that is related to that particular object is retrieved from the database and sent to the web browser on the client side. Since users typically view an object with an interest in only a subset of this information, the complete object description is

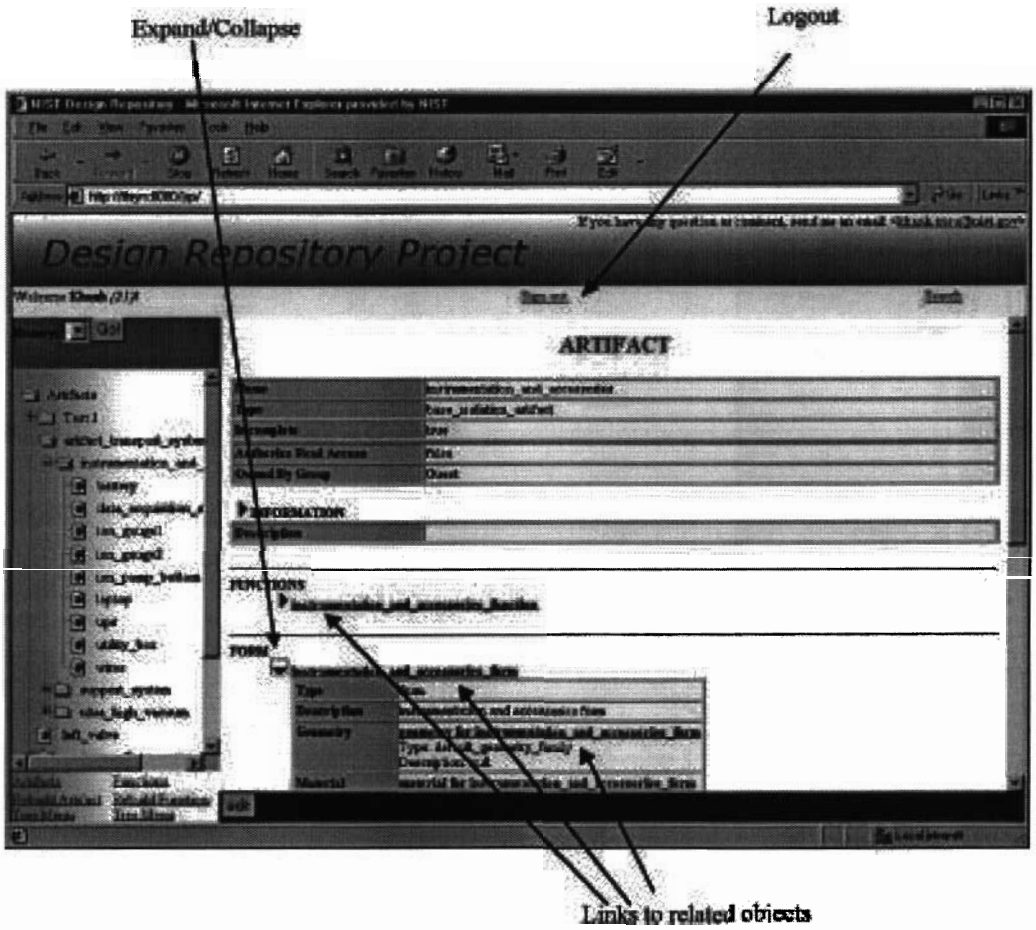

Figure 19. Screenshot of the interface in browser mode

generally more than a user wishes to see at once. Nevertheless, retrieving all of the information at once provides a significant advantage in terms of responsiveness of the interface. Although the time required to retrieve the complete object description is slightly longer than the time it would take to retrieve only the subset of information that a user is interested in, this difference is relatively small. Once the information is on the client side, the user can selectively view different subsets of information without the server having to process any new requests from the client. This approach thereby eliminates even brief delays that would be needed for additional client requests to be processed, information retrieved from the database, and sent back to the client.

To avoid overwhelming a user with an excess of detail, the Design Repository Browser initially displays only a subset of the information that is related to a given design repository object. The detailed information has been sent from the repository to the web browser on the client side, but the information is hidden using an expandable/collapsible display structure implemented using Dynamic HTML. In the interface, black triangles serve as visual cues to show where information can be expanded (when the triangle is pointing to the right) to show more detail, or collapsed (when the triangle is pointing down) to hide detail. In Figure 19, the first two triangles indicate that additional information about the current object and its functions is available. The third triangle is pointing down, indicating that the user has already expanded the view to show additional information about the form of this artifact.

Clicking the "edit" button in the Control Frame switches the interface from Browser mode to Editor mode. A screenshot of the interface in Editor mode is shown in Figure 20. This mode displays much

of the same information that the user can view in Browser mode, but changes the interface to a form, allowing the user to edit information that previously could only be viewed.

In addition to editing textual information, in Editor mode the user can also edit links between design repository objects. For instance, the user can assign an existing function object to an artifact, or identify a set of existing artifact objects as being sub-artifacts of the current artifact. This is done by selecting the type of design repository object to be linked to the current object. Once the object type is specified, the interface changes to a new view shown in Figure 21. The user can either type into the blank text field the name of the object to be linked, or can instead select an existing object of the specified type from a tree menu list that appears in a new sub-frame on the right side of the Main Frame.

Once the user is done editing, clicking the "browse" button commits changes to the database and returns to Browser mode. The user may also choose to create a new design repository object by selecting an object type from the pull-down list in the Control Frame and clicking the "Go" button. Doing so will switch the interface into Author mode. Figure 22 shows a screenshot of the Design Repository Editor in Author mode during the process of creating a new artifact object. As noted previously, although Author mode is implementationally distinct from Editor mode, there is no need for the user to be aware of the difference since permissions for editing and authoring are the same. From the user's perspective, the Design Repository Editor can be used either for editing existing information or for authoring new information.

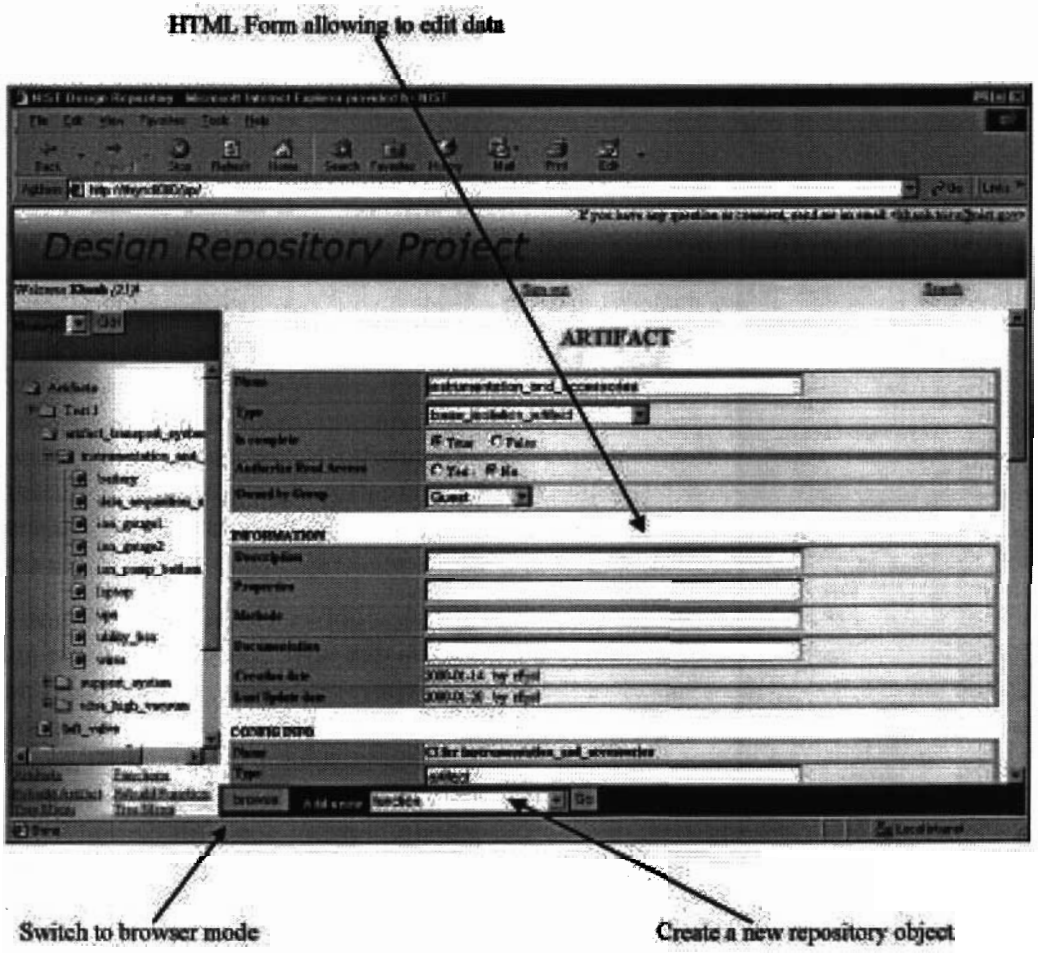

Figure 20. Screenshot of the interface in editor mode

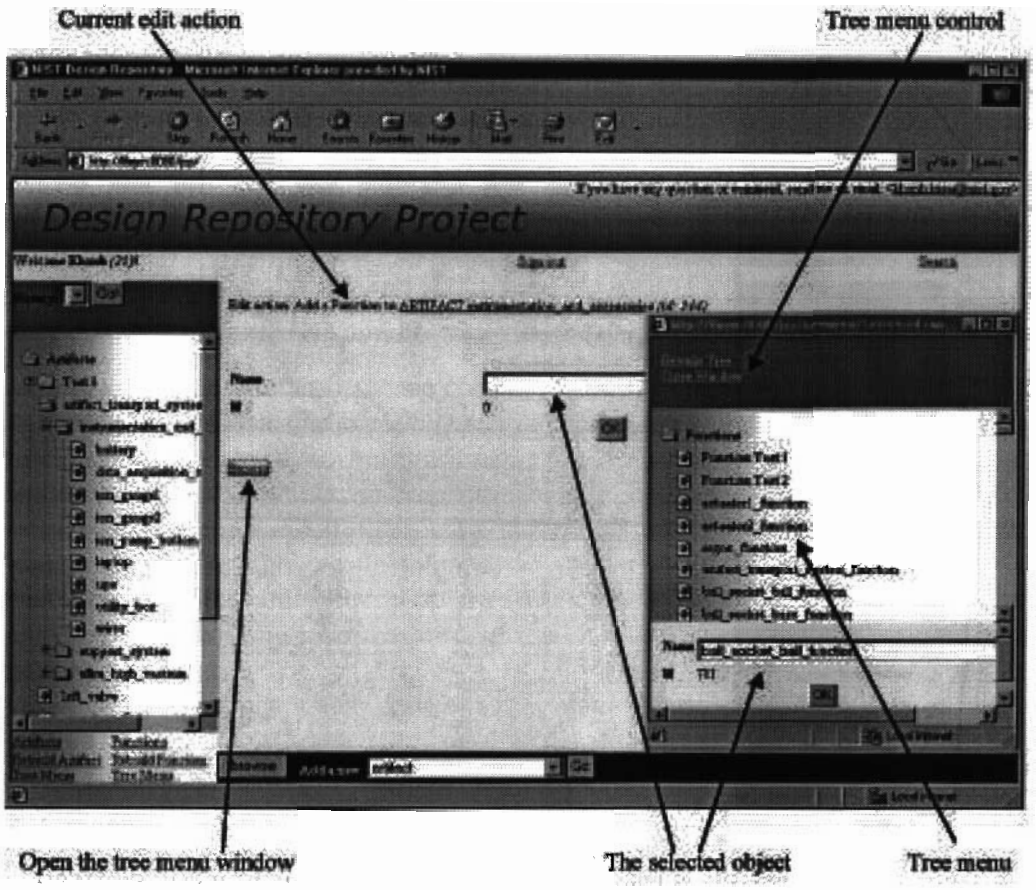

Figure 21. Screenshot of the interface when adding a sub-object

3.3.2 Core Application Logic

This section provides in greater technical detail regarding the implementation of the Design Repository Browser and Editor interfaces. More specifically, a more in-depth description of how the Design Repository System responds to user requests, both on the input side (the Request Handler) and the output side (the user interface seen at the client side), is given. This information is provided for the benefit of developers/implementers involved with the NIST Design Repository Project, and will be of less interest to the general reader.

The Request Handler is implemented mainly using Java Servlets, although a few controls are included in JavaServer Page files. The Request Handler consists of a set of seven Servlets, each of which responds to a specific type of request. Figure 23 shows the Java Servlet class diagram for the Request Handler. The next portion of this section provides additional information about each of the Servlets, including their aliases (when applicable), brief descriptions of what the Servlets do, and more detailed descriptions of the functionality of the methods[7] that they implement.

---

[7] Note that Figure 23 shows doGet and doPost methods for all of the Servlet classes because they extend (i.e. they are subclasses of) the standard Servlet `HttpServlet`, which has those methods. The functionality of a method is only described below when a method overrides the standard doGet or doPost method. If one of the methods is not explicitly mentioned in a Servlet description, it means that the Servlet does not override the generic method provided by the standard `HttpServlet` Servlet.

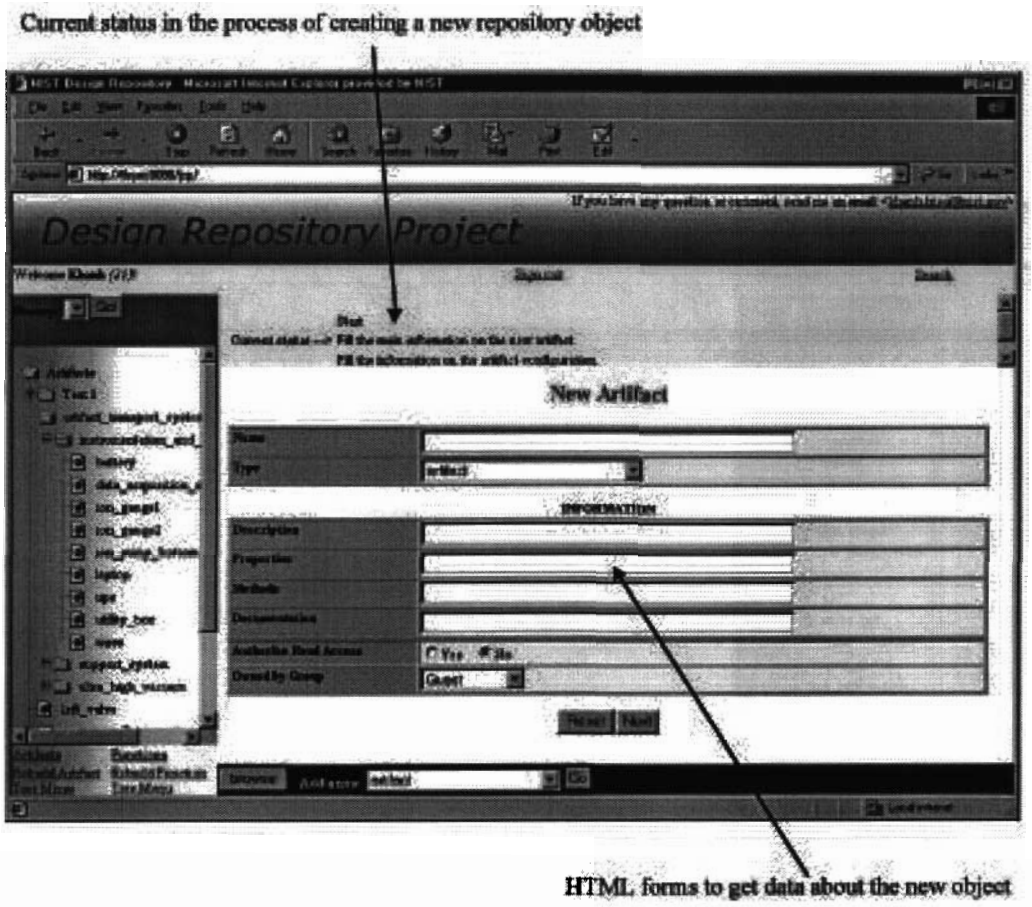

Figure 22. Screenshot of the interface in author mode

`BrowserServlet`

Alias: DRPBrowser

This Servlet handles user requests for viewing information about a design repository object. It also handles requests for updating the main information about an object.

doGet method:

- Retrieves parameters from the request and session objects. These parameters are data that identify the Logged User and the object that the user wants to view.
- Verifies that the user has the permissions to view (and edit, if in Editor mode) the object. If not, redirects the user to the login page. Otherwise,
- Creates a new instance of the corresponding JavaBean (`ArtifactBean` for an artifact object, `FunctionBean` for a function object, and so on).
- Directs the JavaBean to retrieve the relevant data from the database and store them in the JavaBean
- Calls the corresponding JavaServer Page to display the data.

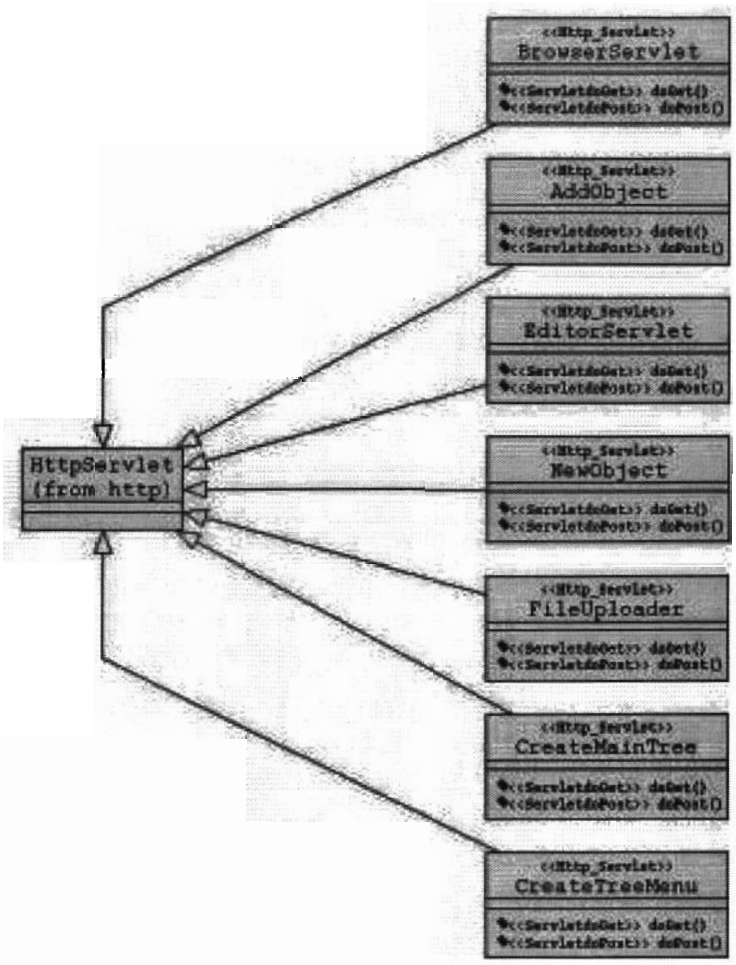

Figure 23. Java Servlet class diagram for the Request Handler

- If the session parameter *update* is equal to *true*, which would indicate that the user has changed some data, updates the database with the new data (the changes are stored in the JavaBean instance)

doPost method: the HTTP POST request is sent only when the user has modified some data. The changes are stored in the JavaBean instance using JavaServer Page introspection.

- Changes the value of the session parameter *update* to *true*.
- Calls the doGet method.
- Changes the value of the session parameter *update* to *false*.

`AddObject`

This Servlet handles user requests for adding a link between the currently viewed object and another one (for instance adding a function object to an artifact object).

doGet method:

- Retrieves parameters from the request and session objects. These parameters are data that identify the Logged User, the currently viewed object, the object to be linked, and the type of association between the two objects.
- For each of the two objects involved with the link, creates a new instance of the corresponding JavaBean.
- Verifies that the object to add exists in the repository. If not, redirects the request to the Servlet DRPEditor, which allows the user to create a new object. Otherwise,
- Creates the link between the two objects.
- Forwards the request to the JavaServer Page file `ObjectAdded.jsp`, which displays a confirmation message.

`EditorServlet`

Alias: DRPEditor

This Servlet handles user requests for creating a new design repository object.

doGet method:

- Retrieves parameters from the request and session objects. These parameters are data that identify the Logged User and the type of object that is being created.
- Verifies that the user has the permissions to create the object. If not, redirects the user to the login page. Otherwise,
- Creates a new instance of the corresponding JavaBean (`ArtifactBean` for an artifact object, `FunctionBean` for a function object, and so on).
- Calls the corresponding JavaServer Page to display HTML forms that allow the user to enter information about the new object.

`NewObject`

Once the user has entered all information needed to create a new design repository object and confirmed that the information is correct, this Servlet creates the new object in the design repository database.

doGet method:

- Retrieves parameters from the request and session objects. These parameters are data that identify the Logged User and the JavaBean instance containing the information about the new object to be created.
- Verifies that the user has the permissions to create the object. If not, redirects the user to the login page. Otherwise,
- Creates the new object in the design repository database.
- Forward the request to the JavaServer Page file `ObjectAdded.jsp` that displays a confirmation.

`FileUploader`

This Servlet handles user request for uploading geometry files through the web server to a specified directory on the machine on which the Design Repository System is running. It uses the Java class `com.oreilly.Servlet.MultipartRequest`.

doPost method:

- Creates an instance of the `MultipartRequest` class. This will retrieve parameters from the request object and start to upload the geometry file.
- Creates an instance of the `FileBean` class and sets its attributes with values retrieved by the `MultipartRequest` object.
- Directs the `FileBean` object to update the database with information about the geometry file (e.g., file name and path name).

`CreateMainTree`

The main tree menus that are displayed in the Menu Tree Frame (the artifact object hierarchy tree and function object hierarchy tree) are built based on static files when user first connects to the Design Repository System. This Servlets handles user requests to rebuild these static files in order to update the hierarchies in the Menu Tree Frame.

doGet method:

- Retrieves parameters from the request and session objects. These parameters are data that identify the Logged User and the type of tree menu to update (the artifact object hierarchy or the function object hierarchy).
- Creates an instance of `TreeMenu` class.
- Directs the `TreeMenu` object to retrieve the information needed to create the tree menu from the database.
- Generates and writes an updated data file.

`CreateTreeMenu`

The Design Repository System makes use of hierarchical object trees in two different ways. One way is to display information in the Menu Tree Frame to aid in browsing a large hierarchical product structure. The Menu Tree Frame displays artifact and function object trees where names of objects can be clicked on to view information about that object in the Main Frame.

The other way that object trees are used is to provide the user a hierarchical list of objects of a specific type, for the purpose of helping the user locate an object that should be linked to another object when using the Design Repository Editor (see the sub-frame in Figure 21). These trees are not limited to artifact and function objects, but may be of other design repository object types as well. They are similar in form to the trees displayed in the Tree Menu Frame, but because they serve a different purpose (e.g. clicking on an object name does not display information about that object), the syntax of the static file from which the trees are built differs. This Servlet is functionally equivalent to the previous one, but is a separate Servlet because of the differences in the syntax of the files for the two types of menu tree.

To provide additional information about the structure of the implementation, Figure 24 shows the package `drp` class diagram. Below are descriptions of each of the class types shown in the figure.

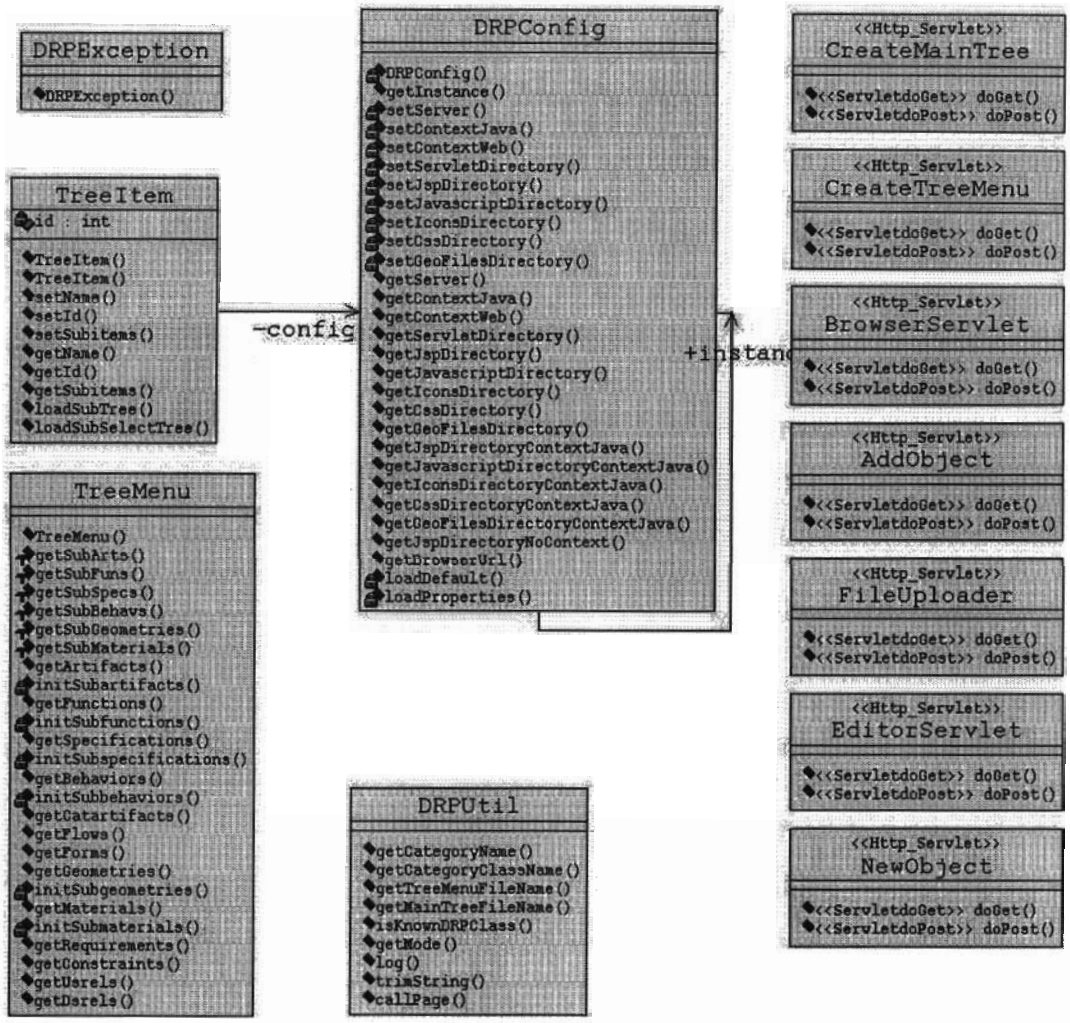

Figure 24. Package drp class diagram

- `DRPUtil`: This class provides a collection of constants and static methods used by the other classes.
- `DRPConfig`: This class retrieves configuration information from the `drp.properties` file. These data include paths and directory names for JavaServer Page files, JavaScripts, graphical icons, etc. This data then available to the other classes and JavaServer Page files.
- `DRPException`: This class is a subclass of the general `Exception` class.
- `TreeItem`: This class allows the representation of a tree menu in memory. Each instance of `TreeItem` includes a design repository object name and ID. These instances represent a node in the tree menu, or a leaf if a given object has no "sub-items" (i.e. sub-artifacts or subfunctions).
- `TreeMenu`: This class retrieves the data needed to build a tree menu from the database. These data are stored in memory using the `TreeItem` class.

Figure 25 shows the package `drp.beans` class diagram. This package groups together all JavaBeans used by JavaServer Page files. This is, fundamentally, the implementation of the Database Exchange Manager.

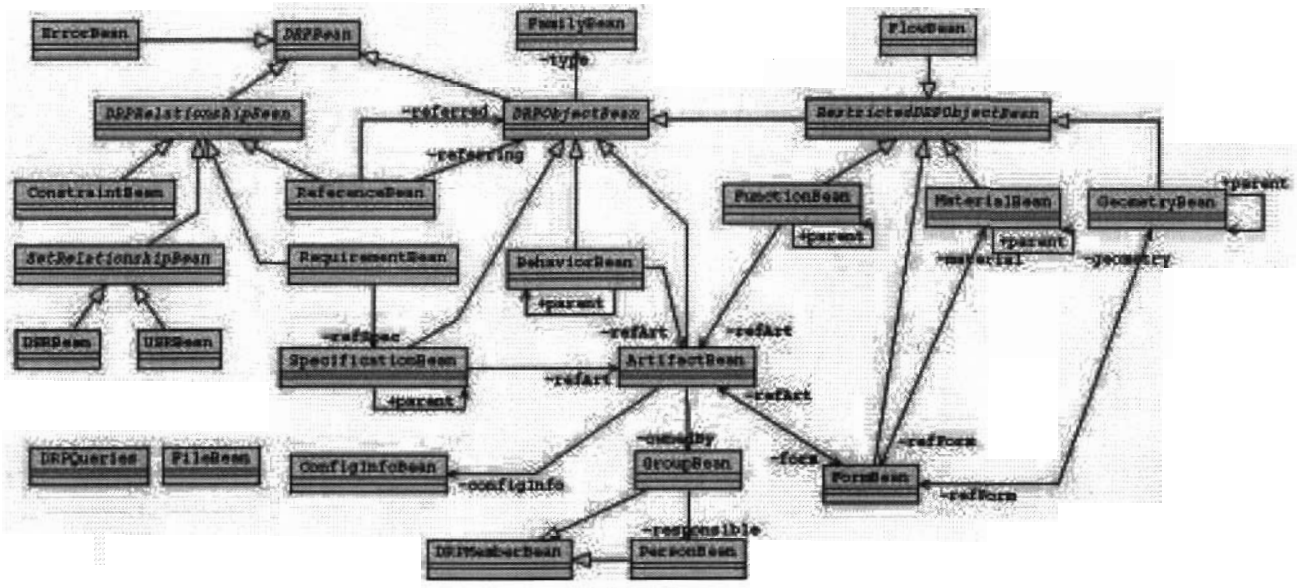

Figure 25. Package `drp.beans` class diagram

3.4 Testing

This section summarizes the tests that were performed on the functionality of the Design Repository Browser and Editor interfaces. Note that the tests are described only at a high level. A test often consisted of several steps, and in cases where there is more than one way to accomplish a given functions from the interface, the functions were tested in multiple ways.

Tree menu test:

- Test: Expanding and collapsing the tree menu.

  Result: No problems detected with Internet Explorer 4 and Netscape Navigator 4.

- Test: Rebuilding tree feature (synchronizes data with the database) verifying consistency of data.

  Result: No problems detected when rebuilding the tree menu. However, with Netscape Navigator 4, sometimes the browser crashes when refreshing the web page. Problem is not consistently repeatable. (This is both good and bad; the problem will affect the user only occasionally, but it is harder to identify the source of the problem in order to fix it.)

Viewing a repository object test:

- Test: Checking consistency of information displayed for each type of design repository object against the corresponding information in the design repository database.

  Result: No problems detected.

- Test: Expanding and collapsing design repository object information in the Main Frame.

  Result: No problems detected.

- Test: Checking hyperlinks to verify that they are active, and that the object retrieved corresponds to the name of the object that was clicked on.

Result: No problems detected.

Editing an existing repository object test:

- Test: For each type of design repository object, edit the data and verify that the data has been correctly updated in the design repository database.

  Result: No problems detected.

- Test: For each type of design repository object, try to add a sub-object (creating a link between the current object and another object) and verify that the data has been correctly updated in the design repository database.

  Result: No problems detected.

- Test: Upload a geometry file and verify that the file has been correctly uploaded and the link to the geometry file contains the correct file path to the file on the server.

  Result: No problems detected.

Creating a new repository object test:

- Test: For each type of design repository object, create a new object and verify that the object has been correctly created in the design repository database, and contains the correct data.

  Result: No problems detected.

4 DESIGN REPOSITORY SEARCH TOOL

4.1 Functionality Summary

The Design Repository Browser provides an interface for navigating information in product models stored in the design repository database. As suggested by the name of the interface, the mode in which it is used is mainly for browsing information that is already of some particular interest. For a repository of interest, the user can view the hierarchical product decomposition and select particular artifacts to view, for an artifact of interest the user can retrieve information about its form, functions, or move to other design repository objects that have some connection to this one (such as sub-artifacts, among others). One of the main motivations for creating design repositories is to archive information so that it can later be browsed, allowing designers to gain insight into how the product was designed, decomposed into subsystems, what the function of various assemblies or components are, how energy flows through systems, and so on.

In addition to providing insight regarding previously designed products, the second motivation for creating design repositories is to enable information retrieval in support of design knowledge reuse for subsequent product development efforts. Whether a designer is attempting to find a known component or assembly or simply trying to find portions of previous designs that can be used to accomplish similar functions in a new design, a browsing interface is not an effective means of retrieving information from a potentially large product knowledge base. The Design Repository Search Tool was developed to facilitate search and retrieval of information from design repository databases.

The Design Repository Search Tool allows the user to search for design repository objects according to their type (artifact, function, flow, etc.). Items can be searched for according to a variety of different search attribute criteria, for example by name or by type. Searches can require exact matches (e.g., an

artifact whose name is *Motor_17B*) or substring matches (e.g., any artifact whose name contains the string *motor*). More complex searches can be constructed using Boolean (and/or) combinations of search criteria (e.g., any artifact whose name contains *motor* AND whose description contains *12 volt* OR whose description contains *12V*).

The Design Repository Search Tool is a separate software component from the Design Repository Browser interface, but the Design Repository System implementation allows the two modules to function in an integrated fashion. Users can access the Design Repository Search Tool from the Design Repository Browser interface, and clicking on a design repository object that is returned with the results of a search automatically brings the detailed information about that object into view in the Design Repository Browser.

4.2 Interface Specifications

4.2.1 Use Cases

Figure 26 shows the relatively simple use case schema for the Design Repository Search Tool. As was mentioned in Section 3, the Design Repository Search Tool can be accessed by any user, even if the user is not logged in. If a user who is not logged in attempts to retrieve information about a design repository object that is included within a set of search results, the user will automatically be directed to the login page. This functionality is handled as part of the *view a repository object* activity within the Design Repository Browser, and thus is not part of the logic of the Design Repository Search Tool.

4.2.2 Activity Diagrams

The activity diagram for the Design Repository Search Tool is shown in Figure 27. This diagram is essentially a detailed activity diagram for the *Search an object* activity shown in Figure 9 (Section 3.2.2). In that activity, the user accesses the Design Repository Search Tool from the Design Repository Browser. The post-condition for that activity is that the user is looking for an object; this is shown as the starting point in Figure 27. The descriptions for the activities shown in the figure are as follows:

`Select object type`

Pre-condition: The user has accessed the Design Repository Search Tool from the Design Repository Browser interface.

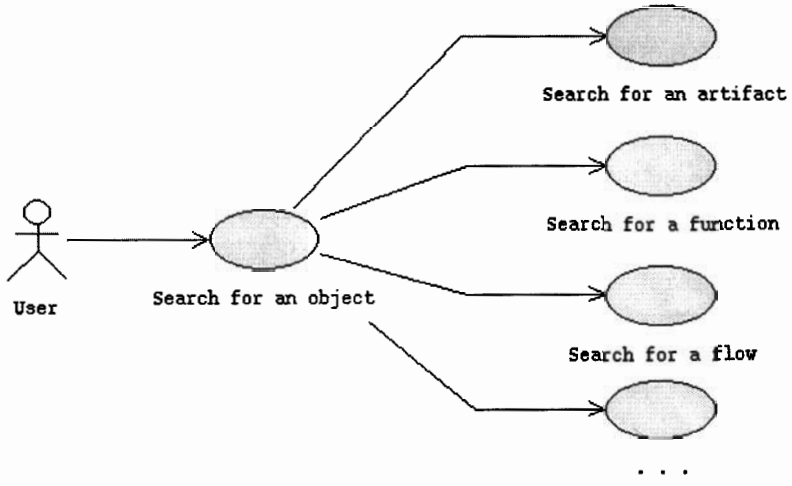

Figure 26. Use case schema for the Design Repository Search Tool

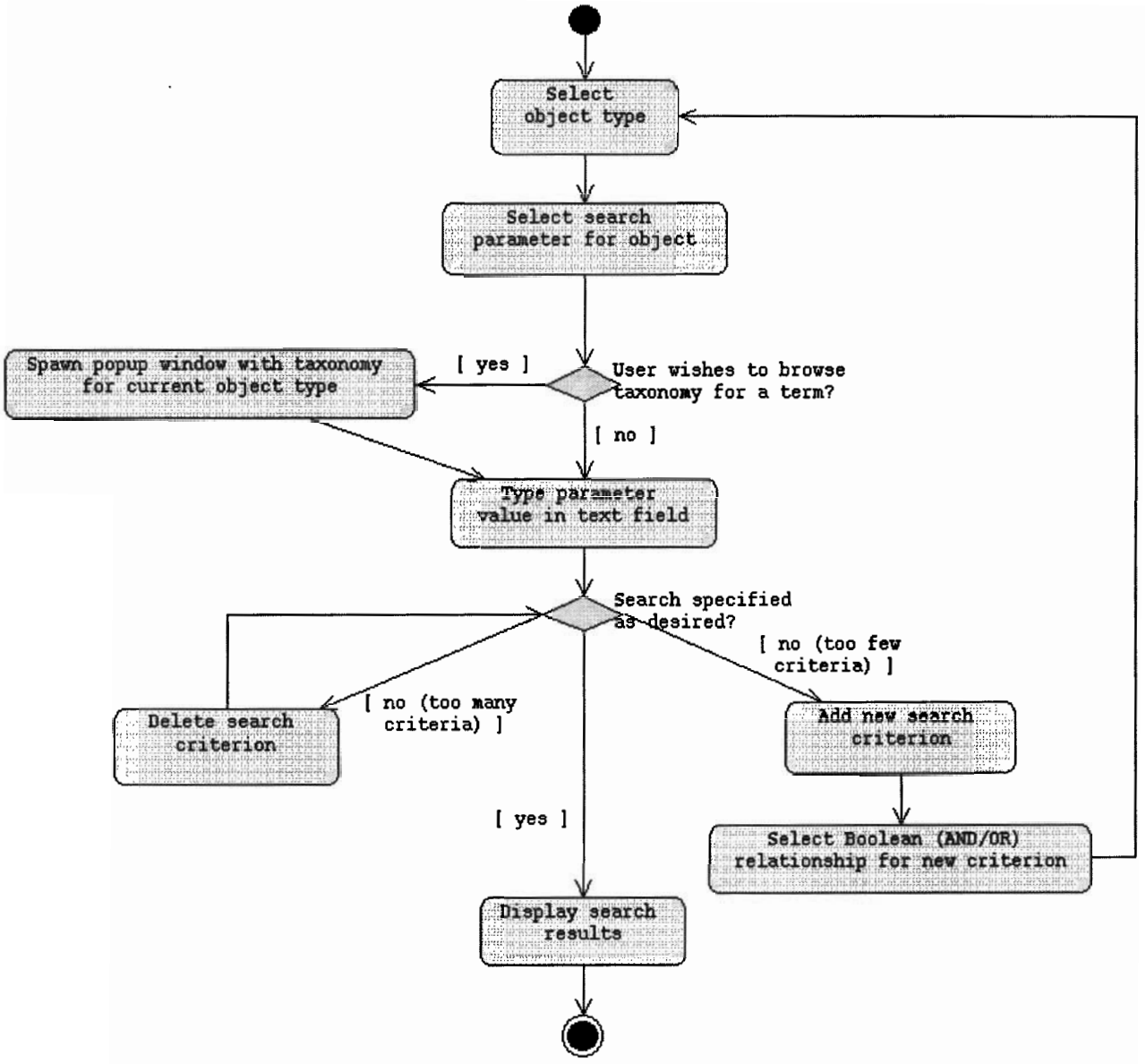

Figure 27. Activity diagram for the Design Repository Search Tool

Action: The user selects the type of design repository object to search for from a popup list.

Post-condition: The object type has been selected, and the list of search parameters is updated based on the type of object that has been selected so that the user can begin specifying search criteria appropriate for that object type.

`Select search parameter for object`

Pre-condition: The user has specified an object type for the search.

Action: The user selects a search parameter appropriate to that object from the popup list.

Post-condition: The search parameter for the current parameter has been specified.

`Spawn popup window with taxonomy for current object type`

Pre-condition: The user wishes to browse the taxonomy for the current Function or Flow object.

Action: The user clicks on the Function or Flow button in the interface.

Post-condition: A separate popup window is spawned, which lets the user browse the various levels of the Function or Flow taxonomy to locate a term that matches what the user is looking for.

`Type parameter value in text field`

Pre-condition: The user has selected a search parameter for the given object.

Action: The user types a value for the parameter in the text field.

Post-condition: The desired parameter value for the search criterion has been entered.

`Delete search criterion`

Pre-condition: The user has not fully specified the search and wishes to delete a search criterion.

Action: The user clicks on the appropriate button to delete a search criterion from the list.

Post-condition: The last search criterion on the list is deleted.

`Add new search criterion`

Pre-condition: The user has not fully specified the search and wishes to add a new search criterion.

Action: The user clicks on the appropriate button to add a new search criterion to the list.

Post-condition: A new search criterion, in the form of a popup menu of parameter types and a text field for the desired value, is added to the list.

`Select Boolean (AND/OR) relationship for new search criterion`

Pre-condition: The user has added a new search criterion.

Action: The user clicks on either the "AND" or the "OR" button.

Post-condition: The user has specified the Boolean relationship for the new search criterion.

`Display search results`

Pre-condition: The user has specified the search as desired.

Action: The user clicks on the "OK" button.

Post-condition: The search results are displayed in the interface.

Figure 28 shows the sequence diagram that corresponds to the activity diagram shown in Figure 27.

4.3 Implementation

4.3.1 Interface Descriptions

The user interface logic for the Design Repository Search Tool is implemented using JavaServer Page (JSP), and communication with the database is handled with JavaBeans. Figure 29 shows the main blocks into which the Design Repository Search Tool code is decomposed. The names of JavaServer Pages and JavaBeans are provided for the benefit of anyone who is browsing the Design Repository System code. The user interface is divided in three different parts (see Figure 30). The top region al-

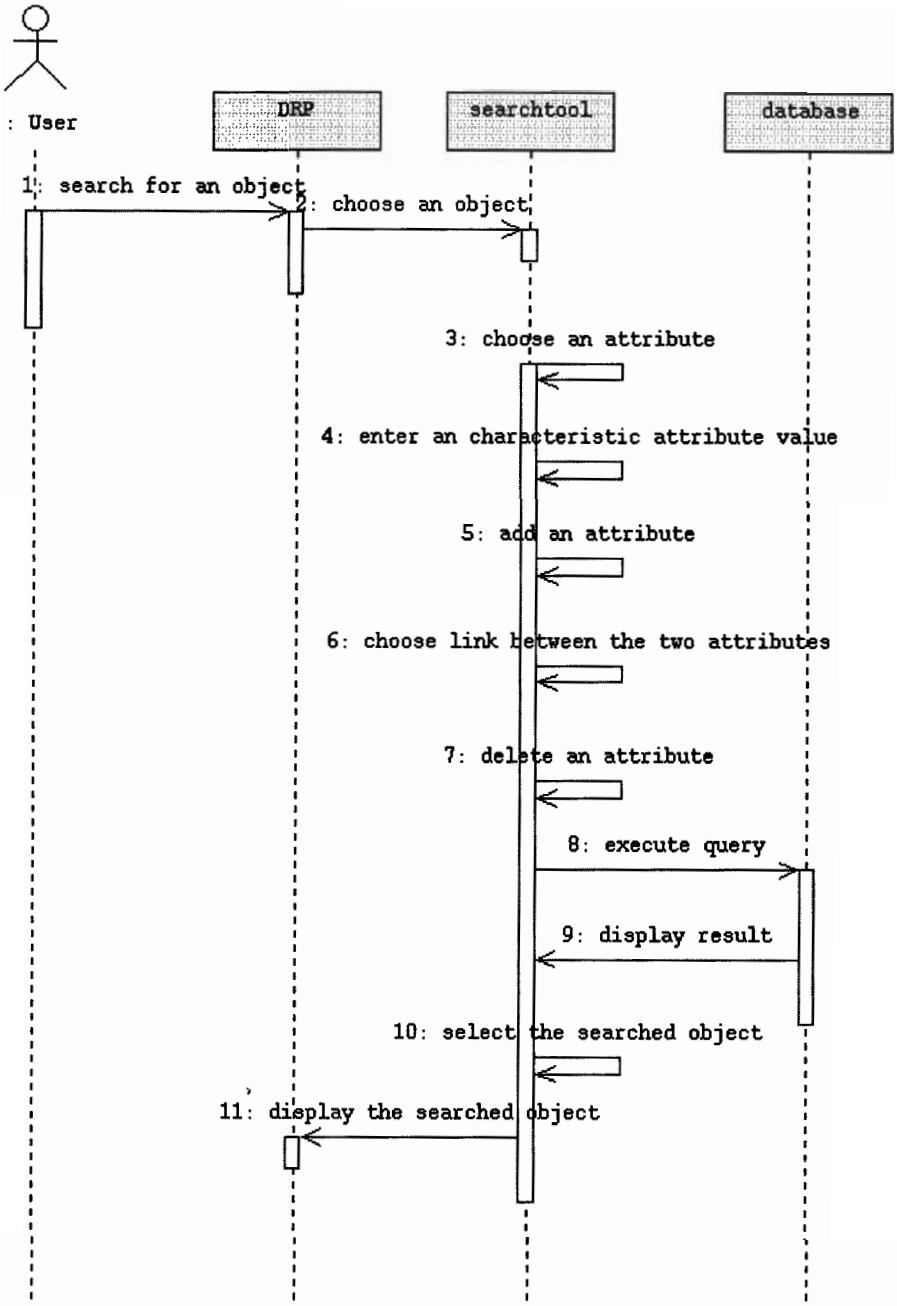

Figure 28. Sequence diagram for the activity *Search for an Object*

lows the user to specify which type of design repository object to search for, the middle region allows the user to specify the parameters for the search, and the third region shows the results of a search.

The first step in conducting a search is to specify which type of design repository object to search for. This is done via a pull-down list. Next, the user specifies a search criterion by selecting a search parameter also from a pull-down list, and typing in a value in a text field. The user can search for any object by type, by name, by matching text in the object description, by creator, and other parameters.

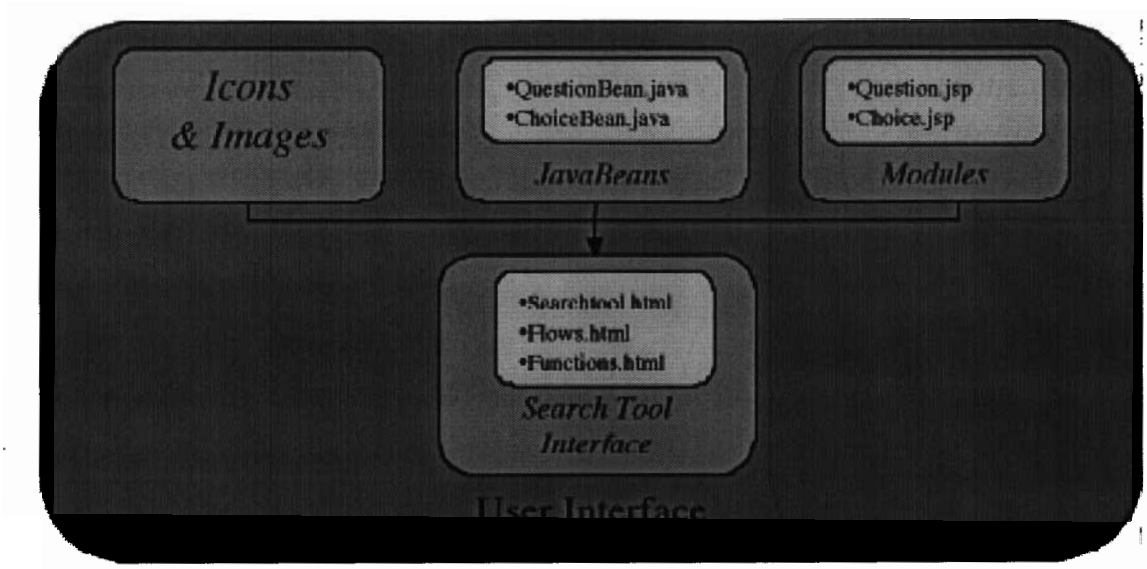

Figure 29. Design Repository Search Tool user interface implementation

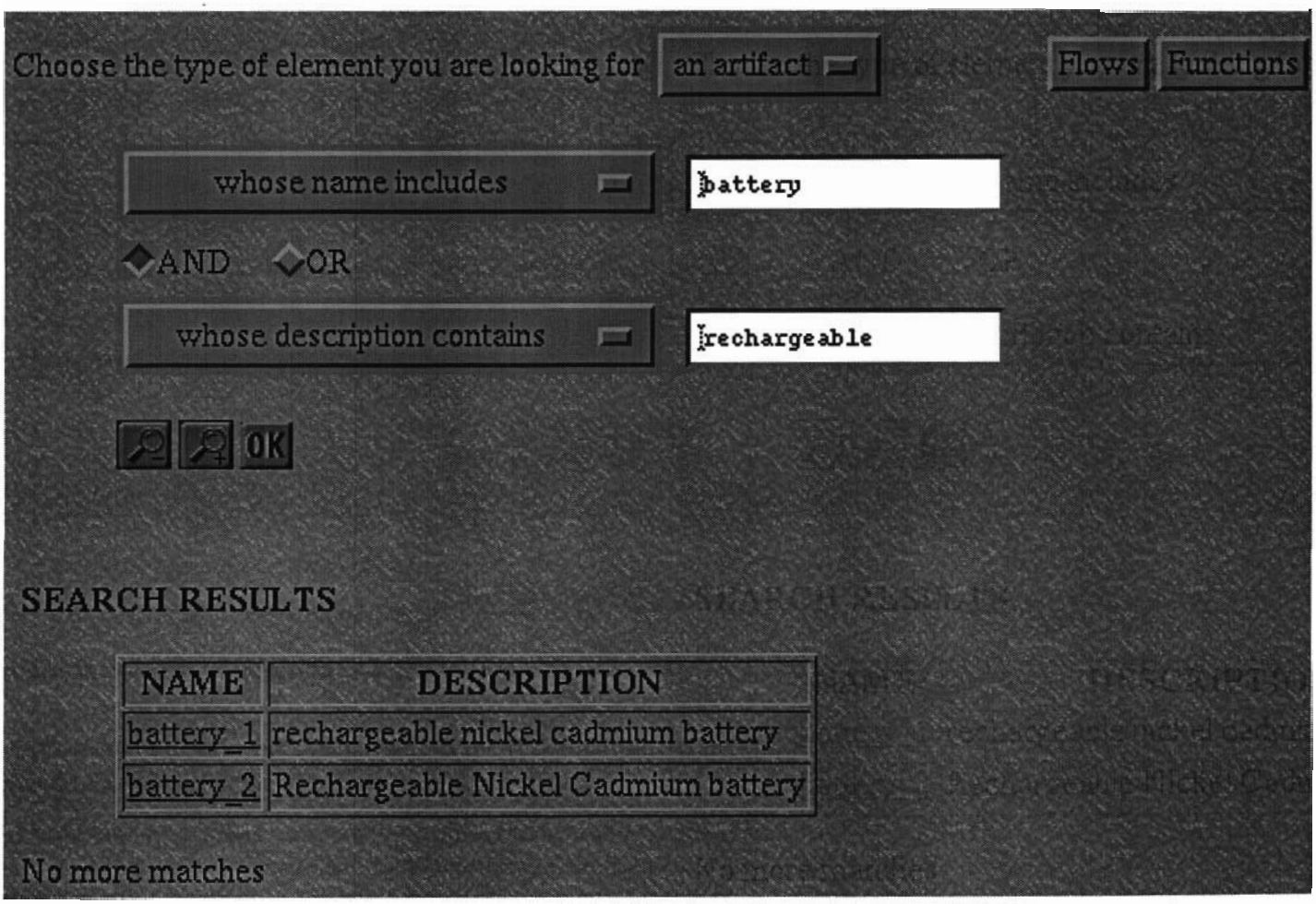

Figure 30. Design Repository Search Tool interface

In addition to these generic parameters, the second pull-down list is customized for each particular object type (e.g., pull-down list for searching for artifacts contains some choices that are not on the list for functions, and vice-versa). The selection of an object type in the first pull-down list automatically triggers an update of the second menu with the appropriate parameters for that type.

At this point, if the user has fully specified the desired search, clicking the OK button will submit the search. In some instances, it is useful to perform a search using multiple search criteria. To accomplish this, the user clicks on the button that shows a magnifying glass and a "+" sign. This adds another search criterion for the user to specify, again by selecting a parameter from the pull-down list and typing a value into the text field. The user also clicks on a button to select a Boolean value (either "AND" or "OR") to indicate how this new criterion should be incorporated into the search. Using "AND" narrows the search by requiring that results of a search match both criteria, while using "OR" broadens the search by matching on results that satisfy any of the criteria.

The user can specify multiple search criteria from the start, or can perform a search with a single criterion and then refine the search by adding additional criteria to expand or narrow the search, based on the results of the previous search. When a search comes up with too few results (or none at all), expanding the search can provide the user with additional results. Conversely, when a search yields too many matches, narrowing the search with additional criteria can prune the list to a more manageable size and can also provide more specific (and hopefully more useful) results. The user can also reduce the number of search criteria by clicking on the magnifying class with a "–" sign to remove the last one on the list (this button is non-functional when there is only one criterion listed since at least one must be specified for a search to be possible).

Once a search has been submitted, the results are returned in a tabular form showing a list of design repository objects that matched the search criteria, along with their descriptions. The objects listed in the results are hyperlinks, allowing the user to bring up object details in the Design Repository Browser interface simply by clicking on the object name.

When searching for functions and flows, it is generally more convenient to search by type than by name. The Design Repository Search Tool therefore includes additional functionality to support type-based searches for functions and flows. Because the function and flow types are specified in a hierarchical taxonomy of terms, the interface provides a mechanism for quickly locating appropriate terms to use in a search. The "Functions" and "Flows" buttons at the top right corner of the Design Repository Search Tool pop up separate windows that allow the user to hierarchically browse the function and flow taxonomies. These two taxonomies are stored in files called Functions.xml and Flows.xml, respectively. The user identifies the correct function or flow term that matches a particular concept by selecting a high-level function or flow type, followed by two additional levels of more detailed subtypes.

If the user selects a term from the bottom level of subtypes, that term is used as the type for the function or flow object search. If the user wishes to be less specific and uses a term from one of the higher levels in the taxonomy, the search will include all of the terms that are subtypes of the higher-level term, and will match on objects having any of those types. For example, if the user enters the term *Separate* as a function, the search will match on functions that are of type *Divide*, *Extract*, or *Remove*, all of which are subtypes of *Separate*.

Once a search has been performed, if the search was too specific and included fewer results than the user wanted, the user can insert a "^" character in front of the function or flow type and repeat the search. This will broaden the search by searching for all of the terms that are on the same branch of the taxonomy as the term initially used. Specifically, the Design Repository Search Tool retrieves the

supertype of the term used, and repeats the search for all of the subtypes of the higher-level term. Returning to the example terms mentioned above, if the user performs a search for the function *Remove*, and then repeats the search after inserting a "^" character in front of the term, the new search will match on functions of the types *Divide*, *Extract*, or *Remove*, as these are all on the same branch below the term *Separate*.

4.3.2 Core Application Logic

This section provides in greater technical detail regarding the implementation of the Design Repository Search Tool. This information is provided for the benefit of developers/implementers involved with the NIST Design Repository Project, and will be of less interest to the general reader. At the implementation level, the Design Repository Search Tool handles its requests much like the request handler of the Design Repository Browser and Editor interfaces (see Section 3.3.2). Providing less complex functionality, the Design Repository Search Tool request handling mechanism consists of only one Servlet called `SearchTool`.

As with the Servlets comprising the Design Repository Browser and Editor Request Handler, the `SearchTool` Servlet extends the generic `HttpServlet` class. This Servlet does not implement a doPost method. A summary of the doGet method implemented for the `SearchTool` Servlet is as follows:

doGet method:

- Parse the interface configuration file to generate the user interface. The interface configuration file defines the contents of the menus in the Design Repository Search Tool interface (such as what kinds of design repository objects can be searched for, or what attributes can be used as search parameters). This configuration information is kept in an XML file to separate it from the code itself. Thus the types of searches that are possible can be reconfigured as new needs arise without having to modify the Design Repository Search Tool code itself.
- Translate the information that a user has specified in the interface to a corresponding set of SQL queries that the database system can execute.
- Obtain a connection to the database from the Connection Manager and submit the SQL queries. (A set of requests is sent sequentially using the same connection.)
- Uses a JavaBean to capture the relevant data from the database.
- Calls the corresponding JavaServer Page to display the results in the client web browser.

To provide additional information about the structure of the implementation, Figure 31 shows an implementation diagram for the Design Repository Search Tool. Below are descriptions of each of the entities shown in the figure that are specific to the Design Repository Search Tool. Those entities in the figure that are not listed below are either associated with another portion of the implementation (e.g., the Connection Manager), or are more generic Java classes that are used by, but not developed within, this project (e.g., the generic `HttpServlet`, and those classes denoted by org.w3c.dom in the figure).

- `SearchTool`: The main Servlet of the Design Repository Search tool, which serves as the request handler for the tool as described above.
- `SearchTool_XML`: The XML file which stores the interface configuration information that defines which design repository objects can be searched for, and for those objects which of their attributes can be selected as search parameters.

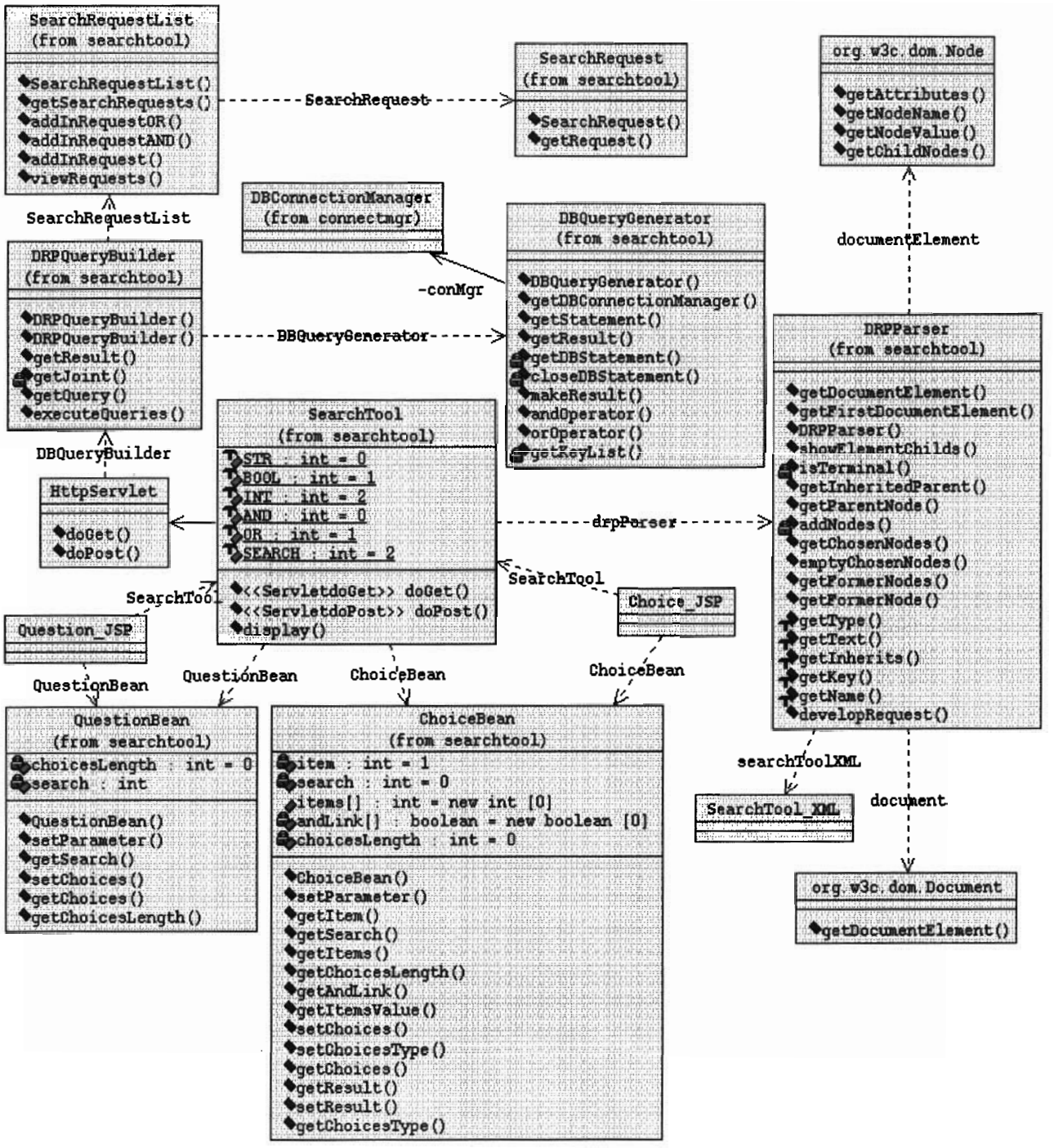

Figure 31. Implementation diagram for the Design Repository Search Tool

- `DRPParser`: An XML parser based on Sun Microsystems' JAXP (Java API for XML Processing), used to parse the `SearchTool_XML` file.
- `SearchRequest`: A class that is used to capture the attributes of a search criterion.

- `SearchRequestList`: Captures the characteristics of the search specified by the user, using a list of `SearchRequest` objects and information about how they are combined in Boolean fashion.
- `DRPQueryBuilder`: Converts the search specified by the user into a set of SQL queries that can be understood by the database.
- `DBQueryGenerator`: Obtains a database connection from the Database Connection Manager and executes the queries.
- `QuestionBean`, `Question_JSP`, `ChoiceBean` and `Choice_JSP`: The two JavaBeans are used to retrieve information that appears in the interface (such as the types of objects to be selected, and the information that characterizes the search that the user is specifying). The two corresponding JavaServer Pages are used to format this information for display at the client web browser, and sends that information to the client via the web browser.

4.3.3 Interface Configuration

One approach to defining the content of the Design Repository Search Tool interface (the types of objects that can be searched for, and the attributes of those objects that can be used as search parameters) would have been to allow the user to perform searches against any of the attributes of any of the data entities in the design repository database. This possibility was rejected however, for a variety of reasons. Some of the data entities that appeared in the database were not design repository objects. As an example, we did not wish to allow a user to search for and retrieve information about groups, user membership in groups, and user/group permissions.

Simply restricting searches to all attributes of all design repository objects was not an adequate solution either. There are many attributes of design repository objects that are an important part of a product representation, but that are not particularly interesting from the knowledge retrieval perspective. For example, it is reasonable to expect a user to search a design repository for artifacts that accomplish a particular function, but including objects and/or attributes in the interface that a user is unlikely to use as search criteria only serves to clutter up the interface needlessly. It was therefore decided to allow an administrator to configure the content of the Design Repository Search Tool interface. As mentioned previously, this configuration information is stored in a separate XML file rather than being embedded within the interface code so that the interface can be reconfigured without having to modify and recompile the code itself.

Within the XML file, each design repository object that can be searched for is specified by defining an XML element. Each element has four attributes:

- `Type`: *This attribute either has a value of "root" or is undefined.* If the value equals "root" the object appears in the list of design repository objects that can be searched for; if it is undefined, the object does not appear in the interface. This attribute allows the administrator to define configuration information for objects even if those objects are not currently intended to appear in the interface.
- `Key`: Specifies the name of the key of the table in the relational database that corresponds to the particular type of design repository object.
- `Inherits`: Some attributes of objects are defined as part of that object, while other attributes are "inherited" from abstract class definitions (see Appendix A). The `Inherits` attribute specifies the table (in the relational database) from which this design repository object inherits some of its attributes. This information is used by the `DRPQueryBuilder` Servlet.

- Text: This defines the text that appears in the first pull-down list from which the user selects an object type to search for. For example, as can be seen at the top of Figure 30, the text that appears in the pull-down list to specify a search for artifacts is "an artifact."

In addition to these attributes, an element has several sub-elements. Where the element corresponds to the design repository object that is being searched for, the sub-elements correspond to the attributes of that object type that should appear in the second pull-down list from which the user selects attributes to use as search parameters. For example, a user can search for artifacts based on their name, their type, their function, etc. Each of these attributes corresponds to a sub-element in the element associated with artifacts. Within the XML file each sub-element has three attributes

- Type: The `Type` attribute in the sub-element definition has a different meaning than at the element level. Here, the `Type` attribute specifies the data type of the information being searched for. In other words, if a design repository object has attributes and values, the `Type` indicates the data type of the value for the attribute. For instance, if a search is being done for an artifact based on its name, the `Type` indicates that what is being searched for is a "string." On the other hand, if a search is being done for an artifact based on its function, the type is "Function."
- Query: When a search is being done for standard data types (e.g., "string," or "number"), this attribute is not used. However, this is not always the case. In the example given in the previous paragraph, an artifact's name involves a standard data type (a string) but an artifact's function corresponds to a separate design repository object in the database. Searches that involve other objects in the database require more complex queries to be issued. In these cases, the `Query` attribute provides additional information used to construct the SQL query associated with the desired search.
- Text: This defines the text that appears in the second pull-down list from which the user selects an attribute to use as a search parameter (e.g., "whose name includes" and "whose description contains" seen in Figure 30).

4.4 Testing

This section summarizes the tests that were performed on the functionality of the Design Repository Search Tool. Note that the tests are described only at a high level. All of these tests have been conducted using Microsoft Internet Explorer 5 and Netscape Navigator 4.

The first set of tests described below involved testing of the Design Repository Search Tool interface. When testing was performed, the four types of design repository objects that could be searched for were artifacts, functions, flows and materials. Additional objects can be added by modifying the interface configuration XML file, in which case some of the tests listed below should be repeated for the new additions to verify functionality.

- Test: For each type of design repository object (artifact, function, flow and material), select the object from the first pull-down list in the Design Repository Search Tool interface and verify that the second pull-down list is updated to include the correct list of attributes that should correspond to searches for the selected object type.

  Result: No problems detected.

- Tests: Select an artifact as the object to be searched for. Then:

  1. Select the attribute `name` to verify that a text field appears (to enter a string),
  2. Select the attribute `created on` to verify that a text field appears (to enter a date),

3. Select the attribute `completed` to verify that a pair of radio buttons appears (to enter a Boolean value corresponding to whether or not an artifact is *completed*, meaning that it is considered final and is not still being modeled).

4. Select the attribute `input flow` to verify that a text field appears (to enter a string, though as described in the previous section, when the search is performed this string will be associated with another design repository object and not a regular string having standard data type),

Results: No problems detected.

- Test: Click the button to add an additional search criterion and verify that another criterion appears correctly (with the correct attributes for the specified object type showing in the new pull-down list).

  Result: No problems detected.

- Test: Click the button to delete a search criterion and verify that the last search criterion is deleted unless there is only one search criterion showing, in which case clicking the button should have no effect.

  Result: No problems detected.

- Test: Submit a query. Verify that the query is sent to the database, that any results returned are correctly formatted and displayed, and that an appropriate message is displayed if no matches are found for the specified search

  Result: No problems detected.

The second set of tests involved testing of the query-building capabilities of the Design Repository Search Tool to verify that the sets of SQL queries that were built to perform searches were, in fact, correctly constructed and that the SQL queries corresponded to the search criteria specified by the user. These tests were performed with two web servers (Sun Microsystems' Java Web Server 2.0 and Tomcat 3.2[8]) and Oracle 7.0 as the relational database management system. Although the SQL queries generated by the Design Repository Search Tool should be pure SQL (which is a platform-independent standard), additional tests should be performed using another relational database management system to verify full compatibility. As with the previous tests, some of the tests below should be repeated if the interface configuration is modified to add new design repository object types as search parameters.

- Test: Perform searches that test the following artifact attributes as search criteria: name, properties, description, created by, created on, complete, type, function, source, destination.

  Result: No problems detected.

- Test: Perform searches that test the following function attributes as search criteria: name, properties, description, created by, created on, complete, type, input flow, output flow.

  Result: No problems detected.

- Test: Perform searches that test the following flow attributes as search criteria: name, properties, description, created by, created on, complete, type, source, destination.

---

[8] Tomcat is a free, open-source implementation of Java Servlet and JavaServer Pages technologies developed under the Jakarta project at the Apache Software Foundation. See `http://java.sun.com/products/jsp/tomcat/faq.html` for more information.

Result: No problems detected.

- Test: Perform searches that test the following material attributes as search criteria: name, properties, description, created by, created on, complete, type.

  Result: No problems detected.

- Test: Perform a variety of complex searches (searches that combine multiple search criteria) to verify that query sets are correctly combined.

  Result: No problems detected.

5 USER MANAGEMENT SYSTEM

5.1 Functionality Summary

The user management system allows for the granting and management of access privileges for users. Upon attempting to access information in the design repository, users forwarded to a login page where they log in with a name and password. Users who are not registered with the system can also fill out a registration form to create an account with the system. Upon registering for the first time, a user is required to confirm the registration using a link or a confirmation code sent to the user via email, in order to verify that a valid and correct email address was used for the registration. A new user is automatically made a member of a group called Guest, which allows access to repositories

There are two levels of privileges:

- Reader: provides read access but does not allow information to be changed.
- Member: allows write/edit privileges in addition to read access.

Privileges are managed at a group level. Each design repository belongs to a group. An individual user may be a member of one or more groups, with the user's access to information in a given repository being determined by which of the two levels of privileges the user has in the group associated with that repository. A new user is automatically made a Member of a group called Guest, which allows access to example design repositories that are owned by the Guest group. Once logged in, a user can edit an information profile, request membership in other groups, manage one's own groups, etc.

Each group has a group administrator who manages group membership. The group administrator can approve requests from new members to join the group (after receiving an email message that is automatically generated by a user requesting access), and can assign or modify the access level for each member of that group.

5.2 Interface Specifications

5.2.1 Use Cases

The use case schema for the User Management System is shown in Figure 32. The functionality associated with this diagram was summarized in the previous section. It should be noted that only first-time users are required to undergo the registration and registration confirmation process. Users who have gone through this process once are registered with the system and can log in directly.

As was discussed in Section 3, Logged Users can perform a number of functions. The description in this section focuses only on functions associated with the User Management System, which have not been described previously. Because user management concepts and functions are generic and most

readers will be familiar with them, the description of functionality given in the previous section is presumed to be sufficient. Thus, unlike use case schemata in previous portions of this document, more in-depth descriptions of the activities represented by the nodes in Figure 32 are not provided here.

5.2.2 Activity Diagrams

The activity diagram for the User Management System is shown in Figure 33. The descriptions for the activities shown in the figure are as follows:

`Display login page`

Pre-condition: None.

Action: The user clicks on the Login link accessible from any Design Repository system interface page (see Figure 16).

Post-condition: The login page is displayed.

`Register`

Pre-condition: The user selects the register link on the login page.

Action: The user is directed to the registration page and follows the registration process.

Post-condition: The user is registered with the system.

`Display profile page`

Pre-condition: The user is not logged in.

Action: The user logs in.

Post-condition: The is now a Logged User and user's profile page is displayed.

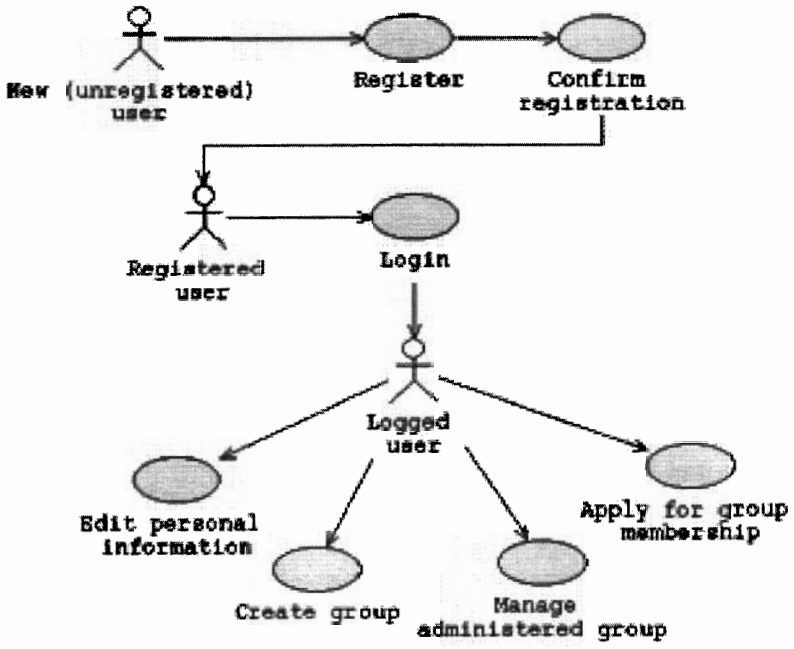

Figure 32. Use case schema for the User Management System

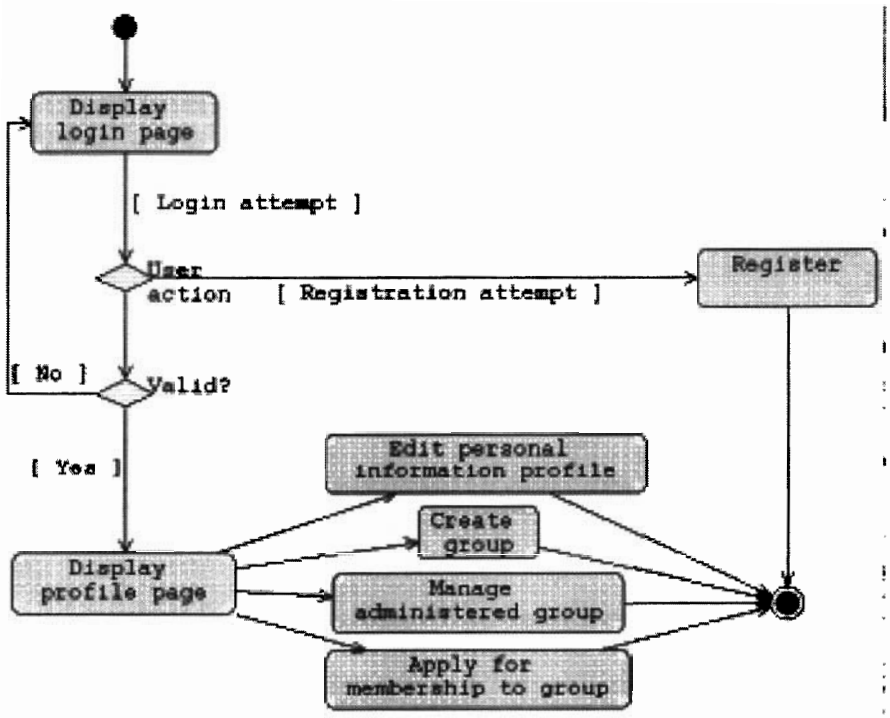

Figure 33. Activity diagram for the User Management System

Edit personal information profile

Pre-condition: The user is a Logged User and selects a link to edit personal information profile.

Action: The user edits the user's personal information profile and submits the changes.

Post-condition: The changes are saved and the user's personal information profile has been edited.

Create group

Pre-condition: The user is a Logged User and selects a link to create a new group.

Action: The user enters the name and a description for the new group and submits the request.

Post-condition: An email is sent to the User Management System administrator, who must approve the request before the new group is actually created.

Manage administered group

Pre-condition: The user is a Logged User and selects a link to manage groups that the user administers. (Managing groups includes: (1) managing privileges for existing group members, (2) approving/denying new requests for membership to the group, and (3) when desired, reassigning the role of group administrator to a different user.)

Action: The user enters the name for the new group and submits the request.

Post-condition: Various, depending on management activity.

Apply for group membership

Pre-condition: The user is a Logged User and selects a link to apply for membership to a group administered by another user.

Action: A list of available groups and their descriptions is displayed. The user can use a pulldown menu to select the kind of privileges (Reader or Member) that are being requested for a given group, and submits the request.

Post-condition: An email is sent to the administrator for that group, who can then approve or deny the request during the administrators own group management activity session.

Figure 34 shows a detailed activity diagram for the *Register* activity that appears in Figure 33. As before, because user management functions are not conceptually complex precise activity descriptions (pre-conditions/actions/post-conditions) for the activities in Figure 34 have been omitted. Silarly, although detailed activity diagram for the *Register* activity is provided for illustrative purposes, a complete set of detailed activity diagrams for the other activities that appear in Figure 33 is not provided.

5.3 Implementation

5.3.1 Interface Descriptions

Figure 35 shows a screen capture of the user interface displaying the login screen for the user management system. This screen is accessible by clicking on the *Sign in* link in the user frame (the horizontal bar below the top frame) that is present from any Design Repository System interface screen. In addition, as has been described previously, when a user who is not logged in attempts to access information about repository objects via the tree menu frame at the left, that user will automatically be directed to the login screen for authentication.

Users who are new to the system can click on the *Register* link to create a new account. The registration screen is shown in Figure 36. The registration process solicits various types of information from

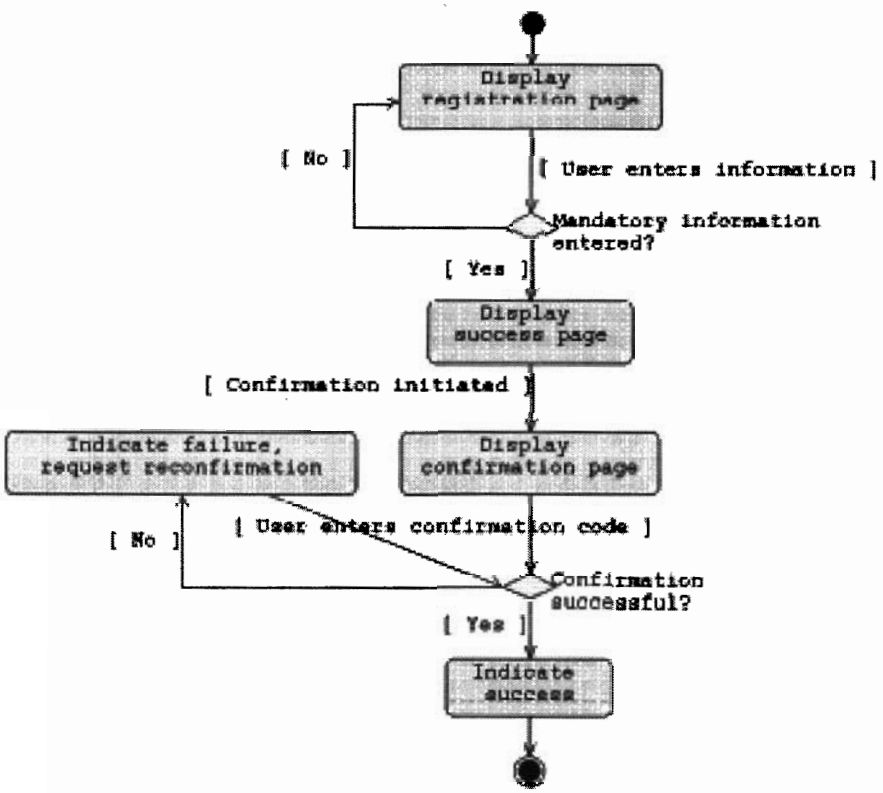

Figure 34. Detailed activity diagram for the activity *Register*

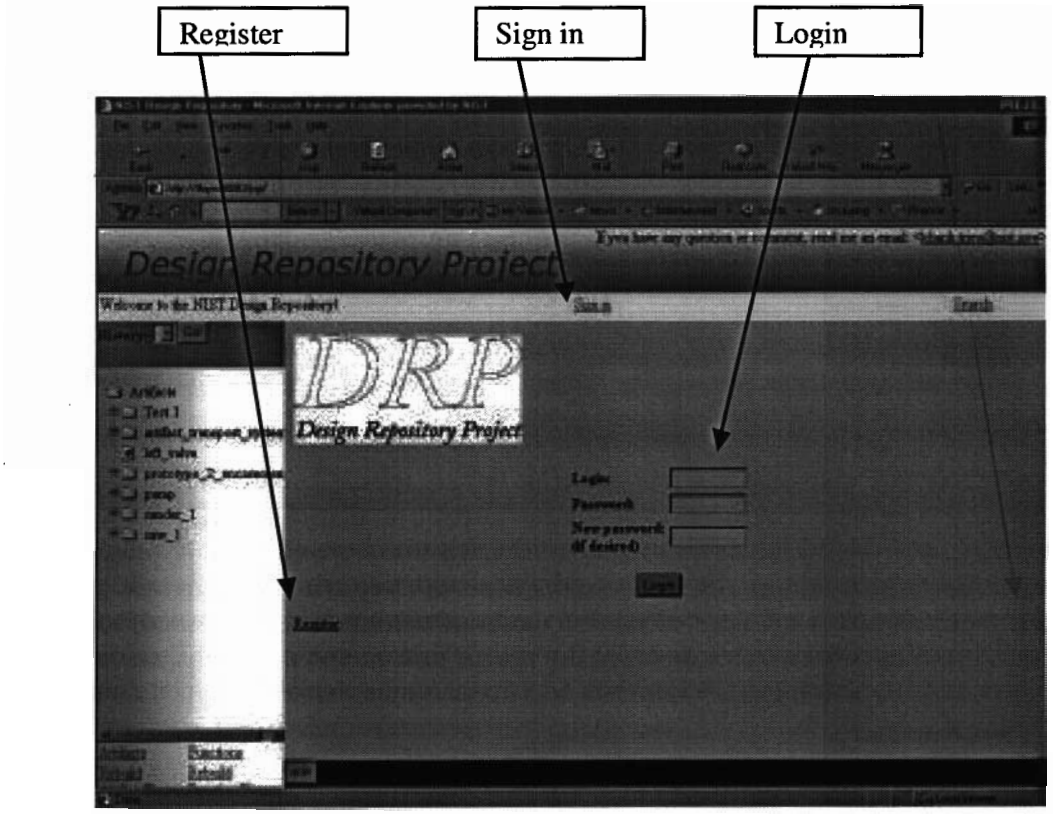

Figure 35. Login screen

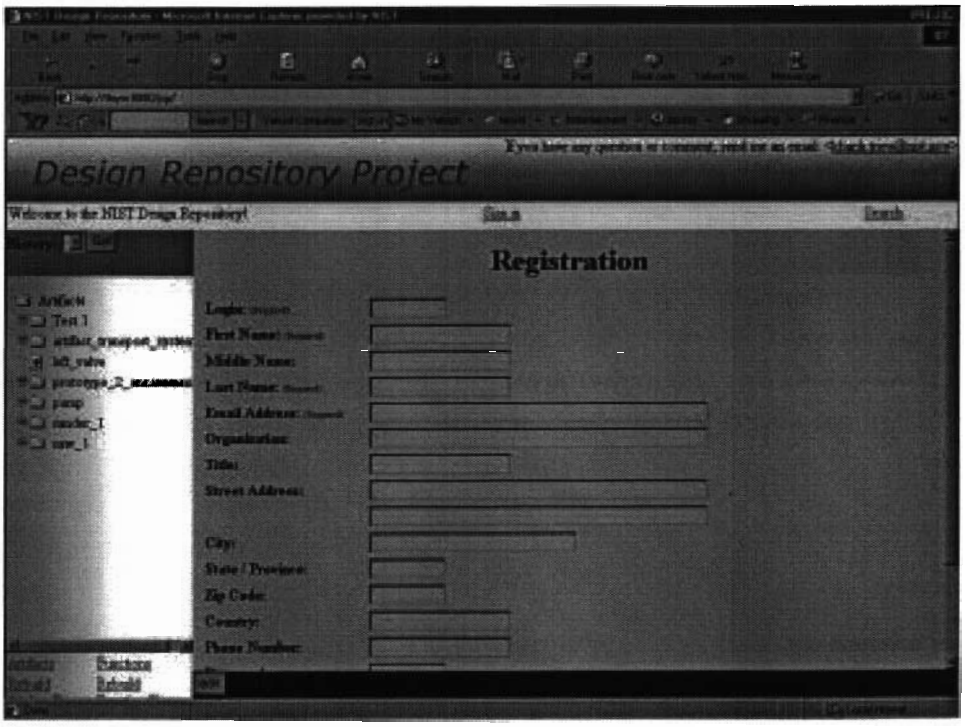

Figure 36. Registration screen

the user, some of which is mandatory and some of which is optional. When the form is submitted, JavaScript code embedded in the page verifies on the client side that the mandatory information is present. If it is not, the user is prompted to provide the missing information. If it is, the system on the server side verifies that the login name is not already in use. If a duplicate login name is given, the user is notified and is prompted to select a new login name.

If there is no problem with the login name, the user is directed to a page that prompts the user for a confirmation code that has been sent via email. This is done as a verification mechanism to ensure that the system has a valid contact for the user (i.e., the user did not accidentally mistype an email address). This process also serves as an authentication mechanism to prevent people from maliciously registering using somebody else's identity, since a user attempting to do so would presumably not have access to somebody else's email to obtain the confirmation code needed to complete the registration process. Once the user enters the correct confirmation code, the user's account is activated and the user may log in to gain access to the system.

Upon logging in, a user is directed to a profile page (see Figure 37). This page provides a summary of the user's profile and displays a list of groups that the user belongs to. From this page, a user can edit his or her information profile, apply for membership to another user group, attempt to create a new group, or view another user's profile. Clicking on the name of a group (shown below the profile table in the figure) directs a user to a group information page.

From that page, users can read a description of the group and view a list of the users in that group. In addition, the user who is the administrator for that group administrator can also modify privileges for existing users, approve or deny requests from new users to join the group, and can also reassign the role of group administrator to another user. Creation of new groups must be approved by a User Management System administrator. The profile page shown in Figure 37 is that of a user who has User Management System administrator privileges, as is evidenced by the *Confirm group creation* link.

Because requests to join a group or to create a new group are approved by users other than the one making the request, an email is automatically sent to the appropriate administrator (group administrator or User Management System administrator) when such a request is made to alert the administrator that a request is pending. When a request is approved or denied, an email message is automatically sent to the user who made the request to notify them of the outcome.

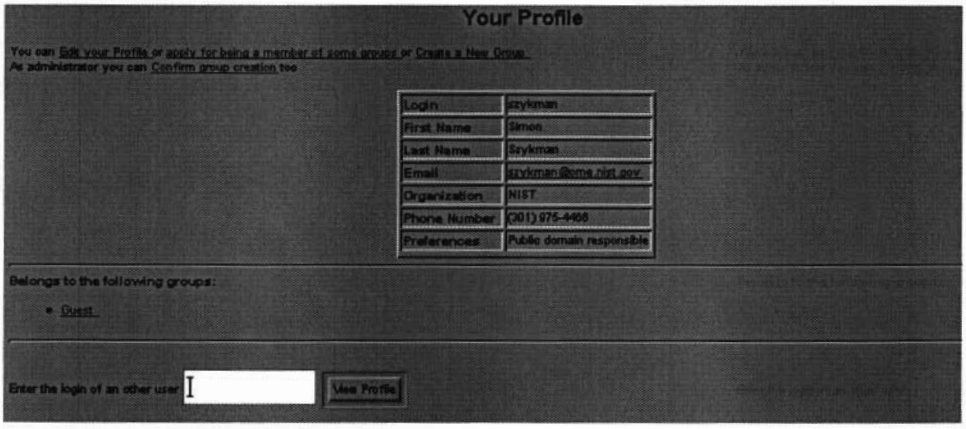

Figure 37. Administrator profile screen

5.3.2 Core Application Logic

The User Management System is built using the same as the other parts of the Design Repository System. The only difference is that because the User Management System handles its own requests. Because the Browser, Editor and Search Tool interfaces all deal with the same kinds of data (product/design data, see Appendix A) and share common kinds of data requests, the handling of requests for this information are centralized in a separate request handler which can serve these various components. Since the User Management System deals with a different kind of data (user data, see Appendix B), however, the servlet that provides user management functions handles its own requests rather than the systems main Request Handler (as was shown schematically in Figure 1). These requests may come from the user (login/logout or permissions management requests), as well as from the other centralized request handler which verifies that user have the necessary privileges before allowing them to view and/or edit data.

Although the User Management System does not use the main system Request Handler, connections used to communicate with the database are managed using the main system Connection Manager. All of the various queries used to communicate with the database are stored in a file called `UserQueries.java` class in order to be easily changed when necessary. The following methods are associated with the User Management System:

- CheckLoginNotUsed: Verifies that a login name selected when a user is attempting to register has not already been chosen by an existing user, or by a user whose registration is pending (i.e., one who has registered but whose registration is not yet confirmed).
- `GetUserInfo`: Retrieves information about an existing user (see Appendix B for details of this information).
- `GetTemporaryPerson`: Retrieves information about a user who has initiated the registration process but whose registration has not yet been confirmed.
- `DeleteTempPerson`: Removes a temporary user from the database once the registration is confirmed (since that user is no longer considered temporary), or if a temporary entry is considered abandoned (an unconfirmed registration that remains unconfirmed for a certain period of time).
- `UpdateUserPassword`: Changes a user's password.
- `UpdateUserPasswordReminder`: Changes the user-selected reminder string that is used to remind a user of a forgotten password.
- `InsertMember`: Adds a user to a group by adding a link between a specified user and a specified group.
- `DeleteMember`: Removes a user from a group by deleting the link between a specified user and a specified group.
- `DeleteMbr`: Removes a user, a group, a temporary user, or a temporary group from the DRP_Member table.
- `UpdateStatusBelongTo`: Used by a group administrator, assigns (or reassigns, in the case when a user's status is changed) the privileges (Reader or Member) that a user has within a group.
- `GetMemberInGroup`: Retrieves information about a specific member of a group.
- `GetGroupResponsible`: Retrieves information about the administrator of a given group.

- `GetGroupId`: Retrieves information about a group (see Appendix B for details of this information).
- `GetGroup`: Retrieves information about all existing groups and their administrators.
- `GetUserGroup`: Retrieves descriptions of the group(s) that a given user belongs to as a Member and/or a Reader
- `GetGroupUser`: Retrieves descriptions of the group(s) for which a given user is the group administrator.
- `UpdateGroupDescription`: Used by a group administrator, changes the text description of a group.
- `InsertTmpGroup`: Creates a temporary group when a user attempts to create a new group. The actual group is not created until the attempt is approved by the User Management System administrator.
- `GetGroupCreationApplies`: Used by the User Management System administrator, retrieves a list of the groups that users have attempted to create, which are still pending approval.
- `GetTemporaryGroupInfo`: Used by the User Management System administrator, retrieves information about a specific group that a user has attempted to create.
- `InsertGroup`: Creates a new group with a specified name, description and group administrator once the User Management System administrator approves a user's attempt to create a new group.
- `DeleteTemporaryGroup`: Deletes a temporary group to remove it from the list of attempted group creations that are still pending approval.
- `UpdateGrpResponsible`: Used by a group administrator, attempts to reassign the group administrator role to another user.
- `GetGrpConfirmCode`: Retrieves the group confirmation code used to confirm the reassignment of group administrator when this responsibility is transferred from one user to another.
- `UpdateGrpConfirmCode`: Changes the group confirmation code.

5.3.3 System Setup

Before a Design Repository System can be used for the first time, the User Management System must be set up by making certain initializations in the database. Specifically, a set of scripts have been created to create the default group that all registered users belong to (called *Guest*), and to assign a group administrator for this group, who is also the User Management System administrator. A user can only view (or edit) information associated with a group for which the user is a Reader (or Member). If a user who is adding new information to a design repository does not specify a particular group to which access should be restricted, that information belongs to the group Guest by default.

To initialized the database, the default group and initial administrator need to be created. To do so, a DRP_Person is created with MemberNum 1. For example a user may be created with the following information (see Appendix B for details regarding the information associated with the various fields):

(1, 'smith', 'John' ,Q, 'Public', 'JPublic@nist.gov', 'NIST', null, null, null, null, null, null, '(555) 555-1212', 'smith', null, 'User Management System administrator')

Note that several portions of the user's personal profile could have been filled in here, but were instead left *null* so that the user could edit the user's profile later. Similarly, the Guest group is created by cre-

ating a DRP_Group with MemberNum 2 and the additional information (again see Appendix B for details) as follows:

(2, 'Guest', 'Public domain group to which all new users belonged by default', 1, null);

Strictly speaking, the MemberNums of the User Management System administrator and the Guest group do not have to be 1 and 2. However, for the system to function properly, these numbers must match the ID numbers for the guestResponsibleID and guestGroupID constants which are specified in DRPUtil.java, which is a file used to maintain a number of constants used by various parts of the Design Repository System. In addition to those two constants being set to match the associated MemberNums as described, the following strings need to be set to the following values in the DRPUtil.java file: allAccess = "Member", readOnlyAccess = "Reader", noAccess = "Refused", noAccessAnymore = "Fired", accessApplyRemove = "Removed".

5.4 Testing

This section provides a high-level summary of the the tests that were performed on the functionality of the User Management System.

- Test: Registration attempted with all the mandatory parameters given.

  Result: Expected result returned (proceeds to request for confirmation code).

- Test: Registration attempted with some mandatory parameters missing.

  Result: Expected result returned (error message, missing parameters requested).

- Test: Registration attempted with some mandatory parameters missing and with JavaScript disabled. Because the verification of mandatory information is done using JavaScript, this test was done to ensure that the system would not allow a registration to proceed when mandatory information was given if JavaScript was turned off in the web browser.

  Result: Expected result returned (unable to assess presence of mandatory information without JavaScript, but returns to the same page and does not proceed to the confirmation request, which is the next step in the registration process).

- Test: Confirmation attempted with the correct confirmation code.

  Result: Expected result returned (registration is accepted and confirmed, user can now edit user's the information profile or proceed to browsing the design repository).

- Test: Confirmation attempted with an incorrect confirmation code.

  Result: Expected result returned (Error message, user prompted for correct confirmation code).

- Test: Login attempted with a valid username and password.

  Result: Expected result returned (user moves on to the user profile page).

- Test: Login attempted with something other than both a valid username and password.

  Result: Expected result returned (error message, password reminder string is displayed and user is prompted to try logging in again).

- Test: Login attempted with a valid username and password, and a new password entered in the New Password field.

Result: Expected result returned (user moves on to the user profile page and the old password is updated with the new one).

- Test: Various portions of a user's information profile changed.

  Result: Expected result returned (information fields are updated with new information).

- Test: New group creation attempted with an unused name, and a group description.

  Result: Expected result returned (user returned to the user profile page which shows the new group, and shows the user as the group administrator).

- Test: New group creation attempted with a name that is already in use by another user or group.

  Result: Expected result returned (error message displayed, indicating that the selected name is already in use).

- Test: View another user's information profile.

  Result: Expected result returned (other user's profile displayed).

- Test: Verification that only the group administrator has a link to the page that allows group administration functions.

  Result: Verified.

- Test: Group administration functions tested: approval of user attempting to join group, assignment of privileges for a new group member, modification of privileges for an existing group member.

  Result: Expected result returned.

- Test: Verification that only the system administrator has a link to the page that allows acceptance of new group creation attempts.

  Result: Verified.

- Test: System administration function tested: acceptance of new group creation attempts.

  Result: Expected result returned.

ACKNOWLEDGMENTS

The NIST Design Repository Project was initiated by Ram D. Sriram of NIST in 1996. Since 1997, the project has been led by Simon Szykman of NIST, who oversaw subsequent development of the project.

The initial Design Repository implementation was done during the summer of 1996 by J. William Murdock (then a summer student from Georgia Institute of Technology, currently a NRC Postdoctoral Research Associate at the Naval Research Laboratory). The second implementation was developed primarily by Christophe Bochenek and Janusz Racz. The third Design Repository Project prototype discussed in this report was developed by Jocelyn Senfaute, Khanh Trieu, Guilhem Assant, Sylvain Archiambault and Jean-François Heintz. The latter group of Guest Researchers authored major portions of the documentation that has been compiled in this document.

The example product models that currently reside in the design repositories were developed primarily by J. William Murdock, Simon Szykman, Matthew Tundermann, Robert J. Fijol and Robert H. Allen. Additional tools that were utilized in this project, but were not integrated into the main system prototypes, were developed by Janusz Racz, Lisa Anthony and Erik Lightman. Steven J. Fenves provided invaluable effort in developing the Core Product Model, the knowledge representation that underlies the knowledge stored in the design repositories.

From its inception to the present, the project has been funded from a variety of sources, including NIST Scientific and Technical Research and Services (STRS) funding, the NIST Advanced Technology Program (ATP), and the NIST Systems Integration for Manufacturing Applications (SIMA) program.

REFERENCES

1. Gorti, S. R., A. Gupta, G. J. Kim, R. D. Sriram and A. Wong (1998), "An Object-Oriented Representation for Product and Design Process", *Computer-Aided Design*, Vol. 30, No. 7, pp. 489-501.
2. Murdock, J. W., S. Szykman and R. D. Sriram (1997), "An Information Modeling Framework to Support Design Databases and Repositories," *Proceedings of the 1997 ASME Design Engineering Technical Conferences (Design for Manufacturing Conference)*, Paper No. DETC97/DFM-4373, Sacramento, CA, September.
3. Szykman, S., R. D. Sriram, and S. J. Smith (Eds.) (1998), *Proceedings of the NIST Design Repository Workshop*, NISTIR 6159, National Institute of Standards and Technology, Gaithersburg, MD, November, 1996.
4. Szykman, S., R. D. Sriram, C. Bochenek and J. W. Racz (1999), "The NIST Design Repository Project," *Advances in Soft Computing – Engineering Design and Manufacturing*, Roy, R., T. Furuhashi, and P. K. Chawdhry (Eds.), Springer-Verlag, London, pp. 5-19.
5. Szykman, S., C. Bochenek, J. W. Racz, J. Senfaute and R. D. Sriram (2000), "Design Repositories: Next-Generation Engineering Design Databases," *IEEE Intelligent Systems*, Vol. 15, No. 3, pp. 48-55.
6. Szykman, S., J. W. Racz, C. Bochenek and R. D. Sriram (2000), "A Web-based System for Design Artifact Modeling," *Design Studies*, Vol. 21, No. 2, pp. 145-165.
7. Szykman, S., J. W. Racz and R. D. Sriram (1999), "The Representation of Function in Computer-based Design," *Proceedings of the 1999 ASME Design Engineering Technical Conferences (11th International Conference on Design Theory and Methodology)*, Paper No. DETC99/DTM-8742, Las Vegas, NV, September.
8. Szykman, S., J. Senfaute and R. D. Sriram (1999), "Using XML to Describe Function and Taxonomies in Computer-based Design," *Proceedings of the 1999 ASME Design Engineering Technical Conferences (19th Computers in Engineering Conference)*, Paper No. DETC99/CIE-9025, Las Vegas, NV, September.
9. Hirtz, J., R. B. Stone, D. A. McAdams, S. Szykman, and K. L. Wood (2001), "Evolving a Functional Basis for Engineering Design," *2001 ASME Design Engineering Technical Conferences (13th International Conference on Design Theory and Methodology)*, Paper No. DETC01/DTM-21688, Pittsburgh, PA, September.

10. Szykman, S., S. J. Fenves, S. B. Shooter and W. Keirouz (2001), "A Foundation for Interoperability in Next-generation Product Development Systems," *Computer-Aided Design*, Vol. 33, No. 7, pp. 545-559.

11. Fenves, S. J. (2001), *A Core Product Model for Representing Design Information*, NISTIR 6736, National Institute of Standards and Technology, Gaithersburg, MD, April.

12. Allen, R. H., R. J. Fijol, S. Szykman and R. D. Sriram (2000), "Representing an Artifact Transport System in a Design Repository: A Case Study," *Proceedings of the 2000 ASME Design Engineering Technical Conferences (20th Computers and Information in Engineering Conference)*, Paper No. DETC2000/CIE-14608, Baltimore, MD, September.

13. Allen, R. H., R. J. Fijol, S. Szykman and R. D. Sriram (2001), "Representing the Charters of Freedom Encasements in Design Repository: A Case Study," *2001 ASME Design Engineering Technical Conferences (21st Computers and Information in Engineering Conference)*, Paper No. DETC01/CIE-21292, Pittsburgh, PA, September.

14. Koch, G. and K. Loney (1995), *Oracle: The Complete Reference (Third Edition)*, Oracle Press/Osborne McGraw Hill, Berkeley, CA.

APPENDIX A: THE CORE PRODUCT MODEL

Notes:

- An abstract class is a class for which instances cannot be created, but that exists for the convenience of grouping attributes that are common to all of its subclasses so that these attributes can be inherited by the subclasses.
- As described in Section 10.2, the type attribute for objects and relationships is a string that is required to be one of the terms within a taxonomy associated with that kind of entity. Abstract classes do not have types since instances for abstract classes do not exist.
- In addition to the entity definitions shown below, a separate set of entities also exists for the organization of terms into taxonomies used for entity type classification. As the focus of this paper is not on taxonomies and terminological issues, these entity definitions are not presented here.
- "`[x]`" represents a pointer to an entity belonging to the class `x`.
- "`{string}`" represents a list of strings.
- "`{[x]}`" represents a list of pointers to entities belonging to the class `x`.
- "`(I)`" indicates that an attribute value and any constraints on that value are inherited from an abstract class. For example, an artifact has an attribute called `name` that is inherited from the abstract class `DRP_Object`. Because the name of any `DRP_Object` is required to be unique and not null, the name of any artifact is as well.
- "`#`" indicates that the rest of the line is a comment.
- "`# (UNIQUE)`" is a comment indicating that a string must have a unique value.
- "`# (NOT NULL)`" is a comment indicating that the field is required.

```
Abstract Class DRP_Object
{
 name                          string           # (UNIQUE, NOT NULL)
 information                   [Information]    # (NOT NULL)
 references                    {[Reference]}
 is_referenced_by              {[Reference]}
 is_member_of                  {[Set_Relationship]}
 is_special_member_of          {[Directed_Set_Relationship]}
}

Abstract Class Restricted_DRP_Object  # (inherits from DRP_Object)
{
 name                          (I)
 information                   (I)
 constrained_by                {[Constraint]}
 references                    (I)
 is_referenced_by              (I)
 is_member_of                  (I)
 is_special_member_of          (I)
 required_by                   {[Requirement]}
}

Class Artifact  # (inherits from DRP_Object)
{
 name                          (I)
 information                   (I)
 references                    (I)
 is_referenced_by              (I)
 is_member_of                  (I)
 is_special_member_of          (I)
 type                          [Artifact_Family]    # (NOT NULL)
 is_specified_by               {[Specification]}
 config_info                   [Config_Info]        # (NOT NULL)
 function                      {[Function]}         # (NOT NULL)
 form                          [Form]               # (NOT NULL)
 behavior                      {[Behavior]}
 subartifacts                  {[Artifact]}
 subartifact_of                {[Artifact]}
 is_source_of                  {[Flow]}
 is_destination_of             {[Flow]}
}

Class Function  # (inherits from Restricted_DRP_Object)
{
 name                          (I)
 information                   (I)
 constrained_by                (I)
 references                    (I)
 is_referenced_by              (I)
 is_member_of                  (I)
 is_special_member_of          (I)
 required_by                   (I)
 type                          [Function_Family]    # (NOT NULL)
 subfunctions                  {[Function]}
 subfunction_of                [Function]
 function_of_artifact          [Artifact]           # (NOT NULL)
}
```

```
Class Transfer_Function  # (inherits from Function)
{
 name                          (I)
 information                   (I)
 constrained_by                (I)
 references                    (I)
 is_referenced_by              (I)
 is_member_of                  (I)
 is_special_member_of          (I)
 required_by                   (I)
 type                          (I)
 subfunctions                  (I)
 subfunction_of                (I)
 function_of_artifact          (I)
 input_flow                    {[Flow]}
 output_flow                   {[Flow]}
}

Class Flow  # (inherits from Restricted_DRP_Object)
{
 name                          (I)
 information                   (I)
 constrained_by                (I)
 references                    (I)
 is_referenced_by              (I)
 is_member_of                  (I)
 is_special_member_of          (I)
 required_by                   (I)
 type                          [Flow_Family]        # (NOT NULL)
 source                        {[Artifact]}
 destination                   {[Artifact]}
 has_external_source           Boolean              # (NOT NULL, Default: FALSE)
 has_external_destination      Boolean              # (NOT NULL, Default: FALSE)
 is_input_of                   {[Transfer_Function]}
 is_output_of                  {[Transfer_Function]}
}

Class Form  # (inherits from Restricted_DRP_Object)
{
 name                          (I)
 information                   (I)
 constrained_by                (I)
 references                    (I)
 is_referenced_by              (I)
 is_member_of                  (I)
 is_special_member_of          (I)
 required_by                   (I)
 type                          [Form_Family]        # (NOT NULL)
 subforms                      {[Form]}
 subform_of                    [Form]
 geometry                      [Geometry]           # (NOT NULL)
 material                      [Material]           # (NOT NULL)
 form_of_artifact              [Artifact]           # (NOT NULL)
}
```

```
Class Geometry  # (inherits from Restricted_DRP_Object)
{
 name                          (I)
 information                   (I)
 constrained_by                (I)
 references                    (I)
 is_referenced_by              (I)
 is_member_of                  (I)
 is_special_member_of          (I)
 required_by                   (I)
 type                          [Geometry_Family]    # (NOT NULL)
 subgeometries                 {[Geometry]}
 subgeometry_of                [Geometry]
 geometry_of_form              [Form]               # (NOT NULL)
}
Class Material  # (inherits from Restricted_DRP_Object)
{
 name                          (I)
 information                   (I)
 constrained_by                (I)
 references                    (I)
 is_referenced_by              (I)
 is_member_of                  (I)
 is_special_member_of          (I)
 required_by                   (I)
 type                          [Material_Family]    # (NOT NULL)
 submaterials                  {[Material]}
 submaterial_of                [Material]
 material_of_form              [Form]               # (NOT NULL)
}
Class Behavior  # (inherits from DRP_Object)
{
 name                          (I)
 information                   (I)
 references                    (I)
 is_referenced_by              (I)
 is_member_of                  (I)
 is_special_member_of          (I)
 type                          [Behavior_Family]    # (NOT NULL)
 subbehaviors                  {[Behavior]}
 subbehavior_of                [Behavior]
 behavior_of_artifact          [Artifact]           # (NOT NULL)
}
Class Specification  # (inherits from DRP_Object)
{
 name                          (I)
 information                   (I)
 references                    (I)
 is_referenced_by              (I)
 is_member_of                  (I)
 is_special_member_of          (I)
 type                          String               # (NOT NULL)
 requirements                  {[Requirement]}      # (NOT NULL)
 specification_of_artifact     [Artifact]           # (NOT NULL)
}
```

```
Class Directed_Set_Relationship  # (inherits from Set_Relationship)
{
 name                          (I)
 information                   (I)
 members                       (I)
 type                          String              # (NOT NULL)
 special_members               {[DRP_Object]}      # (NOT NULL)
 special_member_role           String
 member_role                   String
}

Class Constraint  # (inherits from DRP_Relationship)
{
 name                          (I)
 information                   (I)
 type                          String                          # (NOT NULL)
 constrains                    {[Restricted_DRP_Object]}       # (NOT NULL)
}
```

APPENDIX B: USER MANAGEMENT DATA

```
Class DRP_Person
{
 PersonNum    # An unique ID identifying  a user, also listed in the MemberNum field  of a
               DRP_Member
 Login
 First_Name
 Middle_Name
 Last_Name
 Email
 Organization
 Title
 Street_Address
 City
 State
 Zip
 Country
 Phone_Number
 Password
 Password_Reminder      # A string to remind a user of a password
 Preferences
}

class DRP_Group
{
 GroupNum  # An unique ID identifying the user, listed in the DRP_Member as a MemberNum
             field.
 Name
 Description
 Responsible  # The PersonNum of the person who is the administrator for this group
}
```

```
class DRP_Member
{
 MemberNum    # An unique ID identifying a user or group
}

class Temporary_Person   # Nearly the same as DRP_Person;  used to maintain information
                           about users who have started the registration process with the
                           system but who are not yet approved/confirmed users
{
 TempPersonNum
 Login
 First_Name
 Middle_Name
 Last_Name
 Email
 Organization
 Title
 Street_Address
 City
 State
 Zip
 Country
 Phone_Number
 Password
 Password_Reminder
 Confirmation_Code       # A randomly generated code used to confirm a user registration and
                           to ensure that a valid email address is given
 Creation_Date           # The system date of a registration, to allow registration entries
                           to be automatically removed if they are not confirmed within a
                           certain period of time
}

class Temporary_Group    # Nearly the same as DRP_Group;  used to maintain information
                           about groups that users have attempted to create but that haven't
                           yet been approved by the User Management System administrator
{
 TempGroupNum            # An unique ID identifying the group, listed as MemberNum in the
                           DRP_Member
 Name
 Description
 Responsible             # The PersonNum associated  with the user who is the group
                           administrator
 Confirmation_Code       # A randomly generated code used to confirm a group creation and
                           to verify the identity of the user creating the group
}

class Belongs_To         # Defines which members belong to which groups, and what their
                           privileges are in each groups (Reader or Member)
{
 Ref_mbr                 # An unique ID identifying the user, listed as MemberNum in the
                           DRP_Member
 Ref_gr                  # An unique ID identifying the group, listed as MemberNum in the
                           DRP_Member
 Status                  # A string (either 'Reader' or 'Member') that defines the specified
                           user's  privileges in the specified group
}
```

```
class Temp_Belongs_To       # Contains the same information as Belongs_To, but used to hold
                              that information when a user initially applies to join a group,
                              before the request has been approved
{
 Ref_mbr
 Ref_gr
 Status
}
```