SERVICE ORIENTED WEB APPLICATIONS:

GLYCOVAULT AS A CASE STUDY

by

SRIKALYAN C. SWAYAMPAKULA

(Under the Direction of John A. Miller)

ABSTRACT

There are many Web application frameworks emerging these days. Most of these frameworks are driven by MVC (Model View Controller) architecture. Web browsers have become the common client for most of these frameworks. Often, there is no other way to access data other than using the end client. With the increasing popularity of SOA, there is a strong need for building Web applications in such a way that the data can be accessed in flexible ways other than using a Web browser. Using traditional approaches, we end up writing the entire application from scratch to support newer ways to access the data. The focus of this thesis is to discuss the emerging framework called SOFEA (Service Oriented Front-End Architecture). This thesis discusses how to automatically generate the Persistent Object Layer, Service Layer etc., from SQL schema. This thesis also discusses a Web application called GlycoVault developed using SOFEA.

INDEX WORDS: Web services, SOAP, Ontologies, OWL, SOFEA, and GlycoVault.

SERVICE ORIENTED WEB APPLICATIONS:

GLYCOVAULT AS A CASE STUDY

by

SRIKALYAN C. SWAYAMPAKULA

B. Tech, Jawaharlal Nehru Technological University, India, 2007

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2010

SERVICE ORIENTED WEB APPLICATIONS:

GLYCOVAULT AS A CASE STUDY

by

SRIKALYAN C. SWAYAMPAKULA

Major Professor:	John A. Miller
Committee:	Krzysztof J. Kochut
	William S. York

Electronic Version Approved:

Maureen Grasso

Dean of the Graduate School

The University of Georgia

May 2010

DEDICATION

To my Mom, Dad, Uncles and Aunts.

ACKNOWLEDGEMENTS

I would like to thank Dr. John A. Miller, my major advisor, for all the guidance and support given to me in the course of this thesis. I would also like to thank Dr. Krzysztof J. Kochut and Dr. William S. York for their guidance and support. I would like to thank my colleague Matthew Eavenson for his help. Lastly, I would like to thank my family members and friends who were always there whenever I need them.

TABLE OF CONTENTS

Page

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

There are many Web application frameworks emerging these days. Most of these frameworks are driven by Model View Control (MVC) [1] architecture pattern and are quite mature. The main problem with these frameworks is that they assume Web browser as their end client, so there is no other way to access the data other than using a Web browser. The main intension of any Web application is to access the data using a user friendly interface. With the increasing popularity of the Service Oriented Architecture (SOA) [2], there is a strong need for developing Web applications in such a way that they can support different ways of data access. Implementing multiple ways for accessing data results in writing the entire application from scratch. This is not a good idea as maintainability becomes a huge issue. So, there is need to develop an API which is language independent, platform/client independent. Service oriented Front-End Architecture (SOFEA) [3] defines a good approach for building the Web applications. The details about SOFEA are discussed as a separate chapter (SOFEA is sometimes referred to as SOUI).

We have developed a Web application based on the SOFEA architecture called GlycoVault. GlycoVault is a Web application which acts as a repository to store the data from scientific experiments. The main objective behind GlycoVault is to share the experimental data. We have developed a user friendly frontend to access this data. In case a research group prefers not to use our frontend, but is interested in accessing the experimental data, they can invoke the Web services upon which our frontend is built. This is one advantage of SOFEA. Most of the experimental data in our case are either in the form of spreadsheets or XML documents. We can

extract the data from XML documents as these documents have a standard consistent structure but the spreadsheets do not have a standard structure. Some of these spreadsheets are automatically generated from the instruments which perform the experiments and produce the results in the form of spreadsheets. We need to extract the information from the important cells of these spreadsheets to populate our database. In order to automate this process, we have defined workflows to extract the data from the spreadsheets and populate the GlycoVault database.

Building any application involves defining/creating several different layers like Persistent Object Layer, Service Layer etc., specific to an application. Creating these layers is quite a tedious task. This becomes especially annoying when changes are made periodically to the design. Most of the Web applications perform four major common steps, i.e., save the data, update the data, delete the data and select/fetch the data (update and delete operations are invoked significantly less often than read or save operations). Most of these Web applications have databases as their backend and data are organized in the form of tables and columns in these databases. The major purpose of the above layers is to read/write the data from/to tables and columns. This is very true even for the GlycoVault. Changes to the design of the GlycoVault are periodically made. GlycoVault also perform the same four operations. So, we have defined an approach using which we can automatically generate the persistent object layer, API and its implementation sub layer , factory sub layer and service layer code. Also, we have added security features in this automatic code generation process to prevent any unauthorized access to the data or any unauthorized writes/updates to the data. We are generating the code from a SQL (Structure Query Language) schema using techniques from reverse engineering. We choose the SQL schema instead of Unified Modeling Language (UML) to generate the code because many

legacy systems do not have a UML design, but they have solid SQL schema. We are interested in supporting every Web application system. Also, UML to SQL Schema generation often requires the purchase of an enterprise version of a UML tool.

The thesis is organized as follows. Chapter 2 discusses background information related to this thesis. Chapter 3 discusses SOFEA. Chapter 4 focuses on the implementation of GlycoVault. Chapter 5 explains the code generation process from the SQL schema. Chapter 6 is presents conclusions and future work.

CHAPTER 2

BACKGROUND

2.1 Web Application Frameworks

Web application frameworks are designed to support the development of Web based applications. Web application frameworks help developers by providing quick and easy solutions to build, maintain and debug Web applications. Most Web application frameworks follow the MVC architecture pattern, which separates the Model, View and Controller modules that helps in dividing the work among different groups like Web designers, business logic developers, etc. Based on directionality of the data access, the MVC architecture patterns are broadly classified into two types 1) Push-based MVC and 2) Pull-based MVC. In Push-based MVC, actions perform the required data processing and push the data to the view layer to display the result. Examples of Push-based MVC are Spring MVC and Ruby on Rails. In Pull-based MVC, the view layer pulls the data from different actions and displays the result. Examples of Pull-based MVC are Struts 2 and Wicket. MVC has two major versions MVC1 and MVC2. In MVC1, the controller is decentralized as the currently displayed page determines the next page to be displayed. In MVC2, the controller centralizes the logic and forwards the requests to correct views based on the request URL, parameters and state of an application. Based on the type of controller, [4] has classified the Web application frameworks into two types 1) Server-centric Web frameworks and 2) Client-centric Web frameworks. In Server-centric Web frameworks, the controller responsible for rendering the data onto the client Web browser resides on the server-side. Most of the existing frameworks are Server-centric Web frameworks. Some examples of Server-centric Web frameworks are Struts, JSF (Java Server Faces), Spring MVC, Wicket and

Stripes. In Client-centric Web frameworks, the controller responsible for rendering the data on the client Web browser resides on the client-side. Client-centric Web frameworks are emerging frameworks and most of them have a controller written in JavaScript. Some Examples of Client-centric Web frameworks are Adobe's flash, Microsoft's Silverlight/Moonlight and JavaFX. Most commonly used Web browsers are built on top of JavaScript engines like WebKit, V8 engine and SpiderMonkey which render the data differently. This is a major concern for the Client-centric Web frameworks. There are many JavaScript libraries like jQuery, Yahoo's YUI, Ext JS and MochiKit which reduces the differences in the rendering styles of different JavaScript engines.

2.2 Model Driven Generation of Web Applications

Model-Driven Engineering (MDE) [5] offers a good approach for building Web applications. MDE automatically generates the code from a model. MDE [6] is extremely helpful in building Web systems whose design changes regularly. MDE specifies that model should be treated as a most important element in any phase of the software development life cycle and should be used in all phases of the software development process. MDE helps developers to focus on the problem and ignore the specification of its implementation or solution, i.e., developers can concentrate on what is the functionality of the system rather than focusing on how to develop the system. MDE defines a new specialized area of engineering under software engineering called "Web Engineering". Web engineering addresses the most common problems faced during the development of Web based software systems. Some Web engineering approaches are OO-H [7], WebML [8] and UWE [9]. The most popular MDE approach is the Model Driven Architecture (MDA) defined by the Object Management Group (OMG) [10]. Most of these approaches automatically build solutions by splitting the problem space horizontally, i.e., layer-by-layer.

2.3 WSDL

Web Service Description Language (WSDL) [11] is an XML notation language used to describe Web services. A WSDL document describes Web services with the following tags/elements types, message, portType, operation, binding and service. The <types> tag is used to define the data types that are used by the Web service. The <message> tag is used to define the data elements of an operation. A message tag can contain one or more part tags where each part tag represent a parameter of an operation. The <portType> tags are used to represent a set of operations defined in this service. Each operation has messages depending up on the type of the operation. The <binding> tags are used to define the message exchange format and protocol details for each operation. The <service> tag is used to specify the location of the Web service. WSDL have two major versions, i.e., WSDL 1.1 [12] and WSDL 2.0 [13]. WSDL 1.1 supports only Simple Object Access Protocol (SOAP) [14] based Web services. WSDL 2.0 supports both SOAP based and Representation State Transfer (REST) [15] based Web services.

2.4 SAWSDL

Semantic annotations for WSDL and XML schema (SAWSDL) [16] are extensions to WSDL to semantically annotate WSDL and XML schema elements using ontology. Annotations are specified using the following attributes “modelReference”, “liftingSchemaMapping”, “loweringSchemaMapping”. A modelReference tag is used to specify an association between WSDL element and a concept in ontology. A liftingSchemaMapping is used to specify the transformation of WSDL data to ontology instance data, i.e., lift the data from WSDL and put it in an ontology instance. A loweringSchemaMapping is used to specify the transformation of ontology data to WSDL XML data, i.e., lower the data from ontology and put it in WSDL.

SAWSDL annotations are usually represented as attributes of WSDL elements. There is one exception in which SAWSDL had defined the <attrExtensions> tag to specify the annotations for operations in WSDL 1.1.

2.5 OWL

The Web Ontology Language (OWL)[17]. It is a standard used to define ontologies in XML. It is built on top of Resource Description Framework (RDF) [18]. OWL is designed for processing information over the Web and is designed to be understood by machines. OWL has three sub languages OWL Lite, OWL DL and OWL Full. These languages are designed based on expressiveness and decidability. OWL Lite is the least expressive sublanguage and fully decidable. OWL DL is based on Description logic and OWL Full is the most expressive sublanguage. OWL Full is not decidable. Every OWL Lite valid conclusion is an OWL DL conclusion and every OWL DL valid conclusion is an OWL Full conclusion. OWL Lite should be used in situations where there is need for the least expressiveness. OWL DL should be used where users need maximum expressiveness and still be decidable. OWL Full should be used only where importance is given to expressiveness and not to decidability. OWL DL is the most popular of the three sub languages. There are two main java packages used for parsing these documents 1) JENA [19] 2) OWLAPI [20].

CHAPTER 3

SOFEA

Service Oriented Front-End Architecture (SOFEA) [3] is an emerging standard for building Web applications. Traditionally, Web application frameworks are developed according to the 3-tier or 4-tier architecture. The graphical representation of the 3-tier architecture is shown in figure 3.1

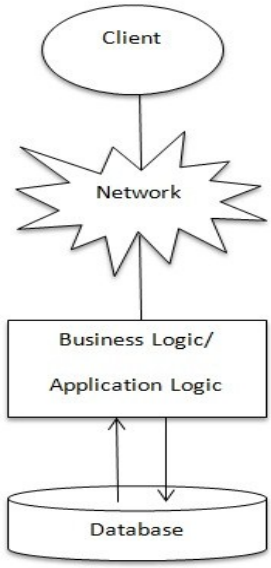

Figure 3.1: 3-tier Architecture.

A client's user interface (UI) communicates with the Business / Application logic over the network and Business/Application logic communicates with the Back-end / Database. This approach works well, but the problem is that there is no other way to access the data other than using the client UI. However, with the increasing popularity of SOA, we need a newer approach

to build Web applications that supports data access in different ways other than using a client UI (e.g. through Web browser). SOFEA helps in providing data access through Web services. This is one major advantage of the SOFEA architecture. The architecture of the SOFEA is graphically represented in figure 3.2

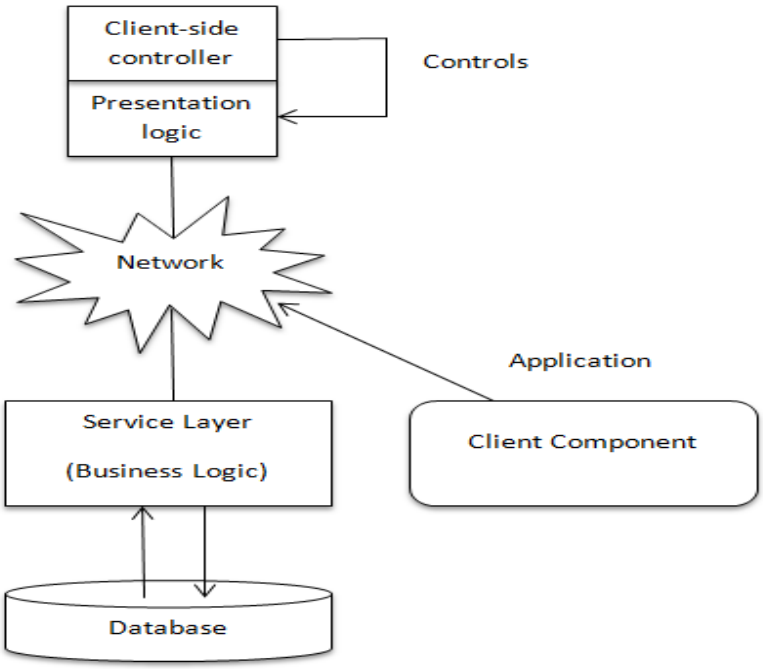

Figure 3.2: SOFEA Architecture.

3.1 Principles of SOFEA

The three major principles of SOFEA [3] are the following:

1. Decouple the different layers in 3-tier or N-tier architecture. SOFEA states that Application logic, Presentation logic, Business logic must be independent instead of sharing a closely connected relationship. Business logic is the part of the software system that performs the required data processing, e.g., extracting the data from the database,

saving the data into database, etc. Application logic is the part of the Web based system that performs the operations, including how to implement security, how to manage and access the user state, etc. Presentation logic is the part of the Web based system which specifies how the data is rendered on the end user's system. This is the main principle of SOFEA.

2. Presentation logic should be controlled by the client-side component. Also, client state must be managed by a client-side controller. This principle strongly opposes the traditional approaches for building Web applications. In traditional Web applications, the presentation logic is typically controlled by a server side component.

3. More emphasis should be placed on the data exchange between the client-side component and the service layer. Since, the client-side component controls the presentation logic, the only thing that the client-side controller needs is the data so more emphasis should be given on the data exchange between the service layer and client-side component.

3.2 Advantages and Disadvantages of SOFEA

The major advantages of SOFEA are the following:

1. Data can be accessed in many ways instead of just providing access via a Web browser.

2. Extensibility becomes easy as one has to write only the presentation logic and client-side controller specific to an application.

The major disadvantages of SOFEA are the following:

1. Having a client side controller to control the presentation logic results in some security issues. For example, a client side controller exposes presentation logic to the end user.

2. SOFEA results in downloading the entire client side application and client-side controller on to the end user's system, so sufficient network bandwidth is needed for SOFEA to be an effective approach. There are situations where the application is smaller than the fully formatted data. For example, animations developed using Adobe's flash. In such situations bandwidth is less of an issue.

CHAPTER 4

GLYCOVAULT

GlycoVault is a Web application built using the principles of SOFEA. GlycoVault acts as a repository for storing data from scientific experiments.

4.1 Architecture

The architecture of GlycoVault is shown in the figure 4.1

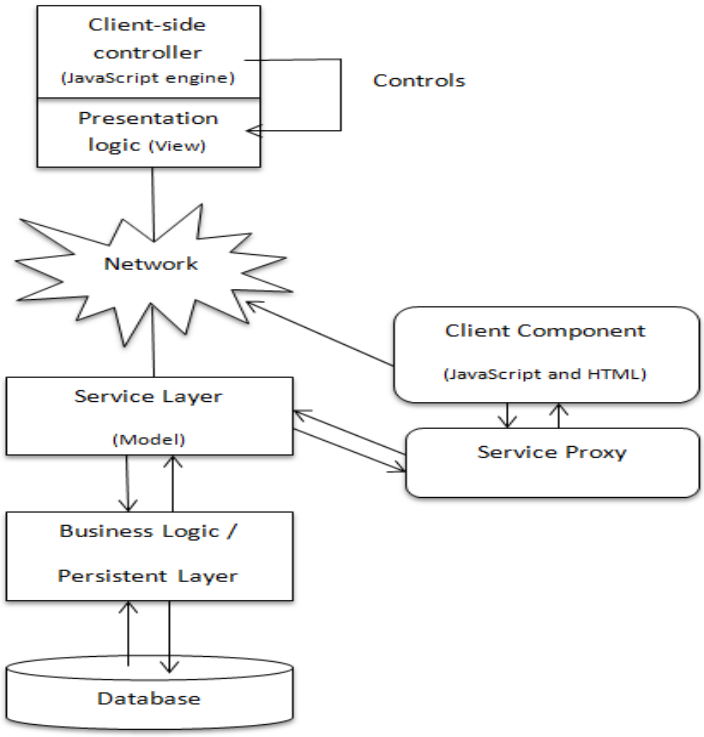

Figure 4.1: Architecture of GlycoVault

As you can see GlycoVault closely follows SOFEA principles with slight variations. We have a Proxy Layer which acts as a mediator between Web services and Client-side controller. We had Proxy Layer to avoid direct invocations of Web services through JavaScript as it makes our development much faster and it is much easier to maintain the code base. The Service Layer is currently implemented using SOAP based Web services and is written in Java. The Client-side controller and Presentation logic are implemented using HTML and JavaScript. The Client-side controller interacts with the Proxy Layer which interacts with the Service Layer. The proxy layer is implemented in Java. The API is implemented in Java.

4.1.1 Modifications to SOFEA

As you can see from the figure 4.1, we have slightly modified the principles of SOFEA because of the reasons we observed during the development process. We believe that having a client-side controller for presentation logic may not be ideal, as it exposes various security issues. We think that the decision of having a client-side controller should not be made mandatory. In fact, we think the decision of having a client-side or server-side controller must be left to the developer/designer group who are responsible for developing the application. A client-side controller is good when the application is fairly small. However, for large Web applications developing/managing client-side controller becomes a challenge.

4.1.2 Proposed Hybrid Architecture

SOFEA and traditional Web frameworks have major advantages, but they also have some major drawbacks too. We are proposing a Hybrid Architecture which is a combination of both SOFEA and traditional Web frameworks which attempts to minimize the drawbacks of each. The major

drawback of the Web frameworks is that they lack access to data other than using a Web browser. The major disadvantage of SOFEA is having a client-side controller. Our proposed Hybrid architecture combines both the ideas by defining a service layer similar to SOFEA, but instead of having a client component, we define a Persistent Object Layer which acts as a database to the traditional Web frameworks.

4.2 Information Model

The UML class diagram of the GlycoVault is shown in figure 4.2. The UML class diagram presents three different views 1) Experiment Design View 2) Experiment Setup View and 3) Experiment Execution View.

Experiment Design View presents the design point of view of the GlycoVault. Experiments usually have a design which is followed while conducting experiments. “ExperimentDesign” is the class that represents the design of an experiment. Each ExperimentDesign have a set of rules or protocol templates which are represented by the class “Protocol” and these protcols are the part of Experiment Design. Each protocol is monitored by a set of observables and is varied by set of parameters, so we have classes called “Parameter” and “Observable” to represent them.

Experiment Setup View presents the view of the GlycoVault in terms of experiment setup. In order to conduct an experiment we make an experiment design first and follow the design to create an experiment setup, which is represented by the class called “ExperimentSetup”. Experiment design have a set of protocols which are varied across various experiment setups, so we have a called “ProtocolVariant” to represent the variations in the

"Protocols". This variation is established by the "ParamValue" class, which represents the value for the parameters involved in the Protocol.

Experiment Execution View represents the implementation point of view of an experiment setup. Each experiment setup is followed by conducting or executing an experiment which is represented by "Experiment" class and each step in an experiment is represented as an instance of the "Task" class. Each step in an experiment produces some samples or data, which are represented by the class "ExperimentalObject". An experiment object can also be used by an experiment.

An experiment object can be classified into "PhysicalObject" to represent the physical samples like biological samples or a "DigitalObject" to represent the digital data. A physical object is further classified into "BiologicalSample" to represent the biological samples used/produced by a task and "MolecularObject" to represent the molecular objects such as genes, enzymes, transcripts, protiens, lipids and glycans involved in a task. An experiment usually starts with some initial sample which is represented by class "SourceSample". A "DerivedSample" class is used to represent those samples which are produced at the end of an experiment step or task. Some experiment steps also yield digital data in the form of files so we have a class called "File" to represent them. These files can be simple plain texts, spreadsheets, images, etc., so their types are represented by the class "FileType". We also extract some important information represented by "Observable" instances, from these files and store them in our database. The extracted data are stored in the "ScalarValue" class. These values can be either strings or floats. The "Vector" class is used to represent the order in which these scalar values are stored.

Experiments are usually carried out by scientists, so we have a class “User” to represent them. The “Laboratory” class is used to represent the labs in which these scientists work. The “ControlledObject” class is used to implement the security features in GlycoVault.

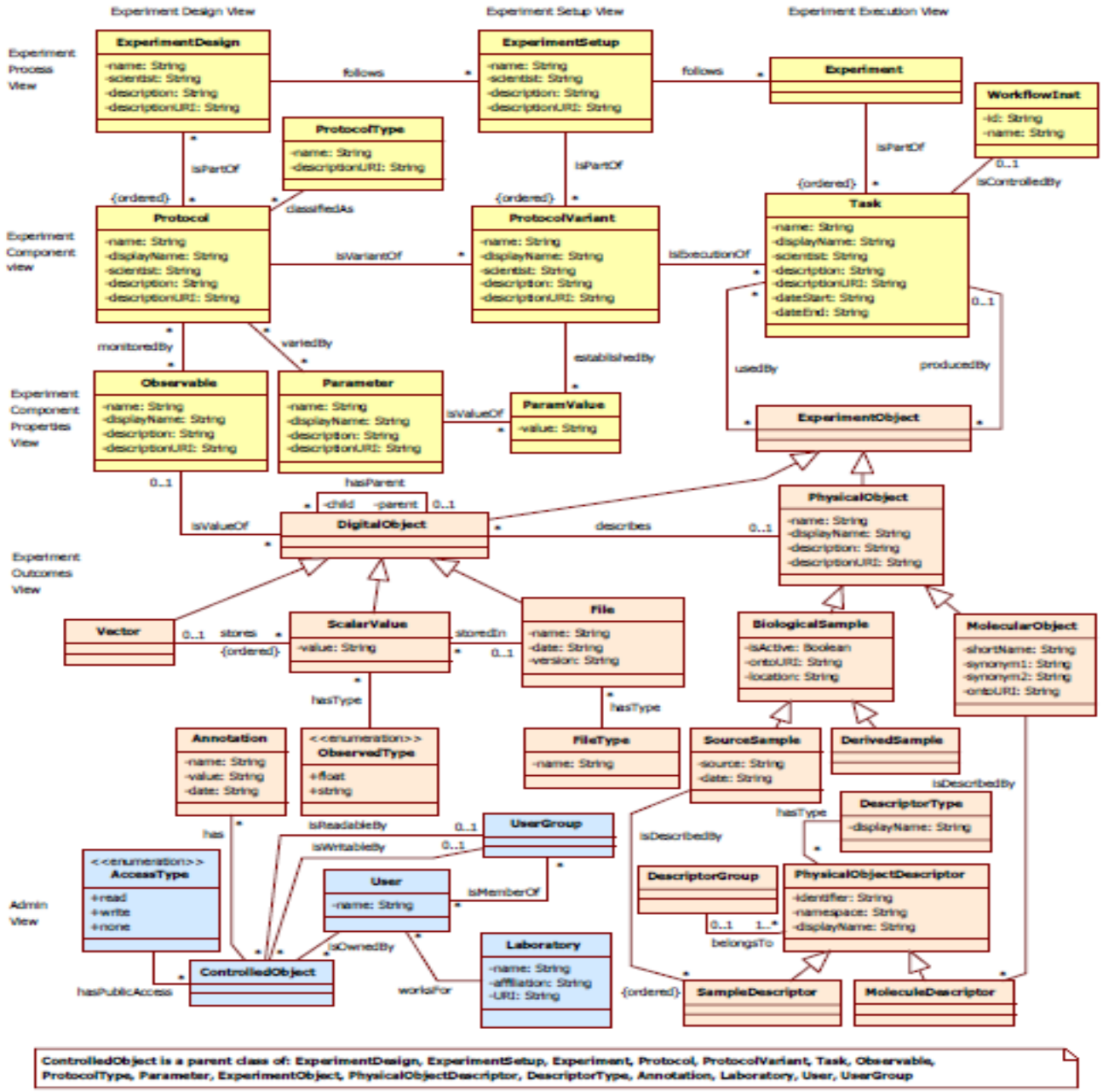

Figure 4.2: UML Class Diagram of GlycoVault

4.3 Persistent Object Layer

A good software system should follow the principles of object oriented programming, including abstraction, inheritance, polymorphism etc., in order to build a secure system. Layers like Persistent Object Layer, Service Layer etc., and sub layers like API, its implementation, factory etc., are defined for a software system to ensure that the principles of object oriented programming are maintained. We have defined an API layer for GlycoVault so that we can hide its implementation and provide the necessary documentation to access the data by an end user. The entire API layer of GlycoVault is generated automatically, so we need to specify the methods that are defined in the API and how they are generated. We are generating this API layer from the database schema and we made sure that this API layer matches our UML class Diagram. For every class in the UML diagram, we have a corresponding class in the API. Each class has getters and setters for the fields defined in the UML class. For the relationships, we have defined methods to get and set the corresponding class objects depending upon the relationship. Details about the code generation process are discussed in chapter 5.

4.4 Web Service Layer

The main intention of GlycoVault is to share the data, so we have defined Web services in the service layer to share the data with other organizations. The service layer hosts SOAP based Web services allow users to the data. As mentioned earlier, we have automated the code generation for most parts of the service layer, except for the login service. Every Web service has a close resemblance to a class in the API and its factory. Each operation in a Web service corresponds to a method in the corresponding class of the API or its factory with one exception, i.e., each operation in the service layer has one additional parameter called “sessionId” which is a string

returned when the user invokes the login service. This sessionId string is used to authenticate the user and also check his read and write access rights.

4.5 Client Component

Most programming languages provide an API for invoking SOAP based Web services. There are many Web 2.0 client libraries services such as jQuery's SOAP Client and codeplex's JavaScript SOAP Client with which one can invoke these Web services. These Web 2.0 client libraries also enable us to embed the service invocation in a Web page which makes it transparent to the end users. End users can interact with these Web service as if they are interacting with any Web site. We have introduces proxy servlets and invoked them using AJAX.

4.6 Security

Security is a major issue for any Web based application. The same is true for GlycoVault. We could have used a security module of the underlying database. The problem with this approach is that the lowest level on which we can impose the built in security of database is at the Table level. Most open source databases cannot impose row level security natively (at present), so we had to implemented the row level security ourselves. We have implemented a UNIX like security system in the GlycoVault. The "ControlledObject" and "UserGroup" are classes used to implement the security in GlycoVault. The Controlled object class is used to determine the access rights of a user. Most of the classes of the GlycoVault are subclasses of the Controlled Object. The Controlled Object class has four important fields, i.e., isOwnedBy, isReadableBy, isWritableBy and publicAccess to provide access restrictions. isOwnedBy is a reference to user which determines the owner of a record in the database. isReadableBy and isWritableBy are

references to userGroup to determine which group has read permissions and write permissions. publicAccess is an enum which can take either “write”, “read” or “none”. We can determine if a user has read rights for a record of a table by extracting the controlled object record associated with the record under consideration. A user can read a record only if he/she is owner of that record or he/she is a member of group that has write access or read access to that record or public access for that record is not “none”. Similarly, a user can write/update a record only if he/she is owner of that object or he/she is a member of a group that has write access to that object or public access for that object is “write”.

We have created user groups to provide access restrictions on the records in the database. The sessionId is used to perform this authentication. A user gets this sessionId string by invoking the login operation of the login SOAP service. Users have to provide the session id while invoking any other service. In case a user does not provide a valid session id, then an exception is thrown with a message to indicate his activity. This “sessionId” string which is generated using “Secured Hash Algorithm 1” with a dynamic key, making it harder to predict the string. We store this “sessionId” as a key in hash map with user details as its corresponding value of the hash map. This hash map record is automatically removed after a certain period of inactivity. In case a user attempts to login more than once, we return the same “sessionId” to avoid duplicate sessions.

4.7 Transactions

Defining transactions for Web services is quite a challenging problem. Defining transactions for an operation is a good starting approach, but the problem with this approach is that it fails to rollback a transaction at the workflow level. Defining a transaction at the workflow level is also

a good approach, but the problem with this approach is that services have no idea on how users are going to define their workflow. In order to overcome these problems, we have defined a Web service to start a transaction and terminate a transaction as operations. User can start a transaction and stop it whenever he/she needs, by invoking the corresponding operations. Also, we made sure that an uncommitted transaction rolls back automatically after a certain period of time and logs out the user to avoid infinite wait times. If a user wishes not to use this transaction service, then he can still perform the rollback by invoking those operations which delete the corresponding objects that he created during the execution of the workflow. In this case, transactions are performed at the operation level.

4.8 Data Loading

Some experiments generate spreadsheet files and we have defined workflows with which we can extract the data from these spreadsheets and populate the GlycoVault database. We also save the spreadsheet file in GlycoVault for a reference. Loading spreadsheet files into GlycoVault database has its own importance. Most of these spreadsheets have a similar pattern or structure which varies across the categories of spreadsheets, so extraction of data becomes straight forward, yet tedious. We plan to explore techniques for automating this in the future. A sample spreadsheet is shown in the figure 4.3. We have defined workflows for extracting the required data specific to the class of the spreadsheet. In order to load these spreadsheets into GlycoVault we need to have some initial data which is present in our ontologies. There are no Web services at present that can extract the data defined in these ontologies into our database. So, we have loaded the required data extracting the spreadsheets which contain the required information and invoke our Web services to insert the data.

	A	B	C	D	E	F	G	H	I	J	K	L
1	Plate #	Re-Run	>=35 in Bold Purple	>=0.5 in Red						Scaled Data		
2	Well	Gene ID	Repl Ct Value	Std Dev Cts	2^-Ct	Norm Value	Normalized	Scaler	Adjusted	Average	Stdev	
3	A1	GT01 M03	27.36		5.80523E-09	3.502E-06	0.0016576	7E-06	0.00165	0.0014	0.0003	
4	A2		27.93	0.2867635	3.9105E-09	3.502E-06	0.0011166	7E-06	0.00111			
5	A3		27.59		4.94974E-09	3.502E-06	0.0014133	7E-06	0.00141			
6	A4	GT01 M15	24.5		4.21468E-08	3.502E-06	0.0120346	7E-06	0.01203	0.01	0.0017	
7	A5		24.93	0.237557	3.1284E-08	3.502E-06	0.0089328	7E-06	0.00893			
8	A6		24.89		3.21635E-08	3.502E-06	0.009184	7E-06	0.00918			
9	A10	GT01 M21	34.38		4.4729E-11	3.502E-06	1.277E-05	7E-06	5.5E-06	5E-06	4E-07	
10	A11		34.34	0.04	4.59865E-11	3.502E-06	1.313E-05	7E-06	5.8E-06			
11	A12		34.42		4.35058E-11	3.502E-06	1.242E-05	7E-06	5.1E-06			
12	B1	GT01 M22	25.99		1.50048E-08	3.502E-06	0.0042845	7E-06	0.00428	0.0046	0.0004	
13	B2		25.94	0.1209683	1.5534E-08	3.502E-06	0.0044356	7E-06	0.00443			
14	B3		25.76		1.75982E-08	3.502E-06	0.005025	7E-06	0.00502			
15	B4	GT02 M01	24.11		5.5229E-08	3.502E-06	0.0157701	7E-06	0.01576	0.0162	0.0016	
16	B5		24.2	0.1429452	5.18889E-08	3.502E-06	0.0148163	7E-06	0.01481			
17	B6		23.92		6.30032E-08	3.502E-06	0.0179899	7E-06	0.01798			
18	B7	GT02M03	23.77		6.99064E-08	3.502E-06	0.0199611	7E-06	0.01995	0.0205	0.0006	
19	B8		23.74	0.0404145	7.13753E-08	3.502E-06	0.0203805	7E-06	0.02037			
20	B9		23.69		7.38923E-08	3.502E-06	0.0210992	7E-06	0.02109			
21	B10	GT02M04	31.66		2.94707E-10	3.502E-06	8.415E-05	7E-06	7.7E-05	6E-05	1E-05	
22	B11		32.15	0.245	2.09839E-10	3.502E-06	5.992E-05	7E-06	5.3E-05			

Figure: 4.3 A Sample Spreadsheet

4.9 Browsing GlycoVault

From the UML Class Diagram (figure 4.2), we can clearly observe that browsing GlycoVault is not a linear process. We have defined two approaches to search for data 1) Task centric which focuses on a path from a selected class to the Task class and 2) Neighborhood centric search which provides a view of a selected class along with its neighboring relationships. A Sample set of tree paths for the Task centric approach is shown in the figure 4.4. Finding out the steps of an experiment instance is the main objective for the several of the users in the GlycoVault. Steps of an experiment instance are represented by the “Task” class. The Task class is placed on the top to indicate that it is the final point for Task centric search. We can navigate to Task from

Protocol Variant, Biological Sample, Source Sample, Molecular Object, etc., so they are represented as immediate children of the Task (figure 4.4) indicating that we can navigate to Task using a single filter. A user can search for steps of an experiment instance starting from any node in the tree and navigating towards Task. For instance, we can search for the steps of an experiment setup which have some specified observables since Observables are not immediate children of Task, so we may have to apply further search filters on Protocol and Protocol Variant class to get the required Task.

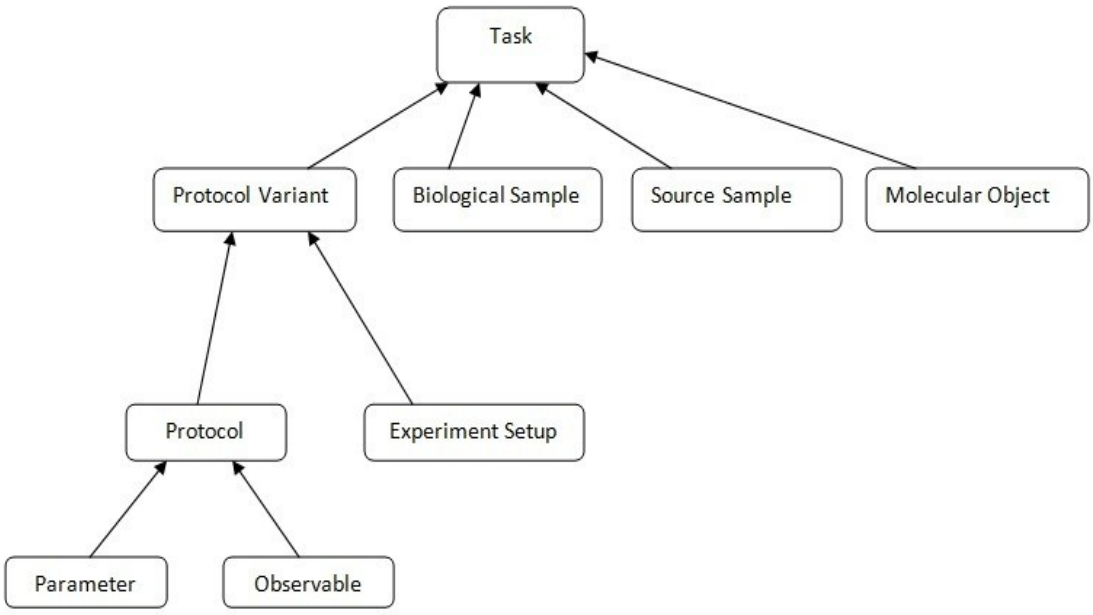

Figure: 4.4 Sample Browse Path

CHAPTER 5

CODE GENERATION FROM SQL SCHEMA

Most of the Web applications are developed to access data from a database through a user-friendly interface. The four most common operations performed by any Web applications are the following:

1) Saving data to the database or backend.
2) Retrieving the data from the database.
3) Updating the data from the database.
4) Deleting the data from the database.

We develop the APIs to read/write a data from/to the database. Most databases organize data using tables and columns. It is quite a tedious task to do similar development for each and every Web application. It is especially annoying if there are periodic changes made to the design of the Web application. These two things are very true for GlycoVault, so, we came up with an approach with which we can generate the code for the Persistent Object Layer and Service Layer and sub layers such as API, API Implementation, factory etc., from the SQL schema using techniques from reverse engineering. We have used an SQL schema instead of an UML design, because there are many Web applications which do not follow UML, but have a solid working SQL Schema. We are interested in supporting each and every system. We generate the code in

such a way that it can prevent any unauthorized access to the data and also any unauthorized writes to the data. In order to understand the reverse engineering technique, one should first have an understanding of how forwarding engineering produces an SQL schema from UML.

5.1 Forward Engineering from UML to SQL

[23] and [24] explains how to generate a schema from a UML Class diagram. An SQL schema may be generated from an UML Class Diagram using following steps:

1) For every class in the UML class diagram, create a Table with the table name as the UML class name. Also, for each field in the class create a column in the table with the column name as the class field name.

2) For each one-to-one association as shown in the figure 5.1, add a foreign key column to the side opposite of the side whose multiplicity lower bounds equals mini (a, b). In figure 5.1, a foreign key should be added to the Table A, if 'b' is smaller than 'a' else the foreign key should be added to the Table B.

Figure 5.1: One-to-One Association

3) For each one-to-many/many-to-one association as shown in the figure 5.2, add a foreign key to the table which is opposite to the one whose multiplicity upper bound is one. In figure 5.2, a foreign key is added to Table A.

Figure 5.2: One-to-Many Association

4) For each many-to-many association as shown in figure 5.3, a new table should be created and the primary keys of both the tables should be added as foreign keys in the newly created table. In figure 5.3, a new table should be created with two foreign keys pointing to Table A and Table B.

Figure 5.3: Many-to-Many Association

5) For each generalization-specialization or parent-child relationship as shown in figure 5.4, make the primary key of the specialized table as the foreign key by referencing it to the generalized table. In Figure 5.4, we need to make the primary key of Table B as the foreign key by referencing it to Table A.

Figure 5.4: Generalization-Specialization Relationship

6) For each aggregation or whole-part relationship, add foreign key columns to part table referencing the primary keys of the whole tables. Note, we are assuming part objects are not shared (i.e., no many-to-many aggregations).

We are also assuming that there are no higher arity relationships (i.e., only binary relationships). Now that we have some background on how the UML-SQL schema transformation occurs. We can proceed with our reverse engineering inspired technique to automatically generate the code.

5.2 Translation from SQL schema to Object Oriented API

We can use the approach discussed in section 5.1 to generate the code from an SQL schema. In order to proceed we need to define few things.

1) In order to check if a table is introduced because of a many-to-many relationship, we need to find the set of foreign key columns for the table and find the set of composite primary keys for the tables. If both sets are not null and set of foreign key columns are a subset of the primary key columns, then it is a many-to-many relationship table, else it is not a many-to-many relationship table.

2) In order to check for parent child relationship, we need to check if the table has a column that is a primary key (not a composite primary key) and also foreign key. If this column is both primary key and foreign key then this table is subclass of the class representing the foreign key table.

Note, we assume that all entity tables have single-column primary keys when they are being referenced. In order to achieve our goal we have defined class which are used to automatically Java generate code from the SQL schema file.

class Table

tableName, // represents the table name of the table

compositePrimaryKey, // represents the list of composite primary key columns.

compositeUniqueKey, // represents the list of column which are composite unique keys.

columns // represents list of columns for this table.

end

class Column

columnName, // represents the name of the given column.

columnType, // represents the language implementation data type for the column.

isPrimaryKey, // true if this column is a primary key.

isForeignKey, // true if this column is a foreign key.

isUnique, // true if this column is Unique.

isNotNULL, // true if this column does not accept null values.

defaultValue, // represents the default value for the column.

isAutomatic, // true if this column gets its value from the database automatically.

foreignKeyTable, // represents a pointer to the table to which this column is foreign key.

foreignKeyColumn // represents a pointer to the column of the foreignKeyTable to which //this column is foreign key.

end

Now that we have defined the helper classes, we can define the algorithms for basic functions.

// this function checks if this table is introduce because of many-to-many relationship or not.

function checkIfManyToMany(table)

foreignkeys=getForeignKeys(table)

if foreignkeys is null **then**

return false

compositePrimaryKey=table.compositePrimaryKey

if compositePrimaryKeys is null **then**

```
            return false
        if foreignkeys ⊆ compositePrimaryKeys then
            return true
        return false
end

// this functions gets the list of tables in which there exists a column which has a foreign key
// table pointing to give table and list of tables. The complete list of tables is passed as the second
// parameter.
function getReferencedByTable(table, tables)
        referencingTables= φ
        for each table1 in tables do
            for each column in table1.columns do
                if column. isForeignKey and column.foreignKeyTable == table then
                    referencingTables.add(table1)
                    break
        return referencingTables
end
```

We can define many other useful functions, but for the sake of simplicity we have defined only the minimum required functions.

5.3 Steps to Generate API

a) parse the SQL schema file and generate the list of tables

b) **for** each table in tables **do**

    1. **if** checkIfManyToMany(table) **then** continue

2. create a class with class name as table name.
3. **if** temp = getParentTable (table) != null **then**
    a. extend your class to represent the subclass relationship.
4. **for** each column in table.columns **do**
    a. **if** column is not foreign key **then**
        i. create getter and setter definitions (mutator) definitions for that column with column type as type of that field.
5. **for** each column in getForeignKeys in table **do**
    a. **if** column is not primary key **then**
        i. create mutator definitions for that column with column type as column.foreignKeyTable
6. **for** each table1 in getReferencedByTable(table, tables) **do**
    a. **if** checkIfManyToMany(table1) **then**
        i. **for** each col1 in table1.foreignkeycolumns **do**
            1. create iterator definitions for the col1 with column type as col1.foreignKeyTable where col1.foreignKeyTable != table1
        ii. **else** go to step b.

b. create iterator definition for table1

In this way we can generate the API code. Similarly, we can also generate the implementation code. The only difference is in API implementation sub-layer we have to define the body of the definitions, add constructors and methods to save, update delete and select data along with methods defined in the API layer.

In order to perform save, update, delete and select data we should store these table structures in the class as static objects and perform the corresponding action. We can also introduce security for preventing any unauthorized activity.

For select we should include all the fields of the class and its parent class and also fields of other classes in which this class is a foreign key table as parameters. The type of each parameter should be a generic filter class if the type is either a string or double. We should use filters as they give all sorts of permutations and combinations for a given parameter. The structure of the filter is

```
class Filter
    isString; //either Double or String.
    minS, maxS; // incase if it is a string filter.
    minD, maxD; // incase if it is double filter.
    isRegex; // incase if it a regex.
    regex; // value of the regex.
end
```

In the factory sub layer, all we need to do is provide a means to create an instance of the corresponding API Class and select method to search for the objects. This can be done by

following an approach similar to the one used for developing the API and its implementation. For Service Layer, we can extend this approach to create a service level object and Web services by providing JAXB annotations.

The problem we faced lately with this approach is to find a way to determine the ordering of a set of pairs. As an ordered set of pairs results in creation of one additional column and we would not have any idea on how to populate this column automatically unless manually specified. So, we have specified an annotation to represent an ordered set by adding @ordered in the comment section of the corresponding column. Most Web applications have a "Select-Option" pane to select an element. This "Select-Option" pane usually displays some text to the end user and hides an integer or a key with which applications find the right element. We have defined annotation called @blindSide to denote a column of a table that appears to the end user and hides the key of the element in a select-option pane. We have defined an annotation called @ignoreAtService to denote those columns which should not be shown to the end user. We have defined an annotation called @apiClassDoc to add the contents following this annotation in the javadoc section of the API.

We could easily generate the entire code within few seconds and found out that it is 90-95% complete. The rest of the codebase deals with adding security snippets which have to be manually specified. This code snippet should be specified only once and should be changed only if there is a change in the underlying security model for the system.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

We have successfully built a Web application based on SOFEA. We have successfully generated Persistent Object Layer and Service Layer using our automatic approach. Except for the security part most of the code is automatically generated. We have successfully defined a few annotations on an SQL schema to improve our automation process. We have manually developed a Web frontend which interacts with the defined Web services and have demonstrated that this system actually works.

In the future, we plan to explore the related issues of caching and transaction management for persistent objects. We plan to handle the UML association classes. To improve our ability to extract data from spreadsheets, we will explore the development of annotated spreadsheet templates. Since we are auto generating the code, we intend to also auto generate a test suite for testing the generated code. Finally, we plan to extend the code generation process to implement REST based Web services.

REFERENCES

1. Avraham Leff, James T. Rayfield, "Web-Application Development Using the Model/View/Controller Design Pattern," Enterprise Distributed Object Computing Conference, IEEE International, p. 0118, Fifth IEEE International Enterprise Distributed Object Computing Conference, 2001.

2. Erl, T. 2005 Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR.

3. SOFEA:http://wisdomofganesh.googlegroups.com/web/Life+above+the+Service+Tier+v1_1.pdf

4. Vosloo, I. and Kourie, D. G. 2008. Server-centric Web frameworks: An overview. ACM Comput. Surv. 40, 2 (Apr. 2008), 1-33.

5. Kraus, A.K.A., Koch, N.: Model-driven generation of web applications in UWE. In: Model-DrivenWeb Engineering (MDWE'07), Como, Italy, July (2007).

6. Fraternali, P. and Paolini, P. 2000. Model-driven development of Web applications: the AutoWeb system. ACM Trans. Inf. Syst. 18, 4 (Oct. 2000), 323-382.

7. Jaime Gómez, Cristina Cachero. "OO-H: Extending UML to Model Web Interfaces".Information Modeling for Internet Applications. IGI Publishing, 2002.

8. Stefano Ceri, Piero Fraternali, Aldo Bongio, Marco Brambilla, Sara Comai, Maristella Matera. “Designing Data-Intensive Web Applications”. Morgan Kaufman, 2003.

9. Nora Koch. “Transformations Techniques in the Model-Driven Development Process of UWE”. Proc. 2nd Wsh. Model-Driven Web Engineering (MDWE’06), Palo Alto, 2006.

10. Object Management Group (OMG). MDA Guide Version 1.0.1. omg/2003-06-01, http://www.omg.org/docs/omg/03-06-01.pdf

11. Christensen, E., et al. W3C Web Services Description Language (WSDL). 2001; Available from: http://www.w3c.org/TR/wsdl.

12. Christensen, E., et al. Web Services Description Language (WSDL) 1.1. 2001; Available from: http://www.w3.org/TR/wsdl.

13. Chinnici, R., et al. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. 2006; Available from: http://www.w3.org/TR/wsdl20/.

14. SOAP. Simple Object Access Protocol 1.2. 2003; Available from: http://www.w3.org/TR/soap12-part1/.

15. Fielding, R. T. and Taylor, R. N. 2002. Principled design of the modern Web architecture. ACM Trans. Internet Technol. 2, 2 (May. 2002), 115-150.

16. Jacek Kopecky, Tomas Vitvar, Carine Bournez, Joel Farrell, "SAWSDL: Semantic Annotations for WSDL and XML Schema," IEEE Internet Computing, pp. 60-67, November/December, 2007.

17. Dean, M., Schreiber, G. (eds.): OWL Web ontology language reference, W3C Recommendation, 10 February 2004. Available at: http://www.w3.org/TR/2004/REC-owl-ref-20040210/

18. O. Lassila and R. Swick, Resource Description Framework (RDF) Model and Syntax Specification, W3C Recommendation, World Wide Web Consortium, Feb. 1999; www.w3.org/TR/REC-rdf-syntax.

19. Matthew Horridge, Sean Bechhofer, and Olaf Noppens. Igniting the OWL 1.1 Touch Paper: The OWL API. OWLED 2007 (submitted), 2007.

20. B. McBride. Jena: A semantic web toolkit. IEEE Internet Computing, 6(6):55–59, 2002.

21. Krys J. Kochut, Amit P. Sheth and John A. Miller, "ORBWork: A CORBA-Based Fully Distributed, Scalable and Dynamic Workflow Enactment Service for METEOR," Technical Report #UGA-CS-TR-98-006 Department of Computer Science, University of Georgia, Athens, Georgia (Fall 1998) pp. 1-11.

22. Ivan Vasquez, John A. Miller and Kunal Verma, "Providing Fault Tolerance for Transactional Web Services," Technical Report #UGA-CS-LSDIS-TR-06-012, Department of Computer Science, University of Georgia, Athens, Georgia (September 2006) pp. 1-33.

23. Davor Gornik, "Entity Relationship Modeling with UML", White Paper, Rational Software. June 2003.

24. Davor Gornik, "Relational Modeling with UML", White Paper, Rational Software, June 2003.

APPENDIX A

Installation Guide

This guide is intended for the developers who would like to generate the code from a SQL schema file. Please make sure that the following software modules are installed on the machine where you would like to run the code generator.

1) JDK 1.6
2) Apache Ant

In order to auto generate the code, developers should have the source code of the package "SQLEater". Open the ParserConstant.java file present under the source folder of SQLEater and change the "DESTINATION_FOLDER" to the path where you would like to have your java code generated (this step is optional). Also, change the "SOURCE_SQL" field to the path where your SQL schema file is located and save the file (this step is optional). You also need to modify the additionalConstantsFile.txt to add the appropriate connection url, user name and password to establish JDBC connection to your database. Open a terminal or cmd.exe and navigate to the SQLEater project and run the following commands

1) To clean the project, execute "ant clean".
2) To build the project, execute "ant".
3) To run the project, execute "ant run".

The step 3 results in generation of the code in the destination folder.

To run the GlycoVault project follow the above steps with our GlycoVault SQL schema as the SOURCE_SQL and generate the code. Build a war file using the generated code along with the appropriate JDBC Driver.

To run the project, you need to install following software on your machine

1) JBoss server.
2) Postgresql database management system.

Run the GlycoVault SQL schema and insert data SQL files to create and initially populate the PostgreSQL database. This creates the required tables and the necessary data for the GlycoVault project. Deploy the above generated war file onto the JBoss server. The server side is now ready for access from either Web service clients or Web browser based user interfaces.

APPENDIX B

SQL Schema for GlycoVault

```
-------------------------------------------
-- Create the database schema for GlycoVault
-- version 1.0
-- Matthew Eavenson, Srikalyan Swayampakula, John Miller
-- Sun Apr 25 13:32:42 EDT 2010
-------------------------------------------

DROP SCHEMA public CASCADE;
create schema public;
SET search_path TO public;

DROP TABLE IF EXISTS ControlledObject CASCADE;
…
.
-- ControlledObject
CREATE TABLE ControlledObject ( -- @apiClassDoc * ControlledObject is used for controlling access.
  sid bigserial PRIMARY KEY,
  isOwnedBy bigint NOT NULL,
  isReadableBy bigint default null,
  isWritableBy bigint default null,
  publicAccess varchar(20) DEFAULT 'none',
  CHECK (publicAccess in ('read','write','none'))
);
-- Laboratory
CREATE TABLE Laboratory ( -- @apiClassDoc * Laboratory provides descriptions of labs.
  sid bigint  PRIMARY KEY,
  name varchar(256) NOT NULL, -- @blindSide
  description varchar(256) DEFAULT NULL,
  descriptionURI varchar(256) DEFAULT NULL,
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid)
);
```

```
-- User
CREATE TABLE "User" ( -- @apiClassDoc * User info for registered users.
  sid bigint PRIMARY KEY,
  worksFor bigint,
  name varchar(255) NOT NULL UNIQUE, -- @blindSide
  password varchar(255) NOT NULL,
  email varchar(255) NOT NULL,
  touched timestamp without time zone NOT NULL DEFAULT Now(), -- @ignoreAtService
  registration timestamp without time zone NOT NULL DEFAULT Now(),  -- @ignoreAtService
  lastLogin timestamp without time zone NOT NULL DEFAULT Now(),  -- @ignoreAtService
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid) DEFERRABLE,
  CONSTRAINT User_worksFor_fkey FOREIGN KEY (worksFor) REFERENCES Laboratory (sid)
DEFERRABLE
);
-- UserGroup
CREATE TABLE UserGroup ( -- @apiClassDoc * UserGroup maintains the set of approved user groups.
  sid bigint PRIMARY KEY,
  -- Group names are short symbolic string keys.
  -- The set of group names is open-ended, though in practice
  -- only some predefined ones are likely to be used.
  -- At runtime $wgGroupPermissions will associate group keys
  -- with particular permissions. A user will have the combined
  -- permissions of any group they're explicitly in, plus
  -- the implicit '*' and 'user' groups.
  groupName varchar(16) NOT NULL, -- @blindSide
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid)
);
-- MemberOfGroup
CREATE TABLE MemberOfGroup ( -- @apiClassDoc * MemberOfGroup holds the user, group relationship.
  "user" bigint NOT NULL,
  grp bigint NOT NULL,
  PRIMARY KEY ("user", grp),
  FOREIGN KEY ("user") REFERENCES "User" (sid),
  FOREIGN KEY (grp) REFERENCES UserGroup (sid)
);
```

```
-- ExperimentObject
CREATE TABLE ExperimentObject ( -- @apiClassDoc * ExperimentObject for inputs/outputs of experiments.
  sid bigint  PRIMARY KEY,
  producedBy bigint DEFAULT NULL,
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid)
  -- FOREIGN KEY (producedBy) REFERENCES Task (sid)
);
-- Annotation
CREATE TABLE Annotation ( -- @apiClassDoc * Annotation holds notes about other controlled objects.
  sid bigint  PRIMARY KEY,
  name varchar(256) NOT NULL, -- @blindSide
  added timestamp without time zone NOT NULL DEFAULT Now(),
  value varchar(2048) NOT NULL,
  annotates bigint NOT NULL,
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid),
  FOREIGN KEY (annotates) REFERENCES ControlledObject (sid)
);
-- ExperimentDesign
CREATE TABLE ExperimentDesign ( -- @apiClassDoc * ExperimentDesign holds a list of Protocols.
  sid bigint  PRIMARY KEY,
  name varchar(256) NOT NULL, -- @blindSide
  scientist varchar(256) NOT NULL,
  description varchar(256) DEFAULT NULL,
  descriptionURI varchar(256) DEFAULT NULL,
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid)
);
-- ExperimentSetup
CREATE TABLE ExperimentSetup ( -- @apiClassDoc * ExperimentSetup holds a list of ProtocolsVariants.
  sid bigint  PRIMARY KEY,
  name varchar(256) NOT NULL, -- @blindSide
  scientist varchar(256) NOT NULL,
  description varchar(256) DEFAULT NULL,
  descriptionURI varchar(256) DEFAULT NULL,
  follows bigint NOT NULL,
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid),
  FOREIGN KEY (follows) REFERENCES ExperimentDesign (sid)
```

```
);
-- Experiment
CREATE TABLE Experiment ( -- @apiClassDoc * Experiment holds a list of experimental Tasks.
  sid bigint  PRIMARY KEY,
  follows bigint NOT NULL,
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid),
  FOREIGN KEY (follows) REFERENCES ExperimentSetup (sid)
);
-- ProtocolType
CREATE TABLE ProtocolType ( -- @apiClassDoc * ProtocolType specifies the category of a Protocol.
  sid bigint  PRIMARY KEY,
  name varchar(256) NOT NULL, -- @blindSide
  descriptionURI varchar(256) DEFAULT NULL,
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid)
);
-- Protocol
CREATE TABLE Protocol ( -- @apiClassDoc * General plan for an experimental step.
  sid bigint  PRIMARY KEY,
  name varchar(256) NOT NULL, -- @blindSide
  displayName varchar(256) NOT NULL,
  scientist varchar(256) NOT NULL,
  description varchar(256) DEFAULT NULL,
  descriptionURI varchar(256) DEFAULT NULL,
  classifiedAs bigint NOT NULL,
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid),
  FOREIGN KEY (classifiedAs) REFERENCES ProtocolType (sid)
);
-- PartOfExperimentDesign
CREATE TABLE PartOfExperimentDesign ( -- @apiClassDoc * PartOfExperimentDesign associates
ExperimentDesign with Protocol.
  design bigint NOT NULL,
  protocol bigint NOT NULL, -- @blindSide
  step int NOT NULL, -- @ordered
  PRIMARY KEY (design, protocol, step),
  FOREIGN KEY (design) REFERENCES ExperimentDesign (sid),
  FOREIGN KEY (protocol) REFERENCES Protocol (sid)
```

```
);
-- ProtocolVariant
CREATE TABLE ProtocolVariant ( -- @apiClassDoc * ProtocolVariant is a variant of a Protocol with parameters specified.
  sid bigint  PRIMARY KEY,
  name varchar(256) NOT NULL, -- @blindSide
  displayName varchar(256) NOT NULL,
  scientist varchar(256) NOT NULL,
  description varchar(256) DEFAULT NULL,
  descriptionURI varchar(256) DEFAULT NULL,
  isPartOf bigint NOT NULL,
  isVariantOf bigint NOT NULL,
  sequenceNumber integer NOT NULL, -- @ordered
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid),
  FOREIGN KEY (isPartOf) REFERENCES ExperimentSetup (sid),
  FOREIGN KEY (isVariantOf) REFERENCES Protocol (sid)
);
-- WorkflowInstance
CREATE TABLE WorkflowInstance ( -- @apiClassDoc * WorkflowInstance indicates which workflow inserted the data.
  sid bigint  PRIMARY KEY,
  name varchar(256) NOT NULL UNIQUE, -- @blindSide
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid)
);
-- Task
CREATE TABLE Task ( -- @apiClassDoc * Task records info about experimetal steps/tasks.
  sid bigint  PRIMARY KEY,
  name varchar(256) NOT NULL, -- @blindSide
  displayName varchar(256) NOT NULL,
  scientist varchar(256) NOT NULL,
  description varchar(256) DEFAULT NULL,
  descriptionURI varchar(256) DEFAULT NULL,
  dateStart timestamp without time zone NOT NULL DEFAULT Now(),
  dateEnd timestamp without time zone NOT NULL DEFAULT Now(),
  isExecutionOf bigint NOT NULL,
  isControlledBy bigint DEFAULT NULL,
  isPartOf bigint NOT NULL,
```

```
  sequenceNumber integer NOT NULL, -- @ordered
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid),
  FOREIGN KEY (isExecutionOf) REFERENCES ProtocolVariant (sid),
  FOREIGN KEY (isControlledBy) REFERENCES WorkflowInstance (sid),
  FOREIGN KEY (isPartOf) REFERENCES Experiment (sid)
);
-- UsedByTask
CREATE TABLE UsedByTask ( -- @apiClassDoc * UsedByTask relates Task with ExperimentObject.
  task bigint NOT NULL,
  expObj bigint NOT NULL,
  PRIMARY KEY (task, expObj),
  FOREIGN KEY (task) REFERENCES Task (sid),
  FOREIGN KEY (expObj) REFERENCES ExperimentObject (sid)
);
-- Observable
CREATE TABLE Observable ( -- @apiClassDoc * Observable indicates the types of experimental outputs.
  sid bigint  PRIMARY KEY,
  name varchar(256) NOT NULL, -- @blindSide
  displayName varchar(256) NOT NULL,
  description varchar(256) DEFAULT NULL,
  descriptionURI varchar(256) DEFAULT NULL,
  -- monitoredBy bigint NOT NULL,
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid)
  -- FOREIGN KEY (monitoredBy) REFERENCES Protocol (sid)
);
-- MonitoredByProtocol
CREATE TABLE MonitoredByProtocol ( -- @apiClassDoc * MonitoredByProtocol relates Observable with
Protocol.
    protocolId bigint NOT NULL,
    observableId bigint NOT NULL,
    PRIMARY KEY (protocolId, observableId),
    FOREIGN KEY (protocolId) REFERENCES Protocol (sid),
    FOREIGN KEY (observableId) REFERENCES Observable (sid)
);
-- Parameter
CREATE TABLE Parameter ( -- @apiClassDoc * Parameter indicates the types of experimental inputs.
```

```
  sid bigint  PRIMARY KEY,
  name varchar(256) NOT NULL, -- @blindSide
  displayName varchar(256) NOT NULL,
  description varchar(256) DEFAULT NULL,
  descriptionURI varchar(256) DEFAULT NULL,
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid)
);
-- VariedByParameter
CREATE TABLE VariedByParameter ( -- @apiClassDoc * VariedByParameter relates Parameter with
Protocol.
  protocol bigint NOT NULL,
  parameter bigint NOT NULL,
  PRIMARY KEY (protocol, parameter),
  FOREIGN KEY (protocol) REFERENCES Protocol (sid),
  FOREIGN KEY (parameter) REFERENCES Parameter (sid)
);
-- ParameterValue
CREATE TABLE ParameterValue ( -- @apiClassDoc * ParameterValue holds values of Parameters of
ProtocolVariants.
  sid bigint PRIMARY KEY,
  -- for generality values are strings, in future may need a type specifier?
  value varchar(256) DEFAULT NULL, -- @blindSide
  isValueOf bigint NOT NULL,
  establishedBy bigint NOT NULL,
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid),
  FOREIGN KEY (isValueOf) REFERENCES Parameter (sid),
  FOREIGN KEY (establishedBy) REFERENCES ProtocolVariant (sid)
);
-- PhysicalObject
CREATE TABLE PhysicalObject ( -- @apiClassDoc * PhysicalObject specifies biological/chemical entities.
  sid bigint PRIMARY KEY,
  name varchar(256) NOT NULL, -- @blindSide
  displayName varchar(256) NOT NULL,
  description varchar(256) DEFAULT NULL,
  descriptionURI varchar(256) DEFAULT NULL,
  FOREIGN KEY (sid) REFERENCES ExperimentObject (sid)
);
```

```
-- DigitalObject
CREATE TABLE DigitalObject ( -- @apiClassDoc * DigitalObject holds data from experiments.
  sid bigint PRIMARY KEY,
  hasParent bigint DEFAULT NULL,
  describes bigint DEFAULT NULL,
  isValueOf bigint DEFAULT NULL,
  FOREIGN KEY (sid) REFERENCES ExperimentObject (sid),
  FOREIGN KEY (hasParent) REFERENCES DigitalObject (sid),
  FOREIGN KEY (describes) REFERENCES PhysicalObject (sid),
  FOREIGN KEY (isValueOf) REFERENCES Observable (sid)
);
-- Vector
CREATE TABLE Vector ( -- @apiClassDoc * Vector is used when DigitalObject is vector valued.
  sid bigint PRIMARY KEY,
  FOREIGN KEY (sid) REFERENCES DigitalObject (sid)
);
-- ScalarValue
CREATE TABLE ScalarValue ( -- @apiClassDoc * Vector is used when DigitalObject is scalar valued.
  sid bigint PRIMARY KEY,
  value varchar(256) DEFAULT NULL, -- @blindSide
  type varchar(15) NOT NULL,
  vId bigint DEFAULT NULL,
  seq bigint NOT NULL, -- @ordered
  storedIn bigint DEFAULT NULL,
  UNIQUE (vId, seq), --  if vId is not null then make sure vId and seq is unique
  FOREIGN KEY (sid) REFERENCES DigitalObject (sid),
  FOREIGN KEY (vId) REFERENCES Vector (sid),
  FOREIGN KEY (storedIn) REFERENCES File (sid)
);
-- FileType
CREATE TABLE FileType ( -- @apiClassDoc * FileType indicates the MIME type of the file.
  sid bigint PRIMARY KEY,
  name varchar(256) NOT NULL, -- @blindSide
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid)
);
```

```
-- File
CREATE TABLE File ( -- @apiClassDoc * File stores a file as a byte array in the database.
  sid bigint PRIMARY KEY,
  name varchar(256) NOT NULL UNIQUE, -- @blindSide
  uploaded timestamp without time zone NOT NULL DEFAULT Now(),
  version varchar(256) NOT NULL,
  contents bytea NOT NULL,
  hasType bigint NOT NULL,
  FOREIGN KEY (sid) REFERENCES DigitalObject (sid),
  FOREIGN KEY (hasType) REFERENCES FileType (sid)
);
-- DescriptorType
CREATE TABLE DescriptorType ( -- @apiClassDoc * DescriptorType indicates the category for
PhysicalObject descriptors.
  sid bigint  PRIMARY KEY,
  displayName varchar(256) NOT NULL,
  -- ex. organism, cell-type, tissue, disease, molecule-type
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid)
);
-- PhysicalObjectDescriptor
CREATE TABLE PhysicalObjectDescriptor ( -- @apiClassDoc * PhysicalObjectDescriptor describe
PhysicalObjects.
  sid bigint  PRIMARY KEY,
  namespace varchar(256) NOT NULL,
  identifier varchar(256) NOT NULL,
  displayName varchar(256) NOT NULL, -- @blindSide
  hasType bigint NOT NULL,
  descriptionGroupNumber integer,
  FOREIGN KEY (sid) REFERENCES ControlledObject (sid),
  FOREIGN KEY (hasType) REFERENCES DescriptorType (sid)
);
-- BiologicalSample
CREATE TABLE BiologicalSample ( -- @apiClassDoc * BiologicalSample records info about samples used in
experiments.
  sid bigint  PRIMARY KEY,
  isActive bool NOT NULL,
  ontoURI varchar(256) DEFAULT NULL, -- @blindSide
```

```
  location varchar(256) NOT NULL,
  FOREIGN KEY (sid) REFERENCES PhysicalObject (sid)
);
-- SourceSample
CREATE TABLE SourceSample ( -- @apiClassDoc * SourceSample records original BiologicalSamples.
  sid bigint  PRIMARY KEY,
  source varchar(256) NOT NULL, -- @blindSide
  dateObtained timestamp without time zone DEFAULT Now(),
  FOREIGN KEY (sid) REFERENCES BiologicalSample (sid)
);
-- DerivedSample
CREATE TABLE DerivedSample ( -- @apiClassDoc * DerivedSample records BiologicalSamples produced by
experiments.
  sid bigint  PRIMARY KEY,
  FOREIGN KEY (sid) REFERENCES BiologicalSample (sid)
);
-- MolecularObject
CREATE TABLE MolecularObject ( -- @apiClassDoc * MolecularObject provides info about molecules used
in experiments.
  sid bigint  PRIMARY KEY,
  shortName varchar(256) NOT NULL, -- @blindSide
  synonym1 varchar(256) NOT NULL,
  synonym2 varchar(256) NOT NULL,
  ontoURI varchar(256) NOT NULL,
  FOREIGN KEY (sid) REFERENCES PhysicalObject (sid)
);
-- SampleDescriptor
CREATE TABLE SampleDescriptor ( -- @apiClassDoc * SampleDescriptor provides descriptions
SourceSamples.
  sid bigint  PRIMARY KEY,
  describes bigint NOT NULL,
  seq bigint NOT NULL, -- @ordered
  FOREIGN KEY (sid) REFERENCES PhysicalObjectDescriptor (sid),
  FOREIGN KEY (describes) REFERENCES SourceSample (sid)
);
-- MoleculeDescriptor
```

```
CREATE TABLE MoleculeDescriptor ( -- @apiClassDoc * MoleculeDescriptor provides descriptions of MolecularObjects.
  sid bigint  PRIMARY KEY,
  describes bigint NOT NULL,
  seq bigint NOT NULL, -- @ordered
  FOREIGN KEY (sid) REFERENCES PhysicalObjectDescriptor (sid),
  FOREIGN KEY (describes) REFERENCES MolecularObject (sid)
);
---------------------------------------------------------------------
-- ALTER TABLE commands required for circular FOREIGN KEY constraints.
-- constraint name required when FOREIGN KEY is NOT NULL.
---------------------------------------------------------------------
ALTER TABLE ControlledObject ADD constraint ControlledObject_isOwnedBy_fkey FOREIGN KEY (isOwnedBy) REFERENCES "User" (sid) DEFERRABLE;
ALTER TABLE ControlledObject ADD FOREIGN KEY (isReadableBy) REFERENCES UserGroup (sid) DEFERRABLE;
ALTER TABLE ControlledObject ADD FOREIGN KEY (isWritableBy) REFERENCES UserGroup (sid) DEFERRABLE;
ALTER TABLE ExperimentObject ADD FOREIGN KEY (producedBy) REFERENCES Task (sid) DEFERRABLE;
```

APPENDIX C

Java Doc of Persistent Object Layer

Package edu.uga.cs.glycovault.api

Interface Summary	
Annotation	Annotation holds notes about other controlled objects.
BiologicalSample	BiologicalSample records info about samples used in experiments.
ControlledObject	ControlledObject is used for controlling access.
DerivedSample	DerivedSample records BiologicalSamples produced by experiments.
DescriptorType	DescriptorType indicates the category for PhysicalObject descriptors.
DigitalObject	DigitalObject holds data from experiments.
Experiment	Experiment holds a list of experimental Tasks.
ExperimentDesign	ExperimentDesign holds a list of Protocols.
ExperimentObject	ExperimentObject for inputs/outputs of experiments.
ExperimentSetup	ExperimentSetup holds a list of ProtocolsVariants.
File	File stores a file as a byte array in the database.
FileType	FileType indicates the MIME type of the file.
Laboratory	Laboratory provides descriptions of labs.

MolecularObject	MolecularObject provides info about molecules used in experiments.
MoleculeDescriptor	MoleculeDescriptor provides descriptions of MolecularObjects.
Observable	Observable indicates the types of experimental outputs.
Parameter	Parameter indicates the types of experimental inputs.
ParameterValue	ParameterValue holds values of Parameters of ProtocolVariants.
PhysicalObject	PhysicalObject specifies biological/chemical entities.
PhysicalObjectDescriptor	PhysicalObjectDescriptor describe PhysicalObjects.
Protocol	General plan for an experimental step.
ProtocolType	ProtocolType specifies the category of a Protocol.
ProtocolVariant	ProtocolVariant is a variant of a Protocol with parameters specified.
Removable	Removable object is used to remove objects from database.
SampleDescriptor	SampleDescriptor provides descriptions SourceSamples.
Savable	Savable is used to save objects into database.
ScalarValue	Vector is used when DigitalObject is scalar valued.
SourceSample	SourceSample records original BiologicalSamples.
Task	Task records info about experimetal steps/tasks.
Updatable	Updatable is used to update objects in the database.
User	User info for registered users.
UserGroup	UserGroup maintains the set of approved user groups.
Vector	Vector is used when DigitalObject is vector valued.

APPENDIX D

List of WSDL Files

1. http://ra.cs.uga.edu:8080/GVsoap2/PhysicalObjectSoap?wsdl
2. http://ra.cs.uga.edu:8080/GVsoap2/SampleDescriptorSoap?wsdl
3. http://ra.cs.uga.edu:8080/GVsoap2/FileSoap?wsdl
4. http://ra.cs.uga.edu:8080/GVsoap2/ExperimentSoap?wsdl
5. http://ra.cs.uga.edu:8080/GVsoap2/ObservableSoap?wsdl
6. http://ra.cs.uga.edu:8080/GVsoap2/VectorSoap?wsdl
7. http://ra.cs.uga.edu:8080/GVsoap2/PhysicalObjectDescriptorSoap?wsdl
8. http://ra.cs.uga.edu:8080/GVsoap2/TaskSoap?wsdl
9. http://ra.cs.uga.edu:8080/GVsoap2/ScalarValueSoap?wsdl
10. http://ra.cs.uga.edu:8080/GVsoap2/ParameterValueSoap?wsdl
11. http://ra.cs.uga.edu:8080/GVsoap2/DescriptorTypeSoap?wsdl
12. http://ra.cs.uga.edu:8080/GVsoap2/DerivedSampleSoap?wsdl
13. http://ra.cs.uga.edu:8080/GVsoap2/BiologicalSampleSoap?wsdl
14. http://ra.cs.uga.edu:8080/GVsoap2/UserSoap?wsdl
15. http://ra.cs.uga.edu:8080/GVsoap2/ParameterSoap?wsdl
16. http://ra.cs.uga.edu:8080/GVsoap2/MoleculeDescriptorSoap?wsdl
17. http://ra.cs.uga.edu:8080/GVsoap2/FileTypeSoap?wsdl
18. http://ra.cs.uga.edu:8080/GVsoap2/UserGroupSoap?wsdl
19. http://ra.cs.uga.edu:8080/GVsoap2/ProtocolSoap?wsdl
20. http://ra.cs.uga.edu:8080/GVsoap2/DigitalObjectSoap?wsdl
21. http://ra.cs.uga.edu:8080/GVsoap2/ExperimentDesignSoap?wsdl
22. http://ra.cs.uga.edu:8080/GVsoap2/ExperimentObjectSoap?wsdl

23. http://ra.cs.uga.edu:8080/GVsoap2/ExperimentSetupSoap?wsdl
24. http://ra.cs.uga.edu:8080/GVsoap2/SourceSampleSoap?wsdl
25. http://ra.cs.uga.edu:8080/GVsoap2/AnnotationSoap?wsdl
26. http://ra.cs.uga.edu:8080/GVsoap2/ProtocolVariantSoap?wsdl
27. http://ra.cs.uga.edu:8080/GVsoap2/MolecularObjectSoap?wsdl
28. http://ra.cs.uga.edu:8080/GVsoap2/ProtocolTypeSoap?wsdl
29. http://ra.cs.uga.edu:8080/GVsoap2/LaboratorySoap?wsdl
30. http://ra.cs.uga.edu:8080/GVsoap2/LoginSoap?wsdl
31. http://ra.cs.uga.edu:8080/qrtPCRReader/ReadqrtPCRExcelSoap?wsdl

APPENDIX E

Unit Testing

We have written a test suite to perform unit testing at the service layer. Since the entire application is automatically generated, it is good approach to perform the tests using a good testing framework. We have used JUnit testing framework for testing the API. We have performed unit testing by creating many objects for each class in the API and invoking the functions of that class.

In order to perform testing at API level, we need to have a copy of our "GlycoVaultTest" package. In order to run the test, we need to make sure that the system has the postgreSQL installed and it is up and running, the glycovault schema SQL and insert data SQL files are also installed in the postgreSQL. Also, make sure that Java 1.6 is installed and Apache ant is also installed. Follow the steps specified in the Appendix A and create a "jar" file not "war" file. Copy the created jar file into the "jarBox" folder of the GlycoVaultTest. Run the suite by navigating to GlycoVaultTest package and run "ant clean" to clean the project, "ant" to build the project and "ant run" to run the suite. If a test case fails then the program will terminate by printing out the exception to indicate the failure.