View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by DigitalCommons@WPI

Worcester Polytechnic Institute
Digital WPI

Major Qualifying Projects (All Years) Major Qualifying Projects

March 2009

Venice Framework: A Municipal Data Object Management Framework

Craig Allen Blanchette
Worcester Polytechnic Institute

Eric Stewart Clayton
Worcester Polytechnic Institute

James J. Fyles
Worcester Polytechnic Institute

Timothy Edison Cheng
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/mqp-all

Repository Citation

Blanchette, C. A., Clayton, E. S., Fyles, J. J., & Cheng, T. E. (2009). *Venice Framework: A Municipal Data Object Management Framework*. Retrieved from https://digitalcommons.wpi.edu/mqp-all/1527

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.

Project Number: GFP0806

Venice Framework: A Municipal Data Object Management Framework

A Major Qualifying Project Report:

submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

____

Craig Blanchette

____

Eric Clayton

____

Timothy Cheng

____

J. Justin Fyles

Date: Tuesday, March 17, 2009

Approved:

____

Professor Gary F. Pollice, Major Advisor

1.) Venice

2.) Municipal Data

3.) Framework

Abstract

Over the past 20 years, the WPI Venice Project Center has been collecting spatial data about the city of Venice, Italy. This document describes our work, during B-term of 2008, on developing a framework for application developers who want to access this data over a network. The completed framework takes the spatial data that has been collected and provides it to Web application developers through the use of Web services.

Acknowledgements

Gary Pollice – Advisor

Fabio Carrera – Co-advisor

Paul W. Davis

Alberto Gallo

Worcester Polytechnic Institute

Università *IUAV* di Venezia

Forma Urbis

United Nations Educational, Scientific and Cultural Organization (UNESCO)

Table of Contents

Abstract i
Acknowledgements ii
Introduction 1
Background 3
Methodology 7
Overview 7
Centralizing the Data 8
Figure 1: Overview of Project Component Tiers 8
Database Technologies Considered 9
Flexibility of DBMS 9
DAO Pattern 10
Figure 2: The DAO Pattern 10
Figure 3: The Active Record Pattern 10
Figure 4. UML for the MDO Manager, Public Art 12
Testing 14
Spatial data and mapping APIs 15
Web Services 17
Code Listing 4 18
Code Listing 5 19
Maven2 20
Application Server 21
Development Environment 21

Development Cycles ... 22
Proof of Concept ... 23
Figure 5: Proof of Concept ... 23
Results and Analysis ... 24
Future Work and Conclusions ... 26
Bibliography ... 28
Appendices ... 30
Appendix A ... 30
REST, SOAP and Their Application to the Venice Framework (Blanchette, Cheng, Clayton, Fyles) ... 30
Works Cited ... 31
Appendix B ... 33
The Broker Pattern (Fyles) ... 33
Example ... 38
Appendix C: Maven2 Tutorials ... 41
Creating the Framework with Maven ... 41
Integrating DBUnit for DB Testing ... 42
Creating a Parent maven2 Project to Share Dependencies ... 43
Porting the WebService to maven2 ... 44
Appendix D: Developing Applications Using the Framework ... 46
Creating a Simple Java Application That Uses the Venice Framework JAR ... 46
Creating a Web Service That Uses the Venice Framework JAR ... 49

Testing the Web Service 49

Appendix E: Deploying the Application to an Application Server 51

Specifying the Database Server 51

Deploying the application to a server 52

Using Images 52

Introduction

For more than 19 years, the WPI Venice Project Center has been collecting data about the city of Venice, Italy. Over 120 Interactive Qualifying Projects (IQPs) and 10 Major Qualifying Projects (MQPs) have been collecting and working with this data. The data encompasses everything from buildings and bridges to public art and canals and has been stored digitally in various Microsoft® Access databases.

While the collected Venice city data is comprehensive, the databases have not been consolidated to a single database or a single location. Many of the databases are stored offline, which makes accessing the data from an offsite location challenging. The few that are online are difficult to access and there are no agreed update procedures for the data in the database. Therefore, applications created from any subset of the collected data are not synchronized with the original copy.

Since there is no central, easy-to-access database, few applications have been developed using the collected data. Most of the data is stored in flat databases rather than more robust solutions, such as relational databases. Also, the data in its current form cannot be remotely connected to and modified.

Our project takes the spatial data that has been collected and provides it to Web application developers in an easy to use format. At the same time, it centralizes all of the existing data. By creating a flexible and scalable design, our framework can be easily generalized to other municipalities. For instance, addresses have been abstracted to be able to adapt to any municipal addressing system. To address the problem of decentralized data, our framework stores all of the data in a single relational database. In order to ease the development of applications with this municipal data, our framework uses object-relational mapping (ORM).

This allows for simple access and modification of the data. In addition, we provide a Web service interface. This Web service can be used to provide other applications with access to all of the features of our framework, including data access. Thus, applications written in a language different than our framework can still benefit from the innovations of our framework.

In our seven-week project term, we have followed an iterative development cycle and have managed to create a framework that consolidates data collected about the city and provides an easy, consistent way for application developers to access this data. We have also put together documentation for this framework to help developers use and expand it, and we have created a proof of concept application that uses the Framework.

The remaining sections of this report focus on our research in municipal data management, how we approached the problem, the techniques and practices that were used, the project's ultimate outcome, and future plans for the framework.

Background

WPI students have collected data on the city of Venice since the Venice Project Center was founded in 1988. These data comprise student-collected municipal statistics (Municipal Data) that are stored electronically. Data collection can range from recording the water taxi (Vaporetto) traffic over different times of the day, measuring the wake pollution and using complex algorithms and programs to analyze the damage over time, and mapping the geospatial data and coordinates of the bridges. In cases such as the latter, students have GPS devices that can pinpoint the exact longitude and latitude of the municipal object. Several past projects (MQPs and others) have used this data for application development. One such project entailed developing a Visual Basic application for viewing Public Art in Venice.

We have worked closely with Professor Fabio Carrera, a full-time faculty member with the Interdisciplinary and Global Studies Division at WPI, to provide a way for this data to be publicly accessible. Professor Carrera founded the Venice Project Center with the goal of preserving Venice by collecting detailed data on the state of the city. This notion of preserving cities through a municipal information infrastructure stemmed from Professor Carrera's PhD dissertation, "City Knowledge". Professor Carrera introduces the concept of "City Knowledge" as a means to centralize municipal data in order to increase effectiveness in "urban maintenance, management, and planning". (Carrera, 2004) Leveraging multipurpose geographic information in a municipal environment allows more in depth spatial analysis of the city. While the municipality benefits from the improved efficiency of centralized municipal data, the combination of geographic information also allows for new interpretations of the data. Professor Carrera explains the "City Knowledge approach" as "the gradual, but systematic compilation of all disparate datasets accumulated by a wide variety of government and non-

government organizations … [which] will prove to be a valid contribution to the creation of emergent comprehensive, updatable municipal information infrastructures.” (Carrera, 2004)

Applications for the data collected over the past 20 years are countless. Applications could be made to aid tourists navigating through Venice, ease the process of municipal officials in maintenance/restoration of in public works, or provide unique and rare insight to university studies on the city of Venice. Our framework is the foundation upon which these applications can be built.

All of the data collected on the city of Venice are separated into two databases — one has information on the properties of the Municipal Data Object (MDO) while the other has mostly spatial information. The former is a Microsoft Access database. The data stored in the Access files can be converted to a comma-separated value file that can be loaded into most popular Database Management Systems. The latter database is a table in a MapInfo file, which is associated with geospatial data. The data includes the coordinates, shape data, and other associated metadata. This was taken and combined with data from the other database in order to start centralizing the data.

Properties of MDOs vary depending on the type of MDO being stored. For instance, a piece of public art has properties for the family who commissioned the piece, whereas a bridge does not have this information. Likewise, a bridge will have its own set of properties, such as the number of steps.

Spatial data is geographic or location data which can be associated with other metadata, such as height or volume. Spatial data can be complex and is not always uniform. At its most simple level, spatial data can be stored as standardized latitude and longitude, also known as a point. Other projections besides standard latitude and longitude exist, including the Universal Transverse Mercator (UTM) or the Universal Polar Stereographic (UPS) projections, both of

which separate the globe into regions. The UTM projection was created by the United States Army Corps of Engineers and takes into account the ellipsoid shape of the Earth. More complex spatial data can be stored as a polygon defined by a list of geographical coordinates. Various formats exist for storing spatial data. The MapInfo (Pitney Bowes MapInfo, 2008) TAB file is one of the more popular formats for storage of geospatial data, capable of storing polygonal location data along with relevant metadata.

Systems that are used for managing spatial data are generally known as geographical information systems (GIS). At its most general, GIS can be defined as a system that "integrates hardware, software, and data for capturing, managing, analyzing, and displaying all forms of geographically referenced information." (What Is GIS?) GIS government systems, known as Spatial Data Infrastructures, include the National Spatial Data Infrastructure (Comittee, 2007) in the United States, and ENSPIRE (ENSPIRE, 2009) in the European Union. Open source standards for GIS systems also exist, created by the Open Geospatial Consortium (OGC) (OGC, 2009). The OGC defines standards regarding GIS on the Internet.

The main goal of our project was to take the spatial data that has been collected and provide it for Web application developers in an easy to use format. A number of Web-based mapping systems exist, including the popular Google Map API. We wanted to create a framework that could be easily expanded and combined with any other services. Most existing frameworks for managing Web-based spatial data are very complicated, so we were looking to fill the gap by providing a system that would be easy for developers of Web applications. The framework has many features that are ready to use and it is built in such a way that any changes or new features that are required can be easily added. The Venice Project Center then decides how the framework should conform to any number of mapping services or mapping format.

We hope that the Venice Framework will aid in future projects done at the Venice Project Center. The project aims to simplify the centralization of the digitized data that has been collected. It also provides a structured way of organizing these data, as well as a standardized way to access the data. While altering the basic infrastructure of the Venice Framework does require some technical knowledge, we have also written documentation and step-by-step tutorials for tasks such as importing data to the database and expanding the framework that should help any future teams make use of the work that we have done. However, if users of the framework wished to add additional data types, we have made the necessary components in our framework so that it may be done with limited technical knowledge. It is also possible that use of this framework could affect city planning and other municipal maintenance needs, although this would require a great deal of expansion of our framework.

Methodology

Overview

The central problem being dealt with in this MQP is how to take spatial data that has been collected from a municipality, centralize it and provide a structured framework that would make it easier for Web application developers to utilize this data.

Centralizing the Data

One of the first decisions that had to be made was how to consolidate the data so that it could be made easily available to application developers. We decided that the data would need to be imported into a single relational database. This database could then be remotely accessed by the framework that would be built. Other Java Web applications running on the same server as the framework could also use the framework to gain a standardized access to the data. In addition, an example Web service layer could be provided that would allow Web applications the use of the Venice Framework's functionalities and would still allow for access control. Figure 1 demonstrates this.

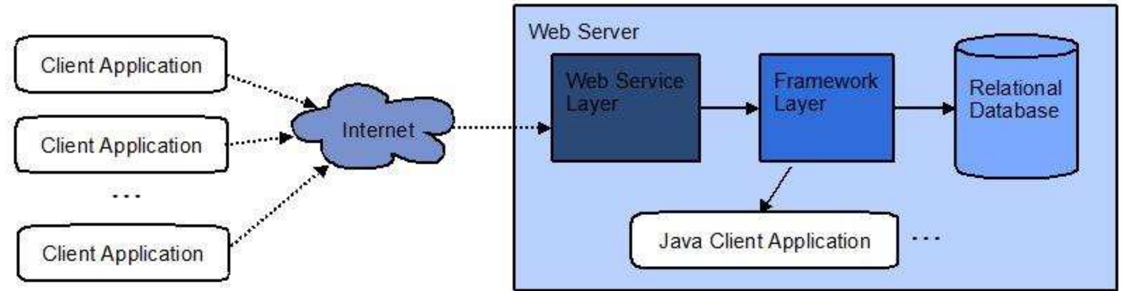


Figure 1: Overview of Project Component Tiers

In order to create a well-designed database with consistent naming and cohesive columns, it was necessary to remove certain data fields that did not make sense in terms of the framework design. However, our framework is designed so that generalization and customization of fields is easy to do when introducing new data sets and/or new ways to manipulate the data. Our framework treats different parts of the municipal data management as distinct layers.

At the top, there is the application layer, which a user can use to view or edit the data through a meaningful user interface. Below that, there is a possible Web service layer that any application can use to add value to the data. Then, there is the municipality specific layer which

each municipality may customize depending on unique features to their city. Next is the framework layer which provides the core services of that are required for any municipality. Last is the data layer which consists of the raw municipal data.

Database Technologies Considered

In order to centralize the collected municipal data, we had to make many decisions from the start. One technology that we took advantage of was Object-relational mapping (ORM). Object-relational mapping is a technique that maps objects and their attributes to rows and columns in a persistent database, respectively. Object-relational mapping can be implemented through a number of libraries. The other option is to use an Object-oriented Database Management System (DBMS). One option which we investigated was the Caché DBMS (InterSystems Caché, 2009). Caché removes the need for having an object-relational mapping by representing all of the data in the database as objects. It can be used with a number of different languages, including Java, which we used to build the framework. Because it is natively object-oriented, it would also provide a small boost in performance to the Venice Framework. While this solution would improve performance, it ties down any developer using our framework to using that specific DBMS. By using a library like Hibernate as a layer above any DBMS that we chose, our framework could be database agnostic. Based on this, we chose to use MySQL instead of Caché so that we would not have to worry about obtaining a student license. In addition, we have had more experience using MySQL and found that it would suit our needs on this project.

Flexibility of DBMS

To allow developers to use any DBMS that they chose to host their municipal data and still allow for the use of object-relational mapping, we turned to Hibernate (Hibernate, 2006). Hibernate is a Java library that is used to perform ORM, mapping java objects to rows in a

persistent database. One of the greatest benefits of using Hibernate, aside from allowing for easy use of ORM, is that it supports almost every DBMS in use today, including MySQL, Oracle, Postgres, Caché, and many others. For this project we chose to work with MySQL because it is a free, open-source product, but, in the future, any programmer using the framework could very easily switch out our DBMS with another thanks to Hibernate. Hibernate allows for the code written for the framework to be compatible to any DBMS'. However, it does not solve the problem of having the data within the database being incompatible to other databases.

DAO Pattern

Hibernate provides easy object-relational mapping, but it does not provide any data access logic to the objects in our project. Many patterns exist to deal with this problem, and some patterns work better than others. Two popular patterns used are the Data Access Object (DAO) pattern and the Active Record pattern.

Figure 2: The DAO Pattern

The DAO pattern is defined as the use of a data access object that "is abstract and encapsulate[s] all access to the data source" (Sun Microsystems, 2002). It is a way of consistently providing data access without having to worry about the underlying database technology. It also encapsulates all data access logic within a single DAO object. The Active Record pattern, created by Martin Fowler, is another pattern that provides

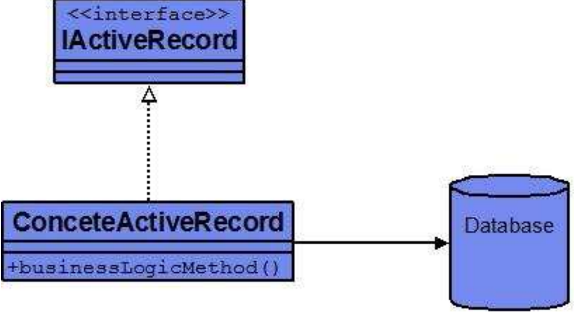


Figure 3: The Active Record Pattern

data access logic to objects. It is defined as “An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.” (Fowler)

This is very similar to the DAO pattern, except that it combines its definition with the definition of ORM. The Active Record pattern also combines business logic with the database access logic in one object. The DAO pattern, on the other hand, completely encapsulates the database access logic. This increased encapsulation led us to choose the DAO pattern over the Active Record pattern for this project.

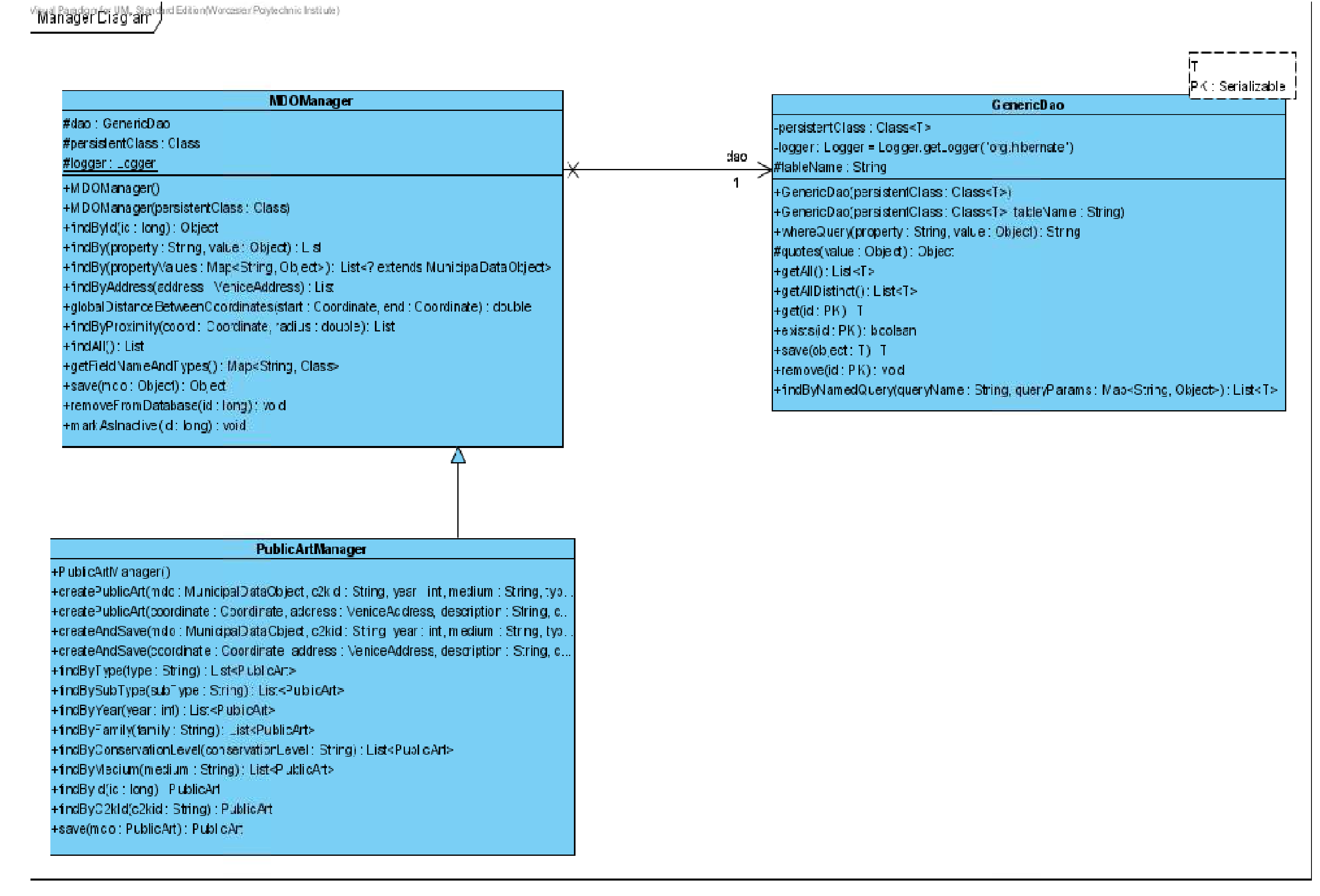


Figure 4. UML for the MDO Manager, Public Art

In our implementation of the DAO pattern, we created a generic DAO class that could be used for data access of any MDO or MDO subclass. This class defined methods to get any attribute of an MDO object, as well as getting all attributes or constructing queries for the Hibernate Query Language (HQL). The following code is the *get by id* operation for our implementation of the DAO pattern.

```
/**
 * Generic method to get an object based on class and identifier.
 */
@SuppressWarnings("unchecked")
public T get(PK id) throws Exception {
    T entity = (T) HibernateUtil.getCurrentSession().get(persistentClass, id);

    if (entity == null) {
        String msg = this.persistentClass + "' object with id '" + id + "' not found...";
        logger.warn(msg);
        throw new Exception(msg);   //@todo consider creating a special exception for this
    }

    return entity;
}
```

Code Listing 1

The code in Listing 1 uses Hibernate to retrieve an entity based on the given id. If the object is not found, then an error message will be logged, otherwise the object is returned. The other methods in our DAO implementation are more complicated, but follow a similar pattern to retrieve objects based on specific attributes.

With the DAO in place to allow for database access, we were then able to develop the other classes that would use the DAO. Our architecture was designed so that every type of Municipal Data Object is described by its own class and has its own manager class. For instance, Public Art objects are described by the *PublicArt* class, and that class is in turn managed by the *PublicArtManager* class. Each MDO that is added to our Framework needs to have both a class and a manager class, and these inherit from a generic MDO class and manager. The MDO manager uses the DAO to retrieve that appropriate requested MDO and also handles persistence of MDOs. The MDO itself stores its own attributes and uses Hibernate annotations

to define the schema of the MDO table in the database. The following is code from MDOManager.java, and uses the *get by id* operation shown earlier in the DAO (Code listing 2). This use allows changes to retrieval of objects to be encapsulated just to the DAO.

```
public Object findById(long id) {
    Object foundObj;
    Transaction tx = HibernateUtil.getCurrentSession().beginTransaction();
    try {
        foundObj = dao.get(id);
    } catch (Exception ex) {
        foundObj = null;
    }
    tx.commit();
    return foundObj;
}
```

Code Listing 2

Municipal Data Object classes and manager classes inherit from the MDO and MDOManager classes and add on other operations specific to that Municipal Data. For instance, the Public Art manager class adds a *findByMedium* operation (Code Listing 3). This operation only makes sense in the context of art, so it is not included in the generic manager. It simply uses a findBy operation defined in the generic manager that will return a list of matching Public Art objects.

```
public List<PublicArt> findByMedium(String medium){
    return findBy("medium", medium);
}
```

Code Listing 3

The generic manager and MDO class make it easy to expand our framework and add new objects. We have also worked to fully document the process of adding new objects to the framework so that anyone unfamiliar with our project should be able to take advantage of it.

Testing

Unit testing is an important aspect of development for any large project, but testing can often be a technical challenge when working with databases. Unit testing is to “take the smallest piece of testable software in the application, isolate it from the remainder of the code,

and determine whether it behaves exactly as you expect." (Microsoft, 2009) There are several options for unit testing when working with databases.

One is to use mock objects. Mock objects are "objects pre-programmed with expectations which form a specification of the calls they are expected to receive." (Fowler, Mocks Aren't Stubs, 2007) While mock objects work when what is being tested is not the database but the rest of the application and any dummy data will do, we wanted a more robust solution.

We needed to test out our connection with Hibernate and would make sure that everything related to the database was functioning correctly, as well as testing the rest of our code. DbUnit is an extension of the well-known JUnit Java unit testing framework. It can be used to put a database "into a known state between test runs." (About DbUnit, 2009) Using DbUnit in combination with HSQLDB (HSQLDB, 2009), a Java database engine, allows for unit testing our application with sample data from a database that can be reset after every test for reliable testing results. This was an ideal choice for our testing needs, since we needed something more robust than simple mock objects. DbUnit also has support to export all of the data and schema in a database that it is using. We were able to use that feature to create a backup of all the data that we had located to the MySQL database we were using.

Spatial data and mapping APIs

One area of research that had to be considered during work on this project was the idea of spatial data. A number of standards currently exist for representation of spatial data. The common term for a system that deals with spatial, or more specifically geographical, data is a Geographical Information System (GIS). GIS extensions for various DBMSs exist, including extensions for MySQL and PostgreSQL. These extensions allow for taking GIS data and looking for intersection and area calculations, as well as other queries that might be performed on geospatial data. While these sorts of queries are important for applications dealing with

complex mapping and simulation, we chose to implement a simpler, more flexible solution that has the ability to be tied into GIS data. The framework can store the latitude and longitude of any municipal data object, and any polygonal GIS data can be related to a municipal data object by using a foreign key in the database. Examples of this GIS data include outlines of buildings or lines defining streets or bridges.

Due to our framework's ability to hold spatial data, it makes sense to combine it with mapping APIs that already exist and are accessible through the Internet. The combination of data from two different sources to create a Web application is referred to as a *mashup*. A popular mapping API that could be combined with our service is the Google Maps API (Google, 2009). The Google Maps API is a service that uses JavaScript to embed interactive maps in a Website. Other APIs, including OpenLayers (OpenLayers, 2009), also exist to provide similar functionality. Because we did not want to tie down our framework to any specific mapping API, we have kept everything agnostic and easy to combine with any API a developer chooses By implementing the aforementioned layered design. We managed this by having our framework use standard Latitude and Longitude units in its calculations and queries and by not coding to any specific API.

In our prototypical example application, we chose to use the Google Maps API in conjunction with our own framework. Integration was a painless process and the displayed maps were detailed and accurate. The Google Maps API is built on top of JavaScript, so we were able to customize it to our needs. Google provides a number of protocols in their API that can be utilized with a small amount of JavaScript. Examples of customization that we performed were changing the default marker icons to "public art" icons that we made and changing the map controls to fit the needs of our application. We were also able to make separate widgets,

such as our view radius slider, that worked with the Google Maps API despite not being part of it.

Web Services

In order to allow access to our data for remote developers, a solution that could provide the functionality of our framework to remote developers needed to be created. The common way to provide a service to remote developers on the Internet is to create a Web service. The World Wide Web Consortium (W3C) defines a Web service as "a software system designed to support interoperable machine-to-machine interaction over a network." (W3C, 2004) The two most popular choices for implementing Web services are Simple Object Access Protocol (SOAP) and Representational State Transfer Web Services (RESTful Web Services).

SOAP is a Web service protocol that accepts remote requests and uses XML to transmit envelopes of data based on those requests. SOAP services are defined using a file known as a Web Service Description Language (WSDL). SOAP is a service-oriented protocol.

REST is a software architecture style that is used in RESTful Web Services. RESTful Web Services consist of resources that are referenced through Uniform Resource Identifiers (URIs). URIs define the paths to specific resources. Every resource mapped in REST has four available Create, Read, Update and Delete (CRUD) actions that can be invoked: POST, GET, PUT and DELETE. CRUD is a model of interaction that describes the basic functionality needed for persistent storage. REST is a resource-based architecture.

Over the past couple of years, there has been a great deal of debate over whether SOAP or REST should be the standard of communication for Web Services. While both sides have reasonable arguments, the whole decision really comes down to whether a Service-Oriented or a Resource-Oriented design is needed. Neither is better or worse because they solve different problems. REST works best for a Resource-Oriented problem, and SOAP works best for a

Service-Oriented problem. In the case of the municipal data framework, a service is being provided that gives access to municipal data. The municipal data could be represented as resources, but querying the data would be much harder, and complex operations that need to be done involving the data would be harder to map. Therefore, a Service-oriented solution makes more sense. SOAP exposes all of the services of the framework, and queries could easily be performed to return all of the objects that match, serialized in XML.

Implementing a SOAP-based Web service required us to first create a Web Services Description Language (WSDL) file that describes the operations that we wanted to expose in our service. Along with the WSDL file, we created an XML Schema Document (XSD), which is an XML file that defines an XML schema structure. In this file, we defined a schema that described the Public Art objects that would be passed. Both of these files can be either written by hand, however some IDEs such as the NetBeans IDE have graphical editors available that make development of these files much easier.

After the WSDL and XSD files were created to describe the Web Service, we used the Java API for XML Web Services (JAX-WS) (JAX-WS Reference Implementation, 2009) to generate Java files to help expedite the writing of our Java based Web service. The NetBeans IDE can work with JAX-WS to generate a service based on a given WSDL, or JAX-WS can be called using other project management tools, such as ANT or Maven. After these classes have been generated, an implementation class can be written to define how the Web Service will respond to requests. The following code (Code Listing 4) is used by our puiblicArtByDistance operation.

```
public List<PublicArt> findByMedium(String medium){
    return findBy("medium", medium);
}
```

Code Listing 4

```
List<edu.wpi.veniceb08.framework.venice.mdo.PublicArt> paList =
   pam.findByProximity(new Coordinate(longitude, latitude), radius);

for (edu.wpi.veniceb08.framework.venice.mdo.PublicArt someArt : paList)
{
   net.wpi.res.slocketrocket.SimpleMDO convArt = new
net.wpi.res.slocketrocket.SimpleMDO();
   convArt.setId(BigInteger.valueOf(someArt.getId()));

convArt.setCoordinateLatitude(BigDecimal.valueOf(someArt.getCoordinate().getLatitude())
);

convArt.setCoordinateLongitude(BigDecimal.valueOf(someArt.getCoordinate().getLongitude(
)));

   returnList.getSimpleMDO().add(convArt);
}
```

Code Listing 5

The code in Code Listing 5 uses a Public Art manager (PAM), to retrieve a list of Public Art objects. It then uses a generated class, SimpleMDO, which only contains coordinates and an id, and writes it to a list that is later returned by the Web service.

After the Web service is written, a client needs to be written in order to communicate with the service. This can be done in almost any language, since SOAP is a very widespread Web service standard. We implemented the Web client in Java, since it was convenient for us to again use JAX-WS. When provided a WSDL, JAX-WS can generate classes used to receive SOAP messages. A servlet can then handle requests to and from the service. The code in Code Listing 6 makes a request to the distance operation that was shown previously and returns that information as an array of JavaScript Object Notation (JSON) objects. JSON is a format, similar to XML, that is used for easy serialization of objects into JavaScript.

```java
protected JSONArray getPublicArtCoordinatesByProximity(double radius,
    double latitude, double longitude)
    throws ServletException, IOException {

    net.wpi.res.slocketrocket.PublicArtByIdPortType port =
service.getPublicArtByIdPort();
    net.wpi.res.slocketrocket.PublicArtList paList =
            port.publicArtByDistanceOperation(radius, latitude, longitude);

    JSONArray jsonArray = new JSONArray();

    for(net.wpi.res.slocketrocket.PublicArt art : paList.getPublicArt())
    {
        JSONObject obj = JSONObject.fromObject(art);

        jsonArray.add(obj);
    }

    return jsonArray;
}
```

Code Listing 6

Maven2

In order to make the framework project portable and easy to build, one of the tools that we took advantage of was Apache's Maven. Maven is a tool that handles project dependencies by keeping a Project Object Model file that defines dependencies and different build profiles. Some of its stated goals are "making the build process easy" and "Providing a uniform build system." (What is Maven?, 2009) It can download, via the Internet, any dependencies from a central Maven repository. When building a specific target, Maven uses the idea of a life-cycle. A life-cycle is a series of goals, such as compile or test that make a dependency list. Each goal in the list needs to be completed before the next one can be executed. The notion of a life-cycle allows for the developer, especially when using plugins to attach goals to certain parts of the lifecycle. For example a GlassFish plugin needs to be executed at the end of the build life-cycle whereas JAX-WS WSDL import goal would be executed during the *generate-sources* phase. The developer would specify in the <plugin> element, an <execution> element. The execution would have a <goal> element that would specify the plugin goal to execute during the life cycle of the build process.

By using Maven for build and dependency management in the framework project, we are able to make our project much more flexible and easy to build. Any Integrated Development Environment (IDE) can work with Maven to use a project that has all of the dependencies it needs and has specific life-cycles already specified. We used the NetBeans IDE as our principal development tool, but other popular Java IDEs, such as the Eclipse platform, could develop using Maven. It is also possible to build it completely independent of any IDE as well. This makes maven since maven projects can be run on a dedicated server without having a GUI and still be able to update through SVN/CVS and be built just as easily as in an IDE.

Application Server

In order to deploy our application, we needed to make use of an application server that is capable of running J2EE applications. An application server can be defined as "a software platform that delivers content to the Web. This means that an application server interprets site traffic and constructs pages based on a dynamic content repository" (Cylogy, Inc., 2009). Popular commercial application servers include IBM's WebSphere, Oracle's OC4J and several others. For our project, we needed to find a free alternative. Free and open-source application servers do exist, including Apache's Geronimo and Sun's GlassFish Application Server. While both projects are completely free and open source, we opted to use GlassFish. GlassFish is natively integrated with NetBeans, which was the IDE that we were using to develop our project. It is also very easy to set up and use, and we had previous experience with GlassFish.

Development Environment

From the outset of the project, we tried to make the project generalized and runnable in any development environment, which helped to inform our decision to use Maven2. In our daily work, we all used the same Integrated Development Environment (IDE), the NetBeans IDE.

An IDE is software that allows for the whole range of software development. IDEs typically allow for the editing of code, the execution of code, integration with code repositories and a host of other functions. The NetBeans IDE is an open source IDE built in Java that has all of the features of a modern IDE. We chose to use this IDE as we were all familiar with its use.

Development Cycles

Over the course of the seven weeks that we spent in Venice, we followed an iterative development schedule. The idea of the short iterative development cycle is a planning concept that has become popular in the Software Engineering world recently due to the popularity of Agile software development (Manifesto for Agile Software Development, 2001). We used WPI's SourceForge tool to create tracker objects of tasks that needed to be completed. These tasks were then ranked by importance and assigned work units, which describe the relative amount of work needed to complete the task by a member of the team. Because we were developing our Framework for WPI's Venice Project Center, Professor Fabio Carrera was the customer for our project. We worked with Professor Carrera to pick the tracker objects that he wanted worked on each week. After the trackers were picked for each cycle, we would break up the work, assigning different trackers to members of our team. As the week progressed, we would close trackers that we finished. If trackers were not finished by the end of the week, we would adjust the work units and allow it to be assigned for the next iteration. Using iterative development helped keep our team on schedule to deliver our product at the end of the seven week term. While there were some trackers that did not get finished, the Framework has been developed to a state where it is working and will be useful for future generations of developers working in Venice or any city-based project center.

In addition to iterative development, we made use of other Agile techniques. When working on tasks, we often took advantage of pair programming. Pair programming is a

technique of collaborative development where two developers work together at the same computer. This helps catch errors that a single programmer might not catch. It also becomes easier to work through problems since there is always another perspective on any situation. While we did make use of this technique, we did not adhere to it strictly, since, for some tasks, it made more sense to work individually. We also saw the need to develop tests for everything at the framework level. This was especially useful when changes were made to our framework. Instead of having to deploy everything to test for bugs, we were able to run a few short JUnit test cases that could tell us if our changes broke anything that worked previously.

Proof of Concept

After creating the base framework, we needed to create a proof of concept to run our project through its paces. A two-layer proof of concept was chosen. The first layer is our SOAP Web service, as previously described. This service simply exposes some of the functionality of our framework so that any remote application can send requests and get back data about Venice. The other layer of our proof of concept is a Java client and servlet that makes requests to our Web service and provides AJAX support to a front end application. This front-end application utilizes the Google Maps API and displays all points of public art within a chosen radius, with close points grouped together.

Figure 5: Proof of Concept

Results and Analysis

Overall, our project was successful. Before starting on the project, our group spent two terms working with Professors Carrera and Pollice to determine what our project would consist of and what would provide the most benefit to the Venice Project Center. While many ideas were thrown around, we all agreed that there needed to be a consistent way for Web applications to have access to data that had been collected. After the main goals of the project were decided, we put together a plan and outlined the architecture that we would develop. Early on, we knew that we would have to stick to a strict iterative cycle if we were going to accomplish everything that we wanted to get done in one term. This early planning time was helpful and allowed us to get a development server up and running, as well as making sure everyone was on the same page in regards to what we were going to work on.

Early on, we did encounter a few setbacks. During the first week in Venice, we did not have any reliable Internet access. We used this time to plan and set up all of the trackers that we wanted to accomplish during the rest of the term. After settling into our development cycle, we were able to start closing trackers and easily caught up to where we needed to be. Throughout the project, we worked with Professor Carrera, who acted as our customer, and kept a consistent iterative development cycle. In the end, we finished the core framework, the Web services, the proof of concept application and the necessary documentation. The work that we did met all of our own expectations and goals.

Our project was received positively. We spent time working with Professor Carrera and attempted to deliver something that would ultimately be useful to him and to the project center. Overall, he seemed pleased with the parts of the project which he could easily evaluate,

especially the front-end proof of concept application. We also presented an overview of our work in a presentation to UNESCO, and the presentation seemed to be well received. Our prototypical application was also successful. It effectively and efficiently used the Web services that we wrote and deployed and it also took advantage of AJAX on the client side, something that Professor Carrera was a big proponent of. The application is reliable and is a good example of what can be achieved using our framework to develop Venice-based Web applications.

The code that we wrote was separated into three main modules in our project, the core framework, the Web services and the client application. The core framework has over 1300 lines of code, and the other two modules both have over 600 lines of code. This does not take into account all of the JavaScript, XHTML and XML that was written in addition to the Java files. Tests for the core framework were written using DbUnit and JUnit and cover almost all of the core framework code.

The best way to measure any project's success is to look at the goals put forth at the beginning of the project and to see how fully they were accomplished. In our own project, we met all of the goals we set out to accomplish and did it within the time-frame that we were given. We also created something that will hopefully be useful to future generations of WPI students who are going to be working at the Venice Project Center.

Future Work and Conclusions

At the end of our project term, we finished all of the components that result in a working framework that we had planned for. Despite this success, there is still a great deal of work that can be done on the Venice Framework. Both expansions of the Framework and tools that work synchronously would benefit the work that we have done.

While we completed the core framework level of our project, use of any new MDO subtypes would require expansion of our framework by a programmer. This is something that we have documented, but it still requires technical knowledge. Adding MDO subtypes can be done by writing Java code, annotated with Hibernate annotations. We have written a PHP generator script that will help automate this process. Alternatively, a tool called hbm2java exists and can take an XML Hibernate file defining some type and convert it to a useable Java file that is already properly annotated.

An extension to our framework that would be helpful for anyone using our framework would be the ability to dynamically add MDO subtypes to the framework without having to recompile everything. While Java is not often thought to be a very dynamic language, this is possible to do and would be an interesting area of future work. Another useful extension upon our work would be an administration console for the framework. The ability to manage dynamically loaded types and specify different database locations without having to touch the code would go a long ways to making the framework more user-friendly. Also, the inclusion of tools, both local and as a Web interface, that could help upload data, would be a helpful addition to our framework. There are tools that allow for data uploading to servers, but one that was custom-tailored to our system would be much easier to use.

Beyond additions to our framework, there is a great deal of work that could be done that could utilize the framework that we put together. While the possibilities are almost limitless, there are some possible applications that would be beneficial to put together in future projects. One possible application of our framework would be for an interactive tourist information system for the city of Venice. Such an application could provide tourists with information on sites near where they are staying, or it could recommend paths on which to tour. Other applications could involve city planning information. Also, our work could be tied together with work on the LOUIS system that had been done earlier.

Due to our work, the Venice Project Center now has a solid framework off of which many Web applications can be developed. We have put together all of the documentation needed to get started expanding and using our work, and we have provided the structure of a database that can be used to centralize the collected data. The data that has been collected can now be shared with application developers around the world and can be put to good use. Ultimately, our framework can be used as a foundation for future MQP students who can expand and utilize our work to better their experiences in Venice and provide value to the Venice Project Center and WPI.

Bibliography

(2008). Retrieved 26 February, 2009, from Pitney Bowes MapInfo: http://www.mapinfo.com/

About DbUnit. (2009). Retrieved February 27, 2009, from DbUnit: http://www.dbunit.org/

Brogden, W. (2006, October 31). *SearchSOA*. Retrieved November 4, 2008, from http://searchsoa.techtarget.com/tip/0,289483,sid26_gci1227190,00.html

Comittee, F. G. (2007, February 20). *National Spatial Data Infrastructure*. Retrieved February 26, 2009, from Federal Geographic Data Comittee: http://www.fgdc.gov/nsdi/nsdi.html

Cylogy, Inc. (2009). *Glossary*. Retrieved February 27, 2009, from Cylogy, Inc.: www.cylogy.com/library/glossary.html

ENSPIRE. (2009). Retrieved February 26, 2009, from ENSPIRE: http://inspire.jrc.ec.europa.eu/

Fowler, M. (2007, January 2). *Mocks Aren't Stubs*. Retrieved February 27, 2009, from martinfowler.com: http://martinfowler.com/articles/mocksArentStubs.html#TheDifferenceBetweenMocksAndStubs

Fowler, M. (n.d.). *P of EAA: Active Record*. Retrieved February 27, 2009, from martinfowler.com: http://martinfowler.com/eaaCatalog/activeRecord.html

Frietag, P. (2005, August 3). *REST vs SOAP Web Services*. Retrieved November 4, 2008, from petefrietag.com: http://www.petefreitag.com/item/431.cfm

Google. (2009). *Google Maps API*. Retrieved February 27, 2009, from Google Code: http://code.google.com/apis/maps/

Hibernate. (2006). Retrieved February 27, 2009, from hibernate.org: http://hibernate.org/

HSQLDB. (2009). Retrieved February 27, 2009, from HSQLDB: http://hsqldb.org/

InterSystems Caché. (2009). Retrieved February 26, 2009, from InterSystems: http://www.intersystems.com/cache/index.html

JAX-WS Reference Implementation. (2009). Retrieved February 27, 2009, from GlassFish: https://jax-ws.dev.java.net/

Manifesto for Agile Software Development. (2001). Retrieved February 27, 2009, from Manifesto for Agile Software Development: http://agilemanifesto.org/

Microsoft. (2009). *Unit Testing*. Retrieved February 27, 2009, from MSDN: http://msdn.microsoft.com/en-us/library/aa292197(VS.71).aspx

OGC. (2009). Retrieved February 26, 2009, from OGC: http://www.opengeospatial.org/

OpenLayers. (2009). Retrieved February 27, 2009, from OpenLayers: http://openlayers.org/

Sun Microsystems. (2002). *Core J2EE Patterns - Data Access Object*. Retrieved February 27, 2009, from Sun Developer Network: http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

W3C. (2004, February 11). *Web Service Glossary*. Retrieved February 27, 2009, from W3C: http://www.w3.org/TR/ws-gloss/

What Is GIS? (n.d.). Retrieved 26 February, 2009, from GIS.com: http://www.gis.com/whatisgis/index.html

What is Maven? (2009). Retrieved February 27, 2009, from Maven: http://maven.apache.org/what-is-maven.html

Appendices

Appendix A

REST, SOAP and Their Application to the Venice Framework (Blanchette, Cheng, Clayton, Fyles)

REST and SOAP are often pitted against each other in the realm of Web service technologies, but in practice, they both have legitimate application. A RESTful Web Service has "key resources" that each have a URI (Uniform Resource Indicator). In William Brogden's article on SOAP and REST (Brogden), he describes the origins of REST. REST was based on the idea that a Web service should not piggy back on a protocol, but rather adheres to the standards of one. In this way, REST states only four commands for its resources: GET, POST, PUT, and DELETE. These commands follow closely with the common HTTP commands. Brogden stresses that REST does not have a specific implementation standard or specification, it is a style.

SOAP has fewer services, but many more commands, or methods, available for each service. These services do not represent resources like in the RESTful situation. They do each have their own interface. Services are provided by messaging using XML over HTTP. In the scope of the Web service that will be built as part of a proof of concept of the Venice Municipal Data Framework, REST and SOAP each have their advantages and disadvantages.

The whole decision really comes down to whether a Service-Oriented or a Resource-Oriented design is needed. Neither is better or worse because they solve different problems. REST works best for a Resource-Oriented problem, and SOAP works best for a Service-Oriented problem. In our case, we are dealing with providing a service that gives access to our Municipal Data. The Municipal Data could be represented as resources instead, but querying the data would be much harder, and we have complex operations that need to be done involving the

data. Therefore, a Service-oriented solution makes more sense. SOAP would therefore benefit the Venice Framework best because using REST would mean that each object exposed by our service could only have four methods (the HTTP CRUD ones) associated with it. This would make doing more complex queries to our Framework difficult. SOAP, on the other hand, could expose all of the services of our Framework, and queries could easily be performed to return all of the objects that match, serialized in XML.

According to Pete Frietag (Frietag), REST is lighter than SOAP and has human-readable results, which he perceives as a benefit. SOAP, he says, can be easier to consume, and has stricter typing. In the scope of the Venice Framework, the strict typing could actually be more of a drawback, since it would require a new Web service every time a new subclass of MDO is created. However, since SOAP does not force a URI standard it allows greater manipulation and customization of the data sent or received. Amit Asaravala wrote in his article 'Giving SOAP a REST', "large amounts of data … can quickly become cumbersome or even out of bounds within a URI." Since our data might include large numbers of data points and related information, a RESTful style might not be appropriate. If another Web service for strictly listing and manipulating single data points or groups of points pivoted on one single common variable were to be created using the Venice Framework, perhaps a more RESTful approach would be appropriate and its benefits could be realized. As such, it is not unreasonable that a long term goal is to have two separate Web service layers built on the framework: one using REST and one utilizing SOAP. Either way, Frietag put it best, "Which ever architecture you choose make sure it's easy for developers to access it, and well documented."

Works Cited

Asaravala, Amit. Giving SOAP a REST. 21 October 2002. 4 November 2008 <http://www.devx.com/DevX/Article/8155>.

Brogden, William. SearchSOA. 31 October 2006. 4 November 2008 <http://searchsoa.techtarget.com/tip/0,289483,sid26_gci1227190,00.html>.

Frietag, Pete. REST vs SOAP Web Services. 3 August 2005. 4 November 2008 <http://www.petefreitag.com/item/431.cfm>.

Heaton, Ryan. Why ROI vs. SOI (read "REST vs. SOAP") Still Matters. 9 October 2007. 4 November 2008 <http://Weblogs.java.net/blog/stoicflame/archive/2007/10/why_roi_vs_soi_1.html>.

Ruby, Sam. REST + SOAP. 2 July 2002. 4 November 2008 <http://www.intertwingly.net/stories/2002/07/20/restSoap.html>.

Tilkov, Stefan. REST vs. SOAP (Oh No, Not Again). 30 June 2006. 4 November 2008 <http://www.innoq.com/blog/st/2006/06/30/rest_vs_soap_oh_no_not_again.html>.

Tomayko, Ryan. How I Explained REST to My Wife. 12 December 2004. 5 November 2008 <http://tomayko.com/writings/rest-to-my-wife>.

Appendix B

The Broker Pattern (Fyles)

Pattern Description

The Broker Pattern helps to decouple components in a distributed system. The pattern encompasses a *broker* that acts as a communication venue between the respective *proxies* of *clients* and *servers*.

Often times, a piece of software is required to access a service already provided by another piece of software. Other times, two distinct pieces of software will need to use a common set of services or data. In both of these cases, the Broker Pattern could be used to provide a constant server-side service engine for these clients, and a way to effectively communicate with it. In both of these situations, the broker pattern encapsulates all of the communication between the client and the server with a set of proxies and a broker.

The Broker Pattern provides one of the most literal examples of reusability available in software development. A server can provide a service (or multiple services) that can be used seamlessly by any number of clients.

The client benefits greatly from this pattern as well. Any number of servers with any number of unique services can register with the broker, and the client can access any or all of the services on any or all of the servers without needing to individually send them information, and without knowing which servers provide which services.

The Broker Pattern is often used in the implementation of Web services, cloud/cluster computing, and is an integral part of a service-oriented architecture.

Pattern Design

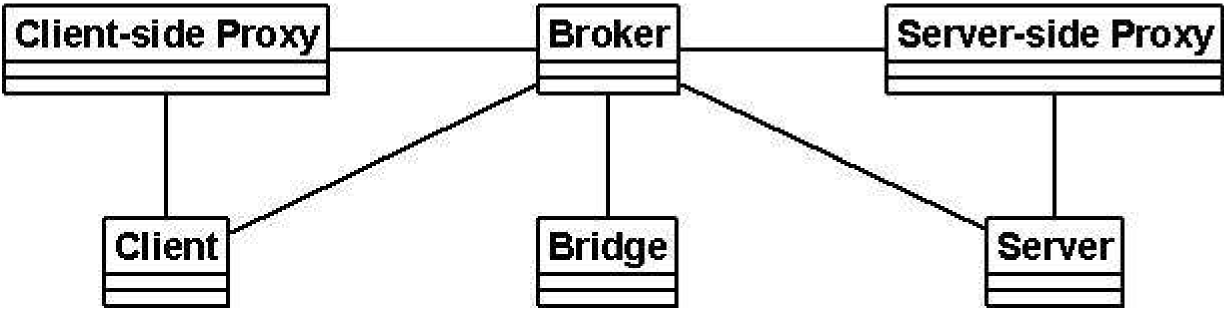


Figure 4: **Basic component diagram of the Broker Pattern. The connecting lines represent collaborators, and do not necessarily imply any dependency or call.**

Components: The Broker Pattern consists of 6 basic components:

Broker: The broker is the central component of the Broker Pattern. It is essentially the messenger between the client and server. The client and server do not make calls to the broker directly, but rather communicate with their respective proxy, which in turn transfer messages to the broker.

The broker is responsible for communicating requests and responses between the client and server. The broker also keeps track of the servers for which it is responsible for facilitating communication. A new server is required to register with the broker to alert the broker of the server's presence.

The broker also offers APIs for use by the servers and clients. These APIs allow the servers to register with the broker, and allow the clients to communicate properly with the methods that the servers offer.

Server: The server is responsible for implementing the services that the client side will have the ability to communicate with. It receives the client's requests, and sends response communication through the server-side proxy. A server is responsible for registering itself with the Broker.

Server-side Proxy: The server-side proxy is a middle man between the server and its communications with the broker. If a client requests a service from the server, the server will accept the request from the broker, adapt it for use with the server, and invoke the proper services in the server. When the server returns some result or exception in response to a request from the client-side, it will travel through the server-side proxy, which will package it up the response and send that to the broker, which will ensure that it gets returned to the proper client.

Client: The client is the user's end of the software. It implements all the functionality that will be available to the end user. It sends requests through the client-side proxy to the broker, which in turn routes the request to the proper server. In this sense, the client is the mash-up result of the client functionality and functionality that is implemented via the server-side.

Client-side Proxy: The client-side proxy, similar to the server-side proxy, acts as a layer of transparency between the client and the broker. Like the server-side proxy, this proxy pulls out of the client the responsibility of adapting the data for use by the client, and all functionality regarding data transfer.

Bridge: The bridge allows for inter-broker communication. This is not needed if transferring data over an existing protocol, such as HTTP.

Flow of Data

Figure 2 demonstrates the sequence of events that would occur if two servers registered with a single broker, and a single client accesses a service in each of the brokers. This sequence will omit the bridge, since we can assume that all of the data transfer is done using HTTP.

In this example, the two service calls are on two separate servers. Since they are each registered with the broker that the client calls, they appear to the client as if they both in a single location and local. This level of transparency is provided by the logic in the broker.

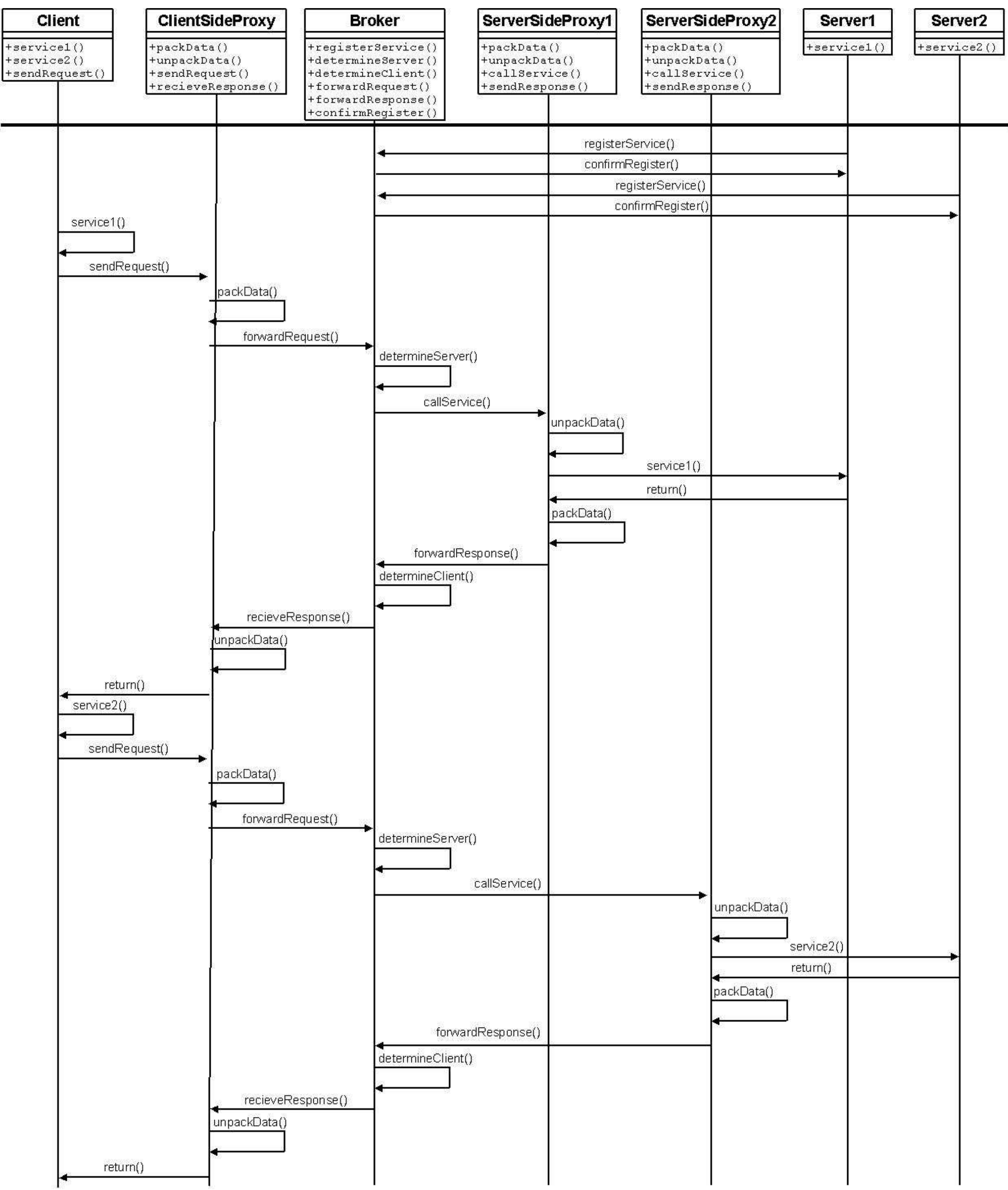


Figure 2: Basic data flow of an implementation of the Broker Pattern. This diagram does not conform strictly to the UML sequence diagram standards, and the components at the top are not necessarily single classes.

Variations: There are many popular implementations of the Broker Pattern. As previously mentioned the most oft-encountered contexts of its use are in distributed

computing and Web services. Regardless of the milieu however, the Broker Pattern is generally met in one of two forms. The first of these forms uses indirect communication between the client and the server sides of the software. Using this design, the client and server proxies send and receive all communications through the broker. The other kind of Broker Pattern implementation uses direct communication, under which the broker connects the client side to the server-side only on the first call from that client to that server. From that point on, the client-side proxy communicates directly with the server-side proxy. Though this method provides a bit of a speed boost, it requires that the client and server sides communicate using the same protocol.

Example

The city government of Venice, Italy wants to create a Web application that maps routes around the city of Venice with regards to several municipal objects in the city, such as public art and bridges. This software will provide a route between two points that the user specifies, and display all of the public art that the user will pass on the route, and all of the bridges that they will cross. The city government has already collected all of the data and put it into a database located on a high-powered server. The initial plan that they came up with was to design a Web application on top of this data using a basic view-data model (Figure 3). In this design, MapView is a Web application that interfaces with the VeniceServer data retrieval application directly to access the prized VeniceData.

Figure 3: The initial design of the Venice mapping Web application. MapView represents the Web-based mapping application front end, and VeniceServer represents the data model that will be used by MapView. The arrows signify the flow of information, not dependencies or relationships.

After the first few development iterations, the city government decided that they wanted to expand the functionality of MapView to allow users to rate the public art. The idea of adding rating functionality into the VeniceData database was initially discussed, but later thrown out because the user rating system was not relevant to the actual VeniceData, but rather an aggregate of public opinion. Therefore, they decided to make another system called RatingServer. This updated design is shown in Figure 4.

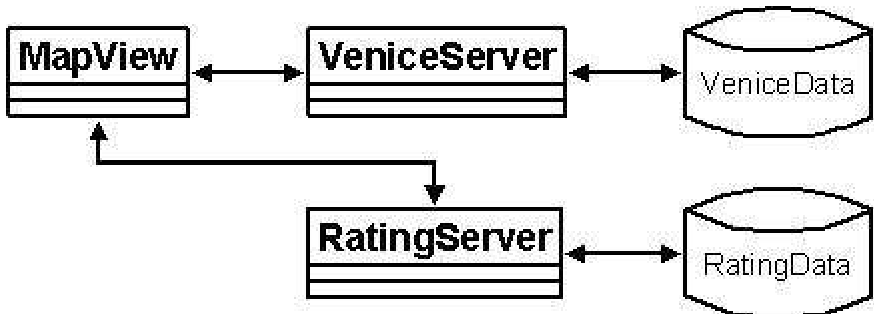


Figure 4: The revamped design of the VeniceServer application framework.

The design never got off the drawing board however, due to some significant drawbacks. The most significant downside was that the MapView application itself was managing the connections to each of the servers when it needed to get data from either of them. The second was that the functionality of MapView had to directly implement the different data transfer protocols used by RatingServer and VeniceServer.

A redesign was ordered, and the software engineers decided to use the Broker Pattern to fix the design and to make it significantly more flexible. The result is shown in Figure 5.

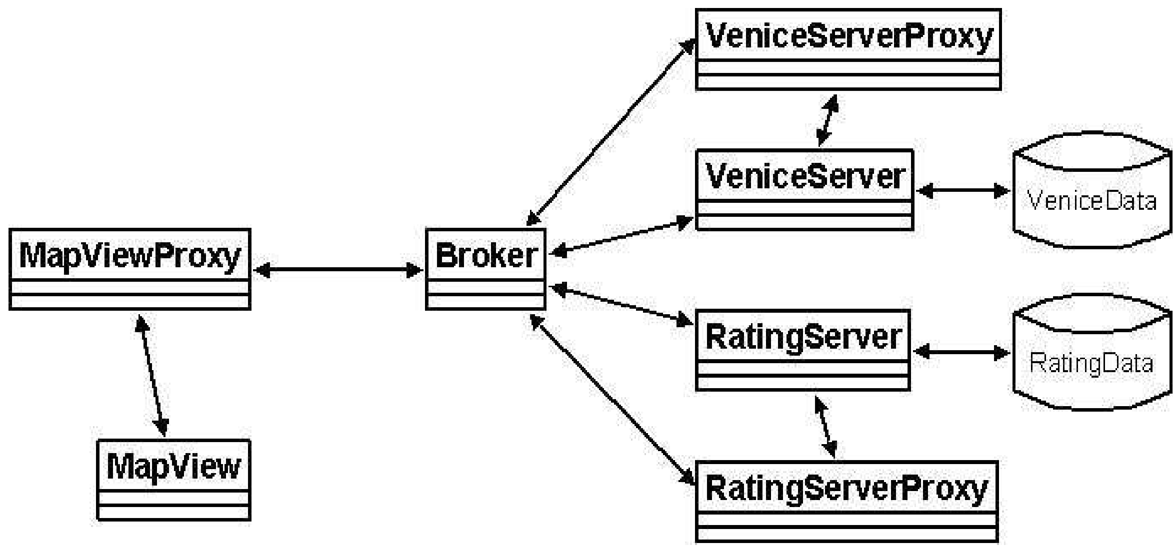


Figure 5: The Broker Pattern applied to the MapView application.

The advantages to the new design were immediately apparent. The broker ensured that the VeniceServer and RatingServer appeared as one entity, and the MapViewProxy ensured that the servers and the services they provided appeared nearly as local data to the MapView component.

Soon after the new design was set to be implemented, the Venice officials were contacted by a company who claimed to have a database of reviews of the public art in Venice. Upon further inspection, they found that not only did this company have all the data they claimed, but they had a service running on a server to allow remote access to it. Though it received requests through its proxy using a different protocol, the switch from RatingServer to ExternalRatingServer was not difficult to make, and did not require any changes to the MapView code.

Appendix C: Maven2 Tutorials

Creating the Framework with Maven

1. Install the maven2 into the command line by following directions here http://maven.apache.org/download.html
2. Download the NetBeans 6.5 or newer. Side Note: There are numerous bugs with the embedded maven2 that NetBeans 6.1 or lower uses, down the line it will be extremely confusing to debug it.
3. Install the maven2 plugin for NetBeans
4. Open the command line and navigate to the directory where you want to create your project (e.g. C:\MyProjects\)
5. Create a maven2 with the quick start archetype command. Follow direction here http://maven.apache.org/guides/getting-started/index.html#How_do_I_make_my_first_Maven_project
6. Import the project into NetBeans by opening the project as a maven2 project.
7. Add the maven compiler plugin by following the directions on their website http://maven.apache.org/plugins/maven-compiler-plugin/usage.html
8. Add the following dependencies in this order (groupId, artifactID, version) by following instructions here

   http://maven.apache.org/guides/getting-started/index.html#How_do_I_use_external_dependencies

   - (org.hibernate, hibernate, 3.2.6.ga)
   - (org.hibernate, hibernate-annotations, 3.3.1.GA)
   - (mysql, mysql-connector-java, 5.1.6) [This may vary depending on which DB you choose]
9. Create the hibernate.cfg.xml file in "src/main/resources" with the follwing...

```
<hibernate-configuration>
<session-factory>
<propertyname="connection.url">jdbc:mysql://hobonator/hibernate</property>
<propertyname="connection.username">root</property>
<propertyname="connection.password">root</property>
<propertyname="connection.driver_class">com.mysql.jdbc.Driver</property>
<propertyname="dialect">org.hibernate.dialect.MySQLDialect</property>

<propertyname="show_sql">true</property>

<propertyname="format_sql">true</property>
<propertyname="hbm2ddl.auto">create</property>
```

```
<!-- JDBC connection pool (use the built-in) -->
<propertyname="connection.pool_size">1</property>
<propertyname="current_session_context_class">thread</property>
```

1. </session-factory>Change the connections settings to your MySQL server settings
2. Add HibernateUtil class like the one shown in the Hibernate tutorial here http://www.hibernate.org/hib_docs/annotations/reference/en/html/ch01.html
3. Add classes with Hibernate Annotations programmatically using the AnnotationConfiguration class also in the Hibernate tutorial
4. Run the project, you should see the table and rows have been created in your mySQL DB.
5. Run "**mvneclipse:eclipse**" in your project folder (assuming that you have maven2 installed correctly in your commandline)
6. Install the eclipse maven plugin from here http://maven.apache.org/eclipse-plugin.html
7. Import the project with eclipse
8. Run the project with eclipse

Integrating DBUnit for DB Testing

1. Backup your database by dumping it to a *.sql file or the db file of your choice depending on the dbms that is being used
2. Get a random sample of your data. Our suggestion for MySQL users is to grab a couple hundred random tuples to test on using the following method (other DBMS users can adapt to their db language)
    a. Order (Total Number of Tuples – Desired Sample Number of Tuples) randomly and delete them. This will result in the desired sample number of Tuples.
    b. This is a concrete example of the MySQL way of doing it, assuming Total Number of Tuples = 1000 and Desired Sample Number of Tuples = 150

    ```
    DELETEFROMMunicipalDataObjectORDERBYRAND(10)LIMIT850
    ```

    c. This leaves 150 random tuples to use as sample data. There RAND() function has a seed so that if the table has foreign keys from a referenced table, it can also be used on those tables with the same effect
3. Create a main class called **ExportSampleData**. Inside the class, follow this small sample code on the dbunit site. http://dbunit.sourceforge.net/faq.html#extract

4. Run the method/class you implemented, take the **sampleData.xml** and copy it to your Test resource folder (e.g. src/test/resources/) for testing later
5. Create an abstract test case class, in this case it is called **VeniceTestCase**. Inside the test case follow the tutorial on the dbunit website http://dbunit.sourceforge.net/howto.html
6. For the getDataSet() method, use sampleData.xml to point to the exported sample data
7. Once this is implemented, the future framework developers need only to extend the **VeniceTestCase** without having to deal with the intricacies of a DBUnit when writing a new TestCase class.
8. In order to use the Assert in JUnit. Import JUnit and call them by using the Singleton Assert class to call Assert.assertEquals(), Assert.assertTrue(), Assert.assertNull(), etc…

Creating a Parent maven2 Project to Share Dependencies

1. Create a root folder to hold all the project folders in, e.g. **MyProject**
2. Create a pom.xml in **MyProject**with the following

```
<?xmlversion="1.0"encoding="UTF-8"?>
<projectxmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
            http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>edu.wpi.myproject</groupId>
<version>1.0-SNAPSHOT</version>
<artifactId>MyProject</artifactId>
<packaging>pom</packaging>
<name>MyProject</name>
<modules>
<module>MyFramework</module>
<module>MyWebService</module>
</modules>
</project>
```

3. Adapt the above to your project. Add all your projects as modules
4. Open/Import the project in NetBeans
5. Go to each of your project's root folder and open their pom.xml and add this to the top them

```
<parent>
<groupId>edu.wpi.myproject</groupId>
<artifactId>MyProject</artifactId>
```

```
<version>1.0-SNAPSHOT</version>
</parent>
```

6. If one of your projects needs the jar/war of the other, simply list them as a dependency in their pom.xml. In the following example the MyWebService project needs the jar for the MyFramework listed as a dependency

```
<dependencies>
...
<dependency>
<groupId>edu.wpi.myproject</groupId>
<artifactId>MyFramework</artifactId>
<version>1.0-SNAPSHOT</version>
</dependency>
...
</dependencies>
```

7. If you are building the projects individually, the dependency of the project must be built first. If you wish to build them automatically in the correct order, clean and build in the root project, **MyProject**. This will build the dependencies first, then the projects that depend on them.
8. This is called aggregation, more information can be found here http://maven.apache.org/guides/introduction/introduction-to-the-

   pom.html#Project_Aggregation

Porting the WebService to maven2

1. Create a new maven project using the quick start archetype. http://maven.apache.org/guides/getting-started/index.html#How_do_I_make_my_first_Maven_project
2. The tutorial assumes you have set up your framework to share its dependencies as well as its jar. Follow the part of the tutorial where you add a <parent> element to a child project.
3. Add the VeniceFramework as a dependency in the pom.xml
4. Open/Import the project into NetBeans 6.5+
5. Change the packaging from **jar** to **war** inside the pom.xml file
6. Since our WebServices uses JAX-WS, in order to also leverage this in a maven2 project, there is a need to use the JAX-WS maven2 plugin. Follow the directions on their website to include the plugin and its dependencies https://jax-ws-commons.dev.java.net/jaxws-maven-plugin/usage.html

7. Create a "webapp" folder in the src/ folder.
8. Move the index.jsp and the WEB-INF folder into the newly created webapp folder
9. Create a "wsdl" folder inside the WEB-INF folder that was just moved
10. Move the WSDL files and its parent folder from the old project inside the new wsdl folder
11. Inside the pom.xml, specify at least the following parameters…
    a. wsdlDirectory (e.g. ${basedir}/src/main/webapp/WEB-INF/wsdl/ServiceImpl)
    b. packageName (e.g. edu.wpi.veniceb08.webservice)
    c. sourceDestDir (e.g. $(basedir)src/main/java/
12. Build the project and make sure it does not fail
13. Now that it can be built successfully, there are two choices to deploy the WebService. The first choice is to use NetBeans, integrated way of interacting with the glassfish server. The second choice is to use the glassfish maven plugin that can be found here https://maven-glassfish-plugin.dev.java.net/ . Since our entire project is done entirely in NetBeans, the following steps will outline the way to do it in NetBeans. To do it using purely maven and glassfish, read the usage page on the glassfish maven plugin website.
14. First make sure that glassfish v2 is in the selection of servers in your NetBeans. You can check this by going to Tools→Servers
15. Right-Click on your WebService project in NetBeans and go to **Properties**
16. Navigate to **Run** on the left side, select **Glassfish V2** as the server and change the **ContextPath** to something more reasonable like "/VeniceWebService" instead of "/VeniceWebService-1.0-SNAPSHOT"
17. Press OK, then right-click on your WebService and click on "Run".
18. The WebService should be deployed and the WSDL should be exposed in its respective URL.

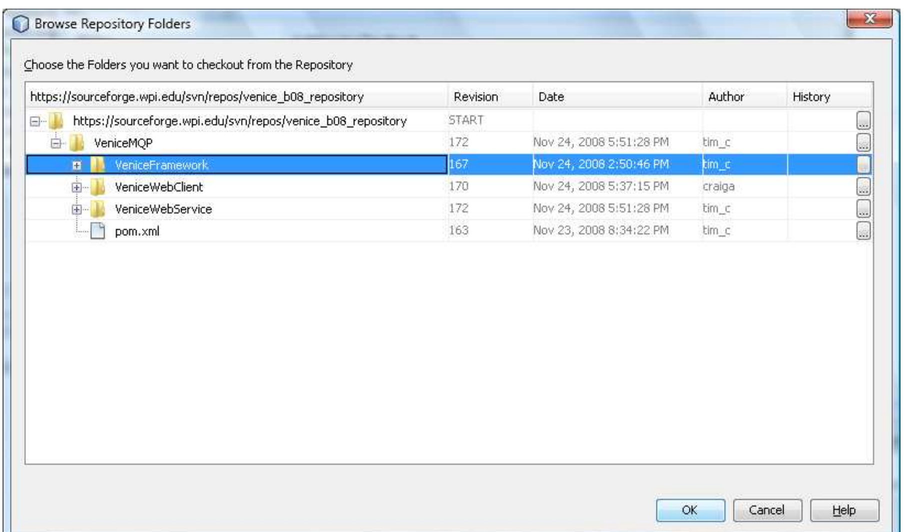
Appendix D: Developing Applications Using the Framework

The Venice Framework was conceived as a venue to allow the rapid development of Venice-specific Municipal-Data-based applications. Developing such an application now requires very little set-up work. The key to this ease of use is that files from Java applications can be aggregated into a Java Archive (JAR) that can be included into another Java project as a library, allowing the Java project to include any classes from the jar. The Venice Framework uses Maven2 as a build automation tool, with a POM file configured to create a JAR on each build.

Creating a Simple Java Application That Uses the Venice Framework JAR

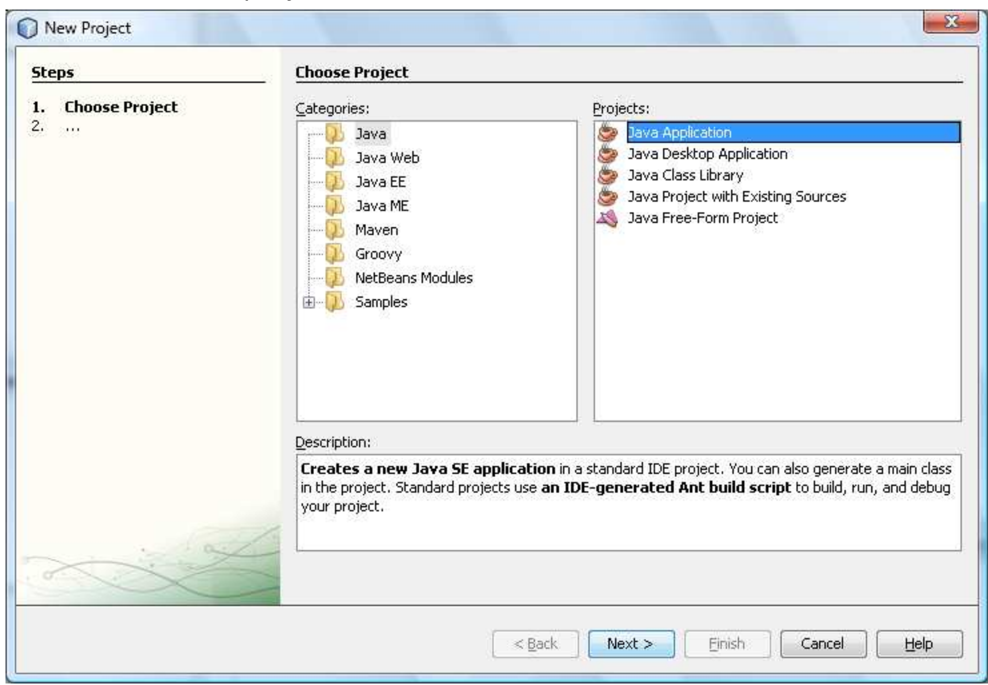
Creating an application that utilizes an external JAR is rather straightforward. An IDE further simplifies the process. In the NetBeans IDE, for example, creating the aforementioned application takes only a few steps.

1. Obtain the JAR file for the Venice Framework.
    a. Get the Maven2 project "VeniceFramework" from the Subversion repository. This will require Maven2 and Subversion plugins.

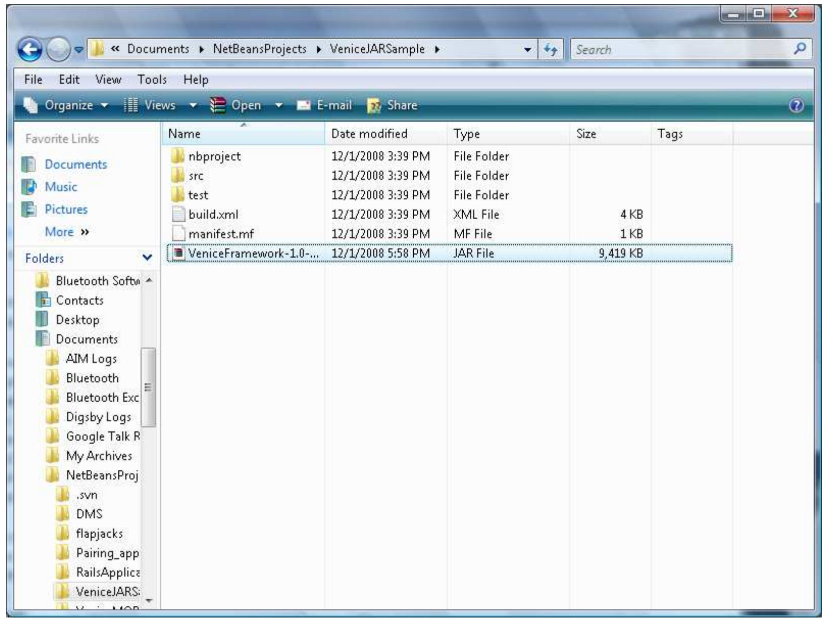


b. Ensure that the project is imported into NetBeans.
c. Open up the command prompt and navigate to the project folder.
d. Type "mvn clean:clean" and hit enter. It should display a success message when completed.
e. Type "mvn assembly:assembly" and hit enter. It should display a success message when completed.

2. Create a new Java project.

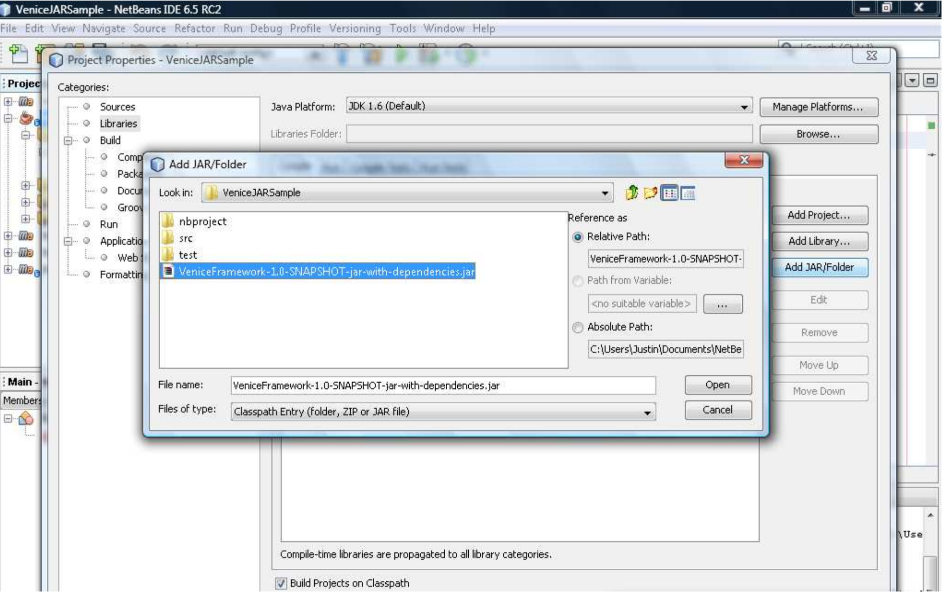


3. Include the JAR in the new project
   a. Navigate to the VeniceFramework project folder. Open the "target" folder. Copy the file "VeniceFramework-1.0-SNAPSHOT-jar-with-dependencies.jar".
   b. Navigate to the folder for the new project, and paste the JAR in the main directory.



c. In NetBeans, right-click on the new project and click “Properties”.

d. Select the “Libraries” category and click the “Add JAR/Folder” button. Navigate to the Venice Framework JAR, select it, and click “Open”. Click “Okay” on the “Project Properties” window.



Classes within the new project can now import classes from the Venice Framework.

Creating a Web Service That Uses the Venice Framework JAR

One type of application that is likely to be created using the Venice Framework is a web service. A web service is an API that can be accessed through a network, allowing one or multiple web clients access to specific server-side functionality. In the context of Municipal Data access, a web service would be a viable solution to allow web application developers limited access to specific transformations of Municipal Data. Creating a simple SOAP-based web service is streamlined by Java API for XML Web Services (JAX-WS), which is integrated into NetBeans and alternatively has a Maven2 plug-in.

The easiest way to create a JAX-WS-generated web service is using the built-in NetBeans project generator. A project generated in this way can later be converted to a Maven2 project if needed. There are several tutorials online that demonstrate how to create a web service in NetBeans. Perhaps the best one is the one on the NetBeans website[22]. To create a web service, follow the NetBeans tutorial, and insert the Venice Framework JAR into the project as per the above section of this document.

Testing the Web Service

Creating a client that consumes the web service is an obvious way to test its functionality. The NetBeans-provided tutorial demonstrates how to create such a client. Testing the web service does not require the creation of a client though. There are many tools that exist for the sole purpose of testing SOAP web services. One of the most fully-features testing utility is soapUI. Although soapUI has both NetBeans and

[22] (NetBeans, 2008)

Maven plugins, the easiest way to test a web service with soapUI is using the standalone application, which can be downloaded for free from http://www.soapui.org. There are tutorials on the site as well which show how to set up and use the tool to test web services. Using the tool will require the web service to be deployed to an application server (localhost is okay), with its WSDL made accessible.

Appendix E: Deploying the Application to an Application Server

Just as the Java Archive (JAR) file format allows easy assemblage and portability of Java applications, Java's Web Application Archive (WAR) file format allows for the easy transfer and deployment of Java-based web applications. In most IDEs, web applications will assemble a WAR file by default when the project is built. Maven2 makes this process more straightforward by also assembling the required dependencies.

Specifying the Database Server

Before assembling a web application using the framework, a custom database needs to be specified in the framework project. The default database is a MySQL server running on localhost with a user *venicemqpuser* and a database called *venicedata*. This can be changed to any database that is supported by the Java Database Connector (JDBC).

To specify a new database:

1. Open *Project Files* in the *VeniceFramework* module inside of *VeniceMQP*.
2. Open *pom.xml*
3. Edit the MySQL profile to match the credentials of your server. If you are not using MySQL, create a new profile that matches your database, changing the hibernate dialect, JDBC driver class name and URL to match up with your database.

   Note: If there are errors with JDBC accessing your database, be sure to check your grant permissions on that particular database.

Deploying the application to a server

After building your application, deploying is relatively easy. Simply install the J2EE application server of your choice (Glassfish, Tomcat, etc.). After installation, simply deploy the application using an application server manager or autodeploy folder. If you have more than one web or application server installed, there may be port conflicts, so make sure this is cleared up before deploying any applications.

Using Images

Images should be stored relatively in the framework's database, although this is not enforced. Because paths are relative, you can put images anywhere on your server so long as they can be served remotely. For instance, using Glassfish, you can put these images in a folder of the docroot of your domain.

If you want to use thumbnails for your application, we have included a bash script for Linux operating systems that will use Imagemagick (which needs to be installed first) to go through and create thumbnails of all images. It may require slight modification to fit your needs. It can also be set up as a CRON job to run in the background and make new thumbnails when new images are added.