Annals of Software Engineering 13, 13–34, 2002


# Using Component-Based Development and Web Technologies to Support a Distributed Data Management System *

M. BRIAN BLAKE † and GAIL HAMILTON {bblake; gail}@mitre.org
*The MITRE Corporation (CAASD), Center for Advanced Aviation System Development, 7515 Colshire Drive, MS N420, McLean, VA 22102, USA*

JEFFREY HOYT jchoyt@mitre.org
*The MITRE Corporation Intelligent Information Management and Exploitation, 7515 Colshire Drive, McLean, VA 22102, USA*

**Abstract.** Over recent years, "Internet-able" applications have been used to support domains where distributed functionality is essential. This flexibility is also pertinent in situations where data is collected and derived to support a distributed set of stakeholders. There are major problems in this distributed data management scenario. One problem is the change that occurs in such domains. Both the schema of the data and the individual needs of the stakeholders evolve over time. Any architecture to support this distributed data management domain must be designed to support these specific changes. One such approach to this architecture is the use of "plug-able" web-based components. As new computational needs arise, new components can be plugged into the architecture. Another aspect of this solution architecture is toward a run-time evolvable process to support the change of the data schemas. At the MITRE Corporation, this architecture has been designed, deployed and tested to support the internal need for a composite data repository. This paper presents the motivation and architecture of this distributed data management system that supports the Center for Advanced Aviation System Development (CAASD). This component-based run-time configurable architecture is implemented using web-based technologies, such as the Extensible Markup Language (XML), Java Servlets, and a relational database management system (RDBMS).

## 1. Introduction: Why a Distributed Data Management System?

With the recent advancements of computing systems, society as a whole has embraced information technology. Almost every domain has become dependent on the need for



† and Georgetown University, Department of Computer Science, 234 Reiss Science Building, Washington, DC 20057, USA. blakeb@cs.georgetown.edu

information resources. Domains ranging from finance to engineering to medical services and many more have a necessity for current and accurate data. With the increase in distributed computing, this data must also be available across multiple enterprise locations and geographical regions. At this point, you may be asking yourself, "Why not use a database management system?". It is true that database management systems (DBMSs) have advanced to handle storage, managing, querying, and concurrency handling of raw data. DBMSs also allow the data to be accessed from distributed locations. Unfortunately, the support given by DBMSs is not sufficient for domains where complex business rules and derivation must occur prior to making the information available to the enterprise-wide stakeholders. In these instances, we suggest a distributed data management system that couples the benefits of a DBMS with the flexibility of a run-time evolvable component-based support architecture.

### *1.1. The DBMS solution*

The idea of web-accessible databases is not a new concept. Currently, most databases have tools and architectures to support the development of web-based graphical user interfaces (GUIs) that can access their DBMS (Oracle's WebDB) [Oracle Corporation 2001]. This architecture is depicted in figure 1.

As in figure 1, the DBMS would supply an interface generation tool that gives the user the option of creating their own GUI. This tool presents the actual schema information and the user would configure a GUI depending on his/her needs. This is a scaleable solution because as the database schema changes, the user can develop new GUIs to support those changes. The DBMS also allows the development of database functions or packages that can be used to derive raw data. These functions run internal to the database server. Moreover, these systems support the return of multiple data formats (i.e., HTML, Delimited text, XML, etc.).

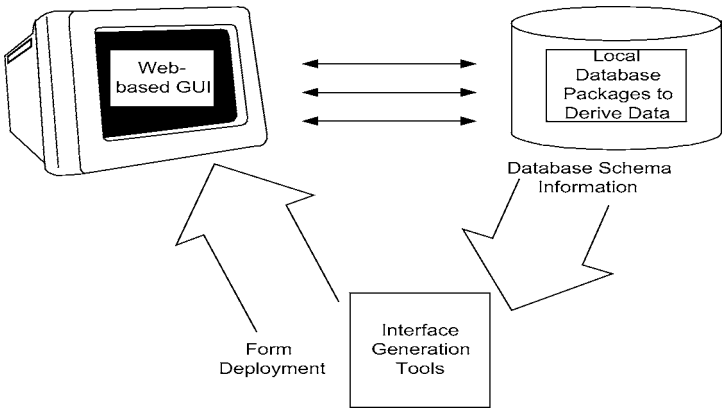


Figure 1. Web-accessible DBMS architecture.

### *1.2. Problems with the DBMS solution*

At the MITRE Corporation-Center for Advanced Aviation System Development (CAASD), we have identified reasons why this architecture, though extensible, is not sufficient for our distributed data needs. The following limitations were experienced in using this generic DBMS solution:

1. *Internal database packages and functions cannot support the rigor of the data derivations*. Some data requests are for many megabytes of information that must undergo a rigorous process to derive original data into more useful information. Using packages to perform these derivations can greatly decrease the performance of the DBMS when multiple users are requesting large database jobs.

2. *Additional "custom" output formats are needed for stakeholders*. Some enterprise stakeholders need output in formats that are specific to their custom applications. Although the notion of a standard data representation is agreeable to all, this notion is not the current reality. Our domain must support the past, present, and future data formats whether they are standard formats (i.e., HTML, XML, delimited text, etc.) or custom user-specified formats.

3. *Some data requests require the dynamic generation of multiple queries* (*due to the introduction of business rules*). Some users require queries that cannot be generated without outside intervention. There must be some additional humans-in-the-loop to supply critical information to complete the query.

4. *Some users have applications that require direct connections to the database*. Some applications need to stream information directly into their applications. This streaming process varies for different custom applications or simulations. Current DBMS solutions do not support the need for data streaming. In addition, the streaming functionality would probably cause additional performance decreases.

### *1.3. Using a distributed data management system*

In this work, we have designed, implemented, and deployed a distributed data management system. This system uses an on-going component-based architecture to support the evolving needs of the user. This system also incorporates the benefits of the DBMS solution. A top-level view of this system is illustrated in figure 2. The Interface Layer contains tools where the user can specify the GUI/query as in the DBMS solution, but the user can also specify special component-based functions that must occur before the query string can be created. The Middle Layer contains a special component that creates a workflow based on instructions from the web-based GUI. This workflow control component uses the workflow to sequentially execute the required business components. As aforementioned, there are a set of autonomous business components that handle special functions such as the generation of dynamic queries, the incorporation of business rules, the generation of custom output formats, and data streaming to custom external

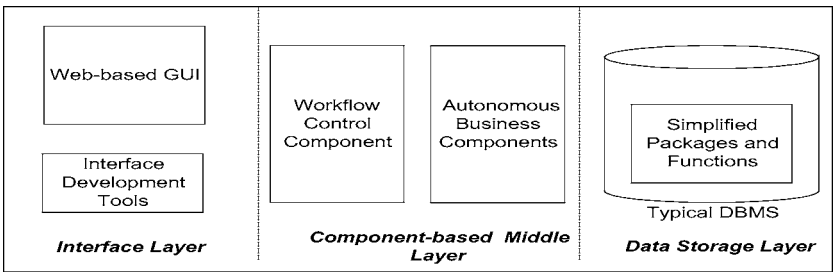


Figure 2. High-level architecture of a data management system.

applications. The Data Storage Layer is basically the DBMS that may contain internal packages for the more rudimentary formatting functions.

At the MITRE Corporation-Center for Advanced Aviation System Development (CAASD), researchers develop simulations for both design-time and real-time analysis. This research constitutes a wealth of knowledge in the area of air traffic management and control. This division of MITRE is split into a large number of individual groups that investigate various problems comprising the air traffic domain. Although the groups analyze different problems, the data to support the investigations are typically the same. Also, these individual groups develop simulations that require the data in different formats (i.e., specialized text files with delimited data, database format, XML with a custom schema, etc.). Moreover, each group looks at different subsets of data that may cross multiple data sources. Researchers are currently provided with data that is gathered from outside sources and distributed by a data librarian. This data is usually distributed in the same media and format in which it is obtained. The CAASD Repository System (CRS) team at MITRE has identified the need for obtaining the desired raw data from outside sources and building a composite data repository that serves the need of this diverse user community [Blake 2000; Blake and Liguori 2001a, b]. The architecture depicted in this article was implemented to serve the needs in this problem domain.

### *1.4. Related research*

In addition to the rudimentary tool support provided by the aforementioned relational database applications, there are several related research projects that are working towards the same end. The Zelig project introduces a schema that can be coupled with HTML to control various CGI-based database executables [Varela and Hayes 1994]. The main difference is that the Zelig project still appears to tightly-couple the interface to the database by hard-coding entire query strings into the HTML documents. The CRS architecture extends this approach by incorporating specific query-building knowledge in the middle-level components. The interface merely has components of the query, while the middle-level components encapsulate the generic intelligence to build query strings. In addition, the CRS architecture has the flexibility to allow additional business-specific components to be plugged in to further format and process the database results.

Moreover, the CRS implementation makes use of the later, more flexible technologies of XML [Weiss 1999; Glushko *et al.* 1999] and Java Servlets [Sun Microsystems 2002].

Another project that is even closer to the CRS architecture is the WebInTool [Hu *et al.* 1996; WebInTool 1997]. In the WebInTool, Hu [1996] specifies a web to database interface building tool. Hu [1996] takes a similar approach as the CRS architecture in promoting a separation of interface and back-end source code. Hu [1996] uses several CGI-based modules but similar to the Zelig project, there is no intelligent query building knowledge in these modules. In the absence of this feature, the WebInTool does not have the same ability as the CRS implementation to dynamically create a large range of queries. In addition, the CRS architecture with the use of later technologies is able to support a larger range of output choices with respect to formatting. Finally, since query strings are not incorporated in the interface, users that understand the databases can build new forms without having intimate knowledge of the source code in the middle-layer components. Users of the WebInTool must have knowledge of both the software (CGI) modules and of the underlying database schema.

This article will proceed by illustrating each of the aforementioned architectural layers in detail in sections 2–4. In section 5, the implementation of this architecture at the MITRE Corporation will be described and illustrated. In the final sections, there are conclusions based on nonfunctional information.

## 2. The Interface Layer

The Interface Layer contains a stand-alone module that allows the user to specify the parameters and special processes that will be used in his/her data query. This module can be accessed through the Internet, using a servlet-based implementation. When a user accesses this module, a servlet is used to present various options for creating a GUI query form. This servlet has access to the database, so the options given to the user contain database-specific information. Once of the user has completely specified his/her options, these choices are used to create an HTML file for this specific user. This HTML file contains a hidden tag with instructions for executing the business-oriented components. This hidden tag uses the Extensible Markup Language (XML) to represent the instructions. Other fields on the form also have hidden information correlating HTML-based fields to database columns. The main functionality in this module is to gather this information, combine the information with database specific information and create this HTML file.

### *2.1. HTML hidden tag and instruction fields*

The GUI query form must contain information that both describes the GUI component and describes how that component is associated with the database. This information can be represented using the name/value structure inherent to form-based HTML pages.

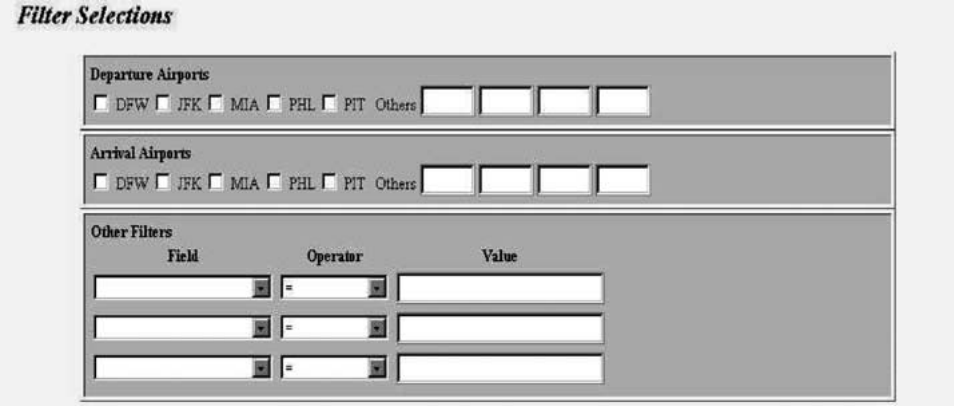


Figure 3. Screenshot of interface form.

The following HTML statement shows an example of a checkbox that is associated to a database field. This HTML code implements the PIT checkbox in figure 3.

```
<input type="checkbox" name="Filter1"
  value="Table:OooiStageData~Field:deptAirport~Value:PIT">
```

The checkbox will be used as a simple filter of data, and therefore a Filter component will be required. The name of the field contains only the type of component that is to be used. The value of the field contains the metadata describing the underlying table. The string is delimited by two characters. The *name/value* pairs are delimited from other *name/value* pairs by a "∼". The *name* is delimited from the *value* by a ":". This is just an example as delimiters can be changed as appropriate for the system. The *name* contains the metadata, (*Table*), and the *value* contains the actual name of the database (*OooiStageData*). These *name/value* pairs can be concatenated together, to contain all required information to associate a field on the form with the database. In most cases, there are multiple Filter components on one form. Therefore, the number "1" is appended to the name. This allows the system to differentiate between components with the same functionality. In addition, it also allows the system to associate different fields on the forms and process them together if required. An example of this is when a user needs to qualify information by greater than or less than. A specific example, in figure 3, is when a user has query options to specify the field, operator, and value. Each of these HTML fields has different *name/value* pairs that must be connected.

Currently there is one case where a *value* tag cannot be used. This case is when the user is entering values manually. In this case a hidden tag is used to contain the metadata. Both the visible and non-visible fields will have the same *name*, e.g., both would have the *name* Filter3, the system is able to process the two fields as if the information was

contained on a single HTML statement. Below is an example of this HTML code that implements the user-entered textbox ("Other") in figure 3.

```
<input type="text" name="Filter3">"Other"
<input type="hidden" name="Filter3"
  value="Table:OooiStageData~Field:deptAirport~Operator:=>
```

There are many other GUI types and database specific functions that we are not able to show in the scope of this article. These aforementioned examples should illustrate some of the flexibility that is afforded by using metadata embedded in the HTML code.

### *2.2. HTML hidden business-oriented instructions*

Another hidden tag tells the architecture how to support specialized functions. The following HTML code is a hidden field used to tell how to process the information supplied from the HTML form.

```
<input type="hidden" name="workflow" value="Building SQL">
```

The above HTML code shows the hidden tag used to specify the workflow of components that need to be called to complete this specific form. The design of the components will be discussed later in section 3. Essentially, the name is equated to *workflow* to differentiate this hidden tag from other hidden tags. There should be only one workflow per form, at least at this point in the development of the system. The value of this tag specifies a particular workflow path to use. When the workflow control component (servlet) is initialized all available workflow paths are loaded into the system. These workflow paths are contained in multiple XML files. The purpose for using XML is to allow flexibility for future external applications to control the architecture by inserting their own XML instructions. This feature is reserved for experienced users as some knowledge of the architecture is needed. These workflow-based XML files are located in a directory known by the architecture. The workflow corresponding to the above HTML code (i.e., *Building SQL*) is illustrated in figure 4.

The workflow path specified in figure 4 is the *Building SQL* path. This is the typical path for a simple form that just requests a standard query. The first step is to parse the information from the HTML page. The StagedParser object performs this. The second step is building a generic query based on information gathered from the form. This process is performed using the *QueryBuilder* object. The final object is the *OutputManager*, which formats the query results into a format that the user specifies in the form. As in the above code sniplet, the workflow can specify instructions at the method level. This is not typically the case, because most components are built with default functionality. However, in this case, the method-level specification shows that the *QueryBuilder* object will be building the specific clauses (Select, From, Where, etc.) of the resulting query. These method-level specifications show the flexibility of the architecture when default functionality is not possible.

```
<WorkFlow>
  <WfPath>
    <name>Building SQL</name>
    <WfObject>
      <name>StagedParser</name>
      <Order>1</Order>
    </WfObject>
    <WfObject>
      <name>QueryBuilder</name>
      <Order>2</Order>
      <WfMethod Order="3">BuildFrom</WfMethod>
      <WfMethod Order="2">BuildWhere</WfMethod>
      <WfMethod Order="1">BuildSelect</WfMethod>
        <Parameter>
          <Type>Number</Type>
          <Value>10</Value>
        </Parameter>
        <Parameter>
          <Type>String</Type>
          <Value>"Select Distinct "</Value>
        </Parameter>
      </WfMethod>
    </WfObject>
    <WfObject>
      <name>OutputManager</name>
      <Order>3</Order>
    </WfObject>
  </WfPath>
</WorkFlow>
```

Figure 4. Workflow path in XML.

### *2.3. Creating new GUIs*

The process to create new HTML forms is a user-driven process. Users enter a generic form that asks a round of questions that eventually drills down to the tables in the database that the user is most interested. The user then has the ability to determine what value filtering, time filtering, sorting, grouping, etc., that he/she wants in the form being created. These choices correspond to the workflow of objects that must be executed to perform the task. All this information is used to create an HTML file with hidden metadata about the database and hidden information about the particular workflow needed to perform the query. This process is illustrated in figure 5.

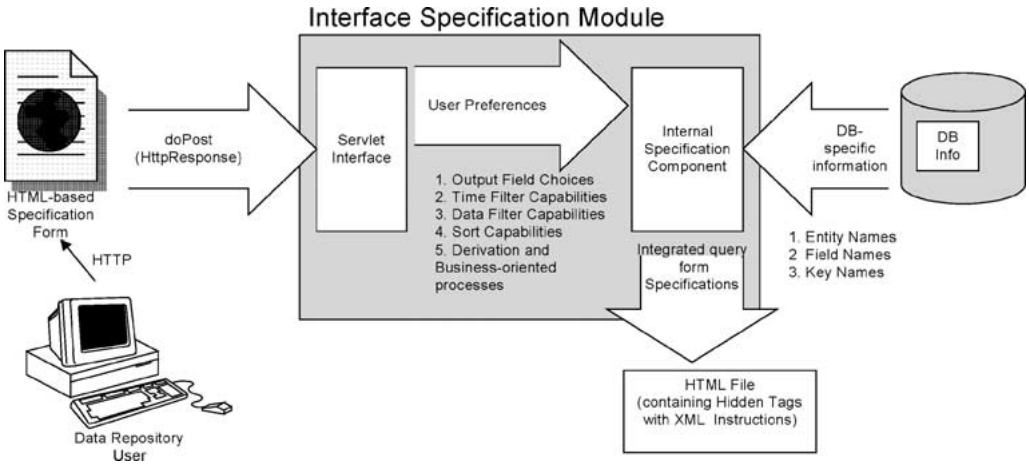


Figure 5. Process for building staged query forms.

## 3. The middle layer

The middle layer is the centerpiece of the architecture. This layer follows strict object oriented principles. All designs were conceptualized using the Unified Modeling Language (UML) [Booch, Rumbaugh and Jacobson 1998]. This layer contains all of the objects that executes the complex business rules, creates the user specified queries and performs the complex data formatting. This middle layer can be separated into two parts, the Workflow Control component and the group of Autonomous Business components. Currently, these parts are implemented with five components. These five components are the MainControl, AdvanceDBManipulation, Parser_ObjectBuilder, Specialized, and Statistical components. In this architecture, components are represented by a group of classes that perform an independent functionality. For clarity, the components can be represented in packages (as defined in the Unified Modeling Language). The five packages/components and the underlying classes are represented in figure 6.

### *3.1. Workflow Control component*

The workflow control component is the part of the architecture that controls what steps will be executed in building the database query and returning the relevant information. Workflow has been defined in many contexts over the past decade. In this work, a workflow can be defined as the specification and execution of a sequence of steps or processes to accomplish a specified job. One problem in most systems is that the workflow can have business rules that are predefined. By essentially "hard-coding" their functionality, these systems are not easily extended. In fact, major revisions are necessary to incorporate even minor enhancements to functionality. However, the workflow control component defined here is an extensible component that allows the workflow to be designed at run-time as illustrated in section 2.2. This component then executes the Autonomous Business components specified in the workflow.

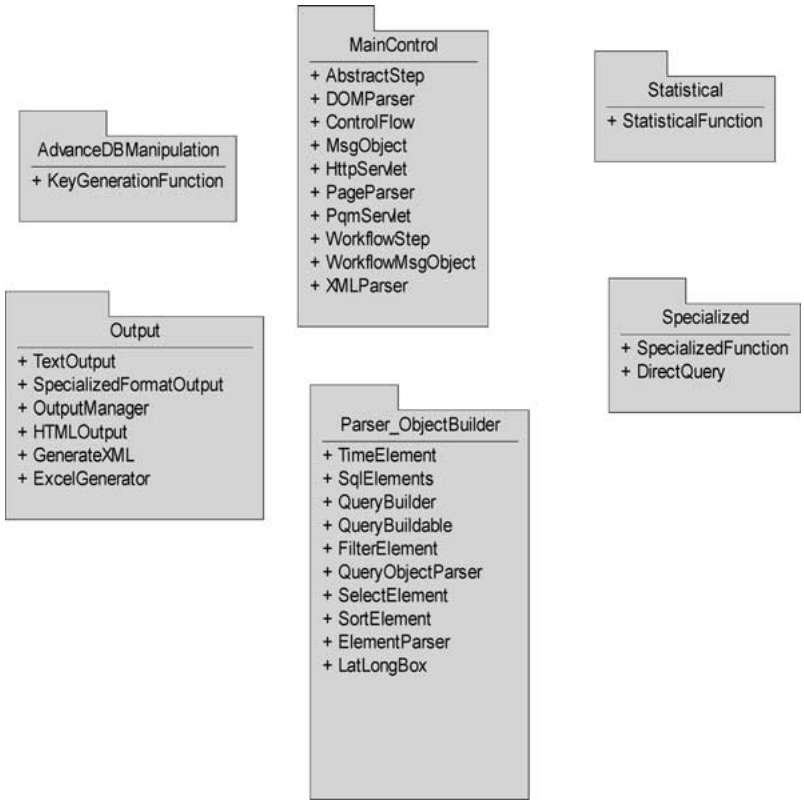


Figure 6. High-level view of middle layer components as packages.[1]

### *3.1.1. MainControl package*

The Workflow Control component is implemented in the MainControl package. This package contains independent classes that can organize, order, and execute the workflow, according to a pre-defined workflow path (see XML document in section 2.2). The main classes of this package are the PqmServlet, ControlFlow, XMLParser, and the MsgObject. The PqmServlet (PQM stands for Presentation Query Management) captures request information that the user submits from HTML forms. The ControlFlow is the main workflow execution engine. This class reflectively executes the autonomous business components (will discuss in detail in the next section). The XMLParser parses the workflow path and captures the information in memory. Finally, the MsgObject has generic data structures that store information that is passed between the autonomous business components. Since the business components are autonomous, this information is generally information at the workflow-level. The class diagram for the MainControl package with the aforementioned classes is illustrated in figure 7.

[1] This diagram and all subsequent object-oriented diagrams were captured using Rose Enterprise 2000 from the Rational Corporation [Rational Corporation 2002].

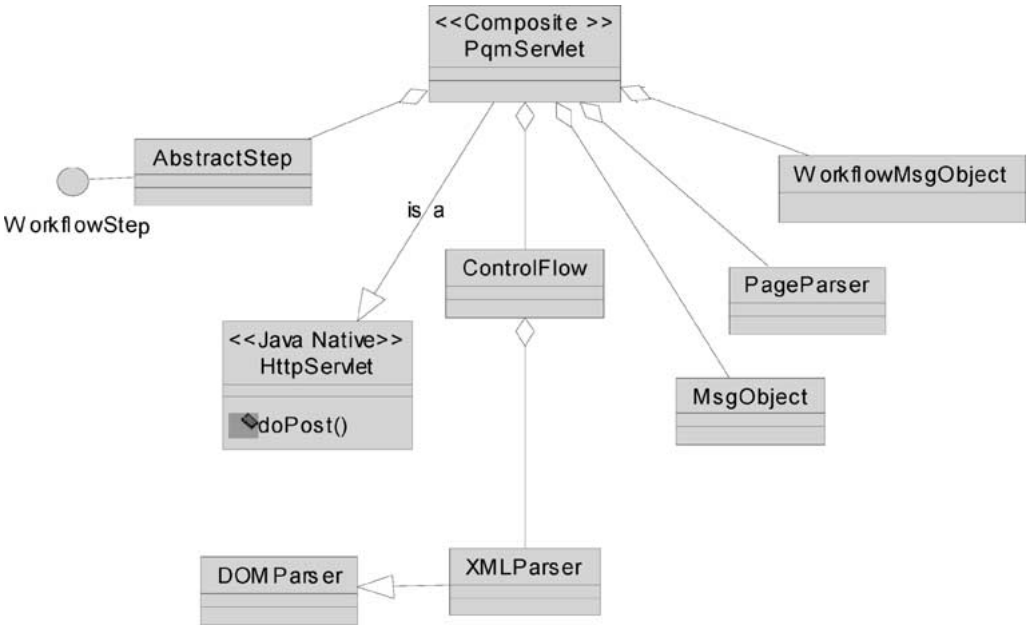


Figure 7. The MainControl package.

The workflow execution process can be summarized in 4 basic steps. When a user presses the submit button on an HTML, the doPost method is automatically executed on the PqmServlet. The PqmServlet populates the MsgObject object with the user's request information (via Java's HTTPRequest). Subsequently, control is passed over to the ControlFlow object. The ControlFlow first parses the XML-based workflow path. Finally, the ControlFlow object uses the workflow information from the XML document and HTTPRequest to create and execute the default or specified entry method for each of the Autonomous Business components. This execution path is illustrated as a collaboration diagram (as defined in UML) in figure 8.

*3.1.2. Reflective capability of the ControlFlow class*

An important part of this architecture is to execute new workflow paths at run-time. Reflective architectures or software can be configured and executed at run-time without compile-time dependencies.

This architecture is a run-time evolvable framework, which uses reflection to convert text-based instructions into software executions. The ControlFlow class has simple method written in Java that performs the reflective execution. The code sniplet in table 1 illustrates the use of Java reflection. As shown in table 1, the ControlFlow object takes the MsgObject as a parameter. This will be the container that the Autonomous Business components use to pass information. The intention of the MsgObject is to provide global workflow-level information without becoming coupled to any specific component. Therefore, the MsgObject contains only implementations that help store

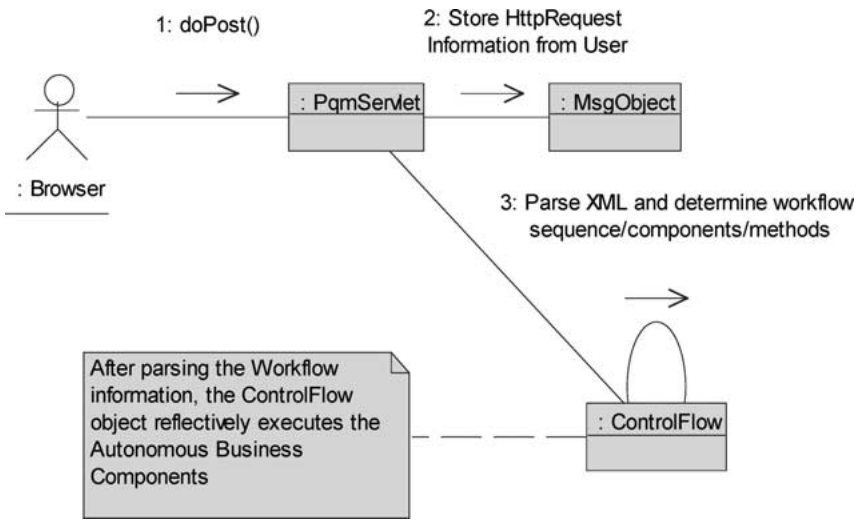


Figure 8. Overview of MainControl package functionality.

Table 1
Sniplet of Reflective Code syntax.

```
public class ControlFlow {

  private MsgObject messageObject = new MsgObject();
  public AbstractStep crsObject; // Parent class of all plug-able components

    /***Controller***/
    public ControlFlow(MsgObject thisMessage)
    {
     messageObject = thisMessage;
    }

    /***Method to create the Object****/
    public void createObject(String className) throws ControlFlowException
    {
    Object object = null;
    try {
          Class classDefinition = Class.forName(className);
          crsObject = classDefinition.newInstance();
          crsObject.execute(messageObject);
         }
    // (CatchExceptions. . .)
```

and retrieve global information. The createObject method takes the className as specified in the XML-based workflow path then reflectively creates an instance of the class. This instance is used to polymorphically run the generic *execute* function that is con-

tained in all the Autonomous Business components. This is one of many cases where this architecture exploits the strength of object-oriented programming.

### *3.2. Autonomous Business components*

The Autonomous Business components are implemented in several packages. These packages, as illustrated in figure 6, are the AdvanceDBManipulation, Output, Parser_ ObjectBuilder, Statistical, and Specialized packages. Each of these packages implements a specific autonomous function toward the data extraction job. The AdvanceDB-Manipulation package is used to collect specific table information from the HTML files and build the complex joins between database tables needed in some data queries. The Output package is used to convert the query results into various query output format options as specified by the user. The Statistical package has internal algorithms that derive the query results based on certain algorithmic criteria. The Specialized package is used to capture all functionality that cannot be generic. One such function implemented in this package is the ability for a user to directly type in a database query in SQL and have it returned in a specified format. Finally, the Parser_ObjectBuilder package has classes that build a generic query from HTML. This package understands request information and generically builds a standard database query string. It is not within the scope of this article to define all of these Autonomous Business components in detail. We have chosen to explain the classes in the Parser_ObjectBuilder because this package is essential to the generic SQL generation.

#### *3.2.1. The Parser_ObjectBuilder package*

The Parser_ObjectBuilder package is an example of the most general process that may be used by any specified workflow path that displays data from the database. In this architecture, independent buttons and fields from the HTML query form have direct relations with the underlying query that will be built. There are underlying classes in this package that associate fields from the HTML form with the underlying database. The duties of these classes are to understand the HTML fields and create the SQL query. The typical classes are SelectElement, FilterElement, SortElement, GroupElement, and StatElement. Each of these classes are derived from the SqlElement class. The SqlElement class implements the common Java interface QueryBuildable. By implementing QueryBuildable, the SqlElement and each of its derived classes must implement the methods buildSelect, buildFrom, buildWhere, buildSort, buildGroupBy, and buildOrderBy. Intuitively, these methods will create the part of the query relevant to the Select, From, Where, Sort By, and Group By SQL blocks. The functions of these classes can be summarized as below.

- SelectElement: converts HTML fields to columns to be displayed from the database.
- FilterElement: converts HTML fields to restrictions in the dataset displayed.

- SortElement: converts HTML fields to order in which results are displayed from the database.
- GroupElement: converts HTML fields to groupings that data is displayed from the database.
- StatElement: converts HTML to statistical functions such as Count().

The class diagram of the Parser_ObjectBuilder is shown in figure 9. The QueryBuilder class collects the list of all SqlElements for a given query. The QueryBuilder class passes in an empty SQL string that would continually get populated as the interface methods are called for each of the derived SqlElements. This is probably the strongest use of object-oriented programming. This function demonstrates information hiding, encapsulation, and polymorphism as each SqlElement is responsible for its own behavior. These elements also are called polymorphically using the *build...* methods (these methods are all implemented via the QueryBuildable interface as shown on the far right side of figure 9).

*3.2.2. Adding new Business components*

As aforementioned, the Autonomous Business components are independent. Each component has an interface class that contains an *execute* method. This *execute* method is polymorphically called by the ControlFlow class. With this design, new components can be added seamlessly or *software plug and play* [Bronsard 1997]. When a new component is added, it merely needs to be derived from the AbstractStep class contained

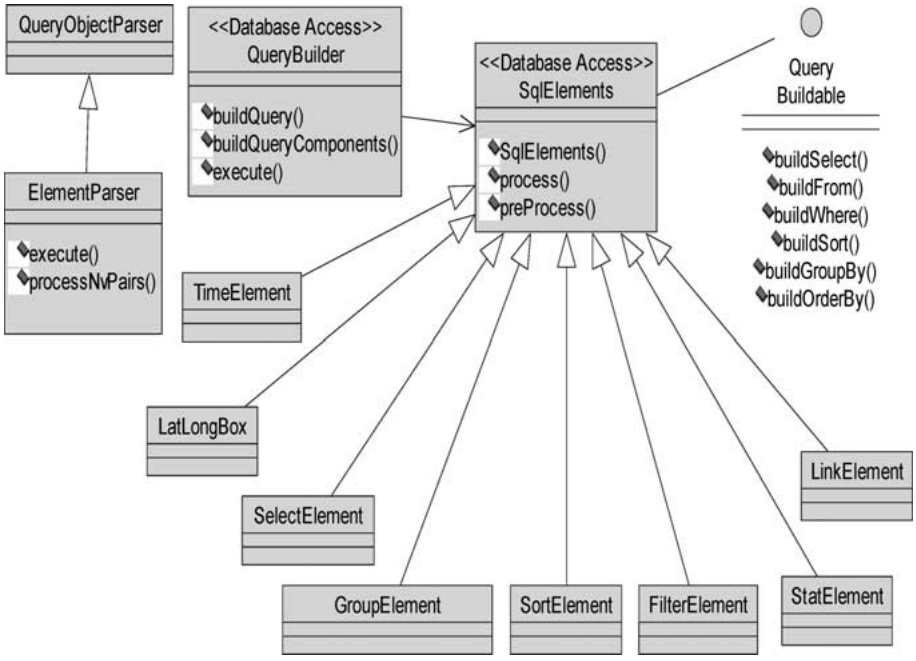


Figure 9. Class diagram of the Parser_ObjectBuilder package.

in the MainControl package (this package was illustrated earlier in figure 7). The AbstractStep class has the interface WorkflowStep. The WorkflowStep interface defines the *execute* method. Subsequently, when a text-based instruction uses the new component in a workflow path, then the *execute* method would be called by default unless as discussed in section 2.2 specific functions are declared.

## 4. The Data Storage Layer

The Data Storage Layer is somewhat customary. The data in this architecture is captured in a relational database management system (RDMBS). In this case, the RDBMS is Oracle version 8i. This architecture does not require that data be normalized, but some normalization helps in the performance of the system as a whole. The following are database functions that our architecture takes advantage of.

1. *Stored procedures to perform simple formatting* (*date, times, latitude, longitudes, etc.*). Some formatting is easily done using the database server rather than exterior Java-based processing. In the future, more stored procedures may be added to further remove some generic functionality from the code. One such functionality may be the addition to drill-down features.
2. *Meta-data to translate database column names into relevant English names*. The system uses a generic query process that uses the direct column names to generate the query. There are meta-data tables in the database that help the architecture translate database-specific column names into the more "user-friendly" names. However, there is also support in the architecture for the "user-friendly" name to be specified in the HTML form.
3. *Indexing for large tables*. As with most RDBMSs, the performance of this system is dependent on an accurately indexed system. A database administrator performs the indexing. Indexes are typically created and updated regularly at low peak hours.

## 5. An implementation of the Distributed Data Management System at MITRE

This section will discuss the application that was developed at the MITRE Corporation based on the architecture presented in this article. The following sections will give an overview of the system that was developed, walk through the query-building process, and show some actual forms used in the process.

### *5.1. An implementation of Distributed Data Management System: The CRS system*

At MITRE-CAASD, the CRS team has implemented the CRS architecture using various Internet technologies. Figure 10 presents the implementation that supports the details in the original architecture diagram. The CRS implementation mainly uses Java-based technologies. The three basic steps in extraction data have been separated into three

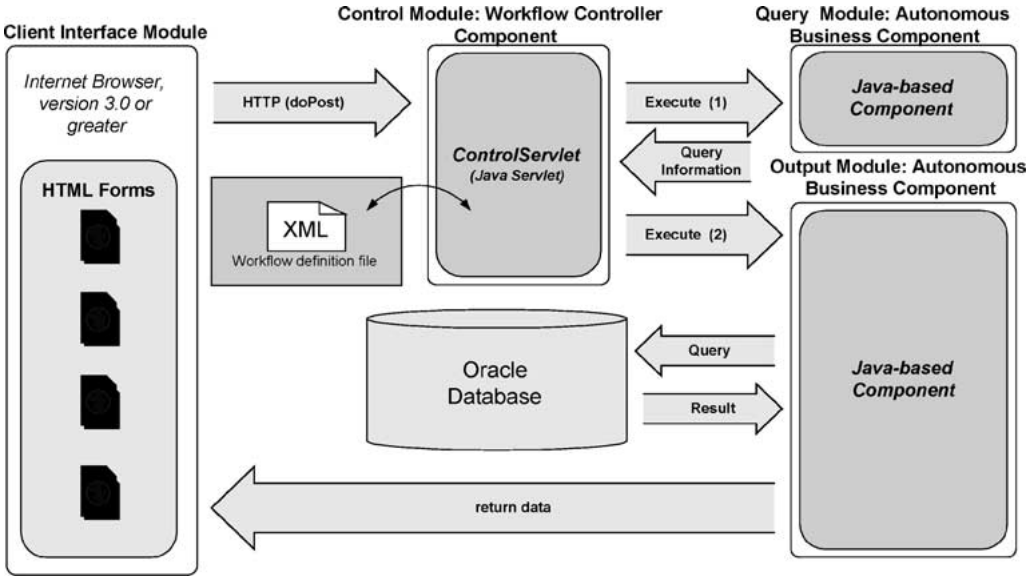


Figure 10. The CRS system architecture.

modules, the Control, Query, and Output Modules. The remainder of this section will discuss these modules in context of the Distributed Data Management architecture described in this article.

The browsers in the Client Interface Modules connect to a Java Servlet-based interface (ControlServlet), which passes the information to the Control Module (Both make up the Workflow Controller Component). The ControlServlet serves as the main interface between the user and the Control Module. The servlet accepts information from the browsers in the Client Interface Module in the form of an HttpServletRequest. The customers' data and format criteria are encapsulated in the HttpServletRequest. As described in section 3.1.1, this information is parsed and stored in a Java object, the MsgObject. This MsgObject is essentially passed to all the steps in turn. Each step extracts what it needs from the MsgObject and modifies its contents, if necessary.

The following sections discuss the low-level design of the Distributed Data Management architecture in context to CRS implementation.

### *5.2. Description of the CRS Modules*

The Client Interface Modules are simply the web browsers used by the stakeholders using the CRS system. Typically, the stakeholders access a standard query form based on the dataset that they are interested in. One particular data schema in reference to Air Traffic Management (ATM) is the OOOI data set. This data set deals with the Out/Off/On/In times when a plane backs "out" from the terminal, lifts its wheels "off" the runway, lands back "on" the arrival runway, and finally pulls "in" to the arrival air-

port terminal. The query form in figure 11 shows a form that is compatible with the CRS distributed data management system.

The form is separated into 5 sections. The first section, Time Selections, allows the user to specify a particular time range used to filter the output information. The second section, Filter Selections, allows the user to specify filter constraints based on return values. The Output Selections section allows the user to determine what columns are requested as output. The Sort selections allow the user to determine how the data should be sorted. This selection requires that sorted columns also be selected in the Output Selections. Finally, the Output Size and Format Selections section allows the user to limit the quantity of output, as well as specify what format is desired as the return. In this particular case, the request is for an hours worth of information starting at January 1, 2001 midnight. This information will be based on the time flights back out of the terminal. These choices are seen in the Time Selections section. In addition, this request will return only information on flights departing either Dallas airport or Pittsburgh airport (as selected in Filter Selections).

The Output Data Selections shows various airplane and airport related information that will be returned for each entry. In Sort Selections, the results will be sorted by the departure airport. Finally, in the Output Size and Format Selections, the output is limited to 5 lines and will be returned as HTML table.

The Control Module is implemented with an XML file and several Java classes. The main purpose of this module is to take the information workflow information and orchestrate the execution of the appropriate modules. The XML file contains a listing of all the workflows supported along with the choice of and order of modules to execute. The module parses the XML file to find the appropriate workflow. The OOOI dataset uses the XML file (workflow) depicted in figure 4.

The Query Module is an Autonomous Business component that is implemented with Java classes. As described in section 3.2.1, SqlElement objects are created and used to create a SQL database query string. Consequently, the resulting query string is stored in the MsgObject for retrieval by the Output Module. The query string for the OOOI example is depicted below.

```
SELECT DISTINCT OooiStageDa.carrierFlight, OooiStageDa.deptAirport, OooiStageDa.
  arrAirport, OooiStageDa.gmtdate, OooiStageDa.outtime, OooiDerivedTi.
  taxiouttime
FROM OooiStageData OooiStageDa, OooiDerivedTime OooiDerivedTi
WHERE ROWNUM < 6 AND OooiDerivedTi.OooiDataId=OooiStageDa.OooiDataId AND
  OooiDerivedTi.outtime >= to_date('2001-01-02 00:00:00', 'YYYY-MM-DD
  HH24:MI:SS') AND OooiDerivedTi.outtime <= to_date('2001-01-02 01:00:00',
  'YYYY-MM-DD HH24:MI:SS') AND (OooiStageDa. deptAirport= 'PIT' OR
  OooiStageDa.deptAirport= 'DFW')
ORDER BY OooiStageDa.deptAirport
```

The Output Module is an Autonomous Business component that is basically a customized interface to the DBMS. This is not a new technology as many vendors have constructed APIs for DBMSs. Here, we specialize one of those APIs for use within the CRS architecture. In this implementation, we use Sun Microsystems' JDBC methods to connect to an Oracle database. All database and user interaction is run in the Output

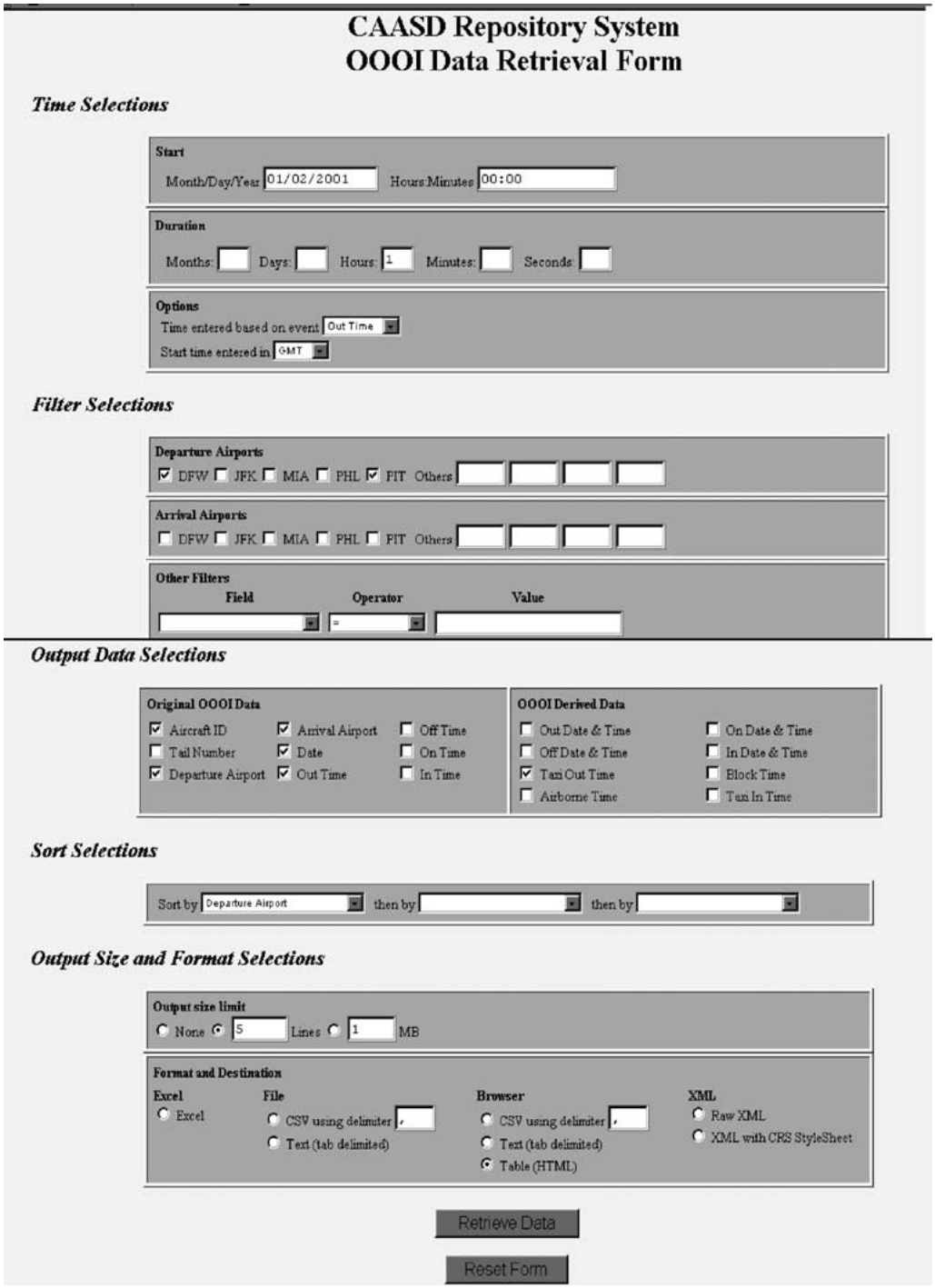


Figure 11. Sample HTML file for OOOI dataset.

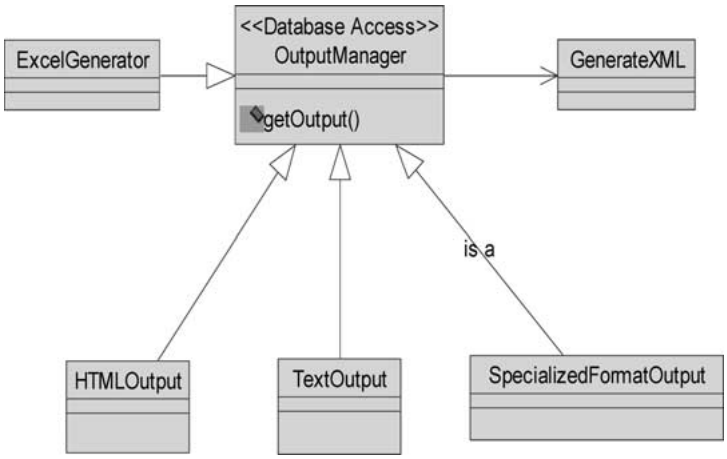


Figure 12. Low-level Class diagram of Output Module.

Module to help performance. The main class in the Output Module is the OutputManager. This object extracts the query from the MsgObject and submits the query to the DBMS. Once the result set is available, it returns the data in the specified format. This class is specialized to generate HTML, ASCII, XML, Excel, and any other customized format that may be desired. A representation of this design is illustrated in figure 12. This Output Module is independent of the query builder and database so that a change in either will not affect its functionality.

### *5.3. The data extraction process*

In figure 13, a collaboration diagram is used to show the sequence of events in creating the query string and returning the results.

The following are a list of software executions taken in the collaboration diagram in figure 13.

1. Initially, the doPost event is captured by the ControlServlet (also PqmServlet).
2. The ControlServlet then stores the information contained in the doPost message in the MsgObject for retrieval by the downstream modules.
3. The ControlServlet creates a Control Flow object, which identifies which modules are to be used. (In this case the standard Query Module and OutputManager Module will be used.)
4. The name of the workflow was stored in the MsgObject during step 2, above. That name is checked against the pre-defined workflows in an XML document to determine the module and classes (and optionally, the methods) to be called and sequence of execution.
5. In the Query Module, there are two steps: the ElementParser and the QueryBuilder. The ElementParser is called first and creates the SQLElements from the information

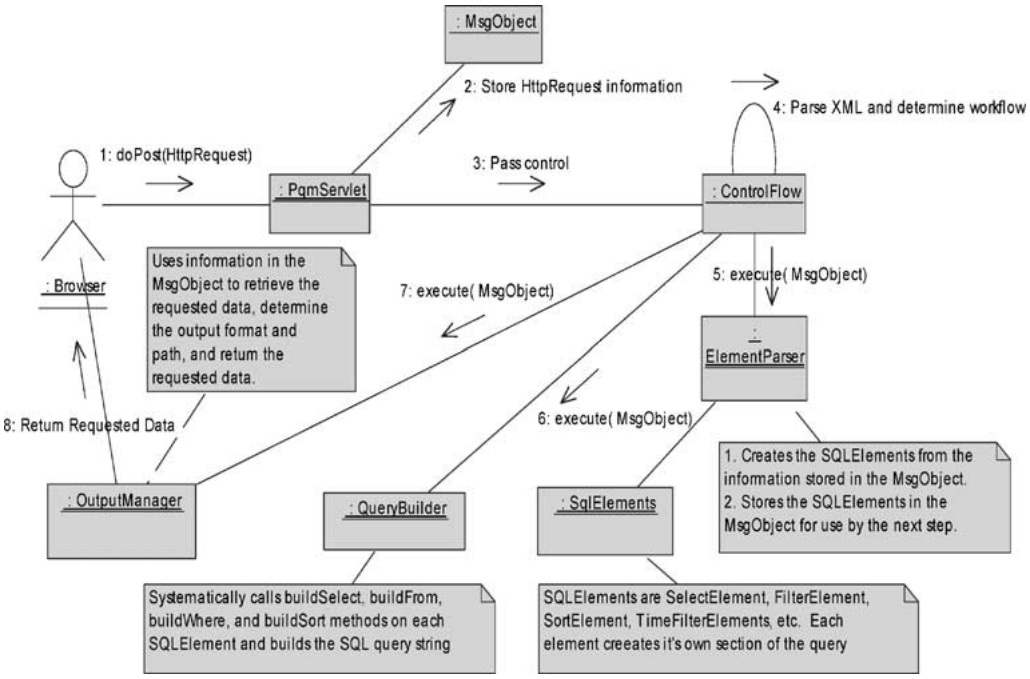


Figure 13. Collaboration diagram of the data extraction process.

stored in the message object. As each element is created, it is stored in the MsgObject for use by the QueryBuilder.

6. The QueryBuilder systematically calls buildSelect, buildFrom, buildWhere, and buildSort methods on each SQLElement, which builds the query string.
7. The OutputManager is then called with the default method to retrieve and return the requested data in the correct format.
8. The data is returned to the client browser and either viewed or saved for later use.

### *5.4. OOOI Query results*

As specified in the query form, the result is an HTML table that shows 5 rows. Each row contains the airline/airport data for flights that departed from Dallas or Pittsburgh airports. These results are shown in figure 14.

## 6. Conclusions

In this work, we have the shown the need for a Distributed Data Management System which enhances the sole use of a RDBMS. This article presents an architecture that is implemented using object-oriented concepts. Component-based structures are used in this architecture to assure the run-time evolvability of the system. Additionally, this

| CARRIER | ORGNLGATEDEPTTIME | ORGNLGATEARRTIME | DEPTAIRPORT | ARRAIRPORT |
|---|---|---|---|---|
| AAL | 141 | 340 | DFW | TUS |
| AAL | 141 | 406 | DFW | LGA |
| COA | 2334 | 2542 | DFW | CLE |
| UAL | 0 | 139 | DFW | DEN |
| USA | 2318 | 2403 | PIT | SYR |

There were 5 records returned. The limit was 5 lines. There were approximately 377 bytes returned. There was no file size limit.

Figure 14. OOOI Query results.

system exploits the latest in emerging web-based technologies (XML, Java Servlets, Java Reflection, and the leading RDBMS).

This autonomous architecture allows a user to specify which data they want and in what format. The application then generates an appropriate query, executes it, and returns the data. By maintaining the autonomous nature of the component modules, the architecture can adapt in the future with minimal lapses. of services. Because the modules are ignorant of the database schemas, this system can be reused for any data schema with minimal effort. The users of this system have merely to understand the output data, formats, and types of filters to operate this implementation. Moreover, as business needs change, the system can adapt to the corresponding changes quickly and easily.

This architecture has been implemented into the CAASD Repository System. The CRS system has seen a great deal of success at MITRE-CAASD. New data schemas and functionalities are being added on an ongoing basis. The architecture has been extremely robust and changes have been minor even with many different data schemas.

Currently, there is work in providing specialized output formats that extend further than the common return output formats of delimited text, XML, or HTML tables. The future approach will allow CRS users to specify there preferred output formats as an XML file, and the system will used this file to process the query response.

## Acknowledgements

This work was done in cooperation with the Center for Advanced Aviation System Develop (CAASD) at the MITRE Corporation. There was a great deal of support given by members of the MITRE-CRS team consisting of Rob Tarakan, Fred Wieland, Tho Nguyen, Ted Cochrane, Jay Cheng, Ali Obaidi, and Gretchen Jacobs.

## References

Allen, R.J., R. Douence, and D. Garlan (1998), "Specifying and Analyzing Dynamic Software Architectures," In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE98)*, Lisbon, Portugal, Springer, Berlin, pp. 21–37.

Blake, M.B. and P. Liguori (2001a), "An Autonomous Decentralized Architecture for Distributed Data Management and Dissemination," *IEICE/IEEE Joint Special Issue on Autonomous Decentralized Systems and Systems' Assurance, IEICE Transactions on Information and Systems*, E84-D *10*, 1394–1397.

Blake, M.B. and P. Liguori (2001b), "An Automated Client-Driven Approach to Data Extraction Using an Autonomous Decentralized Architecture," In *Proceedings of the 5th International Symposium of Autonomous Decentralized Systems (ISADS2001),* IEEE Computer Society Press, Dallas, TX, pp. 311–319.

Blake, M.B. (2000), "SABLE: Agent Support to Consolidate Enterprise-Wide Data Oriented Simulations," In *Proceedings of the 4th International Conference on Autonomous Agents* (*AGENTS2000*), *Workshop on Agents in Industry*, Barcelona, Spain.

Booch, G., J. Rumbaugh, and I. Jacobson (1998), *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA.

Bronsard, F. *et al.* (1997), "Toward Software Plug-and-Play," In *Proceedings of the 1997 Symposium on Software Reusability*, pp. 19–29.

Garlan, D. and M. Shaw (1992), *Software Architectures, Perspectives on an Emerging Discipline*, Prentice-Hall, Upper Saddlehill, NJ.

Glushko, R. *et al.* (1999), "An XML Framework for Agent-Based E-commerce," *Communications of the ACM 42*, 3, 106–107.

Hu, J., D. Nicholson, C. Mungall, A.L. Hillyard, and A.L. Archibald (1996), " WebInTool: A Generic Web to Database Interface Building Tool," In *Proceedings of the 7th International Conference and Workshop on Database and Expert System Applications (DEXA96)*, IEEE Computer Society Press, Zürich, Switzerland.

Oracle Corporation (2001), "WebDB Application 3.0," `http://oradoc.photo.net/ora816/webdb.816/a77075/basics.htm`.

Rational Corporation (2002), "Rational Rose Enterprise Edition," `http://www.rational.com`.

Shrivastava, S. and S. Wheater (1998), "Architectural Support for Dynamic Reconfiguration of Large Scale Distributed Applications," In *Proceedings of the 4th International Conference on Configurable Distributed Systems (CDS'98)*, IEEE Computer Society Press, Annapolis, MD.

Sun Microsystems Inc. (2002), "Java Language Specification and the Distributed Event Model Specification," `http://java.sun.com`.

Varela, C.A. and C.C. Hayes (1994), " Zelig: Schema-Based Generation of Soft WWW Database Applications," In *Proceedings of the 1st International Conference of the World Wide Web (WWW94)*, Geneva, Switzerland, Elsevier Science.

WebInTool (1997), `http://www.ri.bbsrc.ac.uk/webintool.html`.

Weiss, A. (1999), "XML Gets down to Business," *Networker 3*, 3, 36–37.