CHAPTER 5 JXDB IMPLEMENTATION AND TESTING

This chapter will describe the proposed JXDB system implementation, including its development environment, coding, testing and evaluation phase. Here, the implementation phase emphasizes on the implementation of the major classes and how these classes are integrated to form a working JXDB system. As stated in Chapter 3, these component systems are independent from each other, and thus can be implemented separately. Each of the component systems is constructed based on a sequence of classes and conceptual design of the system listed in the previous chapter. The final system is the integration of these major classes.

5.1 System Development Environment

In this section, a brief explanation on the system development environment, which includes the software development tools and environment used in developing JXDB system, will be elaborated according to its suitability and functionality reasons in the system. At the end of the development, all the classes and required components will be bundled together as JXDB.jar file. It contains all the necessary JXDB classes and libraries, third party libraries, such as XML parser or XQuery engine and also images to operate JXDB system. To run JXDB application, we need to execute "Run.cmd" file, which will launch JXDB if the correct Java ClassPath environment variable is set in the running machine.

5.1.1 Software Development Tools

One of the software development tool used is Borland JBuilder 8 Enterprise Edition for Java. It is a comprehensive integrated and leading cross-platform development environment for building enterprise Java applications. Some of the benefits are speed coding and debugging with an integrated, extensible source code editor, graphical debugger, compiler, visual designers, time-saving wizards, support for Java standards and so on. Currently, it also supports Java 2 standards (Java version 1.2 and above), which include support XML and Web Services. As for designing and modelling of UML diagrams, Rational Rose is used for drawing the use case diagrams, class diagrams, sequence diagrams and so on. Whereas, Microsoft Visio 2002 is used to draw the system architectural diagrams as it provides easy-to-use system architectural and deployment stencils.

5.1.2 Development Environment

The development environment is conducted on an Intel machine running Windows XP 2000 operating system with hardware specification of 256 MB RAM and 40 GB hard disk space. Besides the installed software development tools mentioned above, this machine also has installed the following software in order to run the JXDB system:

- Apache Xerces XML Java parser for parsing XML data
- Oracle 9i
- Microsoft SQL 2000
- MySQL
- JDBC Driver for Java applications accessing back-end database

- Third-party XQuery processor, i.e. QuiP (version 2.2.1.) from Software AG.

5.2 JXDB XML-based Interfaces Application System

The application system of the JXDB provides XML-based interfaces to its end users. Generally, the main screen of JXDB is divided into two main parts; the top part is used for displaying XML files whereas the bottom part for displaying multiple database connection details. Figure 5-1 shows the main screen of JXDB implemented by JXDB application system. The main screen has one frame component derived from JFrame swing class, which consists of one main panel component at the center that is derived from JPanel swing class. In this main panel, there are another two more panels separated by a JSplitPane component; the top panel and the bottom panel. In addition, the JMenu swing class is added at the top of the main screen to create the menu bar as a container for JMenu components such as JMenuItems that will response to menu events invoked by any user.

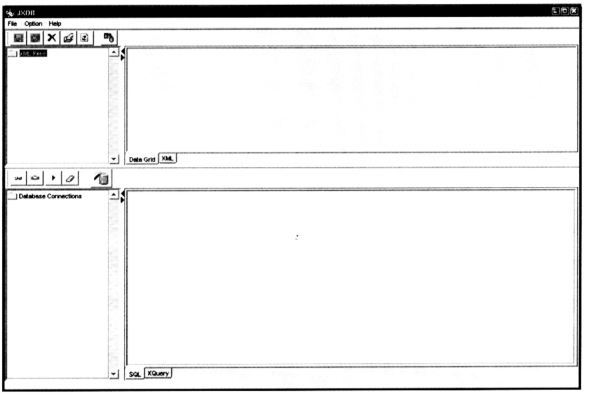

Figure 5-1: JXDB Main Screen

Basically, the top panel contains two more components. The first component is JTree swing class, which is located at the top-left interface to display multiple XML files in a tree view and it is separated from the top-right interface by JSplitPane swing class. JSplitPane is used to divide two (and only two) components. These two components can be graphically divided based on the look and feel implementation, and the two components can then be interactively resized by the user when the user clicks to expand the top-left or top-right interface. The top-right interface is formed by two JTabbedPane components with one to contain JTable and another one to contain JTree. JTable is used to create the data grid interface that will display the XML data in tabular form, whereas the JTree is used to display these XML data in a tree view, such as their elements or attributes in one XML document.

On the other hand, the bottom panel consists of two components, they are JTree and JEditorPane. The JTree swing class that is used to create the tree view at the bottom-left is responsible to display multiple database connection details. This component is also separated by a JSplitPane from the bottom-right component, which is the JEditorPane component. Similarly, this bottom-right interface is formed by another two JTabbedPane components; they are used to contain two JEditorPane components where one JEditorPane for each JTabbedPane component. One JEditorPane component is used as a text component that allows users to enter any SQL commands to be executed and return the results from the database to be displayed at the data grid interface that is derived from JTable swing class, which is located at the top panel interface. Meanwhile, the other JEditorPane component is used to allow users to enter XQuery expressions to be executed to extract the XML data from multiple XML data sources.

For parsing and validating XML files, there are many third party open-source XML parsers that can be used for this purpose. One of the most popular parsers is Apache Xerces2 Java Parser that is used for Java applications. Xerces2 fully conforms to XML schema processor and supports JAXP 1.2 and the two dominant models; they are DOM Level 2 and SAX 2.0. Generally, SAX is an event driven API that calls event methods whilst parsing an XML document as it provides methods that can response to data in the document when reading that data. However, this is useful when only a few parts of a document are needed, and these parts can be located within the stream of SAX events, or when reading the data in sequence. On the other hand, DOM parses the entire document and creates a corresponding Document object that can be browsed using suitable method calls. Since DOM creates the object from the XML file, it needs adequate memory to hold that file. In fact, it will be a drawback if the XML file is very large and the allocated memory is very small. In effect,

DOM provides programmatic access to the entire document, in a non-liner order. Nevertheless, the DOM model is much easier to use, whereas the SAX model provides faster parsing and requires less memory.

For this implementation, Apache Xerces2 Java Parser (version 2.4) was selected because it is able to provide a seamless XML parser and a XML builder as one integrated API for XML. Basically, it is a hybrid of DOM and SAX, maximizing the benefits of both and minimizing their drawbacks. This software is freely downloadable from apache website in JAR or ZIP format (http://xml.apache.org/xerces2-j/index.html) and has well documented technical supports and tutorials available.

5.3 JXDB Component Systems

Here, the implementation process starts from the systematic layered and object-oriented design of the JXDB application systems to all its independent and integrated component systems. These component systems are represented by various business entities located at the business specific layer, which form the objects found in the main JXDB XML-based interface. The design model specified earlier is traced to the use case model in Figure 4-1, which consists of Load XML Documents, Insert XML Data, Update XML Data, Delete XML Data and Read XML Data use cases that are integrated together into JXDB application systems. In fact, each of these use cases is developed to deliver the main functionality first, such as before a user can proceed to do any updating and transferring to relational databases, the user has to load XML documents prior to any updating or transferring of XML data. During the development process of these use cases, they will be

developed incrementally and iteratively, and upon the completion of each increment, the next set of use case is then refined and implemented.

5.3.1 Implementation of Database Connection Component Systems

One of the important component systems in JXDB system is its database connection component systems. Here, the classes are responsible for accessing the back-end databases and retrieving the results from these databases via designated Java database drivers. JXDB's database design is a three-tier architecture because it has three layers or tiers that are designed to perform specific tasks based on each different tier. The front tier is the presentation layer that presents the user interface at the client's side, whereas the middle tier is the business layer that is responsible to perform the business rules via the business components. The last tier is the data access layer, which consists of data access components to create database connection and manage data retrieval. In this case, the database connection component systems act as the data access layer that uses JDBC Driver for database connectivity. Among the tested back-end databases are Oracle9i, Microsoft SQL 2000 and MySQL.

Apart from that, it contains database connection wizard to assist users to connect to different back-end databases. The wizard dialog box will prompt the users to select the required database driver and its respective connection string. They are also required to enter their username and password as well. In this case, Figure 5-2 shows the popup database connection wizard where the user with username DEMO was trying to connect to JXDB database instance in Oracle9i using Oracle JDBC driver via Oracle default port 1521.

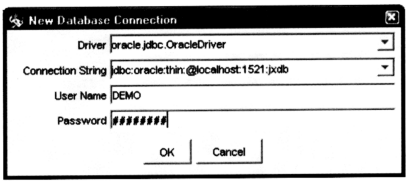

Figure 5-2: Database Connection Wizard Dialog Box

Below is part of the java code inside the Database Connection Wizard class to connect to Oracle9i database:

```
java.sql.DriverManager.registerDriver((java.sql.Driver)Class.forName(this.getConnection
Driver()).newInstance());

this.connection=java.sql.DriverManager.getConnection(this.connectionString,
this.userName, this.password);
```

Once it has successfully connected to the back-end database, the system will display the word *connected* at the bottom of window's status bar. The respective database connection string will be sent to JTree component to add to its node and display the connection string in the respective tree. Following that, we can enter and execute a SQL statement at the SQL tab located at the bottom panel, for example, to select all the results from Demo table. Referring to Figure 5-3, once the valid SQL statement is executed by clicking on the Run button, the system will display the returned results in the data grid control at the Data Grid tab located at the top panel, generate a corresponding valid and well-formed XML file based on these returned results, and lastly add it to the XML Files' tree. These automatically generated XML files are stored in a user-defined working folder where all

these files correspond to their respective tables from the database. In fact, each of these XML files represents a table generated from the SQL statement executed by the user.

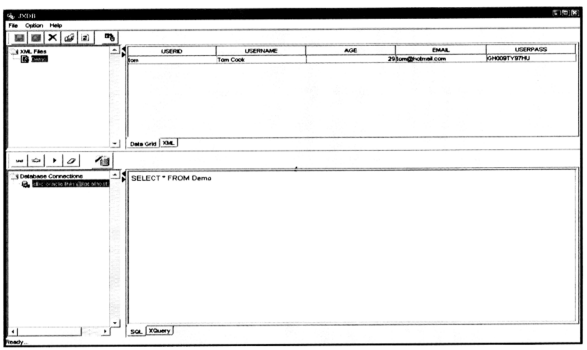

Figure 5-3: Execute SQL Statement Screen

5.3.2 Implementation of XQuery Wizard Component Systems

Generally, by creating an XQuery Wizard and adopting it as part of the JXDB framework to assist end users to use the framework more efficiently and learn to appreciate many advantages the framework has to offer for efficient and seamless integration of XML with heterogeneous relational databases. Having such an application wizard can lead the end users going through the steps of using the XQuery interpreter without having to learn the complexity of XQuery, which is at this stage still in its preliminary stage. Thus, by having a user-friendly user interface in this wizard, any user can quickly manipulate XQuery expressions on the XML data without having to go through the high learning curve.

The core XQuery engine of JXDB is implemented using a third party API component named Quip from Software AG. Software AG's Quip is a prototype of W3C XML XQuery language implementation. Basically, quip is designed to be user-friendly for end users to learn and use the language. To execute and process XQuery syntaxes, JXDB will invoke the Quip APIs to perform the task and these queries can be made available to the XQuery engine from XML files, which are stored in local FileSystem. Most of the XQuery language has been implemented in Quip and conformed to a large number of examples of W3C use case queries. The following lines give a simple example on how to invoke Quip APIs from JXDB java source code:

```
java.util.Properties prop = new java.util.Properties();

prop.put("quipcmd", "E:\\xdb\\lib\\quip.exe");

com.softwareag.xtools.quip.xqueryAPI.QMachineQueryResult qr;

com.softwareag.xtools.quip.xqueryAPI.QMachineConnection qmc = new
com.softwareag.xtools.quip.xqueryAPI.QMachineConnection("file:///E:/xdb/work", prop);
```

Figure 5-4 depicts the main user interface of the XQuery Wizard. By using this wizard, we can extract data from multiple tables or XML files from the Table drop-down list. After selecting the required table or XML file, we can then select its respective columns or elements before applying the available operators from the operator drop-down lists. For example, Equal, Greater, Less, Not Equal, And, Or and so on are some of the examples of available supported operators to generate FLWOR (FOR, LET, WHERE and RETURN) expressions. Upon clicking the OK button, the system will run the XQuery engine to parse these XQuery expressions and generate returned results in XML file format.

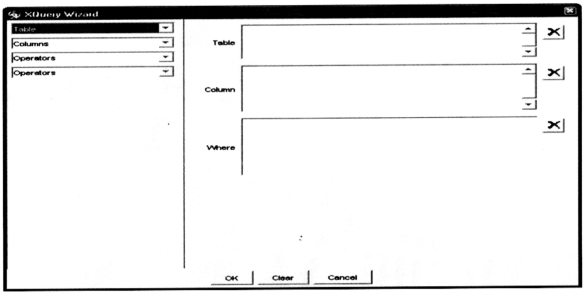

Figure 5-4: JXDB XQuery Wizard Dialog

Figure 5-5 shows the screen how to build XQuery expressions using the simple designed wizard. In this case, the user has selected two XML files, which are book.xml and demo.xml, from the combo drop-down list component named Table. Then these selected XML files are shown in text area component named Table. Following that, the user has to select the required columns to be displayed from the combo drop-down list component named Columns; these columns are only available from the selected XML files. Once the user has completed building the expressions, he/she can click on the OK button to close the wizard. In short, the user is trying to generate a query from book.xml and demo.xml where only the following columns are extracted to be displayed; they are BOOKID from book.xml and USERID and AGE from demo.xml.

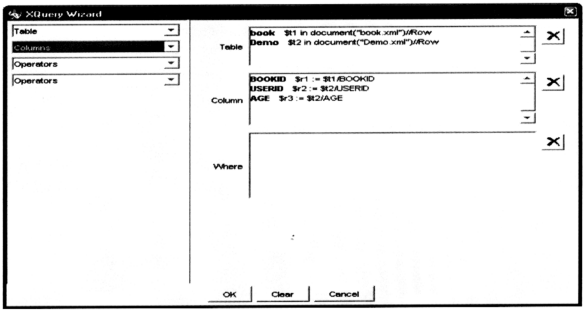

Figure 5-5: Building XQuery Expressions Using XQuery Wizard Screen

After closing the wizard window, JXDB will build the required expressions and display the generated XQuery expressions to the XQuery text area component of the XQuery tab as shown in Figure 5-6. To process the following XQuery expressions, the user has to click on the Run button. This in turn will invoke the Quip XQuery engine that will parse and process the XQuery expressions. At the end of the processing, JXDB will generate a XML file named query1.xml, add the respective filename to the tree node and display the data to the data grid or table component of Data Grid tab.

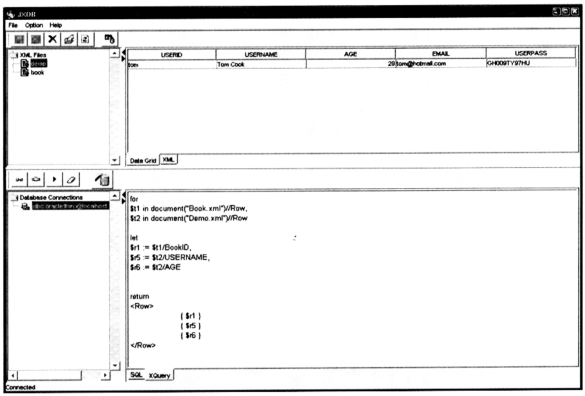

Figure 5-6: Execute XQuery Expressions Screen

5.3.2.1 FLWOR Expressions

According to XQuery's working draft, the most powerful constructs to generate new structures or sequences, are FLWOR expressions (Cagle et al., 2002). FLWOR is short for **FOR**, **LET**, **WHERE**, **RETURN** and FLWOR expressions (pronounced as FLWoR) are not programs. In particular, as explained from previous chapters, there are vast differences between the data model of SQL and XQuery even though both are tools to manipulate databases but they were designed to handle databases with significantly different data models. This feature has been integrated into the XQuery wizard as one of its features. It is obvious that FLWOR is the XQuery version of similar resembles of full-blown SQL Select clauses shown below:

FOR *variable IN expr*
LET *variable := expr*
WHERE *expr*
RETURN *expr*

Below are the common SQL Select clauses as shown below:

SELECT *column-list*
FROM *table-list*
WHERE *expr*
GROUP BY *column-list*
HAVING expr

From the comparison above, it is obvious that there are some similarities between SQL and XQuery clauses, though both of them have different data models. Nevertheless, SQL is known to be less flexible compared to XQuery because it can only deal with tuple-sets. For example, the *column-list* stated above is composed of columns from queried tables, expressions involving columns from the queried tables and literal values. On the other hand, XQuery FLWOR expressions are more flexible compared to SQL, for example, some of the features include FLWOR expressions always return a sequence of nodes or simple data types, all FOR, LET, WHERE, and RETURN clauses take expressions as arguments, and this can include other FLWOR expressions. For example, one of the use cases in W3C XML Query Use Cases Working Draft November 2002 states that, to find cases where a

user with a rating worse (alphabetically, greater) than "C" is offering an item with a reserve price of more than 1000. Below is the solution stated in XQuery expressions:

```
<result>{

FOR $u in document("users.xml")//user_tuple, $i in document("items.xml")//item_tuple

WHERE $u/rating > "C" and $i/reserve_price > 1000 and $i/offered_by = $u/userid

RETURN

< warning>{$u/name}

{$u/rating}

{$i/description}

{$i/reserve_price} </warning>  }</result>
```

This will yield the following expected result:

```
<result> <warning>

<name>Dee Linquent</name>

<rating>D</rating>

<description>Helicopter</description>

< reserve_price>50000<reserve_price>

</warning></result>
```

5.3.3 Implementation of Data Transfer Component Systems

A common problem now in our XML community is how to map and transfer XML data to and from back-end databases. There are many different methods of transferring data but most are basically doing similar approach that is by mapping and conversion of data structure before transferring to databases. Moreover, as one already knows that XQuery still does not support direct updating of XML data to databases (Igor et al, 2001). In particular, this section discusses briefly JXDB's data transfer component systems, which is the core data mapping and transferring engine inside JXDB system. This component system consists of several components to collaborate together to perform the task, they are Query Interpreter, Mapping, Transform, Transfer and DBTransaction components. In order to integrate XML data between XML documents and relational databases, we have to do some mapping on the XML structure defined in XML schema, such as DTD, to the database structure defined in database schema (Bourret, 2001). The transformation and transfer components in this case are built on top of the mapping component. To transfer data from multiple XML documents to relational databases, there are few steps involved. First, the Query Interpreter component uses XQuery wizard where the user enters the XQuery expressions to query and retrieve XML data from multiple XML documents, and it will generate new virtual XML results. Some of the XQuery expressions supported here are Path expressions, Element Constructors, FLWOR expressions (FOR, LET, WHERE, RETURN), XQuery functions and operators.

Subsequently, it will display this virtual XML results on the data grid control and also create DOM objects in memory for further manipulation of the XML results. From this data grid control, the user can insert, update or delete the in-memory XML results before

mapping and transferring these data to relational databases. Before transferring, the Mapping component will perform the necessary mapping from XML schema to database schema based on the built-in mapping files or templates. JXDB provides each different database a specific data access builder adapter or rather database table builder adapter that is designated to perform the required mapping processes because of the different data types supported by these database vendors. Here, JXDB will save a copy of XML results in physical XML format in the predefined destination. Once the mapping is in place, the Transform component will transform the XML data to Java objects before invoking Transfer component to transfer the data to the back-end relational database. The DBTransaction component will manage the whole transaction during the data transferring process, and it is also responsible to establish and manage connection pooling to the databases.

There are two commonly used mapping strategies to model XML data in XML documents; they are table-based mapping and object-relational mapping or object-based mapping (Bourret, 2001). There is a need to do mapping from its XML schema to database schema before transferring the data between XML documents and relational databases because XML structure is not the same as database structure. In fact, both mappings are bidirectional meaning that they can be used to transfer data both from XML documents to the databases and from the database to XML documents. We have studied the pros and cons of both mapping strategies, and decided to customise and use the table-based mappings, though it only works with a limited subset of XML documents depending on the application requirements. Despite this, this mapping satisfies our requirements even though there are some tradeoffs in return. These table-based mapping strategies are based on predefined built-in mapping templates.

On the other hand, to transfer data from relational tables to XML documents, the implementation strategy is basically similar. First, the users issue a SQL SELECT statement to the database to select the required data from a table or multiple tables using a JOIN statement, and then download the result sets to the client's side and generate virtual XML views using DOM API to transform these data stored in persistent Java objects format to XML format. Next, the system also generates its respective XML schema embedded inside the XML document itself after it has downloaded the result sets to the client's side because XML schema information might be needed later on in further mapping process. The newly generated virtual XML view is presented back to the users via the data grid control. Here, the user can continue to manipulate the data like before, such as insert, update or delete, before going through the same process of mapping and transferring the modified data back to relational databases for updating. Figure 5-7 shows the graphical overview of JXDB's data transfer strategy.

5.4 JXDB Testing Strategies

In order to develop and deliver any robust and reliable system, a high level of confidence that each of the components in JXDB system will behave as specified, collective behaviour is correct and no incorrect collective behaviour will be produced (Bahrami, 1999). It is important to understand that not only each of the components needs to be tested thoroughly, but their collective behaviour must also be examined to ensure maximum operational reliability. Here, JXDB system is tested with the intention of verification and validation to determine whether it conforms to the specifications of requirements and have met all the functional specifications as specified in the earlier chapters as well as any added new requirements during the software development process. Furthermore, during the software development process, a test model has been designed to confirm the validity of the other models produced during the software development cycle life conducted in the previous chapters. Besides that, it is essential to test the whole system and detect any bug or error that might occur in the system. One must realize that testing must take place on a continuous and refining basis (Bahrami, 1999). This refining process must be continued throughout the entire development process until satisfied results are obtained. During this iterative testing process, smaller pieces of the application is being testing simultaneously throughout the system development life cycle, and from these smaller pieces of software, the prototype will be transformed incrementally into an actual and complete application.

JXDB testing strategies are designed to apply each use case instances specified in the use case model shown in Figure 4-1 as test scenarios. These identified use cases produced during the analysis process can be used to test and verify the JXDB design as well. For example, once the design is complete, the users can walk through the steps of the scenarios

5.4 JXDB Testing Strategies

In order to develop and deliver any robust and reliable system, a high level of confidence that each of the components in JXDB system will behave as specified, collective behaviour is correct and no incorrect collective behaviour will be produced (Bahrami, 1999). It is important to understand that not only each of the components needs to be tested thoroughly, but their collective behaviour must also be examined to ensure maximum operational reliability. Here, JXDB system is tested with the intention of verification and validation to determine whether it conforms to the specifications of requirements and have met all the functional specifications as specified in the earlier chapters as well as any added new requirements during the software development process. Furthermore, during the software development process, a test model has been designed to confirm the validity of the other models produced during the software development cycle life conducted in the previous chapters. Besides that, it is essential to test the whole system and detect any bug or error that might occur in the system. One must realize that testing must take place on a continuous and refining basis (Bahrami, 1999). This refining process must be continued throughout the entire development process until satisfied results are obtained. During this iterative testing process, smaller pieces of the application is being testing simultaneously throughout the system development life cycle, and from these smaller pieces of software, the prototype will be transformed incrementally into an actual and complete application.

JXDB testing strategies are designed to apply each use case instances specified in the use case model shown in Figure 4-1 as test scenarios. These identified use cases produced during the analysis process can be used to test and verify the JXDB design as well. For example, once the design is complete, the users can walk through the steps of the scenarios

to determine if the design enables the scenarios to occur as planned. Hence, these use cases and its scenarios can be applied as the test scenarios that drive the test plans. Ultimately, the sequence of testing activities is designed to be closely matched with the sequence diagrams in Chapter 4. Particularly, the outcome of a testing execution is the estimated test result. This test result validates the consistency of each model as well as mapping to its respective other models. Moreover, as mentioned in Chapter 3, these systems' test results also need to be tested for accuracy and consistency. Furthermore, JXDB testing strategies include testing from its subsystems and then proceed to the whole layered system, and are carried out iteratively from as early as in the design phase in order to refine and validate the requirements. Hence, in this case, the testing begun with the subsystem testing and once all the subsystems have been tested, it will continue with the function testing, integration testing and finally system testing. Generally, these are also known as bottom-up testing approach. As a matter of fact, during the system verification phase, white-box testing will be conducted to test how well the implementation of the system design is. On the other hand, black-box testing of the system is to test whether or not the system's external behaviour matches that described by the requirement specification.

At the initial testing phase, each subsystem in JXDB system is treated as individual components and they are tested as part of the class or an instance of a class, which is also known as object, that represents the smallest unit that can be tested during the subsystem interaction. From here, the testing will proceed to other related classes or sets of classes that will interact with each other during their interaction. In object-oriented development, white-box testing concentrates on module that consists of classes' operations or states. Hence, all the classes in JXDB's component systems were tested accordingly where the state of each class is tested according to its sequence of operations defined in that respective class. For

example, white-box testing consists of testing the operations of a class such as passing of messages, logical loop, and data flow and so on. Basically, white-box testing, which is also known as structural testing, is the most common testing method to test these subsystems and to determine all the possible paths within the application system, such as in Database Connection and Data Transfer component systems, and so on.

As mentioned earlier, use cases were used as test scenarios as they can describe how the system is to be used. Black-box testing or functional testing is performed using sample data to test the overall functionality of the JXDB system. For example, this testing includes testing all the interfaces and the interaction between the actors and the system as described in the use cases. All these testing must be done iteratively starting from design to implementation phase where each phase is refined until all the system functions work according to their functional specifications.

The final stage of the testing strategies is integration and system testing for the whole layered system. The integration testing is performed to ensure that the integration between the subsystems was correct and can perform the defined functionalities, and also conform to the overall system specifications. In contrast, system testing involves examination of the whole JXDB system, which includes all the software or hardware components and any interface available. At this stage, the whole layered system is then checked for validity and verification of whether it has conformed to all the JXDB system's objectives and functionalities as defined during the requirement and analysis stage. Last but not least, it is important that testing must take place on a continuous and refining basis throughout the development process until the users are satisfied with the end results.

example, white-box testing consists of testing the operations of a class such as passing of messages, logical loop, and data flow and so on. Basically, white-box testing, which is also known as structural testing, is the most common testing method to test these subsystems and to determine all the possible paths within the application system, such as in Database Connection and Data Transfer component systems, and so on.

As mentioned earlier, use cases were used as test scenarios as they can describe how the system is to be used. Black-box testing or functional testing is performed using sample data to test the overall functionality of the JXDB system. For example, this testing includes testing all the interfaces and the interaction between the actors and the system as described in the use cases. All these testing must be done iteratively starting from design to implementation phase where each phase is refined until all the system functions work according to their functional specifications.

The final stage of the testing strategies is integration and system testing for the whole layered system. The integration testing is performed to ensure that the integration between the subsystems was correct and can perform the defined functionalities, and also conform to the overall system specifications. In contrast, system testing involves examination of the whole JXDB system, which includes all the software or hardware components and any interface available. At this stage, the whole layered system is then checked for validity and verification of whether it has conformed to all the JXDB system's objectives and functionalities as defined during the requirement and analysis stage. Last but not least, it is important that testing must take place on a continuous and refining basis throughout the development process until the users are satisfied with the end results.

5.5 JXDB Evaluation

Although we have reviewed and compared different XML middleware approaches and looked at how several relational databases handling integration with XML data, we have not looked into how well these XML middleware approaches perform the execution of XML XQuery and the performance rate of XML data transfer as well as updates to the databases. In this section, we will evaluate and compare JXDB with other XML middleware benchmark. Besides that, we have used relatively simple benchmark parameters to do JXDB evaluation and comparison from the aspects of performance, usability, scalability and reliability. However, the hardware used for performing the evaluation tests was not adequate to run extensive tests of large volumes of XML insertion records and execution XQuery queries. As XML middleware technologies are relatively new in the market comparatively with relational databases, more intensive research work needs to be done on XML middleware approaches to make them viable commercial products.

We have selected XML-DBMS as the benchmark that could provide a better quantitative comparison with JXDB. In this study we tried to compare the three different approaches to help the application developers choose the right approach for their specific application. Nevertheless, none of these three approaches can satisfy requirements of all the different kinds of applications in today enterprise world. One of the important criteria to consider when choosing the right XML technology is that XML middleware approach seems to be a good option in a multi-tier application environment, especially when it is distributed. Others good candidates for middleware approach may be from applications that are not highly-coupled with the back-end database or even applications using legacy databases and

where the XML document format is not being changed frequently. Both JXDB and XML-DBMS are XML middleware software for transferring data between XML documents and relational databases. Middleware software is a software component that sits between an application and a database. Usually most XML middleware are sets of Java API packages where they are not standalone applications and requires a reasonable amount of Java programming skills and other skills to use, such as SQL, XML and XQuery. Even though they usually provide a straightforward programming interface that makes them fairly easy to use. Developers who choose to use this approach would require writing their own sample application to implement these APIs to transfer data, such as XML-DBMS. For example, this sample application can be run from the command line to transfer data between a database and an XML file. However, this would incur unexpected errors or bugs if the developer does not understand well enough how to use these APIs.

5.5.1 JXDB

Nevertheless, now with JXDB, any developer can start using the system without having to acquire any in-depth Java programming skills or even SQL, XML and XQuery skills. This is because JXDB is a GUI-based XML middleware where it provides a user friendly graphical user interface together with sets of object-oriented and reusable Java API packages. Furthermore, the user can transfer data to any relational database with just a few clicks of buttons without having to write any code. Thus, the learning curve is low and the productivity rate is high comparatively if the user were to build the same system using JXDB. Alternatively, JXDB and XML-DBMS are able to preserve the hierarchical structure of an XML document, as well as the data itself, which includes character data and

attribute values in that document. Nonetheless, XML-DBMS handles more efficiently compared to JXDB because it is also able to preserve the order in which the children at a given level in the hierarchy appear (Bourret, 2001). Despite this, for data-centric applications, such order is not significant and the application will run much faster without this feature. JXDB and XML-DBMS are software that are designed to transfer only the data inside the document and not the document, therefore it does not preserve document type declarations, nor does it preserve physical structure such as document encodings, CDATA sections or entity use. Above all, it does not attempt to implement a document management system on top of a relational database system.

In order to transfer data in any XML file to the database using JXDB, the user only needs to click on the selected XML file icon, select which connection node to use to transfer the data and finally click on the transfer button to begin the data transfer process. Please refer to APPENDIX C for further details on how to transfer data via JXDB application. At the moment, JXDB supports multiple connections to several popular databases, such as Oracle 9i, Microsoft SQL 2000, MySQL and Access. However, JXDB does not incur much JDBC connection overheads when accessing the data from multiple databases because once these result sets are extracted from the database, they are downloaded and saved as XML file format on the client's desktop. Most importantly, JXDB supports data transfer process across multiple databases by extracting from multiple XML sources using powerful yet versatile XQuery technology.

Figure 5-8 depicts the overall JXDB implementation approach. The core feature of JXDB that differ from XML-DBMS is that it provides built-in mapping files and database table builder adapters, hence the user do not need to create a separate mapping file and also relational schema file when retrieving or transferring to and from databases. In fact, JXDB provides XQuery technology to extract XML data from multiple XML files. In short, JXDB hides all detailed implementation from the user without having the user to learn tedious coding skills.

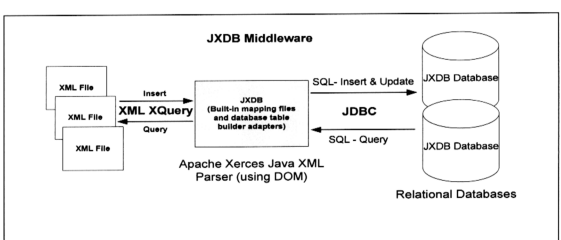

Figure 5-8: JXDB Middleware Approach

5.5.2 XML-DBMS

On the contrary, if you were using XML-DBMS to transfer data from any XML file to the database, you need to create your own mapping file using XML-DBMS mapping language in a ".map" format. For example, to transfer the data from Users.xml document to the database, we need to map according to the mapping file defined in Users.map. You might

need to use the following code to transfer the data from Users.xml document and this uses the Map, DOMToDBMS, and DocInfo classes from XML-DBMS.

```
// Use a user-defined function to create a map object, conn1 is the created database connection handler.
Map map = createMap("Users.map", conn1);
// Use a user-defined function to create a DOM tree over Users.xml.
Document doc = openDocument("Users.xml");
// Create a new DOMToDBMS object and store the data.
DOMToDBMS domToDBMS = new DOMToDBMS(map);
DocInfo docInfo = domToDBMS.storeDocument(doc);
```

One of the drawbacks is that XML-DBMS does not work well with Microsoft SQL Server because the DOMToDBMS class, which is responsible to retrieve and transfer data from the database, requires more than one result set (JDBC connection) to be opened at any given time, whereas SQL Server only allows a single open JDBC Statement per Connection (Bourret, 2001). The work-around solution to this problem is to pass a JDBC 2.0 Data Source to Map object and allow it to create Connection objects whenever it needs them. However, at the moment, the DOMToDBMS class is only able to retrieve data from a single table if it were to be used with SQL Server. Still, it does not provide transferring of data back to SQL Server.

Figure 5-9 depicts the overall XML-DBMS implementation approach. XML-DBMS requires the user to create a mapping file using its mapping language when retrieving and transferring data to and from the databases. Instead of directly retrieving and transferring the data to and from the databases, it needs to do object-relational mapping and transformation of the XML data using the created map file and relational schema every

time they transfer the data to the database and vice versa. This situation gets more complex if the XML schema and/or map file of the XML file are changed frequently. Another similar subtle bottleneck is that if there are lots of different types of XML file structures and frequently new types of XML are being introduced, it would cause more complicated issues in mapping and transformation of schema. Basically, JXDB and XML-DBMS use XML DOM to parse and process XML files; this would encounter other application bottleneck problems, such as performance, scalability and reliability, or out-of-memory resources, when processing large volume of XML data. In the near future, XML-DBMS will be able to use SAX instead of DOM to parse large volume of XML data to overcome this issue.

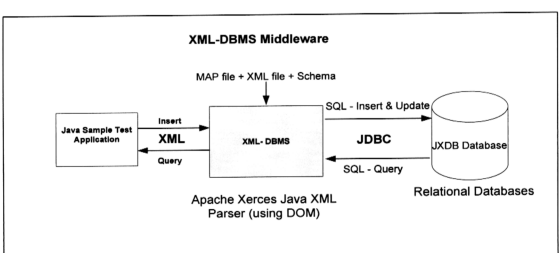

Figure 5-9: XML-DBMS Middleware Approach

5.5.3 Comparison between JXDB and XML-DBMS

Based on the evaluation above, we have summarised the review results in Table 5-1.

Table 5-1: Comparison between JXDB and XML-DBMS

Features	JXDB	XML-DBMS
Performance	Use DOM object to process XML files. Main bottleneck is when processing large volume of XML data. JXDB has optimised its performance by downloading data to client's side before any XML processing is done.	Also use DOM object. Map object needs to create more database connections when it needs them during retrieving and transferring data to and from the database.
Scalability and Reliability	Provide load balancing between middleware and databases without impacting applications. Useful in distributed and multi-tier architecture applications.	XML-DBMS is also a middleware, so it can be optimised to provide load balancing in distributed and multi-tier applications.
Usability	Provide interactivity using wizard-based approach.	Lack of GUI that leads to complexity in using Java APIs and classes.
Mapping Rules	Allow flexibility of specifying mapping definitions via its embedded built-in functions and declarative approach for different XML files. Hide the complexity of mapping from the users.	Need to create mapping files and relational schemes for different XML files. A Map object is used to describe an object view of the element types and attributes in an XML document and how to map this view to the database via object-relational mapping approach.
Relational Database Support	Oracle, MS SQL, MySQL and Access. Support any database with simple customisation of a database table adapter.	Support most databases, does not work well with MS SQL; maybe due to the features of JDBC Driver for MS SQL.
XQuery Support	Support XQuery.	No.

5.6 Summary

In summary, this chapter describes the overall JXDB implementation and testing strategies. Above all, Java 2 technology was selected to be the main programming language used to develop JXDB system and Borland JBuilder 8 Enterprise was used as the IDE for the development and testing environment. The core XML architecture of JXDB framework for XML processing and parsing is built using DOM model from third party software, which is Apache Xerces2 Java Parser.

Basically, the implementation of JXDB can be divided into several business specific component systems, which were developed independently before integrating them together during integration phase. Firstly, JXDB XML-based Interface application system is the first to be implemented because it is the core framework's user interface of the application. Its main screen consists of all the required java swing components and containers to form the user interface of JXDB system, such as the interface to load the XML files and display as a tree view. Following that, the next phase of implementation is to develop the Database Connection component system that is responsible to create and manage the database transaction activities and data retrieval activities. The third phase of implementation is to create the XQuery Wizard in order to assist users to begin using XQuery expressions without going through difficult learning curve. Inside this XQuery Wizard is a simple XQuery processor to interpret these XQuery expressions. For example, the XQuery expressions supported here include Path and FLWOR expressions (FOR, LET, WHERE, RETURN) and so on. By using this XQuery Wizard, the users are able to extract the required XML data from multiple XML files and generate a new XML file. The last component system developed is the Data Transfer engine. This subsystem is the most

important as it is the core engine of JXDB framework. This is because it is responsible to perform the key objective of JXDB functionality that is to integrate XML data to and forth databases seamlessly and efficiently using JDBC connectivity. Here, it consists of several core components that will work together to perform the data transfer processor activities; they are Query Interpreter, Mapping and Transform, and Transfer components. Once each of these sub-component systems is in place, they will be integrated together to work to perform the required tasks of JXDB system. During the entire implementation phase, testing and debugging processes are also performed simultaneously to test for errors and bugs such as logical errors. Lastly, the final phase before releasing JXDB system to its end users is the testing phase. After implementing the framework architecture as a layered system, JXDB is then tested by interfacing its system components to its underlying domain business specific component systems, middleware components and system software components. Besides that, each component system is tested standalone and also as part of the layered system. This is essential because by testing the interoperability of the components acting as a packaged subsystem is important to ensure the high degree of reusability of these components. Last but not least, the use case diagrams and interaction diagrams presented during the analysis and design phase provide the necessary testing scenarios and guidelines in testing how well the final implementation works and also validating the system functionalities whether they conform to the defined system objectivities and specifications as stated earlier.