RepCom: A Report Generator Component System using XML-driven, Component-based Development Approach

Leong Chee Hoong and Lee Sai Peck
University of Malaya, Kuala Lumpur, Malaysia.

chleong98@hotmail.com saipeck@fsktm.um.edu.my

Abstract

It is undeniable that report generation is one of the most important tasks in many companies regardless of the size of the company. A good report generation mechanism can increase a company's productivity in terms of effort and time. This is more obvious in some startup companies, which normally use some in-house report generators. Application development could be complex and thus software developers might require substantial efforts in maintaining application program code. In addition, most of the report generators use a different kind of format to store the report model. An application is no longer considered an enterprise-level product if XML is not being used elsewhere. This paper introduces a XML-driven and Component-based development approach to report generation with the purpose of promoting portability, flexibility and genericity. In this approach, report layout is specified using user-defined XML elements together with queries that retrieve data from different databases. A report is output as an HTML document, which can be viewed using an Internet browser. This paper presents the approach using an example and discusses the usage of the XML-driven report schema and how the proposed reusable report engine of a customisable report generator component system works to output an HTML report format. The customisable report generator component system is implemented to support heterogeneous database models.

Keywords: report model, report schema, report generator, XML, Component-based development

Introduction

It is undeniable that report generation is one of the most important tasks in many companies regardless of the size of the company. A good report generation mechanism can increase a company's productivity in terms of effort and time. This is more obvious in some startup companies, which normally use some in-house report generators.

Application development could be complex and thus software developers might require substantial efforts in maintaining application program code (Cleaveland 1988). In addition, most of the report generators use a different kind of format to store the report model. In common practice, a big company normally uses more than one report generator to cater for their reporting needs. The lack of a generic format of report model has the impact that reports generated in one report generator very unlikely work on another report generator due to the proprietary format used by different vendors.

An application is no longer considered an enterprise-level product if XML is not being used elsewhere (McLaughlin 2002). This paper introduces a XML-driven and Component-based development approach to report generation with the purpose of promoting portability, flexibility and genericity. In this approach, report layout is specified using user-defined XML elements together with queries that retrieve data from different databases. A report is output as an HTML document, which can be viewed with an Internet browser.

The report generation mechanism of this approach supports heterogeneous database models, and therefore, reports generated by an application are independent from other database models. The report layout and content can be specified using XML elements, which eventually made up the report schema. The XML report schema can be used to help application developers create reports even much more faster as well as code maintenance can be relatively done much more easier.

In Section 2, the conceptual architecture for realising the XML-driven and Component-based development approach to report generation will be described. Section 3 presents the report schema with an example of the usage through an application system. Section 4 presents the report model and report engine, which is the core component of the system. Section 5 describes the data mapping to reports. Finally, a conclusion is drawn in Section 6.

Overview

The proposed conceptual architecture for realising the XML-driven and Component-based development approach to report generation consisting of various conceptual components is depicted in Figure 1. The database contains dynamic data, which are changed from time to time, and therefore, the corresponding contents of a generated report will change accordingly whenever the report model is executed (Chan 1998).

The report schema is in XML format. The report schema defines the possible layout of the report. Each report has its own report schema and report model. The report model is the representative of the report schema in Java format. It is derived from the report schema after the report engine has parsed it. The parsing process is done by three main components, namely report parser, DOM parser and lastly the report factory component. It takes the report schema as input and produces the report model as output.

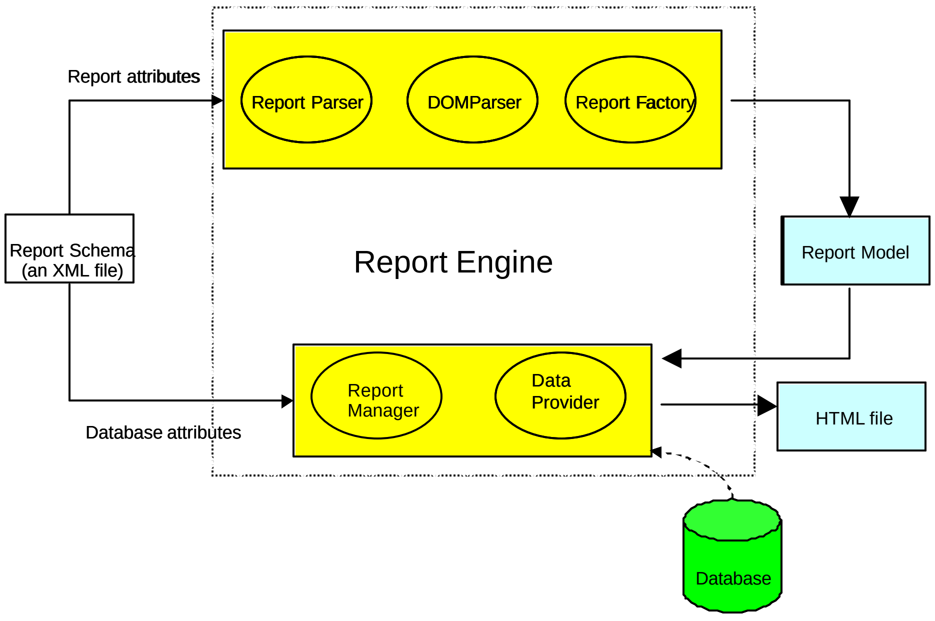

Figure 1: A Conceptual Architecture.

Only a successful parsing will generate the report model.

The report model consists of the report layout and attributes such as report title, report name, report header, report data, report queries and report footer, database driver type and URL. Any queries specified in the report schema will be converted into proper SQL statements by the report engine. The data are drawn from the database to fill the report. Data will be restructured in the report model before it can be understood by the report manager. The report manager is only capable of reading data row by row.

In this approach, it is assumed that different databases use queries expressed only in SQL query language which is widely used in multiple database models such Oracle, DB2, Informix and SyBase. Database queries imposed on a specific database schema and database are specified in the report schema. These SQL query statements defined in the XML report schema are used to retrieve the data from the database to fill the report. As reports are desired to be presented via an Internet browser, therefore, HTML reports are considered for this purpose.

Report Schema

The modeling of a report starts with the definition of a report schema, which captures the layout structure of the report. The report schema will be parsed and transformed into a report model by the report engine. A good report schema will help in further processing of the report model as well as generating the final report using the proposed approach described in Section 1.0. It is important to note that one report will have only one report schema and one report model respectively. Specifying a report with this approach is a matter of putting in the heading of the report and titles for various rows and columns with the data contents left to be specified with the help of the SQL query language. The example of the full report schema for a student report is shown as below:

```
<?xml version="1.0"?>
<!DOCTYPE JavaXML:Report SYSTEM "JavaXML.dtd">
<JavaXML:Report id="StudentReport">
        <JavaXML:Header>
              <JavaXML:Text>UM Student Report </JavaXML:Text>
              <JavaXML:Date>Date: </JavaXML:Date>
              <JavaXML:Time>Time: </JavaXML:Time>
              <JavaXML:PageNo>Page No : </JavaXML:PageNo>
        </JavaXML:Header>
        <JavaXML:Title>
             <JavaXML:Component id="name" length="20">Student Name</JavaXML:Component>
             <JavaXML:Component id="phone" length="20">House Phone</JavaXML:Component>
             <JavaXML:Component id="result" length="20" isTotalRequired="true">Marks</JavaXML:Component>
        </JavaXML:Title>
        <JavaXML:Total>
             <JavaXML:Component id="average" length="40">Average</JavaXML:Component>
             <JavaXML:Component id="total"   length="20" totalMask="##0.0" fid="result"></JavaXML:Component>
        </JavaXML:Total>
        <JavaXML:Body>
             <JavaXML:Data>
                          <JavaXML:Query id="name">
                                        <JavaXML:SQL>select studentname from table_student</JavaXML:SQL>
                          </JavaXML:Query>
                          <JavaXML:Query id="phone">
                                        <JavaXML:SQL>select phone from table_student</JavaXML:SQL>
                          </JavaXML:Query>
                          <JavaXML:Query id="result">
                                        <JavaXML:SQL>select result from table_student</JavaXML:SQL>
                          </JavaXML:Query>
```

```
            </JavaXML:Data>
            <JavaXML:RowTotal>
                        <JavaXML:Query id="total" fid="result">
                                    <JavaXML:SQL>select avg(result) from table_student </JavaXML:SQL>
                        </JavaXML:Query>
            </JavaXML:RowTotal>
        </JavaXML:Body>
        <JavaXML:Footer>
            <JavaXML:Text>End of Report</JavaXML:Text>
        </JavaXML:Footer>
        <JavaXML:DataProvider>
             <JavaXML:Driver>sun.jdbc.odbc.JdbcOdbcDriver</JavaXML:Driver>
            <JavaXML:Url>jdbc:odbc:Student</JavaXML:Url>
        </JavaXML:DataProvider>
</JavaXML:Report>
```

The report schema starts by defining the heading of the report, which consists of elements such as Text, Date, Time and PageNo. In this example, the heading of the report consists of the Text element, which represents the title of the report and followed by the Date, Time and PageNo descriptions. As shown in the above example, the student report contains 3 columns, which are defined by the Title element. The isTotalRequired attribute of the sub-element of JavaXML:Title is used to indicate the column having a total. Having a column with a total, the report schema will have to specify the JavaXML:Total element to reflect the requirement. A JavaXML:Total takes two inputs, one is the caption text and another is the spaceholder for the total value. Typically, the JavaXML:Total is an empty element, in other words, it is legal to specify that this element contains no textual data but attributes. The fid attribute of the JavaXML:Total must always reference back to the component id that is defined in the JavaXML:Title. The impact of wrong reference will cause improper display of the total value in the report.

The report engine depends on the DataProvider component to fill up the report. The JavaXML:Driver and JavaXML:Url elements define the database connectivity protocol and database schema respectively. These two elements are used by the DataProvider component to access the source of the data. As shown in the example, the JDBC OBDC driver is used to access the *student* database schema. It should be noted that the application developer needs to modify the JavaXML:Url and JavaXML:Driver element data for constructing different types of reports, for instance, sales report.

The report ends by specifying the footer element, which contains the *Text* element for the application developer to define the footer description. Finally, a student report can then be constructed according to the content structure described in the report schema. After the SQL query statements are executed, the report will become completely filled. The output is illustrated in Figure 2.

```
UM Student Report              Date:Apr 28 2002   Time:10:00   Page No :  1
______________________________________________________________________________

Student Name      House Phone      Result
______________________________________________________________________________

Leong Chee Hoong   03-80681220       100

Yee Choi Len       03-80681220        96

Lim Choon Peng     03-80681220        65

Average:                              87

End of Report
______________________________________________________________________________
```

Figure 2: A Sample Student Report.

A Data Type Definition (DTD) file is used to constrain all the report schemas by defining the structure of the data. In the StudentReport.xml document, the file JavaXML.dtd, which is located on the local filesystem is declared as the DTD for this XML document by the DOCTYPE syntax. The main purpose of this DTD file is to define how data must be formatted. It defines each allowed element in the XML file, the allowed attributes, the nesting and occurrences of each element, and any external entities. The JavaXML.dtd file is defined like the following after all the allowed element nestings are determined:

```
<!ELEMENT JavaXML:Report
(JavaXML:Header,JavaXML:Title,JavaXML:Total,JavaXML:Body,JavaXML:Footer,JavaXML:DataProvider)>
<!ELEMENT JavaXML:Header (JavaXML:Text,JavaXML:Date,JavaXML:Time,JavaXML:PageNo)>
<!ELEMENT JavaXML:Title (JavaXML:Component+)>
<!ELEMENT JavaXML:Total (JavaXML:Component+)>
<!ELEMENT JavaXML:Body (JavaXML:Data,JavaXML:RowTotal)>
<!ELEMENT JavaXML:Data (JavaXML:Query+)>
<!ELEMENT JavaXML:RowTotal (JavaXML:Query+)>
<!ELEMENT JavaXML:Query (JavaXML:SQL)>
<!ELEMENT JavaXML:Footer (JavaXML:Text)>
<!ELEMENT JavaXML:DataProvider (JavaXML:Driver,JavaXML:Url)>
<!ELEMENT JavaXML:Text (#PCDATA)>
<!ELEMENT JavaXML:Date (#PCDATA)>
<!ELEMENT JavaXML:Time (#PCDATA)>
<!ELEMENT JavaXML:PageNo (#PCDATA)>
<!ELEMENT JavaXML:Component (#PCDATA)>
<!ELEMENT JavaXML:Driver (#PCDATA)>
<!ELEMENT JavaXML:Url (#PCDATA)>
<!ELEMENT JavaXML:SQL (#PCDATA)>
<!ATTLIST JavaXML:Report id CDATA #IMPLIED>
<!ATTLIST JavaXML:Component id CDATA #IMPLIED>
<!ATTLIST JavaXML:Component length CDATA #IMPLIED>
<!ATTLIST JavaXML:Component isTotalRequired CDATA #IMPLIED>
<!ATTLIST JavaXML:Component totalMask CDATA #IMPLIED>
<!ATTLIST JavaXML:Query id CDATA #IMPLIED>
<!ATTLIST JavaXML:Query fid CDATA #IMPLIED>
```

Transformation of Report Schema to Reports

Before a report model is created, the report factory component has to be invoked to transform the report schema to the report model as depicted in Figure 3. After the transformation, the report model will be made up by a set of reusable report components (Jacobson 1997) as shown in Table 1.

Report model

In the report model, a report object is a generic object or a container object for other objects like head object, title object, data object or foot object as explained in Table 1. The header report object and footer report object have only one head object or foot object respectively, whereas, the data report object consists of the combination of title object and data object. This logical report model applies to the above example report schema (StudentReport.xml) where there is only one liner of heading, footer, title description and one whole chunk of report content. A report object is essential in a report model to become a major object holder and this is taken care by the report manager component, which will be discussed in the next section. A report object can have a one-to-one relationship or one-to-many relationship with other objects like title object and data object. It is therefore possible to construct a report object to have more than one title and data object depending on the complexity of the report.

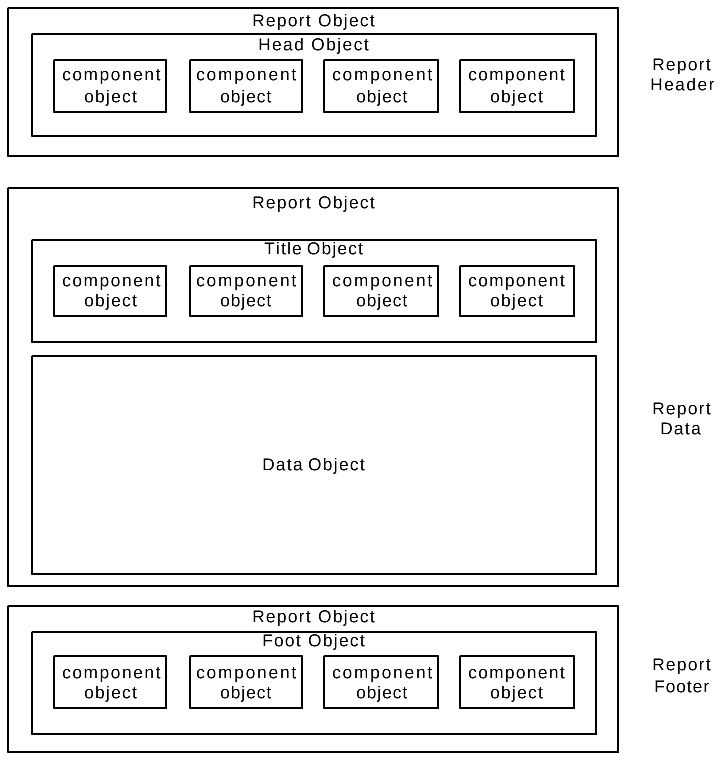

Figure 3: A logical report model.

Objects	**Purpose**
Report object	The root object of the report
Head object	Heading object of the report
Title object	Title object of the report
Data object	The content object of the report
Total object	The total object for a particular column
Foot object	The footer object of the report
Component object	An inner object that contains string value

TABLE 1: Objects in report model

Another key component in the report model is the component object. A component object contains the information like alignment (left, right, justify), text value, length of the text, etc. As shown in the Figure 3, the component object is the inner-most object in a report model. For instance, the JavaXML:Text, JavaXML:Date, JavaXML:Time and XML:PageNo in the header element will be converted into individual component objects respectively.

Report engine

The report engine is the main component of this approach. In general, the report engine is made up of several sub components such as the report parser component, a third-party DOM parser component, report factory component, report manager as well as data provider component and a set of reusable report components.

The report parser component is built on top of the DOM parser. The DOM parser gives a tree structure as output and allows the application to process the "tree" object by accessing the data at each node of the tree. A set of interfaces and classes that define and implement the DOM is specified in the report parser component. The report parser component parses the XML document and prepares the tree structure for the report factory component. A document object will be created after the parsing. The document object is obtained in the report parser component and passed to the report factory to process. A set of methods is defined in the report factory component to process the XML document object. Each node in the "tree" object is processed accordingly to build the report model accordingly.

The transformation from the report schema to the report model is done by the report factory component. It begins by mapping the JavaXML:Header node into the head object in the report model. Subsequently, the JavaXML:Title node will be transformed into the title object, JavaXML:Data node will be the data object and JavaXML:Footer node will be the foot object. The header attributes defined within the JavaXML:Header node such as text, date, time and page number are the component objects in the report header.

As part of the report engine, the report manager is perceived as an interpreter of the report model. The interpretation of a report model is a matter of converting the report objects into a more presentable format. In this case, the report model is converted into HTML format for output display to the user. Each object of the report model is restructured and added the HTML attributes such as font size, font type, alignment, etc. For example, the JavaXML:Header when converted to head object is justified to fit in the default report length with an underline to separate from the content.

Mapping Data to Report Model

The SQL query language forms the contents of the report (Oracle 1995). In other words, in order to map the data retrieved from the database to a report, SQL query language is widely used in this approach. Data can be retrieved either directly from a particular column or specified table views that the application developers need to define or from a particular SQL function.

Figure 4 shows a mapping between the data from the database to the report objects defined in the report model. A report can be simply viewed as a tree. The tree starts from the report node and branches out to header, footer, title and body nodes. In this case, only the body node is further broken down into several sub nodes for discussion.

Using the example of the student report, the student record is mapped directly to the body object. Name, phone and result tuples are extracted from the database. Data will be restructured and mapped to the query under the data node respectively. Each query object under the data node as shown in the figure represents a physical column in the report except the rowtotal's query object. Another possible mapping is the average tuple, which is mapped to the query object of the rowtotal node. The rowtotal's query object is an ordinary row data which represents the running total, for example, the average result from the three students in this case.

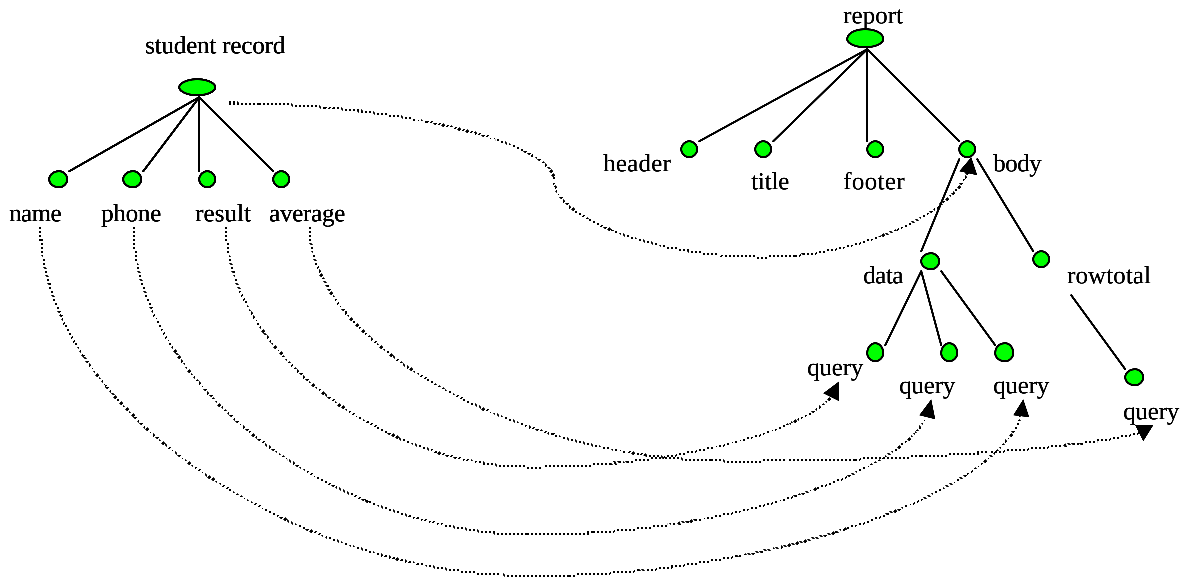

Figure 4: Mapping data to report model

It should be noted that the number of query objects in the data node must be the same with the number of component objects in the title node which is not demostrated in Figure 4. A wrong reference for data node to the title node will cause runtime exception even though the mapping from data to report is correct. The mapping is a straight forward process as long as the XML rules and constraints are followed and used correctly. The report displays the result according to the order of the query element in the report schema if the given *id* attribute of the query element is accurate.

Conclusion

RepCom is designed to help users generate web reports with minimum understanding of programming knowledge. In this approach, a set of user-defined XML tags are developed to allow web reports of different complexities and sizes to be unified for different database models. With this approach, data are retrieved from the database using SQL queries and reports are specified independently as HTML documents. The report structure is defined using the XML language, which provides a lot more flexibility and simplicity. However, application developers have to follow the strict conformity of the XML specifications used in this approach to develop reports.

RepCom was developed to allow application developers to build reports either using the XML report schema or reusable report components directly. The latter provides more satisfaction to the application developers though it is a bit tedious and requires more efforts to master the new components.

The benefits of this approach are simplicity, genericity and independence. The approach is simple because the user-defined XML-based report schema is very easy to undersand and does not require much expertise in any programming languages. Due to the use of HTML format, this approach also provides some degree of genericity compared to other formats. The approach is said to be independent because it does not bind to any database models. Each individual report can be specified to retrieve data from any database system. This is not usually the case with other approaches. In a nutshell, this approach promotes two important technologies (XML and Java) to offer the ease of generating reports and the flexibility of retrieving data from any database systems.

References

Oracle Corporation, U.S.A..(1995). *Building Reports Manuals*. 2.5 edition.

Daniel Chan. (1998). A Document-driven Approach to Database Report Generation. *Proceeding of the Ninth International Workshop on Database and Expert Systems Applications*, Le Chesnay, France.

Ivar Jacobson. (1997). *Software Reuse Architecture Process And Organization For Business Success*. Addison-Wesley , pg 38.

Brett McLaughlin. (2000). *Java and XML*, pg 16-19.

J.Craig Cleaveland. (1988). *Building Application Generators*, pg 25-33.

Biography

Lee Sai Peck is an associate professor at Faculty of Computer Science & Information technology, University of Malaya. She obtained her Ph.D. Degree in Computer Science from University of Pantheon-Sorbonne(Paris I) in 1994 and her Master of Computer Science from University of Malaya in 1990, and her Diploma d'Etudes Approfondies(D.E.A) in Computer Science from University of Pierre et Marie Curie(Paris VI) in 1991. She has published a number of research papers in several computer science journals as well as in local and international conferences. She is a member of IEEE Computer Society.

Leong Chee Hoong is currently a student in the Master of Software Engineering at Faculty of Computer Science & Information technology, University of Malaya. He obtained his Bachelor Degree of Information technology from University of Northern Malaysia in June 1992. His research interest is Component-Based Development, Internet Computing and Reporting Tools.