REPRESENTING VIEWS IN XML

Joumana Dargham, Zeina Alti, and Deema Zurayk
Department of Computer Science
American University of Beirut
Bliss Street, P.O. Box: 11-0236
Beirut – Lebanon

ABSTRACT
View Programming offers software developers a way to write applications having real objects with changing behavior during their lifetime. The benefit of this new approach of programming is mainly revealed in decentralized development where each programmer can view objects from her/his own perspective. When we think of decentralized and distributed development, portability of information comes to mind. A widely accepted portable format for exchanging information is nowadays the eXtensible Markup Language (XML). XML documents are platform independent; thus, can be easily exchanged. In this paper, we will use a new notation to represent views by XML documents and explore what it promotes.

KEYWORDS
XML, Separation of Concerns, Distribution, Views, Viewpoints.

1. INTRODUCTION
With Object Oriented Programming, developers have the opportunity to implement a well modularized, encapsulated, easy to maintain and to reuse code with emphasis on the object [1]. These OOP privileges may be lost when constructing large applications with separate developing teams having each its own subjective view of the same real world object. To solve this problem, View Programming realizes reusability by the following approach: an application object consists of a core object to which we can add and remove functional slices, or Views, reflecting the changing roles of the object during its lifetime. Thus, View Programming offers the ability to handle new concerns as they arise throughout the software lifecycle. It also supports the decentralized development of integrated OO applications and removes ownership dependencies that reduce the reusability of the resulting applications [3]. Since XML technology provides solid foundations in integration of information, we propose to make use of it to help us integrate different classes, Views and Viewpoints, evolving at different phases of the software lifecycle manipulated and used by different persons. Defining views of the core object in XML syntax or defining interactions among agents [5], which can be thought as views/classes in our case, help supporting their implementation and composition. In fact, the idea of interacting roles in [5] is similar to the idea of views in view programming where a core object may have several roles and behaviors according to the views attached to it.

The rest of the paper is organized as follows: In section two, we explain how we benefit from the XML representation. The third section explores how the view (class behavior) may be defined using XML. In the fourth following section, we will discuss the need to combine views and how mapping them to XML documents may help in this composition. Finally, we conclude and present possible future work.

1 Professor Dargham is on leave from University of Quebec at Outaouais
2. WHY XML?

The eXtensible Markup Language (XML) represents data in a familiar tagged form, explicitly separates data from its representation, and provides a self-describing output. Further, markup tags can be user defined in XML documents. These documents may be written according to rules described in a Document Type Definition (DTD) or in a schema. Consequently, XML is capable of representing whatever kind of data and entity one wants; such as documents, services, objects… etc. For our interest, XML can be exploited in OO applications for both the description of a class and the operative use of it. If a class can be described by an XML document that respects a well-defined DTD then we can make use of all present XML technologies to help us in the software development especially in View Programming. Such technologies include querying to retrieve relevant information from the XML documents describing different views of a class. To illustrate, querying using XPATH expressions or XSQL queries can identify methods or variables having same name in different classes. Other technologies to exploit would be parsing the XML document (using XP or SAXON) or better processing the document to transform it according to rules specified in XSLT stylesheets [8]. An XSLT stylesheet is made up of templates that are pure functions to be applied on an XML input document to transform into another. It is analogue in view programming to the viewpoint concept that describes a template rule to be applied on classes to generate views. Finally, the most important benefit that XML offers to view programming is the portability and easy data exchange between different sites. This clearly helps in the decentralized development of OO applications that view programming supports [4]. In fact, we need at a final step to integrate the different views of application developers residing at different sites.

3. DEFINING VIEWS IN XML

In View Oriented Programming a view is defined, like a class, by a set of variables and operations; it specifies a certain functional aspect and represents a certain role for its related objects. The viewpoint construction determines the attributes of a view, and it could be applied to a set of classes that we call base-classes. Basically, a view is an instantiation of a viewpoint with respect to a certain given base-class that should meet the requirements of the viewpoint in order for the view instantiation to take place. In turn, the viewpoint construction provides to each base-class a set of characteristics it defines. In XML terminology, we may think of base-classes and views as elements in an XML document and of viewpoints as rules we may apply on some class elements to generate view elements. But first how a class can be represented in XML syntax as a tagged element? An answer to this can be found in [7] where the author implemented a way to convert any OO source code written in Java to XML notation (JavaML) and vice versa. To support view programming in this framework, all we need is to add definition of view element to the JavaML DTD. However, in the scope of this paper, we do not need to specify how the class is really implemented. What we are interested in is what functionalities and variables the class encapsulates. To illustrate, we provide in figure 1 a simplified DTD describing a class element.

View declaration is similar to class declaration except that when declaring a view, the programmer has to specify the identifier of the view, the base-class name, the viewpoint, and the correspondence list between the base-class and the viewpoint. Mainly, a view element can be decomposed into a viewdef subelement to handle the correspondence with its base-class and a class subelement.

```xml
<!ELEMENT class (name, description?, keyword*, variable*, method*)>
<!ELEMENT variable (var_name, type, description?)>
<!ELEMENT method (meth_name, keyword*, action, description?)>
<!ELEMENT action (description, ret_value?, parameter*)>
<!ELEMENT parameter (par_name, type)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT keyword (#PCDATA)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT ret_value (#PCDATA)>
<!ELEMENT var_name (#PCDATA)>
<!ELEMENT meth_name (#PCDATA)>
<!ELEMENT par_name (#PCDATA)>

<!ELEMENT viewpoint (name, requires, provides)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT requires (variable*, method*)>
<!ELEMENT provides ((variable | method)+)>
```

Figure 1. DTD defining a class.
Remains to define are viewpoints in XML syntax. Viewpoints may be defined independently of the classes to which they apply supporting decentralized development [4]. A viewpoint is a theory that can be applied to a class element satisfying its requirements to generate a view. The generated view will provide extra variables or functionalities to the base-class. Following this description, we can abstract the definition of a view generated for the Employee class and the finance viewpoint in an XML document as in figure 2.

To illustrate, consider the development of an application where we have the Employee class as a core class possibly having instance objects with different behavior. The changing behavior may depend on the user. If the user belongs to the payroll department, she/he may need to print information about the employee’s salary. However, if the user is from the human resource department she/he may need to only print the hours worked by an employee. Two viewpoints may be identified here: payroll and human resource. The first would require the id of the employee and provide a method that prints the salary computed from the number of hours worked by the employee. The second, that of the human resource department, would require the unique id of the employee also but provides a printing method for the number of hours worked only. A sample XML document describing the resulting payroll view is shown in figure 3. XML document describing the human resource perspective will be similar.

Now, our separate views are each in XML documents residing possibly at different site of a distributed software development environment. The problem of variables and functions multiplicities may arise for views attached to the same base class. To elaborate, we can notice that in the previously stated example both views attached to the Employee class involved a print method. Therefore, we must think of a way to combine such views together.

Figures 3. Payroll view of employee class.

4. COMPOSITION OF VIEWS

To solve the problem of variables and functions multiplicities within different views of the same core object, a new view called “composed view” needs to be introduced. This resulting combination view would have to aggregate the behavior of the original ones. Therefore, we need a comparison step to identify conflicts that may occur. Some situations faced seem similar to those when integrating two schema definitions of an XML document in [6] because we need to identify semantic and behavioral relationships between variables and methods defined in the class subelements of our views. We need to point that problems encountered for methods multiplicities involve a wide comparing approach to handle all possible matching cases such as
parameter differences and pre- and post- conditions. For the moment we suppose that an explicit correspondence of methods can by specified by the programmer to resolve any matching ambiguity. As for variables: **Identical** variables, those that have the same name and belong to the same class, and **unique** variables, those that have different names and different definitions of any other variables in the given views to compose, will not cause any problem. However, synonym naming arise when dealing with different names but same definition, which corresponds, to the equivalent semantic relationship. This cannot be resolved without the programmer explicitly specifying correspondence between these variables. For this reason, we propose to treat them as different variables. Our second naming conflict, the homonym naming conflict arises when variables have same name but different definitions; thus, they are **incompatible**. Incompatibility occurs due to data type differences. We may solve data type conflict by expanding data types, by adjusting their scale or taking the union of the disjoint data types. For example, an integer variable and a double variable may be at the end stored as a double. However, if we had a string and an integer we may need to define a union struct. Our aim in presenting composition rules would be to try to solve our composition conflicts at the XML level. In other words, what we are now trying to compare and integrate are really XML documents and existing models for such an integration can be exploited.

5. CONCLUSION

This paper presented an XML representation to define views that possibly do not belong to the same application. The exploitation of XML language offers flexibility, decentralized development and allows the definition of views at different phases of the software application lifecycle. Some research directions for future work maybe oriented towards using this portable representation to help in the implementation of view programming. Leveraging XML technologies such as querying and processing transformation on XML documents may be explored too. Also, we can further study how to benefit from XML technology for automated view document generation. Why not write the viewpoint rules in XSLT stylesheets? We can write template rules in these stylesheets to be applied on the XML class documents in order to transform them into documents defining views. An XSLT stylesheet processor can then take as input the class document to which we wish to apply this viewpoint and an XSLT stylesheet where we describe what transformation to apply on the class to generate the view. A brief description of how this can be done follows as such: first what we need to match are the requirements of the viewpoint in the input class document. For the matching condition, we try to find the exact name of variables and methods in the class element or we take the first ones with a satisfying type. Once the match request is satisfied, the transformation is done. The resulting output would be a view document having a class subelement differing from the input class by methods or variables that the application of the stylesheet added.

REFERENCES


