"SERVICE LOGIC MARKUP LANGUAGE": USING XML SCHEMAS TO REPRESENT AN APPLICATION'S LOGIC

White paper

Brainwave

May 2007

www.brainwave.es

Keywords: XML. Software Development. Software Engineering. Modelling. Development tool. Diagram.

Abstract: The need for Business Processes integration within and among organizations has motivated the emergence of Architectures oriented at Services and the standardization of interfaces between those services using XML schemas. This approach has proved to be an efficient manner to make the cooperation of legacy and new applications easier without any consideration about how each individual service is implemented. In the meantime, at a smaller scale, the development of the services themselves (i.e. code-cutting) has remained mainly a handcraft activity, The purpose of this paper is to present an architecture, which (similarly to e.g. BPEL for process modelling) contains all the necessary components of a service logic, their properties and relations using a specific XML syntax: i.e. a Service Logic Markup Language (SLML). SLML schemas are self-describing and independent of the runtime environment, which actually executes the application on a particular hardware and software platform. The usage of appropriate tools to design the service and generate the SLML schemas used at runtime provides the right level of abstraction so that loose-coupling services can be created. Based on a field-proven implementation of the proposed model, the document specifies the basic XML schemas that shall be parsed and handled by a particular runtime implementation and provides a high level description of a Diagramming Tool used by developers to represent an application logic through the combination of predefined building blocks. These Diagrams are then translated into the corresponding SLML documents and downloaded on a particular runtime platform for execution.

1 INTRODUCTION

B2B and so-called "social computing" are driving the way to new standards, including "Web Services" and "Web 2.0". These models are formalized through specifications such as SOA (Service-Oriented Architecture) or REST (Representational State Transfer) respectively.

Although they address different entities (i.e. the Business community & Communities of individuals) and apply to very different environments, (i.e. Business processes & contents aggregation), these approaches share some fundamental aspects.

Both consider turning applications into Platforms (instead of e.g. modules), which are delivered as services and use presentation interfaces to communicate with end-users and other services. Both consider a loose-coupling implementation (Karl Weick, 1982), i.e. the way the service is built so that it is completely transparent to (either software or human) clients.

They also assume stateless services, uniform interfaces as well as layered and reusable components. Finally in their standardization effort, the industry has formalized these models using XML schemes (Business Process Execution Logic -BPEL- and Simple Object Access Protocol –SOAP- for Web Services and Rich Site Summary –RSS- for syndication of contents).

Those two architectures address standardization of "computing in the large" i.e. linking together software elements in order to cover complex and stable processes (e.g. Business processes), involving heterogeneous environments, geographically spread in the network, built on any software platform and inter-operating through public standard interfaces.

Eventually the advantages of these architectures lie in an easier collaboration among several applications without any consideration of the actual implementation of each service itself; so the different pieces can perform an operation and return the result while ignoring everything about the rest of the components that together form the system. Legacy and new developments can then easily cooperate within a single organization or among different businesses.

By contrast to the concept of "programming in the large", "programming in the small" considers traditional software development and code cutting, using formal languages and data manipulation. The main difference between the "Large" and the "Small" is that while in the former dependencies among services are well established and stable, in the latter, which addresses each of the involved processes, changes in the requirements are frequent and their implementation is dependent on the environment where they reside.

Providing a right level of abstraction in order to isolate the business logic could potentially boost software development lifecycle because applications' designers would be freed from dealing with the details of a particular implementation of the solution and exclusively concentrate on the Functionality. This is achievable by using XML schemas to map functional building blocks and media objects in a similar approach to BPEL for processes interdependencies within a particular business environment and by removing any consideration about the way in which the information is presented to the external world (i.e. a person or another application).

This paper argues that an XML based framework for service execution can describe all the necessary components of an application, independently of its actual implementation.

SLML (Service Logic Markup Language) offers a formal structure containing all the necessary information about code objects, database and media objects as well as their dependencies.

At runtime, a Parser loads this information in memory for execution. The runtime is in charge of taking care of the specifics of the execution environment (e.g. Linux vs. Windows, C vs. Java, Oracle vs. MySQL, HTML vs. VoxML, etc).

Using the right tools (e.g. graphical interface) for designers to represent the business logic regardless of any other implementation aspects, and to translate this business logic into the SLML documents used at runtime, will allow full advantage to be taken of this architecture and offers the following main benefits:

- Better users' satisfaction because the architecture favours a fluid iteration process with users, by quickly presenting a prototype and then implementing progressively additional features and easily including initially unplanned specifications along the development lifecycle.

- Faster time to market due to a higher level of abstraction compared to traditional software development techniques.

- Easier testing and maintenance of the application along its lifecycle because of the systematic reuse of software objects, the self-describing nature of Diagrams and of the XML syntax.

- Vendor independence because while a common SLML schema is shared, developers are free to use a particular implementation of the runtime or any Design Tool that complies with SLML specifications.

In the following sections this document describes the principles of SLML architecture, as well as some implementation's considerations and finally presents an overview of a SLML Service Creation Tool.

2 SLML VS. OTHER MODELS

Traditional software development models as well as Object Oriented design define different entities and organize them in a hierarchical way.

Typically in a Procedural approach, an application is composed of one or several programs built out of different routines, which in turn are made up of statements written in a specific programming language. The code uses parameters and data structures to maintain a session context and apply the application logic. Similarly, Object Oriented architecture defines classes of objects, methods, sentences, parameters and interfaces. Eventually, and independently of the abstraction methodology that is used to design an application, these entities are both necessary and sufficient to specify a particular logic and execute this logic at runtime.

SLML uses an analogous hierarchy as shown in the table below:

Service Logic Markup Language	Object Oriented Programming	Procedural Programming
PROJECT	APPLICATION	APPLICATION
SERVICE	CLASS	PROGRAM
DIAGRAM	METHOD	ROUTINE
NODE	STATEMENT	STATEMENT
FIELD/ATTRIBUTE	PARAMETER	PARAMETER
COMPLEX	INTERFACE	DATA STUCTURE

Table 1: Comparison of different Programming models and hierarchy of components.

A SLML document will then describe each of these elements, including their properties, dependencies and relationships in a formal manner.

At the top of SLML schemes is a Project, which encompasses all the elements that together constitute a computerized solution to a problem.

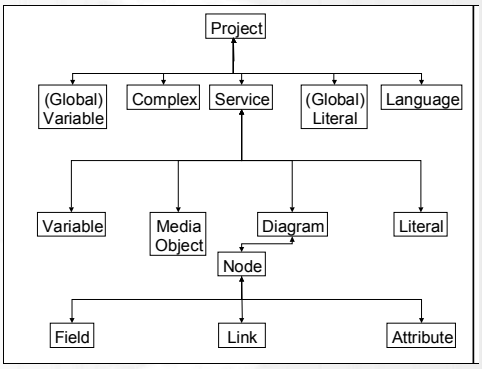

Figure 1: Hierarchy of SLML components.

As the result of an analysis of the problem, a designer will have grouped the Functionality into functional blocks called Services. A Project will always include at least one Service.

Similarly to a Project, a Service will always include at least one Diagram. Diagrams actually describe the Business logic as it can be represented by a flowchart resulting from a detailed design of the application. In the same way, Variables and Literals can be declared (either at the Project or Service level) and used within a Service; in this case they are not seen by other services and can only be used in the Diagrams depending of the Service they belong to.

Media Objects point to templates documents in which the data will be embedded at runtime for presentation to an external user (a human being or another application). In order to do so, the templates enclose tags, triggered on the fly and replaced with the actual data.

Finally, a Diagram is built out of a combination of Nodes. Nodes represent a piece of code equivalent to a simple statement or a more complex piece of logic (e.g. an algorithm, a data input format, etc). The main advantage of the concept of Node is that they incorporate a piece of reusable encapsulated code; so code cutting may be reduced to the creation of additional Nodes.

In the same way that a Media Object points to an external stored document template, a Node points to an external piece of code. The language and API used by that code is managed by the runtime and not relevant to SLML.

Attributes and Fields attached to a Node are used as parameters at execution time and can have the format of a text, a constant, a Boolean value, a value within a discrete number of options, a reference to a Project or Service Literal or Variable, a numeric expression, etc. The difference between Attribute and Field is that the number of Fields for a particular Node can vary from one instance to another (e.g. number of columns as the result of an SQL query).

Links connect Nodes to each other. One Node has generally at least one inbound and one outbound Link. The value of a Link indicates the following executable Node that will follow the Business Logic within a specific Diagram (e.g. OK, NOTOK; ERROR).

3. SLML IMPLEMENTATION CONSIDERATIONS

As it is shown in the description of the XML schemes in the previous section, a SLML document is completely self-describing. All relevant elements, their properties and relationship are defined within an SLML scheme without any assumption about the tools used to create it or the way it is executed. SLML makes no assumption on Design Tools, Operating Systems, Databases, Programming Languages, etc. So SLML can potentially be implemented under any open or proprietary environment. The SLML implementation referred in this document is already available for C, .net and J2EE environments and support provided for Oracle, DB2, MySQL, SQL Server, etc.

Another goal of the SLML model is that by using a modular and hierarchical architecture (represented by the relation Project -> Service -> Diagram -> Node) it encourages the development and encapsulation of reusable software components (i.e. Nodes) or the reuse of a combination of them (i.e. Diagrams and Services).

The level of abstraction of the software coded in a Node is equally independent of any SLML consideration. For example, a Node can encapsulate a simple instruction (e.g. CASE) or a more complex set of sentences building a Business function (e.g. Send a Message where the message format is selected as an Attribute: SMS, SMTP, etc.).

It is important to emphasize that the focus of SLML architecture is put on the reuse and combination of logical pieces of code encapsulated in Nodes. How this code is generated or created is not within its scope. However, the implementation of the SLML architecture, discussed here uses libraries of predefined classes of code objects. The reasons for that approach are reliability, performance and designers' productivity.

Using traditional programming techniques, a developer would write a set of sentences using some formal language to perform a function. Instead SLML architecture provides the means to perform a similar function by simply telling the runtime to execute a pre-packaged Node using Fields or Attributes and, depending on the result (Event) to follow through the corresponding exit branch to the next Node. That allows the representation of each Node within a graphical Software development tool as e.g. an icon and a dialog box for provisioning parameters. At execution, the runtime would take care of executing the corresponding code as specified by the designer.

The following example shows how a graphical tool can represent the Diagram logic using icons to represent a "Switch" and "Menu" Nodes and the Links between them, depending on the Value of a Variable (1, 2 or other):

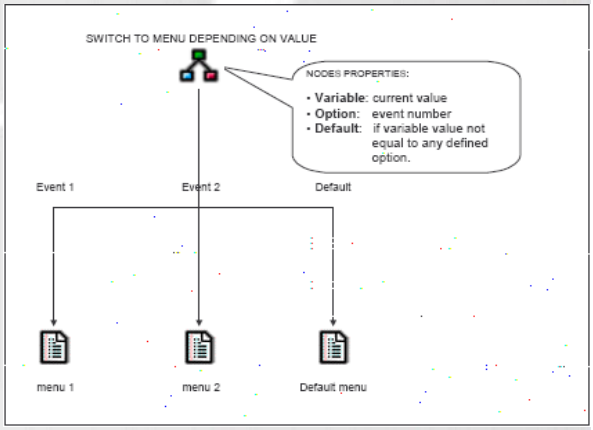

Figure 2: Diagram representation of Service logic.

The following lines show the SLML schemas used for the representation of the "Switch" Node within a Diagram as part of an SLML document:

```
+<diagram id="3">
  <title>Nodos</title>
 -<nodelist>
  +<node id="4" class="show">+<node id="3" class="show">
  +<node id="2" class="show">
  +<node id="0" class="start">
  -<node id="1" class="switch">
   -<attributelist>
     <variable scope="local">1</variable>
    </attributelist>
   -<linklist>
    -<link id="1_1">
     -<attributelist>
       <event>1</event>
       <from>1</from>
       <to>2</to>
      </attributelist>
     </link>
    -<link id="1_2">
     -<attributelist>
       <event>2</event>
        <from>1</from>
        <to>3</to>
      </attributelist>
     </link>
    -<link id="1_D">
     -<attributelist>
        <event>D</event>
        <from>1</from>
        <to>4</to>
      </attributelist>
     </link>
    </linklist>
   </node>
  </nodelist>
</diagram>
```

Figure 3: extract of an SLML document, describing a Switch Node.

The runtime would execute the following (Java) code to extract the value of the Variable and corresponding branch option.

```
“      case SWITCH:
           attribute = node.getAttribute("variable");
           env.setOption(Api.getStringValue(env,attribute));
           break”

“ id = node.getNextNodeId(event);
  if (id == null) {
     id = node.getNextNodeId(DEFAULTC);
   } else if (function == SWITCH) {
     id = node.getNextNodeId(DEFAULT);
   }”
```

Figure 4: Java code executed at runtime.

As shown in the example above, using a Graphical tool to create the Diagram logic, there is no need for the designer to write any code or even to know anything about the actual code executed at runtime. Moreover, as the code itself has hopefully been debugged, tuned and optimized for a particular environment; there is no risk of introducing defective or low performing code.

Actually, the Designer should be protected from any interference or constraints due to the system execution environment or the interfaces used to communicate with the external world; SLML has been thought so that designers can concentrate exclusively on the Business Logic and Users' requirements.

As soon as a designer becomes familiar with SLML, e.g. using some tool or SDK to build Diagrams and combine them into Services under the umbrella of a Project, productivity in terms of the number of programmed sentenced that are produced in a period of time as well as the quality of the output (as measured in the number of software defects) is boosted. Providing that a comprehensive library of Nodes is made available, the designer will not have to write a single line of code (i.e. develop new Nodes) in order to create and develop a Project. Field experience proves that designers do not even need to have any specific knowledge of the environment (e.g. Proprietary, J2EE), not even of the programming language (e.g. C, Java) used in the Nodes.

A compliant SLML scheme shall be transparently understood by any Parser (just ignoring possible extensions not supported by a particular implementation); and Reverse engineering of an SLML document toward a development tool easy to implement because of the formal and univocal structure of the XML standard and its self-describing nature.

4. SOFTWARE CREATION TOOL

Figure 5 shows a screenshot of a tool designed for creating an SLML output based on a Diagramming graphical interface. The tool provides Designers with libraries of pre-programmed Nodes that are combinable into Diagrams and represented by Icons. A particular SLML engine executes at runtime the different Services based on the SLML document produced by the tool. The tool also provides for Designers' profiling and versioning.

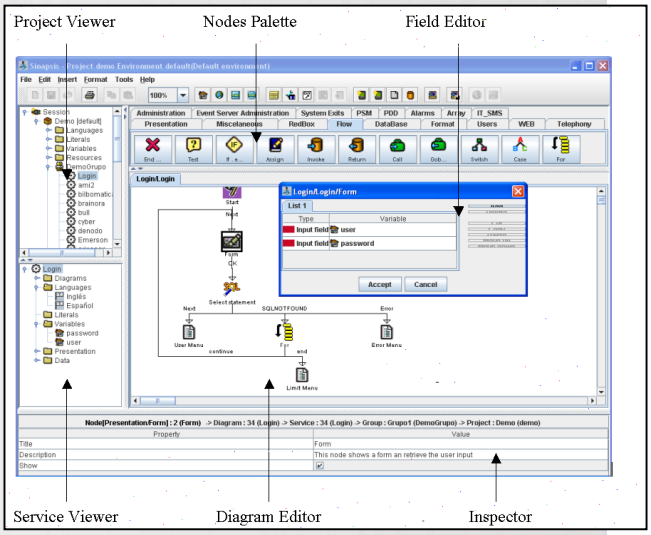

Figure 5: Screenshot of a SML Service Creation tool.

The main reason for using a graphical user interface instead of a text-based language is that it is more intuitive and much easier to follow the logic of a Diagram flow than in the case of formal languages like C or Java. It also has the advantage of avoiding any syntax error and helps detecting and fixing logic bugs more quickly and efficiently in the development cycle.

The modularity of the development environment facilitates small-team work within a Project, by spreading designers into several Groups working on different Services attached to one Project; thus peer-review and team management is more effectively performed.

As important as the clarity of the logic that a graphical environment provides, is the capability to create in a very short time the skeleton or the prototype of a service, often starting from generic or partial requirements, and show the result to the end user, collect feedback and suggestions before developing the next phase of the application. In this way, iteration with the customer throughout the whole Project duration guarantees that not only deviations regarding client's expectations will be quickly identified; it also provides a simple way to face unplanned changes in the requirements to be incorporated in the application throughout the whole Project lifecycle.

The Nodes do not produce code, instead they have been written once and are reused infinitely. Indeed the code contained in many of the Nodes has been executed in production literally billions of times so it can be guaranteed bugs-free and optimised for performance.

These characteristics make the software created with the tool self-describing (what a Node does and its Links to the logical following one) is immediately understandable, predictable (i.e. deliver what was expected), stable (i.e. bugs-free) and efficient (i.e. using available resources).

Reusability and Combinability are a consequence of the modularity of the tool. For example in the same way that for a Node, a Diagram or a Service can be created once and reused everywhere it is necessary in a Project.

Automatic Reconfiguration, Location Independence, Load Balancing and High Availability are provided by the Runtime; while abstracting completely the developer from any specific knowledge.

5. CONCLUSIONS

The usage of XML schemas to represent software Services' components, properties and relations, offers a univocal, self-describing and self-sufficient description of a whole Project with the following benefits:

- Being completely independent of the hardware and software environment in which the application will run, thus fully independent of a particular implementation of the model.

- Easing the integration with Service Creation tools that make a systematic reuse of software components (i.e. Nodes), and therefore contributing to improve dramatically both developers' productivity and the quality of the applications produced.

Field experience in tens of projects (e.g. used for the development of an e-learning platform for 100,000+ employees of a major Spanish Retailer), running in different environments has demonstrated that this approach was still effective for complex applications involving a number of development and users' groups working in parallel. It also shows that the learning curve for designers was considerably shorter than using traditional methods even for low-experienced programmers.

REFERENCES

Frank DeRemer, Hans H. Kron, 1976: *Programming-in-the-Large Versus Programming-in-the-Small.* IEEE Trans. Software Eng. 2(2): 80-86.

Grady Booch, 1994, *Object Oriented Analysis and Design.* The Benjamin/Cummings Publishing Company, Inc.

M. Cusumano, A. MacCormack, C.F. Kemerer , W. Crandall, 2003, *A Global Survey of Development Practices; Paper 178.* A research and education initiative at the MIT Sloan School of Management.