SHOULDERSCORP
THE STRENGTH OF eBUSINESS SOLUTIONS™

An N-tier Rule Based XML Architecture for a Contract Management System

ABSTRACT
ShouldersCorp has developed an architecture for enterprise business applications. A key component of this architecture is messaging between software agents. This paper will describe an implementation of this architecture for a contract management system. This system integrates varied data types and data sources (such as ERP systems, inventory systems, and data warehouses) using multiple natural languages, hardware systems, and business policies. Additionally, this architecture allows efficient customization for specific customer requirements via a table driven rule system. This rule system allows for rapid changes to a single rule base used by all relevant applications. As enterprises grow, they encounter the need to integrate diverse data sources into an enterprise system without the requirement for every entity to use exactly the same enterprise applications. Functional pieces of the ShouldersCorp architecture are factored into appropriate software agents which can be attached to new data sources and which can communicate via XML messaging.

J. W. Wooten
Chief Technology Officer
Shoulders Corporation
555 Heritage Rd.
Suite 100
Southbury, CT 06783
Phone 1-350-860-4331
Email jwooten@shoulderscorp.com

KEYWORDS
XML messaging; XSLT; Autonomous Agents; N-Tier Architecture; Rule based

I. INTRODUCTION

XML is a powerful new standard for communicating between disparate application elements. While XML is an extensible, self-describing language for recording the meaning of a document, it can also be used to support messaging between software agents. This paper describes the application of ShouldersCorp's architecture to a contract management system. It provides a solution for some important structural challenges related to contract management in a way that is easily customizable to meet specific needs of a client.

The structure of the ShouldersCorp architecture and the function of its components are discussed, as is the design of its messaging semantics. While not specifically designed for contract management, the architecture is directly applicable to systems where access to diverse information stores, complicated business rules, tightly regulated workflow, and role-based security is important.

Many large corporate enterprises have grown through acquisition. This requires the integration of the new IT system into the existing enterprise system. Large enterprises rarely agree upon a single standard for an ERP system, data warehousing system, purchasing system, etc. Specifically, contract agreements for specific discounts on equipment and supplies continue to reside on both the purchased company's systems, as well as the enterprise systems.

Users in each business unit are accustomed to a specific interface and specific functionality. It is insufficient to provide multiple logins in order to separately verify whether special offers may, in fact, be available under some other portion of the organization.

What is needed is an "enterprise listener", that is, a software agent that listens to all transactions from a part of the enterprise and identifies those that it is interested in and should be reported. This agent, working quietly in the background, compares transactions against a set of rules and applies specific actions determined by the rules agreed upon by the enterprise.

A "contract management listener" would listen for changes, additions, and deletions to specific contract management database tables it is monitoring. Those events are then reported, using a publish/subscribe mechanism to the listener's "controller" -- another agent that listens for messages from their operatives. Depending upon the messages and previous information it has received from other operatives, the controlling agent takes their own actions depending upon another set of pre-defined rules.

For instance, one corporate entity has a contract agreement to purchase triple pronged double slotted widgets at a volume discount of 50% from Acme Suppliers under agreement number 700856. Another corporate entity, oblivious to this agreement, prepares a purchase order to Acme or some other supplier for a large quantity of tiple pronged double slotted widgets, which they discover can be obtained with a discount of 20%. Note that the original preparation of the contract agreement meant that the controlling agent was notified of the agreement and remembered it. The second corporate entity's issuance of the purchase order, meant that an event was triggered on the second entity's system that resulted it the listener on that system sending a message that was received by the controlling agent. There the data is evaluated and the action results in the second entity's purchasing agent receiving an email that the material can be obtained, for a discount of 50%, using agreement number 700856.

One of the nice features of XML-based messaging systems is that they can be implemented without knowing the totality of the system. That means that one can examine a particular piece of the corporate system, decide at that time what pieces of information might be of interest to other parts of the enterprise, define the XML layout, and begin publishing those events or subscribing to particular other events that would be of interest to that system. Each new system can then be augmented in exactly the same fashion. Development of the agents for those systems can proceed in parallel. The location of data and its machine dependent nature are ignored in the messaging architecture.

Later, when consolidations are done, data is migrated, or systems are changed, the agent is simply moved to the new system, its' interface to the new system implemented and it continues to work. Other systems are not aware of the change.

The contract management architecture presented here also allows rapid customization for individual corporate entities. In the following sections, we describe: 1) the architecture, 2) system design principles, 3) a use scenario with the associated XML message semantics, and 4), and how specific design issues are addressed by this architecture.

II. SYSTEM ARCHITECTURE

Figure 1 shows the ShouldersCorp system architecture. Each box represents an autonomous software agent that interacts with other agents via XML messaging using standard network protocols.

Client interface agents are shown on the left side of the figure. Most clients would interact with a browser interface, connected to a typical ERP system. They may use the ERP system's proprietary interface, but that doesn't matter as the actual messaging system deals primarily with detecting changes in the affected ERP systems' database tables.

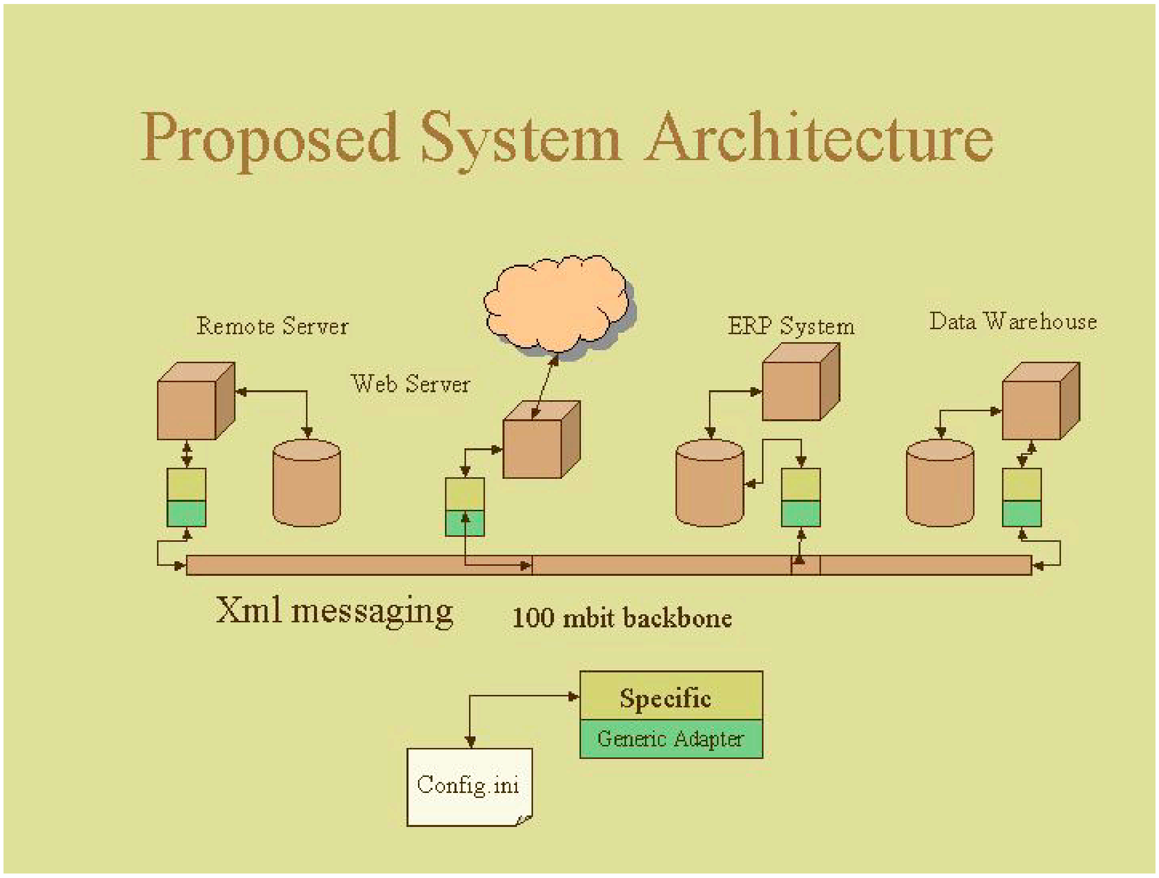

Figure 1. System architecture for the Contract Management System. Two-colored boxes indicate software agents, each comprised of a standard adapter and a specific adapter tailored for a particular database type or proprietary system interface

Database agents, shown as adapters in Figure 1, handle the interfaces to various data stores. All commercial databases support "triggers" that activate associated procedures when specified actions occur. The database agents or adapters are configured, by means of a set of rules, so that the specific trigger action is coupled with the software procedure to use and the information that is to be obtained from the database, when that trigger occurs. In addition, a specific message topic, is passed to the rule set. This topic will be use when event information is "published". Additional rules may define other messages, subscription topics, and the database "stored procedure" to use.

Note that the database agents or adapters consist of database specific and generic or common elements. The database specific part would be an implementation for a vendor specific database, ERP system, or another architecture. The generic part of the agent reads the configuration rules, configures the specific adapter, and subscribers to and listens for events that are covered by its' rule sets.

Upon receiving a message for which it was configured, the agent contacts the proper listener in the specific adapter to process the XML message, it converts the message elements into the specific format required by the associated database adapter, and then hands the requested service to the database. The same, in reverse, happens when a trigger occurs. Data is handed to the generic adapter, where it is packaged according to the rules and published appropriately.

Policy engines can be located, as appropriate, throughout the system or can be handled as a central facility. This is a performance and policy decision that each enterprise can make individually. Nothing in the architecture favors either solution. For instance, all purchasing contract decisions could be made in the Purchasing Policy Engine located in central purchasing and subscribing to messages for Central Purchasing. Inventory decisions could be made in the Inventory Policy Engine located in Central Warehousing or they could be centralized in one Corporate Policy Engine.

The design of the Policy Engine in all of these cases is the same. They also consist of a messaging service adapter and a set of rules that determine what action to carry out if a relevant message is received. These message types and actions are all configured via the rule set for that specific engine.

In Figure 2, we show the underlying components that are used to build the adapters and message structures. These exist as J2EE servlets and EJB's that are enormously scalable and flexible and that have been tested and used in numerous implementations.

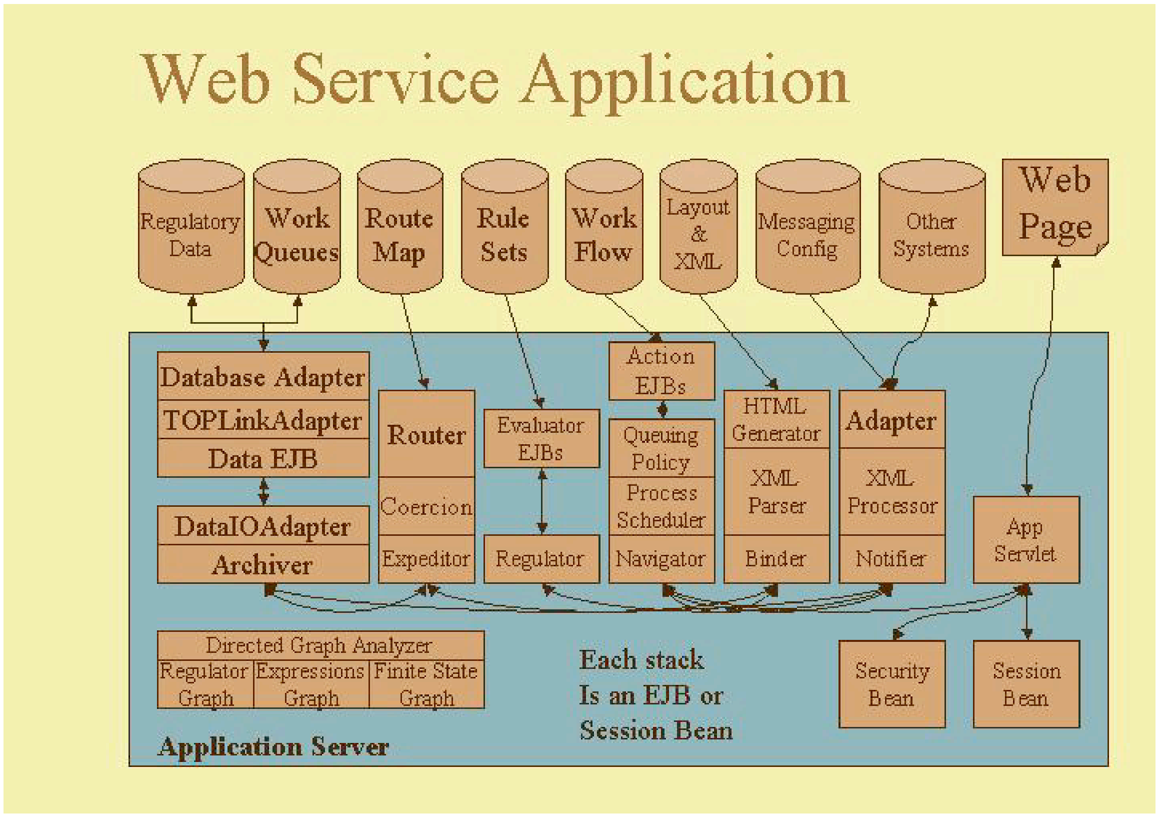

Figure 2. Architectural components used to implement the Policy Engines and the generic adapters for the system shown in Figure 1.

III. SYSTEM DESIGN PRINCIPLES

Autonomous Agents
The system architecture is based on a set of autonomous agents that intercommunicate using XML messaging. The agent approach allows systems to encapsulate the processing as well as provide a simplified interface. An object component model permits encapsulation of information within objects. However, it does not encapsulate the processing of the information, since that *may* involve interactions with distributed and distinct systems. Each agent is charged with providing a particular set of functionality. The agents are autonomous in that information and how it is processed is encapsulated in the rule sets for the agents and their interactions via the publish/subscribe topics. This approach provides a clean separation of responsibilities and partitioning of the system.

Separation of functionality
Each agent does one thing well. The system is partitioned into functional units that are assigned to agents. For example the agent assigned to provide information when the ORACLE database table CONTRACT_TABLE is changed is configured for that explicit table. The specific ORACLE adapter will understand how to communicate with ORACLE (the presumed database for this example) and will activate the correct method (defined by the configuration) when this table changes. While the generic aspect of communicating with ORACLE is the same for all adapters that communicate with ORACLE, the specific information, obtained from the generic adapter via a set of "rules" is specific to the CONTRACT_TABLE. If other tables need to be monitored in this same database for other purposes, it is a good practice to provide either a separate adapter or a separate set of adaptor rules with clear delineation of functionality so that later changes in database structure will not impact the agent.

All decisions are made in the policy engines.
All decisions are policy based. All decisions that can be made are made in policy engines. The adapters or agents gather the information

determined by their rule sets. They then forward that information to the appropriate policy engine. In the policy engine, the specific actions to take as a result of the event are determined through another set of rules. The important distinction is that one set of rules determines *what* is collected and the policy engine to notify, while the policy engine uses a different set of rules to determine the *action* to be taken. These actions can be complex and may involve the role of the individual that initiated the event or have other parameters determined by the enterprise.

Publish/Subscribe topology
This principle is a corollary to the previous one. In order for decisions to be made by the appropriate policy engine, the message must be sent so that the appropriate policy engine can retrieve it. By using a publish/subscribe messaging system, the need for a location, name, or hardware address of the policy engine is avoided. During system implementation, the desired policy engine names are determined. Distributed agents are then configured so that, according to the nature of the event, the appropriate policy agent becomes the subject of the published XML information on the event. This design also reduces the possibilities of congestion, since parallel policy engines can be configured and serviced by a policy engine whose only function is to distribute work to the parallel engines.

Internet protocol communication
All agents communicate with each other using IP (internet protocol) communications, including agents are on the same computer. This provides the flexibility to later separate the agents or reconfigure them with little impact on the system.

XML messaging
XML where possible; XSLT where not. All messages passed between agents and policy engines are in XML format. The Policy Engine only "speaks" XML. Therefore, in addition to responding to messages from the Policy Engines, the interface agents may be required to convert the XML messages into other formats such as HTML pages, e-mail messages, and SQL statements. XSLT transformations are used to convert between XML and other formats.

The situation is more involved going from other formats to XML, as there may be no standardized way of converting these formats into XML. This transformation can be handled by using the rule-set provided on a targeted agent to convert the statement into XML. XSLT would then convert the XML into the specific format used for messaging within the system.

All messages are via a publish/subscribe mechanism with guaranteed delivery. The configuration of the system establishes escalation and failure behavior if messages cannot be delivered in the maximum time specified.

Rule based configuration
Use a rule- based system to maximize reusability. Rather than use a specific adapter for each agent, a rule-based system is used by the agents. Here, a specific adapter is required only for different database types or proprietary interfaces such as an ERP system. The XML information provided must be converted using XSLT into the format required by that vendor. The rule-based system allows a generic adapter (the same for all adapters) to read in a set of configuration rules and set up the necessary triggers in the vendors' system. The methods referred to in the rules are those methods located in the actual specific adapter. The output, of the methods, is targeted to a few methods in the generic adapter that are used to publish the information to the messaging backbone.

IV. SCENARIO AND XML MESSAGE SEMANTICS

The XML tag hierarchy is a powerful way of encapsulating the information model of inner-agent messages in this architecture. At the highest level, messages are sent between agents, and may contain several request or response blocks. These blocks refer to security, user information, event type, and message data. A request or response contains several processing instructions and the context data to support these instructions.

The following use case shows a moderately complex workflow and its associated messages. The policy decisions shown here are representative, but could be arbitrarily modified for specific processing on a case-by-case basis.

USE CASE:

1. Initially the software agent responsible for monitoring purchase orders and perhaps a few other tables is configured using:

```
<config>
        <adapter name=”oracle_purchasing_personnel_adapter”/>
        <event event_type=”INSERT” type=”TABLE” value=”PURCHASE_ORDERS”
                method=”change”, xlst=”oracle_insert_setup” />
        <event event_type=”DELETE” type=”TABLE” value=”ANOTHER_TABLE”
                method=”change”, xlst=”oracle_insert_setup” />
                …
        <policy_engines>
                <policy_engine name=”PurchasePolicy” id=”PURCHASING”/>
                <policy_engine name=”TerminationPolicy” id=”PERSONNEL”/>
        </policy_engines>
</config>
<events>
        <event method=”change” xlst=”oracle_event” >
                <expression table=”PURCHASE_ORDERS” subject=”PURCHASING”
                        action=”publish” />
                <expression table=”EMPLOYEES” subject=”PERSONNEL”
                        action=”publish” />
                <expression subject=”Configure” action=”subscribe” />
        </event>
</events>
```

2. This results in the software agent subscribing to the message type with a subject of Configure and being set up to respond to changes in particular tables via trigger events.
3. The Purchasing Officer later issues a Purchase Order via existing system.
4. The existing system prepares a row of information containing price/unit, description, item, etc.
5. The row is inserted into the database table PURCHASE_ORDERS in the purchasing system.
6. A Trigger Event occurs that calls the method *handleEvent(event)* in the specified adapter.
7. The method *handleEvent(event)* receives the event and requests the table row via standard SQL.
8. The information is retrieved and using XSLT is transformed into an XML data object:

```
<event id=”id#”>
        <event_info type=”TABLE” value=”PURCHASE_ORDERS”
        event=”INSERT”
        row =”3045” />
<context_data >
<item_name>widget</item_name>
<item desc>double pronged triple slotted< /item_desc>
…
</context_data>
</event_id>
```

9. This information is handed to the *change()* method of the generic adapter as specified above.
10. There the expressions are evaluated using the information provided in the XML tagged data.
11. Since there is a match between the table name of an expression and the event data, that expression is consulted to obtain the topic to publish (i.e. the policy engine which will be listening).
12. The message is constructed.

```
<message>
<subject name=”Purchasing Policy Engine” time=”xxx” from=”this agent” />
```

```
<data>
        the above XML data object
</data>
</message>
```

13. The message is published with the topic “PurchasePolicy” as described by the configuration data.
14. The policy engine, which listens for messages with the topic “PurchasePolicy”, retrieves the message.
15. The policy engine then strips out the XML element, parses it and determines that a rule applies to the specified event
16. A request might then be made to the agent managing the CONTRACTS database, a record that corresponds to one or more keys contained in the XML message.
17. The agent returns an XML package that specifies the contract number and discount that applies to contracts for this kind of item.
18. The policy engine then queries its’ rule set to determine the action to take when it finds a match.
19. The rule set may carry out a notification by email, by phone (using voice handling software), or other through another mechanism as determined by the rules. The important thing is that each action is then carried out by other agents specifically designed for the task. Thus, many different policy engines might choose to use voice handling notification, yet only one engine is required to support this and the actual code need not be replicated throughout various engines or systems.

V. DESIGN ISSUES

This section analyses design issues for the Shoulders contract management system. The advantages (and disadvantages, where applicable) of a modular component architecture using rule based XML messaging are discussed.

Transaction Volume: The minimum installation requires at least one system where contracts are maintained and another where purchase orders are prepared. Depending upon the size of the organization, and the number of contracts and purchase orders, the work load could be handled by a single NT machine with an appropriate amount of memory and disk space. At the other end of the scale, a large enterprise might require several servers with separate machines dedicated to collections of agents for several databases.

A modular messaging architecture is critical to achieving this scalability. Since software agents communicate using network protocols, they will work independent of their location. Agents or Policy Engines with high loads can be replicated and handled via a load distribution agent.

Processing Speed: Since this operates as a “background” task, the performance requirement is only that the task be processed before the purchase order is approved or paid. The user does not see any of the activity, as it is “behind the scenes”. If a reasonable fraction (often as little as a few percent) of current purchases that do not use the contracting mechanism can be converted to the contractual purchases, the annual savings completely exceed the initial implementation cost. Subsequent acquisitions or extensions are completed at a fraction of the original implementation cost.

Communications: There are different levels of communications infrastructure available. Typically a given corporate entity has a relatively high bandwidth corporate backbone with a lower bandwidth inter-corporate connectivity. As long as the communication is IP-based or IP can be layered into it, then new corporate entities are “connected” to the policy engines as soon as agents are deployed for their databases or proprietary applications. Using a publish/subscribe mechanism also provides for the establishment of routing nodes so that only one copy of a given message is deployed outside the intra-net and expanded into multiple messages inside the receiving intra-net. All of these messages can be securely encrypted and delivered through corporate firewalls.

Database Interface: The entire design of the system, which of course, can be applied to areas other than contract management (for instance data warehousing) is predicated on the concept that data can be obtained from one database, and exchanged with another, independent of the actual database design or vendor implementation. Standard SQL queries and triggers, supported by all major vendors, are used. Where not available (such as a mainframe ISAM file), separate triggers, based upon the file activity, can be implemented and configured.

Seurity: Clearly security is a vital consideration in a system where sensitive corporate data is exchanged via agents. XML publish/subscribe and messaging systems provide a complete encryption mechanism that exceeds those used by most securities firms. Private encryption mechanisms can also be used, if desired.

Denial of service attacks: These are not specific to this system, but would fall under the general denial of service attack from an outside agency. The standard monitoring, isolation, and tracing functions available in most large IT shops would be used to track and remedy such attacks. Inside the intra-net, messages would continue to be queued for later transmission. Some cost savings opportunities may be lost, but information for future purchases is made available when the attack is remedied.

Security from insiders: While there is a potential for misuse by insiders, the agents can be configured to report numbers of transactions, amount saved, vendors listed, contracts used, etc. Any data that is available to the system via the agents can be monitored, reported on periodically, and compared with external cross-check data. Changing a rule, or other information used in configuring either the agents or policy engines, can itself result in an event that is published to a logging facility.

Cost: The cost obviously needs to be minimized. This architecture helps to minimize cost by:
Minimizing and structuring the customization work. The system clearly separates the base infrastructure from the custom aspects. The system makes extensive use of XSLT for transformation of internal XML messages for database and agent interactions; this means that most customization is obtained via changes in XSLT stylesheets, and from the system and agent behavior which is maintained in XML rule sets.

Using existing technology standards. XML and internet technologies are well developed, extensively documented, and benefit from commodity pricing (as well as many free software components). This is of considerable advantage when compared to the typical systems that are based upon proprietary standards. Additionally, the use of XML lends itself to providing many of the agent services as Web Services and WAP services available via wireless devices.

Customizability: Customization takes into account that policy rules may change, as well as the action that should be carried out when a policy decision is invoked. By using a rule-based approach, this customization is achieved without major software changes. A new database, proprietary connection, or non-SQL or event based connection, will require some development, but this is minimized by the extensive use of patterns in the design.

VI. DISCUSSION

This paper presents the ShouldersCorp contract management system architecture. Autonomous software agents intercommunicating using XML messaging in order to handle complex workflow involving interactions of multiple databases and users. XML technology provides key advantages in this architecture:

1) A flexible message infrastructure between components,
2) A database neutral data format, and
3) A flexible transformation and formatting engine, using XSLT.

In addition, the use of a rule based configuration system with a generic and a specific adapter design, means that fewer components have to be developed and tested. Using proven software patterns, these components are built for maximum reuse and extensibility. New adapters and policy engines can be added without software changes through XML based configuration rules.

Designing a flexible, sophisticated contract management infrastructure is a challenge. It must be built so that new acquisitions are easy to add, integrate, and have no impact on current applications. As economies of scale are recognized in the organization (often by analyzing the information obtained from the agents and policy agents), the system may be changed to improve efficiency without affecting distant parts of the enterprise that interacts with the system, are not impacted by the changes.

The architecture described herein, while described for a contract management system, is designed to become the backbone architecture for the modern enterprise. It allows integration to proceed incrementally and without the continuous exhaustive analysis that has been required in the past only to become outdated as soon as it is completed.

BIBLIOGRAPHY

[1] Alex Ceponkus, Faraz Hoodbhoy, Applied XML, Wiley, New York, NY, 1999.
[2] Eric Greenberg, Network Application Frameworks: Design and Architecture Addison Wesley, Readying MA, 1999.
[3] Nicholas R. Jennings, Michael J. Wooldridge Agent Technology: Foundations, Applications, and Markets Springer-Verlag, 1997.
[4] Oracle Using XML in Oracle database applications http://technet.oracle.com/tech/xml/info/htdocs/otnwp/about_xml.html Nov. 1999.

BIOGRAPHY

J. W. Wooten is the Chief Technology Officer at Shoulders Corporation where he is responsible for architecture, research and development in technologies for web services, and solutions for really hard problems. Previously he held a position as Strategic Planner for the office of Science and Technology at the Oak Ridge National Laboratory. He has a Ph.D. from the University of Nebraska in Theoretical Physics and also did a stint as a high school teacher of physics and advanced mathematics. His interests involve flying (he is a private pilot), sailing, and the study of theoretical gravitational physics.