SOA in the Real World – Experiences

Manoj Acharya, Abhijit Kulkarni, Rajesh Kuppili, Rohit Mani,
Nitin More, Srinivas Narayanan, Parthiv Patel,
Kenneth W. Schuelke, and Subbu N. Subramanian

Tavant Technologies, 3101 Jay Street, Suite 101, Santa Clara, CA 95054

Abstract. We discuss our experiences in building a real-world, mission-critical enterprise business application on a service-oriented architecture for a leading consumer lending company. The application is composed of a set of services (such as *Credit Report Service, Document Management Service, External Vendor Service, Customer Management Service*, and *Lending Lifecycle Service*) that communicate among themselves mainly through asynchronous messages and some synchronous messages with XML payloads. We motivate the choice of SOA by discussing its tangible benefits in the context of our application. We discuss our experiences at every stage of the software development life cycle that can be uniquely attributed to the service oriented architecture, list several challenges, and provide an insight into how we addressed them in real-life. Some of the hard design and development challenges we faced were related to modeling workflow interactions between services, managing change analysis, and contract specification. In addition, SOA architecture and asynchronous messaging introduces fresh challenges in the area of integration testing (e.g. how do we test a system whose interface points are asynchronous messages) and in testing the robustness of the system (e.g. how do we deal with out of order messages, duplicate messages, message loss?). To address these challenges, we built a tool called *SOA Workbench*. We also discuss the techniques we adopted to address scenario-based validation that go beyond traditional document-centric validation based on XML Schema. Monitoring and error recovery, two key aspects of any mission-critical system, pose special challenges in a distributed SOA-based, asynchronous messaging setting. To address these, we built a tool called *SIMON*. We discuss how SIMON helps error detection and recovery in a production environment. We conclude by listing several opportunities for further work for people in both academia and industry.

1 Introduction

Many mission critical enterprise applications share some common characteristics – they comprise of a variety of functionalities, feature complex interactions among them, should be easy to manage, need to be fault tolerant, and should be isolated in failure. In addition, *constant evolution* required to keep pace with the ever changing business requirements and *distributed ownership of the functionalities* spread across several teams are two other crucial characteristics of such systems. The challenge of building and maintaining such systems is not very different from the challenge of building a system comprising of complex subsystems (for example a car or a computer) that are products by themselves, have their own product life cycle, and have clearly defined services that are exposed via agreed upon contracts. In this paper, we discuss our experiences in building such an enterprise application for a large consumer lending corporation.

1.1 The Consumer Lending Application

In this section, we briefly introduce the consumer lending application. The application handles the entire lending life cycle that begins with the procurement and management of millions of potential prospective customers (called *leads*). The application has the ability to scrub large amounts of lead data, classify the leads according to various categories, and distribute the leads based on various criteria to the company's sales force. The sales functionalities include ability to make calls to potential customers and keep track of the progress of the conversation and follow-ups via reminders, real-time management visibility to sales force performance, ability to quickly assimilate data critical to the loan offering (such as income, property details, appraisal etc) *real-time* while the sales person is on the phone with the customer, and the ability to order and instantly receive the customer's credit report. The sales functionalities also include the ability to capture the desires of the customer and perform *what-if* scenario analysis to offer the loan product that optimally matches the customer's desire. On successful completion of the sales activities, the system has a set of fulfillment capabilities, also called loan processing capabilities, that involves validating the data obtained from the customer during the sales cycle (such as income verification, appraisal verification, title verification etc). These verifications during the loan processing stage are performed either via supporting paper documentations obtained from the customer such as W2's and income statements or via automated verifications performed through specialized electronic services (such as credit report services or appraisal services) provided by external vendors. The loan processing stage also involves dealing with exceptions that may arise during the verification phase and performing an analysis of their impact on the loan product. Other crucial functionalities in the consumer lending application include (1) a pricing module that given a set of inputs such as the borrower's credit score, income, and property value generates a loan product with the rate, points, and fees information (2) a compliance module that ensures that the loan product does not violate any of the state, federal, and corporation-specific compliance laws (3) a document service that manages storage and retrieval of electronic documents, and (4) a task management service that keeps track of the list of activities (and their statuses) that need to be performed to take the loan application from one stage to the next along its life cycle. Finally, the system has the ability to take a validated and approved loan through a funding process that involves electronic transfer of funds between financial institutions.

1.2 Motivation for SOA

As can be observed, the consumer lending application consists of a set of distinct, related set of functionalities. Not surprisingly, the consumer lending corporation has departments that specialize in these functions. For example, there is a marketing department that owns the lead acquisition and related functions, a sales department that owns the sales functionalities, and a loan processing department that owns the fulfillment functions. Besides taking ownership, these departments also want the ability to evolve their functions and related IT capabilities independent of the others, manage the applications and data, and not be affected by glitches in the other systems. Naturally, the scalability and service level agreement needs of the functions are also different. For example, the marketing functionality is used by a handful of users in the corporate office where as the sales functionality supports thousands of field agents with an expectation of sub-second response time. In addition, there is a need for functionalities to be reused across multiple applications. For example, the document service related functionalities are required by several sales, marketing, and fulfillment applications.

The above set of requirements lend themselves to a natural organization of the software artifacts that comprise of this application as a set of independently deployed components that expose a set of services that can be invoked via predefined messaging protocol – in other words an architecture based on SOA.

Note that the core functional requirements of our lending application can be realized in a traditional, monolithic, non-soa architecture. Indeed, prior to our system, there existed a basic version of the application built on a client-server platform. However, such a tightly-coupled system would not support several critical features such as independent evolution and scaleability of components, isolated deployment and manageability, efficient reusability of common features, and isolated failure.

2 Application Architecture

Figure 1 captures the application architecture of our lending application. Each component (e.g. appraisal service, credit service) is an independently deployable,

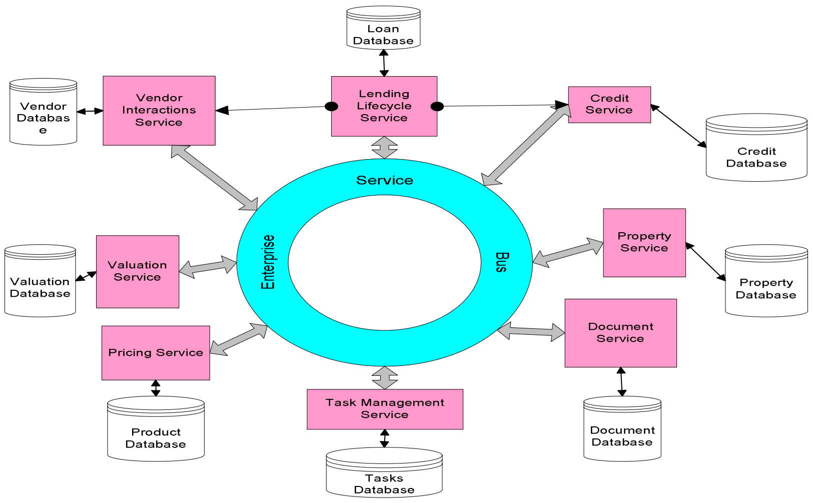


Fig. 1. Application Architecture

maintainable “product” and exposes a set of services related to their specialization that can be invoked by other components that require them in the context of some business workflow. The services can be accessed asynchronously via message interchange on an *Enterprise Service Bus* (ESB) [1] or synchronously via webservice calls.

As illustrated in the figure, the ESB forms the hub of the messaging infrastructure. At a basic level, the ESB provides a reliable messaging infrastructure (we use a commercial ESB product from a third party vendor) that is based on JMS [4]. In addition, it acts as a message router that delivers messages to the appropriate services based on some well-defined routing rules. In order to realize a business transaction, the services communicate among one another via messages that are exchanged on the ESB. In order to standardize the message format and facilitate understanding across teams, we adopted the definition of messages in the form of Business Object Documents (BODs) as defined by The Open Applications Group Integration Specification (OAGIS) [6]. BOD messages are named using a pair consisting of a standardized verb (such as *Get, Show, Process*) and a business relevant noun (such as *loan, credit*).

2.1 Usecase Illustration – Credit Pull

Figure 2 illustrates the realization of a sample business process flow in our architecture. One common usecase in the context of the lending application is the “Credit Pull” workflow – i.e. the functionality that allows a loan sales person to obtain the electronic credit report of a customer in *real-time*. The lending life cycle service initiates the credit pull as a response to a user request on the UI by sending a credit request message to the credit service. The credit service listens to this message, registers its activity with the task management service, and passes the request to the external vendor service which in turn places a request with the credit vendor. The external vendor service obtains the credit report from the right vendor using vendor selection rules (based on established business agreements, service level agreements etc). Once

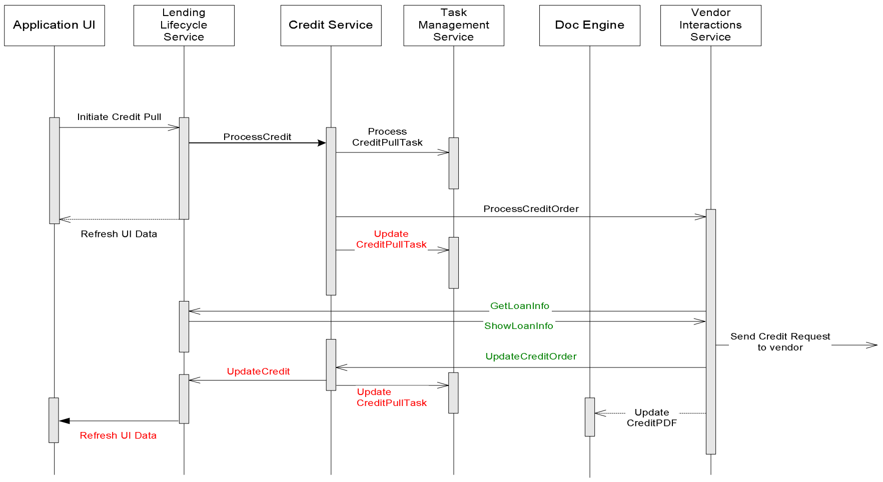


Fig. 2. Credit pull sequence

the vendor responds with the credit report, it is imaged and stored in the document service. The credit report is also sent to the credit service which stores it locally and returns it to the lending life cycle service which performs some local processing and renders the credit report on the UI.

All the service interactions are achieved via asynchronous message interchange on the ESB. Also, the message requests and their names adhere to the OAGIS BOD standards. For example, the lending life cycle service initiates the credit pull request by dropping a *ProcessCredit* message on the ESB.

3 Design Time Challenges

In this section, we motivate the need for new design time tools for SOA applications and describe the *SOA Workbench* tool that addresses these issues. The central elements of the Consumer Lending application described earlier is the notion of “workflows” or “sequences” that is a construction of a higher business services by composing various individual services in interesting ways (e.g., the Credit Pull described in Section 2.1). We need a way to describe such sequences along with their meta-data in a structured way including the details about all its steps, the communication mechanism used for a given step (synchronous via web services or other protocols, or asynchronous via messaging), the structure of the information exchanged (e.g., XML schema info), and several other details specific to the integration between these services. In earlier applications, such information was specified just through design documents. The SOA Workbench is a tool that captures the sequence metadata described above at design time. It then uses this information to do other interesting tasks during the design, test, and production monitoring phases. Additional metadata related to data validation can also be added and is described in the following sections.

3.1 Content Validation

Since the interactions in the sequence are between loosely coupled systems that are usually developed by different teams, it is critical to capture as much information as possible on the validity of the documents exchanged between the services. While some structural and semantic constraints can be expressed in the XML schema, there is a need for validation constraints that cannot be expressed in the schema. For example, the same documents (i.e. same schemas) can be used in different sequences (or even in different steps within a sequence) and the validation constraints may different across these sequences (or across the steps within a sequence). A typical case is that some elements in the schema are mandatory in one sequence but not in another. As discussed in Section 2, we have embraced the OAGIS style of defining documents where key business entities are represented as “nouns” and an XML schema can embed one or more of these nouns within it. Often the same noun is embedded in different schemas that represent different uses of it. For example, we have a Credit noun representing credit information that is used in both ProcessCredit as well ProcessCreditOrder steps in the Credit Pull sequence (Credit noun is also used in several other sequences). The ProcessCredit step mandates some elements within the Credit noun to be present whereas the ProcessCreditOrder mandates a different subset of elements within the Credit noun. The SOA workbench supports such validations by allowing the user to specify mandatory data elements for each step of the sequence. When the communication is asynchronous, our application also uses several custom JMS properties [4] to communicate – the ESB uses the JMS properties to route the message. The SOA workbench tool allows the user to list the JMS properties used in each step and specify whether they are mandatory.

3.2 Advanced Content Validation

XML schemas and the additions described in the previous section about specifying required elements that are sequence-step specific still only validate the message from a structural perspective. The SOA workbench goes further in addressing how one can validate the content of an element (i.e., the *value* of an XML element) as well. In our application, as is typical of many enterprise applications, the data elements in the XML messages are related to or derived from data stored in a database. Services usually consume a message, update the database and generate more messages in response. The content of these generated messages are derived from data in a database. In such scenarios, we need the ability to specify validation checks on the content of the XML messages by validating it against its corresponding data in the database. The tool allows us to specify such validations for each step in the sequence. The XML element to be validated is usually specified via an XPath expression[1]. The tool allows the user to write SQL queries against the database and add a validation that checks if the result of the XPath expression is the same as the result of a SQL query. For example, in the Credit Pull scenario, the Lending Lifecycle Service sends a ProcessCredit document to the Credit Service with borrower's name, social security number (SSN) and address. The Credit Service stores the name and SSN data in its database, but does not have a need to persist the address in its database. It then generates a ProcessCreditOrder document with the borrower name, SSN and address (the address element is just transferred from the ProcessCredit document) and sends it to the External Vendor service. We may want to add a validation constraint that the borrower name and SSN in the Credit Service database is the same as the borrower name and SSN in the ProcessCreditOrder document. SOA workbench allows for specifying such validations.

Notice how the address field was just transferred by the Credit Service from the incoming XML document to the outgoing XML document. In our application, such transient flows of information across XML documents in the sequence are fairly common. It would be useful during testing to validate that the address field in the ProcessCreditOrder document is the same as the address field in the ProcessCredit document. SOA workbench allows to specify whether an XPath expression on a document at a certain step in the sequence has the same value as an XPath expression run on a previous step in that sequence.

[1] Ideally, these should be XQueries instead of XPaths as that makes it easier to express more complex validations on the whole document instead of at an element by element basis. This is a simple extension to the current tool and is planned for a future release.

3.3 Reviews, Approval and Impact Analysis

The SOA workbench also allows the interactions to be reviewed and approved by each of the participants in the service. In a loosely coupled system, such review and approval processes are essential to communicate changes and to get all parties to agree to the proposed design.

Another big advantage of laying out the sequences in SOA workbench is its ability to deal with changes. In our applications, we frequently face the need to make changes to the XML schemas for various reasons. Prior to the SOA workbench, it was extremely difficult to manage these changes. A change to a specific element could impact certain sequences and the person making the schema change was not able to easily identify the affected sequences. To address this, the SOA workbench offers a feature by which the person making the schema change can do an impact analysis and identify all the sequences and the specific steps within the sequence where a BOD is used. Furthermore, if a change to a specific element is made, the tool can identify the sequences as well as the exact steps where the changed element is listed as a *mandatory* element. This will help the user to deal with changes in a more controlled manner. In a future version, we plan to add a change request workflow to the tool where a user can propose a change to a schema and all the owners of the services impacted by that change would be required to approve such changes before it is published.

3.4 Comparison with Workflow Tools

It is useful to compare the SOA Workbench to existing Workflow (BPM) tools in the industry. SOA Workbench is similar to BPM tools in that it helps in designing workflows composed of many services and interactions. Workflow systems focus on the ability to change workflows dynamically whereas the SOA workbench primarily deals with the problem of defining sequences across loosely coupled services and managing the *design contracts* (specifications that help answer questions such as what are the required data elements in a XML schema or required JMS properties in each specific interaction, what constitutes a valid document in the context of a specific step in a sequence, how should elements be validated against data in a database etc.), monitoring, and testing of these services. Recently, BPEL[7] has emerged as a potential standard that provides a portable language for coordinating the flow of business process services. BPEL builds on the previous work in the areas of BPM, workflow, and integration technologies. There are a few commercial implementations of BPEL. Weblogic Integration [8] is one such tool that originally focused a lot on integration and workflow capabilities with proprietary ways of defining workflows (called Java Process Definitions) and more recently starting to offer better support for BPEL.

While BPEL and several commercial implementations address the issue of process definition and execution in a distributed SOA environment, they mainly focus on integration and orchestration of services. In particular, they do not address critical aspects associated with design contract definition. The BPEL tools also do not address other design time activities such as reviews and approvals. Furthermore, they also do not deal with the challenges in testing and monitoring as explained in Sections 4 and 5.

Another recent trend is the emergence of tools providing ESB functionality. As explained in Section 2, we use a commercial ESB tool that provides reliable messaging and acts as a message router. Some ESB vendors also provide value added features for service orchestration on top of the basic ESB. Again, in our experience, such features do not focus on specifying design contracts to the level of detail that we have described and also do not sufficiently address the monitoring and testing needs of a SOA application.

We also wish to point out that the work we have done in the areas of design contract specification, testing, and monitoring in the context of SOA are complementary to the current efforts on BPEL and related commercial tools. In fact our work can be easily integrated into the standards or commercial tools as valuable extensions.

4 Challenges in Testing

The main challenges we faced in testing our SOA application were in the areas of checking conformance to contracts specified at design time, automating tests for systems with asynchronous interfaces, testing robustness of applications built based on asynchronous messaging, and testing services in isolation. We now describe the features that we built in the SOA workbench to address each of these challenges.

4.1 Auto-validation During Manually Triggered Tests

In Sections 3.1 and 3.2, we described how a user could add validation criteria to the steps in a sequence at design time. The SOA workbench also provides additional features that enforce these validation rules at *runtime*, which can be leveraged for the testing of the application. The tester would trigger business sequences from the application -- for example, request a Credit Pull for a borrower from the application. This would exercise the entire sequence. All the messages exchanged at each step (including the JMS properties and the payload) are recorded in a Central Logging Database through a tool called SIMON (see Section 5.1). Through the SOA workbench, the tester can then query for the instance of the credit pull sequence that she just triggered (the query can be based on an application specific property such as say the Borrower's Social Security Number) and then "validate" that instance of that sequence. During validation, SOA workbench queries the Central Logging Database for all the messages that are part of that instance of the sequence, and then validates the message at each step against the Content Validation definitions that were specified at design time – i.e., it tests whether the message at each step has all the required elements, and tests the advanced content validations such as checking if the values in the document match the result of the specified queries in the database or if they match the value of an element from a previous step etc. Notice that while this mode of testing automates whether each step in the sequence adhered to its contracts, it still relies on a user to manually start the sequences through the application and to explicitly use the SOA workbench to validate each instance of the sequence. It does not provide a fully automated regression testing mechanism.

4.2 Fully Automated Regression Test Suites

For web-based applications, there are several testing tools that can be used to automate the user interaction to create automated regression tests. Such tools are not common for message-driven applications. To address this, SOA workbench allows a user to create “scenarios” for a sequence each of which represents a test case for that sequence, and then attach sample input messages for the first step in the sequence. An automated test runner just publishes the message to the service that is the message consumer of the first step in the sequence. After the sequence completes, the test runner validates the messages at each step as described in the previous section.

4.3 Proxy ESB Router

The SOA workbench also provides an additional feature by which it can act as a proxy ESB router whereby it routes the messages to the various services instead of letting it happen via the real ESB. This provides the benefit of being able to inspect and validate the messages immediately when the messages go through the proxy ESB as and when the services publish them, instead of waiting for the entire sequence to finish. This feature also eliminates the dependency on other tools (such as SIMON and the Central Logging Database) for SOA workbench to do its testing.

4.4 Robustness Testing – Duplicate, Lost and Out of Order Messages

The proxy ESB feature of the SOA workbench is a key component for executing robustness tests. An application built around messaging has to deal with issues such as lost or timed out messages, duplicate messages, messages arriving out of order or in orders that the application did not normally expect (this can happen because the speed of consumption and processing times of queues can vary dramatically causing events to happen in an order that a programmer didn’t imagine in the “normal” flow). While the messaging infrastructure may provide certain guarantees about their quality of service with respect to duplicate and lost messages, some of these issues have to be dealt with by the application in any case. For example, the messaging infrastructure can go down causing messages to not arrive in time, or an application that we cannot control can send a messages twice, or a message may arrive in an order that does not conform to the programmers normal flow of thought. The SOA workbench allows such cases to be simulated in the testing cycle by injecting such behavior (such as losing a message, sending a message twice, or routing messages in different orders) during the routing of messages. Such tests are crucial in creating a robust application that can deal with such situations when they happen in real environments. The proxy ESB makes these tests easy to create which would otherwise be extremely hard to simulate.

4.5 Testing Services in Isolation

Testing a service or groups of services independent of the rest of them is important because (a) not all services may be available at the same time due to different development lifecycles (b) logistical problems can cause services to be unavailable in some environments and (c) it is easier to test a large system by incrementally assembling subsystems and testing them. To facilitate this, the SOA workbench allows the user to attach sample messages for each scenario to the intermediate and last steps in a sequence. When a service is unavailable, the ESB proxy uses these messages as replacements for the real message that would have been produced by the real service. This allows the sequence to be tested even when some services within it are unavailable. A common example in our application is testing the Credit Pull when the External Vendor Service is unavailable (it is difficult to coordinate availability of the External Vendor Service for testing because of external dependencies).

5 Monitoring and Error Recovery

So far we have described the challenges during development and testing phases of building SOA applications. We now describe the challenges that arise when the application is deployed on a production environment. In particular, we discuss the challenges in the areas of monitoring and recovering from error scenarios in a production environment.

5.1 Monitoring

As described earlier, a sequence representing a business process involves interactions with several different services that are deployed independently. A distributed system such as this makes it hard to monitor the application. For example, if the Lending Lifecycle Service initiated a Credit Pull and has not received the credit report back within an estimated time, the problem could have been in any of the several services involved in the sequence. We built a tool called SIMON that makes it easy to monitor the sequences and report on their activity and performance. The architecture of SIMON is illustrated in the figure below.

Each service registers an event when it sends a request to another service and when it processes a request from another service. These events are recorded in a database (called Central Logging Database) on a central server that we call the Central Logging Server. SIMON allows the user to define SLAs (service-level agreements) for the completion time for the various sequences. It then runs a background task periodically (the frequency of which can be configured) that looks at all the sequences that have started and whether they all have completed within the defined SLA time period. If some sequences have gone past the SLA time and are still incomplete, SIMON can identify them as exceptions and raise alerts (such as sending an email to the production support team). The level of alerts can be also configured based on the percentage

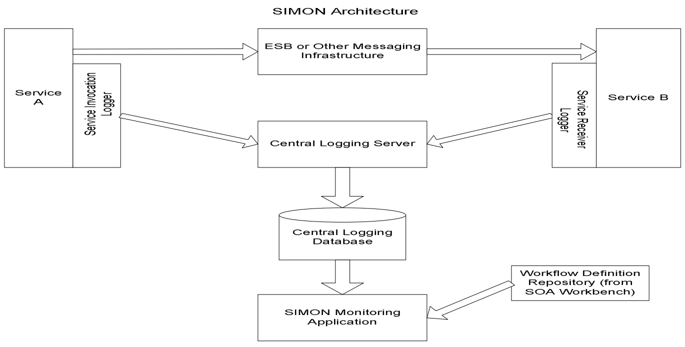

SIMON Architecture
Service A
Service Invocation Logger
ESB or Other Messaging Infrastructure
Service B
Service Receiver Logger
Central Logging Server
Central Logging Database
Workflow Definition Repository (from SOA Workbench)
SIMON Monitoring Application

Fig. 3. SIMON Architecture

of occurrence of failures. For example, if less than 1% of the sequences fail, the alert could be a warning, whereas if more than 20% fail, the alert can be made critical. SIMON can also report on the overall performance of the sequences by providing reports such as the average time it took for sequences to complete and also provide response time breakdowns for each step in the sequence. Such reports are very useful to understand the performance of the overall SOA application and to determine the source of bottlenecks within it.

5.2 Error Recovery

Many errors in a production environment can be attributed to (1) services that are down for unexpected reasons, (2) services that didn't produce the correct message as per the contract, or (3) services that didn't consume the message properly due to defects in the code. If a sequence is stuck in the middle because of such problems, we need a way to recover from them. Temporary problems such as intermittent server crashes are usually fixed by using the redelivery features of messaging providers - a message can be delivered a certain number of times if there are failures in processing them. However, some problems such as defects in the code take longer to fix and the design of most messaging systems don't permit messages to be kept in them for a long time. Also, sometimes, we have to fix the message itself to recover from the problem. To deal with these situations, we built a utility that moves messages that have been tried multiple times from the queues into a database. An application allows users to query these messages in the database and move them back to the queue (which will be done once the defects are fixed and the service is redeployed). Also, if the problem is in the content of the message itself, it allows one to transform the message by applying XSL transformations to it before sending it back to the queue. Having these failed messages stored in a database also gives us the flexibility to query for messages related to a specific customer. For example, if a particular customer's credit pull sequence failed and the business wants the business flow for that customer to be completed first, we can query for all the failed messages for that customer and move them back to the queue so that they get processed first.

6 Learnings and Conclusion

SOA architecture offers promise in its ability to integrate loosely coupled systems. However, the principles underlying the design of applications based on SOA are not well established yet. In this section, we discuss the learnings from our experiences with SOA.

A key issue is defining a service at the right level of granularity. Our original architecture started out with defining services around every domain object (noun). For example, in addition to the services discussed earlier, nouns such as *insurance* and *address* were modeled as services in our original architecture. However, we quickly realized that a system based on such fine grained services will have unwanted development, deployment, and performance overhead. What we have learnt is that a service should represent the right abstraction that both IT and business care about. Importantly, it is something that the company wants to *manage independently*. Other factors that defines a service are does it need to be *released separately* from other components, does it provide *services to many different and varying systems*, should its *down time not affect other* *systems*, does it have *different hardware/scalability requirements*, is there *unique licensing requirements* etc. There is extra cost to managing something as a service and it should be backed by a strong business justification and business ownership.

The area of tools for building SOA applications needs further attention. While the solutions we described in this paper suit most of our needs, there is scope for further improvement. We would like to see more tools that use different approaches to address the challenges in the design, testing and monitoring of SOA applications. We also expect existing workflow and integration tools to start addressing some of these challenges. An interesting area of work is the simplification of the entire development lifecycle by utilizing higher level tools that are fundamentally aware of SOA. As an example, model-driven architectures (MDA [2]) around SOA is an interesting area of study. Another important area of work is the development of systems that intelligently manage schema versioning in SOA.

Adoption of an event driven SOA where back-end services drive events is a difficult challenge for architects and programmers accustomed with the classic UI-driven (e.g. Web) application development. In the web application paradigm, users drive system events by clicking buttons or hyperlinks. The underlying application(s) wait and process individual requests as they arise. Errors are typically handled by raising exceptions that are relayed back to the users, and relying on users to resubmit requests after correcting data and other input problems. In the event driven SOA paradigm, back-end services essentially replace human users. Thus, the back-end systems must be programmed to handle unreliable services, ensure data integrity up-front, resubmit requests, manage transactions etc. This problem is compounded by the fact that services inherently do not have knowledge of the business transaction in which they are a participant. The tools discussed in this paper describe some of means used to mitigate these issues. True defense in depth for the enterprise may require additional audits and batch processing "underneath" the event driven SOA.

Finally, application developers expend valuable time dealing with and designing for failure modes consciously during development (see Section 4.4). Besides the large amount of effort involved, it is difficult to ensure that the developers have thought through the failure scenarios for every use case in the application. It is an imperative to have better programming models and/or more integrated tools that can address these issues in an easier way that removes the burden from the application developers.

Acknowledgements: We would like to acknowledge Fabio Casati for his review of an earlier draft of the paper that has helped improve its quality. We also would like to thank the many members of the Tavant team who helped realize the concepts and features discussed in this paper.

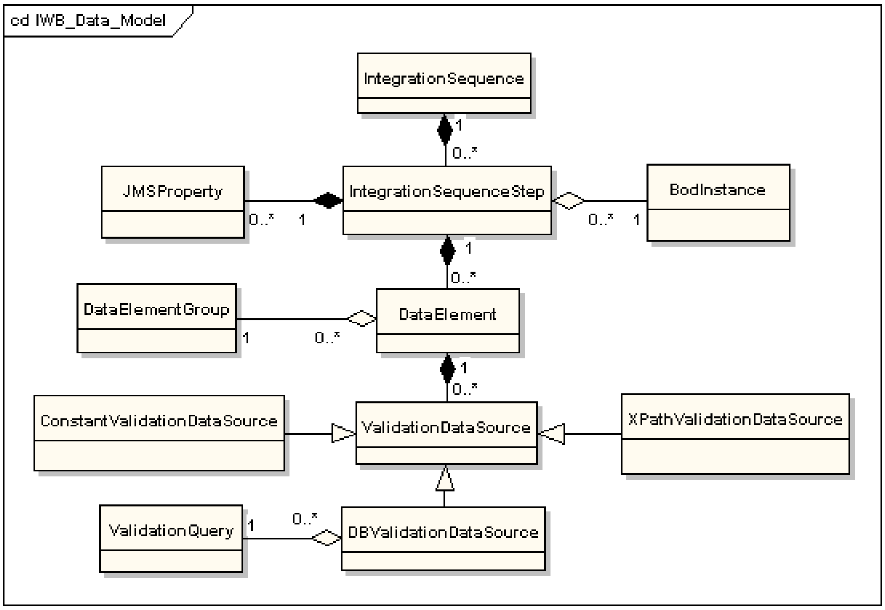
References

[1] Enterprise Service Bus. David Chappell. O'Reilly 2004.
[2] MDA Guide Version 1.0.1 Joquin Miller and Jishnu Mukerji. <http://www.omg.org/docs/omg/03-06-01.pdf>, 2003
[3] Java Business Integration JBI 1.0 http://java.sun.com/integration/1.0/docs/sdk/introduction/introduction.html
[4] Java Messaging Service Specification version 1.1 http://java.sun.com/products/jms/docs.html

[5] XML Schemas http://www.w3.org/XML/Schema
[6] OAGIS Open Applications Group Integration Specification http://www.openapplications.org/downloads/oagis/loadform.htm
[7] BPEL.The BPEL4WS 1.1 Specification. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/
[8] Weblogic Integration http://e-docs.bea.com/wli/docs85/index.html

Appendix -- SOA Workbench Data Model

Figure 4 shows a simplified UML class diagram for the internal object model of the SOA workbench. The class IntegrationSequence represents the notion of sequences (such as the Credit Pull) described earlier. Each IntegrationSequence consists of several steps that is represented by IntegrationSequenceStep. As described in Section 3.1, the data elements used in an IntegrationSequenceStep and whether they are mandatory for that step is represented by the class DataElement. Each DataElement is internally represented as an XPath expression on the XML schema used for that IntegrationSequenceStep. The JMS properties associated with a IntegrationSequenceStep is represented by the class JMSProperty. The ValidationDataSource represents the more complex data validations described in Section 3.2. The class DBValidationDataSource represents the fact that the data element needs to be validated against the result of some query which is represented by the class ValidationQuery. The class ConstantValidationDataSource validates the data element against a constant value. The class XPath-ValidationDataSource validates the data element against the result of another XPath expression on a previous IntegrationSequenceStep within the same IntegrationSequence. In the interest of space, we have omitted illustrating the classes for other parts of the SOA workbench such as those related to Reviews and Approvals etc.


cd IWB_Data_Model
IntegrationSequence
1
0..*
JMSProperty
0..*
1
IntegrationSequenceStep
BodInstance
0..*
1
1
0..*
DataElementGroup
1
0..*
DataElement
1
0..*
ConstantValidationDataSource
ValidationDataSource
XPathValidationDataSource
ValidationQuery
1
0..*
DBValidationDataSource