Implementing Enterprise Integration Patterns Using Open Source Frameworks

Robert Thullner, Alexander Schatten, Josef Schiefer

Vienna University of Technology,
Institute of Software Technology and Interactive Systems
{thullner, schatten, js}@ifs.tuwien.ac.at
http://www.isis.tuwien.ac.at

Abstract. Enterprise Application Integration is needed in many organisations to improve the functionality of their IT systems and offer new business functions to end-users without implementing and deploying new applications. With growing complexity of the IT infrastructure and the pressure to bring new features in ever shorter time, application integration became a challenging task. One approach for integration is to exchange messages between participating applications. Patterns have been described which express best-practices of enabling message-based integration. The patterns are known under the term enterprise integration patterns. As Open Source middleware becomes more and more important as stable and flexible infrastructure components in enterprise IT, this paper discusses the support for architects of enterprise integration patterns by Open Source frameworks, focussing on Apache Camel and Mule.

Key words: Enterprise Application Integration, Enterprise Integration Patterns, Frameworks, Open Source, messaging

1 Introduction

Enterprise Application Integration (EAI) is a continuous challenge for most companies and organisations as soon as a certain size of the organisation is reached [1]. Various computer systems should share data and functionality to support new requirements, ideally in an agile manner, i.e., it should be possible to introduce new functionality fast and without effecting other parts of the system [2]. Hence, in the last decade, a multitude of integration approaches was discussed and a large number of software (middleware) systems were introduced by commercial vendors and developers gained experience with several EAI approaches.

Patterns in Software Engineering are an important means in transporting such experiences. Gamma et.al. introduced the well-known design patterns [3]. However, in enterprise integration efforts patterns on a higher level of granularity are needed and certain efforts to describe such have been undertaken. Martin Fowler for example, describes patterns of enterprise architecture [4]. However,

in the field of enterprise integration the patterns collected and described by Hohpe and Woolf in their book *Enterprise Integration Patterns*[5] were most influential in the community. Hohpe and Woolf's patterns focus on message and event-based integration scenarios, which is the dominating strategy in current integration efforts [6–8].

2 Related Work

As mentioned already, various authors described patterns that can be used in enterprise architectures. Most however, do *not* have the main focus on enterprise integration. It appears to us, that the patterns described by Hohpe and Woolf are the most comprehensive description of patterns to be found on enterprise integration issues. Not by accident, most frameworks (as will be seen later) refer to the categorisation of these authors.

The patterns described by Hohpe and Woolf are organised into six categories: (1) channel patterns dealing with various different types of channels which enable messages to be exchanged between applications, (2) construction patterns which describe how messages can be constructed, (3) routing patterns describing ways how messages can get routed from a source to one or more destination applications, (4) transformation patterns which deal with transformation issues for messages, (5) endpoint patterns showing different ways how applications can be connected to messaging systems and (6) system management patterns dealing with management and governance issues of message-based EAI solutions.

These patterns are of particular interest as most Open Source projects in the domain reference this book and claim that their very framework (Mule, Camel) would implement many of the mentioned patterns or would support the implementation of these patterns.

In the Open Source communities, a couple of frameworks have been released that can be placed in the EAI domain which help build EAI solutions [9]. Thus, there are Open Source projects on the one side and enterprise integration patterns on the other side. What shall be done in this paper, is to show how patterns are supported and can be implemented with the help of Open Source frameworks. The used frameworks are Apache Camel[1] and Mule[2]. Other frameworks which are also worth noticing but will not be covered in this paper in detail are Apache ActiveMQ[3] as well as Apache ServiceMix[4] [10, 11]. These are related to Camel or can be used to enhance it as all three frameworks are mainly supported by the same community in the background. Hence, Camel is fully integrated into both projects, so everything which is implemented with Camel can be reused in either of the frameworks.

[1] http://activemq.apache.org/camel
[2] http://mule.mulesource.org
[3] http://activemq.apache.org
[4] http://servicemix.apache.org

2.1 Apache Camel

Apache Camel is a framework which supports the developer in implementing integration patterns. Patterns must either be configured by writing Java code or by using Spring-based XML configuration files. When using Java for implementing patterns, a domain-specific language (DSL) is available, which eases the implementation and readability of patterns.

On the one side, Camel offers a pattern implementation which can be used easily and, on the other side, it offers a lot of transport types where data can be received from or written to. This means that one can easily read from a file in periodic intervals with Camel or write messages to a file. It is also possible to write to databases or HTTP endpoints. For a full list of available transport components, see the project website. Furthermore, Camel provides a built-in XSLT engine for transforming messages.

In a sense, Camel covers a significant part of what one would expect from an Enterprise Service Bus [12]. In fact, the borders get blurry between different products in the Apache line: ActiveMQ, ServiceMix and Camel. Yet Camel is well prepared to be used as a routing and mediation engine in the Apache ActiveMQ message broker, the Apache ServiceMix JBI-based Enterprise Service Bus, and also works together with Apache CXF[5] (a webservice framework) and Apache Mina[6], which is a rather low level network application framework.

Using integration patterns in Camel is straightforward: All that has to be done is to create a *CamelContext* in a Java class, add all pattern rules to the context and finally start it by calling the *start* method. Alternatively, as mentioned above, rules can be defined in XML. Since version 5.0 of ActiveMQ, Camel is fully integrated into the message broker. This means that all rules can be defined in the configuration files of ActiveMQ; either directly as XML or by providing a reference to Java packages that include all pattern rules. The integration of Camel and the features of ActiveMQ should make a good player in the EAI-framework field. However, when mixing ActiveMQ configuration code and Camel pattern rules, keeping an overview of what parts belong to which framework might get harder than using them separately.

2.2 Mule

Mule calls itself a messaging platform based on ideas from Enterprise Service Bus (ESB) architectures. Mule provides many transport types which can be used for communicating with various applications in an EAI solution. Business logic can be implemented with POJOs which are called Universal Message Objects (UMO) in Mule. For transforming different transport protocols Mule offers a lot of transformation classes which can be applied. In the case that no valid transformer is offered by Mule, custom transformers can be implemented very easily. Configuration of all necessary UMOs and their transport types is done via XML configuration.

[5] http://cxf.apache.org
[6] http://mina.apache.org

Mule offers more functionality than Camel for EAI projects as it is a real server for hosting an EAI solution. Camel just provides a library which has to be used in combination with a message broker like ActiveMQ or an ESB like ServiceMix.

3 Approach

The approach for finding out implementation support of introduced frameworks for enterprise integration patterns is done by analysing the frameworks and implementing scenarios with the help of the frameworks. The scenarios cover nearly all routing and transformation patterns and a reasonable amount of patterns of other categories. *Channel* patterns like *Point-to-Point Channel*, *Publish-Subscribe Channel*, *Guaranteed Delivery* are used nearly in every scenario. *Construction* and *endpoint* patterns already come with the used messaging protocol and do not need much additional support.

For example, JMS already offers support for *Correlation Identifier*, *Return-Address*, *Selective Consumer*, etc. Thus, the focus of the scenarios are *routing* and *transformation* patterns. The scenarios are settled in different application domains. These are airport-, airline-, train- and hospital information systems. All scenarios are imaginary but are partly derived from actual projects that we have done, however with little adjustments made in the mentioned domains. The hospital scenario will be covered in more detail in this paper, details of the other scenarios can be found in [13].

3.1 Communication within Hospital Information Systems

Description This scenario is settled in the hospital domain. When a patient arrives at a hospital, the personal-, health history- and payment data are inserted into the information system. After that, the information has to be sent to an information system used by a doctor and to an accounting system. For the privacy of the patient it is important that each system only gets the information it needs to fulfil its task. Therefore, the *doctor information* system should only get the personal information and the health information of a patient, but not any payment data. On the other side the *accounting* system should not see any health data of the patient but only data which is relevant for accounting. After a doctor has checked the patient, additional information has to be added to the message. This information includes an initial finding and instruction for further therapies. For each therapy, a message has to be sent to the appropriate therapy system. After a therapy is finished, a message including the results is sent to the doctor information system. Another message containing only the patient's personal data and the carried-out therapy is sent to the accounting system so that the patient can get charged. At this point, privacy and security is very important again. The accounting system should never see any health data of a patient and therapy and doctor systems should never get any accounting information of a patient. Another system that is connected is used for generating statistics which

only receives therapy and therapy outcome information for generating reports, but does not receive any personal data of a patient.

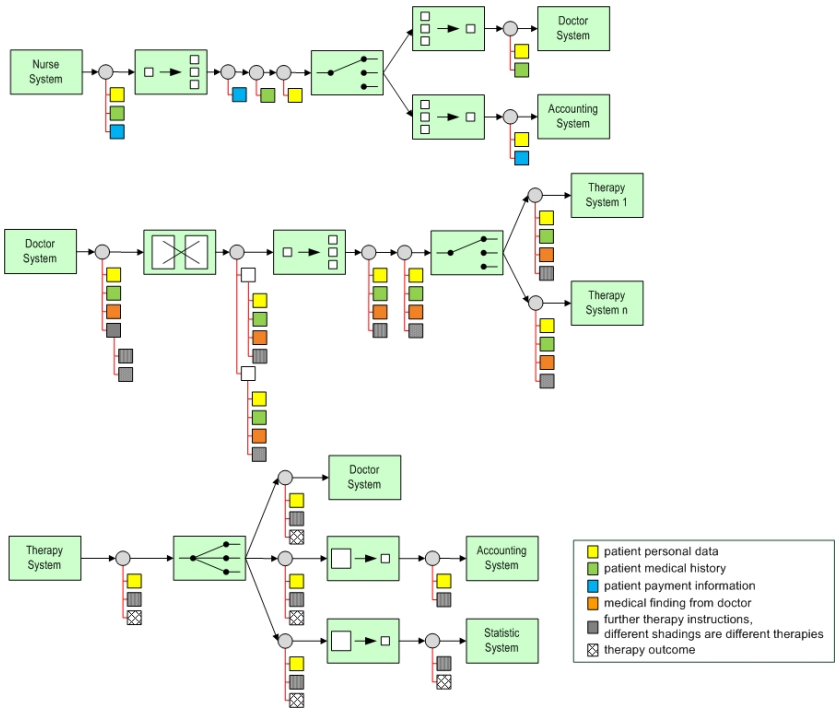

Fig. 1. Hospital scenario: message flow

Modelling The modelling can be seen in Figure 1. The message produced by the *nurse* system consists of three parts. The parts are split and sent to a router which delivers the parts to two different aggregators which aggregate the message again, one time for the doctor system and one time for the accounting system. It can be seen in the figure that the message for the doctor and the accounting system have different content, each containing only relevant data for the specific system. The doctor system adds two parts to the message. One part represents the initial finding for a patient and the other part consists of a list of further therapies. After that, the message is sent to a message translator to bring each further therapy up one level in the XML hierarchy so that they can be split easier. After splitting, a router sends the messages to the appropriate therapy systems. When a therapy is finished, the therapy system will add its outcomes to the message and send it back to the doctor system and the accounting system so that they can charge the patient for the therapy. Before the accounting system receives a message, a content filter has to be used that filters out all health

related information about a patient which is not relevant for the accounting system. The same is done for the statistics system where, only a different filter is applied

Modelling alternatives are rare for this scenario, as security and privacy of data are important for this scenario, a publish-subscribe channel should not be used. What could be used for this scenario is some kind of process manager to support the workflow of all systems. For the first part of the system where the nurse system sends a message to the doctor and accounting system, a splitter and aggregator is used for sending the correct parts of a message to the appropriate system. At the last part of the system, a content-filter is used for that. This is done for showing the usage differently from diverse patterns for implementing the same functionality.

Implementation The implementation of the first part of the scenario with Camel can be seen in Listing 1.1. The implementation with Mule is shown in Listing 1.2. When looking at the Camel code, one will notice that it is very self-explanatory and one can image what it is about very soon without knowing anything about the Camel syntax and DSL.

The first part of the code shows how messages can get split according to an XPath expression, the second part shows routing where XPath is used again for making routing decisions and the third part shows how messages can get aggregated again. Aggregation is done according to the JMSCorrelationId of the messages. The aggregation of the accounting messages is not shown anymore, as it is equivalent to the aggregation for the doctor system.

When looking at the Mule configuration, one will notice that everything has to be defined within a *service*. The first service is responsible for receiving messages from a JMS queue. After receiving a message, it is split at the *outbound* section of the service and sent to the next service. This service sends the split messages along to two further services according to an XPath expression which is used for sending the parts to the appropriate aggregator. The last service shows an aggregator which is used for aggregating messages which will be sent to the doctor system. The Java code for the custom aggregator is not shown here. The custom class has to subclass the *CorrelationAggregator* class which is provided by Mule. The method *aggregateEvents* has to be implemented and provides the business logic for the component. Like in the Camel example, the aggregator for the accounting system is not shown in the listing, as it is equivalent to the one for the doctor system.

Listing 1.1. Splitter, router and aggregator in Camel

```
XPathBuilder splitNurseSystem = new XPathBuilder("/document/*");
from("hospital:camel.hospital.nurse.send")
.splitter(splitNurseSystem)
.to("hospital:camel.hospital.nurse.split");

Predicate patientPredicate = new XPathBuilder("count(/patient)=1
    ");
```

```
Predicate historyPredicate = new XPathBuilder("count(/history)=1
    ");
Predicate paymentPredicate = new XPathBuilder("count(/
    paymentInformation)=1");

from("hospital:camel.hospital.nurse.split")
.choice() //route parts to different endpoints
   .when(patientPredicate)
      .to("hospital:camel.hospital.nurse.aggregator.doctor",
         "hospital:camel.hospital.nurse.aggregator.accounting")
   .when(historyPredicate)
      .to("hospital:camel.hospital.nurse.aggregator.doctor")
   .when(paymentPredicate)
      .to("hospital:camel.hospital.nurse.aggregator.accounting")
   .otherwise()
      .to("hospital:camel.hospital.nurse.aggregator.failure");

//aggregate doctor parts to one message
from("hospital:camel.hospital.nurse.aggregator.doctor")
.aggregator(header("JMSCorrelationID"), new
    MyAggregationStrategy(context))
.to("hospital:camel.hospital.doctor.receiver");
```

Listing 1.2. Splitter, router and aggregator in Mule

```
<model name="muleHospital">
  <service name="nurseReceiver">
    <inbound>
      <inbound-endpoint ref="fromNurse"/>
    </inbound>
    <outbound>
        <!-- split messages according to XPath expression -->
      <mule-xml:message-splitter splitExpression="/document/*">
        <!-- send messages to next service -->
        <vm:outbound-endpoint path="routerAfterSplitting"/>
      </mule-xml:message-splitter>
    </outbound>
  </service>

  <service name="myRouterAfterSplitter">
    <inbound>
      <!-- receive split messages -->
      <vm:inbound-endpoint path="routerAfterSplitting"/>
    </inbound>
    <outbound matchAll="true">
      <!-- forward messages according to an XPath expression -->
      <filtering-router>
        <vm:outbound-endpoint path="toDoctorAggregator" />
        <mule-xml:jxpath-filter pattern="count(/patient)=1"/>
        <transformer ref="addGroupSize"></transformer>
```

```
      </filtering-router>
      <filtering-router>
        <vm:outbound-endpoint path="toDoctorAggregator" />
        <mule-xml:jxpath-filter pattern="count(/history)=1"/>
        <transformer ref="addGroupSize"></transformer>
      </filtering-router>

      <filtering-router>
        <vm:outbound-endpoint path="toAccountingAggregator" />
        <mule-xml:jxpath-filter pattern="count(/patient)=1"/>
        <transformer ref="addGroupSize"></transformer>
      </filtering-router>
      <filtering-router>
        <vm:outbound-endpoint path="toAccountingAggregator" />
        <mule-xml:jxpath-filter pattern="count(/
            paymentInformation)=1"/>
        <transformer ref="addGroupSize"></transformer>
      </filtering-router>
    </outbound>
  </service>

  <service name="doctorAggregator">
    <inbound>
      <vm:inbound-endpoint path="toDoctorAggregator"/>
      <!-- custom aggregator for aggregation incoming messages
          -->
      <custom-inbound-router class="eip.hospital2.mule.util.
          CustomAggregator"/>
    </inbound>
    <outbound>
      <outbound-pass-through-router>
        <!-- send message to a JMS queue, where the doctor
            system is listening -->
        <outbound-endpoint ref="toDoctor"/>
      </outbound-pass-through-router>
    </outbound>
  </service>
</model>
```

4 Discussion

Implementing this and other scenarios with either framework could be achieved without many problems. Both frameworks offer support for the used patterns and implementing or configuring them is straightforward.

However, when looking at the code of both examples the clear structure of rules defined in Camel is very impressive. At this point, the strength of a DSL can be seen immediately. For anyone who is familiar with basic programming knowledge the meaning of the defined rules will be clear very soon.

On the other side, when configuring all patterns in Mule, one has to know what specific XML tags are used, where they have to be applied (inbound or outbound section of a service) and what their meaning is. So some initial learning or guidance is needed to understand all parts of a Mule configuration and the patterns used in there. This does not mean that writing a Mule configuration is more difficult than coding Camel rules, however when taking a first look at both alternatives the Camel code can be interpreted much faster. This does further not mean that a Java DSL is always more readable than an XML configuration or should even be preferred when there is a choice. It is a very personal decision of integration developers which approach—either the Java DSL or the XML configuration—that they prefer for implementing their EAI solutions.

Pattern Name	**ActiveMQ**	**Camel**	**ServiceMix**	**Mule**
Pipes and Filters	oos	oos	oos	oos
Message Router	oos	s	s	s
Content-Based Router	oos	s	s	s
Message Filter	oos	s	s	s
Dynamic Router	oos	s	s	s
Recipient List	oos	s	s	s
Splitter	oos	s	s	s
Aggregator	oos	s	s	s
Resequencer	oos	s	s	s
Routing Slip	oos	ns	s	s
Process Manager	oos	ns	s	s

Table 1. Mapping of Message Routing Patterns to Frameworks

When talking more general about pattern support, it has to be said that the target patterns of Mule and Camel are routing and transformation patterns for which they provide nearly full support. The support for routing patterns is shown in Table 1. The abbreviations used in the table are the following: *s=supported*, *ns=not supported*, *oos=out of scope* (not expected to be supported by a framework) and *di=design issue* (just a design issue when designing an EAI solution - no special support by frameworks is needed). The tables also show the pattern support of Apache ActiveMQ and ServiceMix as they were covered in [13] from where the tables are taken. However, explanation in this paper will only focus on Camel and Mule. The patterns *Composed Message Processor* and *Scatter-Gather* are not listed in the table, as they are just a combination of other patterns. Also the *Message Broker* pattern is not listed as a broker and is not something that can be supported but is a real application. ActiveMQ is a message broker implementation and one can say that it therefore implements this pattern. From the table it can be seen that the only patterns where Mule is superior to Camel are the *Process Manager* for which Mule can use its built-in jBPM process engine and *Routing Slip* for which Mule also provides support. These two patterns are not supported by Camel.

Pattern Name	ActiveMQ	Camel	ServiceMix	Mule
Control Bus	s	s	s	s
Detour	oos	di	di	di
Wire Tap	oos	s	s	s
Message History	oos	oos / di	oos / di	oos / di
Message Store	s	di	di	di
Smart Proxy	oos	oos	oos	oos
Test Message	oos	di	di	di
Channel Purger	s	oos	oos	oos

Table 2. Mapping of System Management Patterns to Frameworks

Table 2 shows the support of system management patterns by the frameworks. A *Control Bus* is a component which helps administrating an EAI solution. Status information of all components and messages sent over channels should be seen on the bus. The pattern could be implemented by sending each message not only to the destination channel but also to the *Control Bus*. By applying this, the message flow can be inspected with the help of the bus. Furthermore, each component in the messaging infrastructure could send status information to the *Control Bus* telling it that it is still alive. All frameworks provide support for administration through JMX. This can already be seen as a *Control Bus* as one can inspect message channels and components of the messaging infrastructure. However, the tool support is rather little. Thus, some more work has to be done to achieve more support for this pattern. A *Detour* routes a message to an alternative path than initially defined. A *Message Router* can be inserted into the solution to provide the functionality of a *Detour*. Again, this is only a design issue. The *Wire Tap* pattern sends a message not only to the indented receiver but also to a second queue. This pattern is explicitly supported by Camel and Mule. A *Message History* deals with adding a history element to messages travelling through the messaging infrastructure. By doing this, each component can analyse the history and then knows the path a message took until it reached the component. This way, the flow of messages can easily be determined. This pattern can either be implemented by participating applications in the EAI solution by setting a property field. When doing this, it is out of the scope for the frameworks. Another way could be to use a *Content Enricher* after each component that inserts a history element into the message. When applying this, the pattern is a design issue. One step further than a *Message History* is the *Message Store* pattern. This pattern stores all messages in some kind of datastore. This could be achieved by reusing the *Wire Tap* pattern and storing all incoming messages to a database. This is a reuse of already existing patterns, therefore, it is marked as a design issue. A *Smart Proxy* can be used to reroute reply messages of an application to different destinations as initially defined. To use this pattern, some implementation work has to be done which ends up in an additional component that has to be inserted into the EAI solution. Therefore, this pattern is marked as out of scope for all frameworks. A *Test Message* can

be used to test the correct behaviour of a component. To achieve this, a generator for *Test Messages* is needed. A message has to be sent to the appropriate message channel to test the correct component. After the messages passed the component, it has to be rerouted again to some monitoring component where the result can be verified. For the last step a *Router* can be used. Thus, it can be seen that some implementation has to be done to support this pattern but, as it relies on already existing patterns, it is also marked as a design issue. The *Channel Purger* is used to delete all messages from a channel. For Camel and Mule this pattern is out of scope as it should be situated directly in the message broker.

Message construction patterns do not relate to the used frameworks very much, as they deal with the design of the message content which is not relevant for the used frameworks, but is important in the technical design phase of an EAI solution. However, support for this category can be achieved by setting properties on messages which can be easily done. The categories endpoint and channel patterns are also very much out of scope for the the frameworks. They are mainly covered by message brokers like ActiveMQ as they deal with low level messaging issues which can be covered by any message oriented middleware.

What also has to be taken into account is that the goal of Camel is to provide a enterprise integration pattern implementation that can be used in an ActiveMQ message broker or the ServiceMix ESB. Thus, it is specialised on providing a pattern DSL for integration developers. The goal of Mule is not to provide direct support for enterprise integration patterns, but be a framework that can be used for integration projects using a message-based approach—providing patterns is just one aspect of the Mule framework. Additionally, it offers a jBPM process engine or a Drools rule engine and more features that can immediately be used when working with Mule. This is where the advantages and strengths of Mule are situated. Such features are not provided by Camel and, therefore, it must be used in combination with further frameworks.

5 Conclusion

After analysing Mule, Camel, ActiveMQ and ServiceMix and implementing a set of scenarios of varying complexity, it turns out that the currently available Open Source frameworks support the enterprise integration architect with a broad variety of features in implementing even complex integration patterns. The first thing that has to be considered when using a message-based approach for integration is some message-oriented middleware. ActiveMQ is a powerful broker that can be used as it supports a variety of protocols and even implements its own protocol that can be implemented by any application for working with ActiveMQ. Through the integration of Camel into the broker, a lot of routing and transformation protocols already get delivered with ActiveMQ and can be used immediately. When there is a need for a standard-compliant EAI container ServiceMix can be used. It also uses ActiveMQ for messaging and Camel can be used for supporting integration patterns. However, messages are limited to XML

content when using ServiceMix as the JBI standard only allows XML messages. This can be mitigated by using XML marshalling and unmarshalling on any objects that need to be transferred.

On the other side, the competitor to these frameworks is Mule. Mule does not rely on any message broker like ActiveMQ but can be used with many different ones as it abstracts from the broker. Like Camel, Mule also offers a broad support for integration patterns and has a lot of transports available. Thus, messages coming from many different sources can be handled by Mule. It is also not limited to XML content and can also send POJOs as a message content. Additional advantages of Mule are the integrated jBPM and Drools engine. Thus, there is no need to use further frameworks in an EAI solution which has to be done when using Camel only.

A guidance like saying Mule is better for situation A and ActiveMQ, Camel and ServiceMix are better for situation B cannot be done. For an educated decision-making process, all requirements of the future EAI solution have to be collected and then mapped to the features of the different frameworks. This paper can give guidance for the mapping process of integration patterns to the frameworks but does not consider other aspects like security or transactions. For this, further investigation has to be done.

6 Future Work

System management patterns have not been considered for implementing the scenarios very much. The reason for that is the additional implementation effort which is required, and they are not delivered with the used frameworks out of the box. Investigation of them was mainly done on a theoretical level. Nevertheless, they are very important for monitoring and maintaining existing EAI solution. Thus, a future work that should be done is to expand the scenarios with system management patterns and implement them with the given frameworks and share experiences gained from it.

All scenarios used a message-based integration approach. Another approach is to use BPEL for orchestrating exposed services and implementing the needed business logic with that. A further possibility is to use SCA and SDO for implementing the given scenarios. A comparison of all three approaches would be very interesting in terms of development effort needed, clarity and simplicity of developed artefacts.

7 Summary

This paper analysed the support of enterprise integration patterns of Open Source frameworks with the focus on Apache Camel and Mule. A sample scenario was introduced to show the different approaches which are taken by the two frameworks for implementing patterns. Camel offers a Java DSL for configuring all necessary patterns which is very clear and self-explanatory. On the other side, patterns have to be configured in Mule in an XML configuration file

which leads to a large and not so clear configuration of patterns which is available when using Camel. However, the main goal of Mule is not to provide a clear definition of patterns, but to be an integration platform. On the other side the main purpose of Camel is to provide support for integration patterns.

When considering patterns in general, both frameworks offer a very good support for all routing and transformation patterns, which are the most important categories to provide a good message flow between applications in an EAI solution. When using system management patterns, some additional implementation effort has to be done. But as most of them are combinations of other patterns, the effort is not too large. Other categories of patterns are covered by other frameworks like ActiveMQ or do not need much support by frameworks, like the construction patterns.

The code snippets shown in this paper show the basic differences when implementing patterns. Everyone can decide for oneself if the Java or the XML approach is more suitable or superior when doing integration projects. Integration can be done with both frameworks, however Mule offers more functionality than a rules- and workflow engine, which is not offered by Camel. For this, an additional framework has to be introduced in Camel EAI solutions. But for simple integration projects, where none of them is needed the lightweight approach of Camel could be superior.

However, patterns are not the only point that has to be considered when doing integration projects. Other aspects like security also have to be taken into account. Thus, this paper only gives an overview of the pattern perspective in an EAI project. For making an educated decision on which framework to use, all requirements have to be collected and matched to the given features of the frameworks.

References

1. Gleghorn, R.: Enterprise application integration: A manager's perspective. IT Professional **7**(6) (2005) 17–23
2. Fowler, M., Hohpe, G.: Agile eai. Technical report, ThoughtWorks (November 2002)
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional (1995)
4. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Professional (November 2002)
5. Hohpe, G., Woolf, B.: Enterprise Integration Patterns. Addison Wesley (2004)
6. Hohpe, G.: Programming without a call stack - event-driven architectures. `http://www.eaipatterns.com` (2006) [Online; accessed 01-March-2008].
7. Marechaux, J.L.: Combining service-oriented architecture and event-driven architecture using an enterprise service bus. `http://www-128.ibm.com/developerworks/webservices/library/ws-soa-eda-esb` (2006) [Online; accessed 01-March-2008].
8. Schulte, R.W.: The growing role of events in enterprise applications. `http://www.gartner.com/DisplayDocument?doc\_cd=116129` (2003) [Online; accessed 01-March-2008].

9. Rademakers, T., Dirksen, J.: Open-Source ESBs in Action. Manning Publications (2007)
10. Binildas, C.: Service Oriented Java Business Integration. Packt Publishing, Birmingham (2008)
11. Vinoski, S.: Java business integration. IEEE Internet Computing **9**(4) (July-Aug. 2005) 89–91
12. Richards, M.: The role of the enterprise service bus. `http://www.infoq.com/presentations/Enterprise-Service-Bus` (October 2006) [Online; accessed 01-March-2008].
13. Thullner, R.: Implementing enterprise integration patterns using open source frameworks. Master's thesis, Vienna University of Technology (2008)