TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

M A S T E R A R B E I T

Implementing Enterprise Integration Patterns Using Open Source Frameworks

ausgeführt am
Institut für Softwaretechnik und Interaktive Systeme
Information & Software Engineering Group
der Technischen Universität Wien

unter der Anleitung von
Ao.Univ.Prof. Dipl.-Ing. Dr. Mag. Stefan Biffl
und
Univ.Ass. Dipl.-Ing. Dr. Alexander Schatten
als verantwortlich mitwirkenden Assistenten

durch

Robert Thullner, BSc.
Friedhofgasse 21, 7123 Mönchhof

Wien, 21. April 2008 _____

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

21. April 2008

Acknowledgements

First I would like to thank my parents for giving me the chance to study and supporting me throughout my years at university.

I also want to thank Alexander Schatten for offering me the chance to write this thesis, all interesting discussions and the guidance for this work. Additionally I want to thank Stefan Biffl for advising my thesis. I further want to thank Tijs Rademakers, author of the book „Open-Source ESBs in Action“, for reviewing parts of this thesis and providing me help at the practical part.

At last I would like to thank all friends and colleagues I met at university for having a good time.

Kurzfassung

Enterprise Application Integration (EAI) ist in vielen Unternehmen ein Begriff. Verschiedene eingesetzte Computersysteme sollen ihre Funktionalitäten mit anderen Systemen teilen damit neue Anforderungen abgedeckt werden können. Eine Möglichkeit um eine Integration durchzuführen ist der Austausch von Nachrichten zwischen Systemen.

Wie für viele Anwendungsgebiete in der IT Branche wurden auch für den Bereich der Applikationsintegration durch Nachrichtenaustausch Patterns gefunden und gesammelt. Patterns beschreiben eine allgemeine Lösung für wiederkehrende Probleme.

In der Open Source Landschaft wurden einige Frameworks entwickelt die in den EAI-Bereich fallen. In dieser Arbeit soll herausgefunden werden, welche Arten von Patterns von Open Source Frameworks unterstützt werden bzw. wie diese implementiert werden können. Dazu werden vier Projekte genauer untersucht. Diese sind Apache ActiveMQ, Apache Camel, Apache ServiceMix und Codehaus Mule. Die Unterstützung der Frameworks wird einerseits durch theoretische Durcharbeiten der Dokumentation und andererseits durch Implementierung von Szenarien erfolgen. Durch die Implementierung soll auch herausgefunden werden wie benützbar und entwicklerfreundlich die verwendeten Frameworks sind oder welche Problem durch deren Verwendung auftreten können. Weiters werden auch verschiedene Teile von EAI Lösungen betrachtet und untersucht welche Teile von welchen Frameworks abgedeckt werden können. Zuletzt wird noch eine kurze wirtschaftliche Betrachtung durchgeführt um herauszufinden ob es für Unternehmen vorteilhaft ist Open Source Frameworks zur Applikationsintegration zu verwenden.

Abstract

Enterprise Application Integrating (EAI) is a concept that becomes more interesting in many organizations. Various computer systems shall share their functionalities with other systems to achieve support for new requirements. One approach for integration is the exchange of messages between participating applications.

Like in other domains of IT, also for integration with messaging patterns were found and collected. Patterns describe a common solution for recurring problems.

In the open source domain some frameworks have been developed that settle in the EAI domain. This thesis shall find out which patterns are supported and can be implemented with the help of open source frameworks. The frameworks used are Apache ActiveMQ, Apache Camel, Apache ServiceMix and Codehaus Mule. To achieve this, the four frameworks will be taken under investigation. The support for patterns shall be found out by studying documentation on the one side and implementing scenarios on the other side. By doing this it shall also be found out how easy it is for developers to use and which problems can arise when using a certain framework. Furthermore parts of EAI solutions will be observed to analyze which of them are covered by the used frameworks. At last a short examination of business aspects will be accomplished to find out if it is beneficial for an enterprise to use open source frameworks for application integration.

Contents

1 Introduction

This chapter gives the motivation for this thesis. A short explanation of enterprise application integration (EAI) and messaging patterns for EAI will be given and it will be shown why it is a hot topic in nearly every mid- or large size company. No technical background on EAI and integration patterns will be illustrated here, this will rather be done in chapter 2.
After that, some motivation for using open source software instead of closed source commercial products is given. It will not be a extensive discussion of open source versus closed source, but some points will be mentioned which were important while working on the practical part of the thesis.

1.1 Motivation for Enterprise Application Integration

When looking at any mid- or large size company today one will notice that there are a lot of different applications running to support the business needs of the company. For example, taking a look at an insurance company one could find the following situation: the company sells life insurances on the one side and car insurances on the other side, where each type is handled by an individual department. As the requirements to a software system are different for the two types of insurances, both departments are running their own software system for managing their insurances and customers. When looking at the company from a customer's view, a customer can have life- as well as car insurances, but as there are two different systems running the data of the customer is stored twice in the organization. This is not a desirable situation when thinking of a simple use case when a customer wants to changes its address. The address has to be updated in both databases which can either double the work for an employee or in the worst case the address is only updated in one system and left with the incorrect values at the other one. This leads to data inconsistencies and can cause problems for the company and the customer. Although storage space is very cheap these days, keeping the same data twice also leads to unnecessary storage costs for a company. This example is not inconceivable, when taking a look at medium- or large-scale enterprises. Most software systems running in a company were build to support one specific business need for a department or the company without having an eye at other applications which are already in use. For an organization that has to fulfill many business tasks, various systems have been developed over the time. During this time the tools and programming languages used for building software also evolved, so newer systems are

usually build with newer technologies and on newer platforms. This way the company's IT infrastructure becomes a very heterogeneous network of applications, built with different technologies and on different platforms, which cannot easily communicate with each other.
Nowadays for a company to survive on a competing market it has to react on changes or even preempt them faster than other competitors [12]. To react immediately applications must be build using a flexible architecture to reduce the time and cost for implementing changes in the system. But as already mentioned before, old systems which are not very flexible are still in use and adapting old legacy systems is a very expensive and time consuming task. Thus another way has to be gone to achieve flexibility in enterprise systems. Gleghorn notes that EAI is one technology to achieve flexibility [12]. EAI allows applications to share data and business functionality across all integrated applications in a company application network. The flexibility gained comes from the point that when the integration solution is designed properly it is not massive work to support new protocols, new business processes or to integrate new software applications into the existing solution [12]. A well structured integration solution will create a loose coupled network of application working together and fulfilling the business needs of a company. Loose coupling also reduces the effect of vendor lock-in because a system can easily be replaced with any other system that provides the same business functionalities.

1.2 Motivation for Enterprise Integration Patterns

The way from a heterogeneous network of individual applications to an integrated solution where all applications can communicate and work together is complex and challenging. As in nearly every field of software development, patterns describing a general solution for recurring problems have been found over the time. Hohpe and Woolf have collected and described messaging patterns for enterprise integration [14]. The patterns will be described in more detail in section 2.5. When applying integration patterns while designing the message flow between applications it can be ensured that it is designed well enough to provide an architecture built with the needed flexibility and extensibility. Thus patterns are very important when designing the message flow for EAI projects.

1.3 Motivation for Open Source

There is a never ending debate between open source and closed source communities, arguing about the better approach for software development and utilization of software. There are a lot of points that can be discussed while comparing the two approaches. Here only a few points which were important while developing the practical part of the thesis will be noted. Other points like security, integration, etc. can be found when

doing a search on an internet search engine. A lot of material can be found by doing this.

1.3.1 Source Code Availability

The greatest advantage of a open source product is, as the name already suggests, that the source code is open. Anyone can view it which also implies that anyone can change it. No software is free of errors, but when one uses open source software one can detect the error and fix it oneself and assure that after correcting the error the software will work as intended. This opportunity is not available when using closed source software. When an error is detected, the software vendor has to be contacted and one has to hope that the error will be fixed and an update will be released. But usually without specific contracts a vendor cannot be forced to fix any detected error. Another point to mention in this context is the time it takes to correct the error. Perhaps finding the part of source code and correcting the code in an open source product can be faster than taking all endeavors for submitting an error to your software manufacturer.
Another advantage that comes with the open source code is that the software can be enhanced by implementing additional customized features for specific needs. When using closed source, the vendor has to be asked to implement a new functionality. But like already stated above, if there are no specific contracts it depends on the mood of the software vendor whether the new requirement is implemented or not. This way one could be left absolutely alone in the dark with a software that does not even meet all business requirements. Of course, as the vendor wants to make money the feature will be implemented pretty sure, but perhaps the time it takes to formally submit the requirement to the vendor even is longer than the time needed for implementing an additional requirement in an open source product.
Even if one is not interested in correcting errors or extending the functionalities of a software it might still be interesting to have a chance to look the source code and see how things are implemented. The knowledge gathered from this can be reused in other development projects being done in the future.

1.3.2 Support

All widely used open source projects offer support through mailing lists or online forums for free. People giving support are usually developers of the project, so the support is a real expert support. If this support is not enough for an enterprise there are usually companies offering commercial support for open source products which of course has to be paid. On the other side also commercial software vendors usually offer support, but the support is often outsourced to some other companies, which do not have to do anything with writing source code. Only in real seldom cases you could directly talk to a developer and ask for support when using a closed source product.

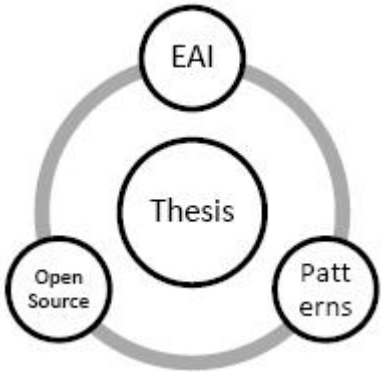

Figure 1.1: Motivation categories

1.3.3 Cost

One point that is always mentioned when talking about open source vs. commercial are the costs of the software. There are no or very low costs for a user when using open source software. This might be an interesting point when using OpenOffice.org instead of Microsoft Office. But when using open source software frameworks for development projects, the cost of the frameworks are usually quite low when comparing the costs to the total project costs and therefore not as important as other cost units in a project. A little comparison of costs between open source and commercial software will be done in this thesis in chapter 6.

1.4 Putting it all together

Figure 1.1 shows the three categories mentioned in this chapter. In the open source communities a couple of frameworks have been released that can be placed in the EAI domain which help building EAI solutions. Thus there are open source projects on the one side and integration patterns on the other side. What shall mainly be done throughout this thesis, is to find out how and which patterns are supported by some open source frameworks and how patterns can be used and implemented to create good EAI solutions. How this shall be achieved and other research goals which are related to the main research question will be described in chapter 3. The used frameworks are *Apache ActiveMQ, Apache Camel, Apache ServiceMix* and *Codehaus Mule*.
What will not be covered in the thesis is the process to come from a requirement specification to appropriate integration patterns. Umapathy and Puroa show a methodology that finds integration patterns for processes which are modeled with the help of the business process modeling notation [21]. To find out all details of the methodology, see [30].

2 Related Work

This chapter deals with topics that are related or build the fundament for the thesis. It starts with the challenges when building EAI solutions, goes over types of EAI projects, technical approaches, integration patterns, architectural concepts and ends at standards in the EAI domain.

2.1 Challenges for EAI

The problem described in the motivation chapter, dealing with an organization that employs two departments where each has its own customer management system does not only arise from the fact that the responsible persons are not informed about a likewise system running in an other department, but there are also other problems. At this point not only technical but also company-political problems arise. Each department has its own budget and there are always power struggles between the heads of the departments to get their budget approved. So why should a system developed for the customer care department and paid with the money of this department suddenly also be used by the support department, which did not pay a cent for developing the system?
Another point is that the customer care department feels that the collected data is theirs. As in these days, data is a very expensive good, even more expensive than the technology behind it, why should the department share their data with other departments?
What should be shown with these two examples is that problems while developing an EAI solution not only occur on technical levels, but even harder to solve problems arise at the political level of a company. Solving these problems is very important because they provide the base that an EAI solution will be accepted by the company as a whole. Thus a rethinking of responsible department heads has to take place from a department-oriented to a more company- and customer-oriented decision making.
There will be no deeper investigation in political problems while developing an EAI solution. The intent was just to note them to show their existence. For the rest of the thesis, the focus will be on technological issues of an EAI solution.
Developing EAI solutions is not an easy task for any developer or software architect. As already stated in the motivation chapter, EAI deals with applications running on different platforms, developed with different technologies and may not even all run in the same networking infrastructure. Thus a good integration developer has to have

expert knowledge in all or many of these fields. It will be hard to find such an expert on the job market. Hohpe and Woolf also identify other challenges that have to be solved for getting an usable EAI solution [14]:

- The first challenge conducts a problem, that was already mentioned under the „political problems“. There must be a shift in company politics. No group or department is the owner of a system anymore, but is only a part of the complete system infrastructure that spans around all groups and departments of an enterprise.
- As the integration solution should integrate nearly all systems in a company, it of course also integrate all critical business applications. So if the EAI solution fails, all critical business applications will also fail, as they all work together. So a correct functioning of the EAI solution becomes vital for the complete company to run their systems without any failures.
- Many participating applications of an integrated solution are legacy systems or prepackaged software applications. An integration developer usually does not have any control over these systems and therefore can not make changes that could help to make them easier to integrate. So integration developers have to find ways to overcome the shortcomings of the applications, which is not an easy task.
- Not many standards are established in the EAI domain, which also slows down the development of integration solutions. XML, XSL and web services are the main players in this domain, but the widely use of web services also leads to many „extension“ and „interpretations“, which result in a lack of interoperability between „standard-compliant“ products.
- Although XML is widely used in the EAI domain this does not mean that all integrated applications speak the same „XML dialect“. There can still be semantic differences for the content of an XML based exchange. Thus finding and eliminating these differences has to be done to ensure that all integrated systems really speak the same „language“.
- As already seen in the previous points, developing an EAI solution is a difficult and exhausting task. Operating and maintaining an existing solution is at least as difficult as the implementation. Through the distributed nature of an integration solution, deploying, monitoring and troubleshooting EAI solutions are complex tasks requiring a lot of specialized skills, which may not even exist in a company.

When looking at all points mentioned here, an EAI solution does not necessarily make the life of an IT department, entrusted with operating of the system, easier but can also make it harder as it was before.

2.2 Types of EAI

Hohpe and Woolf give six types of categories where integration projects typically fall into [14]. One specific solution does not necessarily fall into one of them, but can also be a combination of more types. In figure 2.1 all six types can be seen.

- **Information portals:** (see figure 2.1 (a))

  It is quite common for users to have access to more than one system for gathering all data they need to complete a business task. An information portal is an integration concept that collects data from many different systems and displays them in one system only. Thus the users only have to log in at one system to see and access all necessary information there and do not have to connect and log in to many different system where only partial information can be seen.

- **Data replication:** (see figure 2.1 (b))

  As already described in the motivation for EAI, individual departments of a company can have their own datastore for customer records and customers may be stored in each of the systems. The data replication concept integrates all datastores in a way, that if a record gets updated in one system, all other records of the same customer in other datastores will get updated, too.

- **Shared business function:** (see figure 2.1 (c))

  Many business functions of software systems are needed in more than one application and each application has implemented the function for its own. An example for that could be a validation function for a social security number or a validation for a zip code. Like sharing of data, business functions can also be shared among applications. Thus if the function needs to get updated for some reason, only one implementation has to get updated instead of updating each application.

- **Service-oriented architetures:** (see figure 2.1 (d))

  A shared business function is often called service. When a collection of services exists that fulfill business functions for the company, managing these services is very important. Applications need a directory where they can look for services. Furthermore each service needs a description of its interface so that applications get to know how they have to invoke the service. These two function are called service discovery and negotiation, which are the key elements of a service-oriented architecture.

  When there are enough useful services available in a company, new applications can be built by reusing already existing services only.

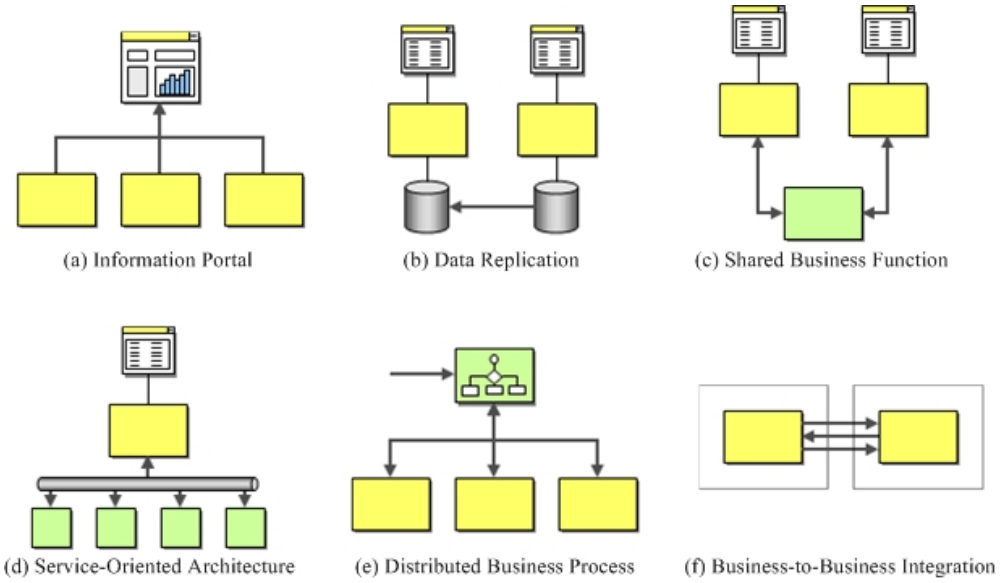

Figure 2.1: Six integration types taken from [14]

- **Distributed business processes:** (see figure 2.1 (e))

  One of the main goals of EAI is to span business processes from one system to multiple system in the network. A business process could be calling a method on one system and then passing the result to an other system for calling a method there. Thus there is a need for some business process management component exists that can handle and manage the execution of a business process between different systems.

- **Business-to-business integration:** (see figure 2.1 (f))

  The previous points were focused on the integration within an enterprise. However a company may also need to communicate and integrate with systems outside its infrastructure. For example a shop system may want to send a request to the shipping company to display the expected time of delivery to the customer. When integrating application between companies issues like transport protocols, security and standardized data formats play an important role when designing and implementing such a business-to-business integration solution.

2.3 Technical Approaches for EAI

Basically there are four different integration approaches for enterprise integration. The list starts from simple and leads to more complex approaches which have been devel-

oped. A little introduction will be given for each approach. For more information on the approaches, see [14].

1. File Transfer
2. Shared Database
3. Remote Procedure Invocation (RPC)
4. Messaging

2.3.1 File Transfer

When using *File Transfer* as integration style an application simply writes data it wants to share with other applications into a file and saves it somewhere on a file system where other applications can read and process the file. To achieve this, all applications must adhere to the same file format and furthermore must have the same semantic understanding for the contents of an exchanged file.
This seems like a fast and easy approach for integration, but also leads to various problems:

- If not only one, but more applications want to process a file, there are no regulations which application is allowed to read the file first or which (if any) application is allowed to alter or delete a file.
- The producing application must implement a unique naming strategy to ensure that no file names are used twice and so an old file is overwritten. The consuming applications must implement this strategy either, to read the correct file at a specific time.
- The management of file system is also important. Someone has to ensure that all applications have read and/or write permissions to an exchange directory or if they do not have the permissions, someone has to copy exchanged files to folders where other applications have sufficient permissions. This can further lead to inconsistency among exchanged files as they are stored at two or more different locations.
- When updates are made it takes time until all applications are aware of the update. As it is expensive for programs to write and read files, files are often written and read at predefined time intervals (nightly, weekly, etc.). So it can take nearly a day for a data update to be passed to all applications. This may not be a problem for some applications, but for many real-time applications this is a tremendous drawback.

This list of problems is not complete now, but it already shows that file transfer is not a satisfactory approach for EAI. Perhaps it may be a good approach for a very small integration project where one can live with the above mentioned drawbacks, but for large-scaled integration projects the drawbacks are too profound to build an integration solution using this approach only.

2.3.2 Shared Database

A *Shared Database* approach for integration uses a database which is shared by all participating applications. This solves the problem of data inconsistencies arisen at the file transfer approach because updated data is immediately saved in the database and all applications can read it. For multiple updates of the same data, database transaction managers can handle the updates so that no inconsistencies occur on the data. But this approach also has some immense drawbacks:

- The design of a shared database is a very hard work. For many applications it is difficult to find a database schema that is satisfactory for one application only. Finding a schema for many applications is a job which nearly cannot be done. This arises also some political problems in a company if applications from different departments need to share one database.
- Shipped, prepackaged applications usually contain their own database and most of the times cannot be integrated into a shared databases.

These drawbacks show that the shared database approach is not an optimal solution for EAI, too. As for file transfer it might be a good solution when only integrating a small amount of applications which one can configure to use a shared database. But as soon as shipped applications, for which configuring a datasource is not possible, come into play this approach cannot be applied anymore.

2.3.3 Remote Procedure Invocation

The first two approaches for EAI use the principle of sharing data among various applications. A different approach is *Remote Procedure Invocation*. Here not data, but functionality of applications is shared with other applications. This is done by simply calling methods of a remote application which are published and accessible by other applications. In the last years web services have been widely used for this integration style, as they are easy to use and platform independent. For programmers, calling a remote method looks like calling a local method. Thus for the programmers point of view, this would be a good solution. But as it looks like a local method call this also leads to a drawback, because the method invocation does not behave like a local method call and can therefore lead to unexpected errors. Such an error could occur, because

the network is down and therefore a method cannot be invoked. As participating applications are distributed across a network there will always be some delay when calling a remote method which is not present when making a local method call. The approach also has an architectural drawback. When calling a remote method the call couples two different applications more together and changes in one application may affect other applications. If a method signature changes and calling applications are not informed about that, they will produce an error while calling the remote method. Even if the calling method gets informed of the change, the application has to be changed to make a correct remote call. This means the application has to be stopped, changed, recompiled, retested and restarted which also takes some time to get an application running correctly again. Thus this approach couples application even tighter together because they are dependent on each other which is not a preferable situation.
This approach could be used when integrating two business-to-business partners through a method invocation on the business partners system. But in an integration project within an organization this approach is neither an optimal solution for EAI.

2.3.4 Messaging

A good approach for loosely coupling applications together is *Messaging*. It can be thought as a file transfer with very little packets where the receiver is notified when a new packet arrived, waiting to be consumed. Messaging combines all three approaches discussed until now into one technique. For sharing data between systems a message can contain all relevant data. If one needs to invoke a method on a remote application a message can contain the name of the method call. The remote application extracts the method name out of the message and invokes it. The result of the method call can be passed back to the caller in a reply message. Thus messaging not only combines all three approaches into one but also brings advantages, like loose coupling, to an EAI solution.
When using this approach some infrastructure is needed for sending and receiving messages. A messaging infrastructure needs to implement some retry mechanism to ensure that all messages get delivered and no message ever gets lost. If the receiving application is not running at the time when a message is sent the infrastructure has to save the message in some datastore and deliver it when the receiving application is running again. This implicitly leads to asynchronity in the EAI solution which means that a sender and receiver of a message do not have to be running when a message exchanges takes place.
Messages can also be transformed while delivering the message to other applications without the sender or the receiver even noticing that the message was transformed. This can be necessary if sender and receiver use a different message format or protocol. Furthermore semantic differences in a message can be resolved with some transformations. In section 2.5 a brief introduction to messaging patterns, including transformation, will

be given.
Although messaging is a good approach there are also some drawbacks that have to be considered when using messaging as integration approach. As the system is of asynchronous nature debugging and troubleshooting the system is more difficult than in synchronous applications. Also programmers and integrators have to think asynchronously which is not the usual way people think and so it is harder to think in this way. Hohpe has written an interesting article about such problems. For more information on that, see [13].

2.4 Loose Coupling

It was already mentioned that messaging integrates applications in a loose coupled way. In this section the basic ideas and the importance for every modern EAI solution will be explained in a few sentences. Loose coupling is the main paradigm for every modern EAI system and even for every software architecture design. Doug Kaye wrote a whole book about this paradigm [16].
When communicating in a loose coupled way, all communicating parties make as little assumptions on each other as possible. By doing this they do not rely on each other for fulfilling their business tasks. An example for a tight coupled system is a local method call in any program. When a method is called parameters can be passed to it and a return value can be passed back. The method can only be called if the correct amount, the correct datatype and the correct order of parameters are known. Otherwise a compile error will prevent the program to be executed. So a lot of assumptions have been made to call a local method. Of course within a class or a program assumptions have to be made because this is the usual way of programming. But when looking a bit further, the class- or domain design for a program should already be done with loose coupling in mind. This leads to an easily extensible architecture for a software system. When taking another look further and looking at the integration of software systems, communication in a loose coupled way can make applications work together but keep them independent from each other for fulfilling their business requirements. Of course integrated systems rely on each other, as the cause for communication is to get information which is needed to achieve a requirement. But when communicating loose coupled a system can easily replaced by an other system, because not more assumptions on participating systems as needed are made, which decouples all systems.
A way to communicate in a loose coupled fashion is *Messaging*, what was already described in section 2.3.4. A practical example for demonstrating the advantage and usefulness of loose coupling could be the following: a client is getting data from an old host-based system for displaying it to users. Over the time a new modern system has been implemented and now the client should get all relevant data from this system. When communication was implemented in a tight coupled way, the client code

implemented all communication infrastructure code for communicating with the host system. Now when the switch to the modern system has to be done, all the communication code has to be rewritten, retested and the whole client application has to be redeployed. When the communication is done in a loose coupled way the client just sends a message to a defined channel and awaits a message on an other defined channel. This way the client does not care, who is receiving the message and providing him with the correct data. This is done by the communication middleware. The message can either be received by the host system or by the new modern system, but this is not relevant for the client anymore. When the host system is turned off, the middleware is just reconfigured to send messages to the new system and the client does not even know that. Thus no code on the client has to be changed.
When building modern EAI solutions all communication should be done in a loose coupled way to get an easily extensible and modifiable integration solution.

2.5 Messaging Patterns for EAI

In 2.3.4 it was already described what messaging is. This chapter introduces patterns that can be used when implemented a messaging approach for EAI. After a brief common introduction to patterns, six categories of messaging patterns will be described.
A pattern is a domain independent solution for a recurring problem. The description of a pattern contains the domain where it can be applied, design constraints, consequences and trades-offs introduced through the usage of a pattern. Instead of providing sample source code, the pattern should rather help designers to describe and communicate design problems and solutions in a particular context.
Patterns were first introduced in the domain of architecture by Alexander et al. in [1]. Gamma et al. first brought the usage of patterns into the software domain [10]. They have gained a lot of acceptance in the software industry and play an important role while analyzing, designing and implementing software systems. Wiley also published a series of five books *Pattern Oriented Software Architecture* dealing with patterns for software architectures [4, 25, 17, 2, 3]. One further book about patterns in enterprise application architecture was published by Martin Fowler [9].
When applying and combining more than one pattern in a solution design a *Pattern Language* is formed. Enterprise integration patterns (EIP) described by Hohpe and Woolf give a pattern language for integration problems describing possible combinations of patterns which have been proven useful [14]. The messaging patterns for enterprise application integration are subdivided into six categories.

1. Messaging Channels
2. Messaging Endpoints
3. Message Construction

4. Message Routing
5. Message Transformation
6. System Management

Each category will be described in the following subsections using one or two sentences per pattern. This gives a rough overview of all patterns, but does not include detailed information. For more information on each pattern, a look at [14] should be taken.
Main target for the integration patterns is the messaging approach for EAI. A different approach for EAI is Spaced-based computing(SBC). SBC is based on *Linda* spaces [11] and provides an alternative to messaging. Martin et al. do a comparison between spaces and messaging with respect to integration patterns [20], which is worth reading for getting a view on different concepts for EAI. However the broadly used technique for EAI is messaging.

2.5.1 Messaging Channels

In the introduction to messaging (section 2.3.4) it was already noted that some messaging infrastructure has to exist when using messaging. One job of the infrastructure is to establish messaging channels between applications for enabling communication. Thus an application has to have access to a messaging channel where it can put messages onto. The receiving application listens on a channel for new messages.
For messaging channels various patterns exist. A *Message Channel* is a simple connection between two applications as already said above. When using a *Point-to-Point Channel* only two applications are connected to the channel. A sender puts a message on the channel and a receiver takes the message off. Only one sender and one receiver can connect to the channel. When multiple message consumers are required a *Publish-Subscribe Channel* can be used. This channel allows that more than one application can connect to the channel and receive messages from a channel. The Java Messaging System (JMS) provides a *Queue* and a *Topic* which represent these two types of channels. However, there are more patterns falling in category. A *Datatype Channel* is a channel where only messages with a specific datatype can be sent to. An *Invalid Message Channel* is a channel where messages that were not expected by the receiver are put onto. A *Dead Letter Channel* is used by the messaging infrastructure to sort out messages that cannot be delivered. *Guaranteed Delivery* can be used to ensure that a message gets delivered, even if the messaging infrastructure crashes while delivering or the receiver is not running at the time a message is sent. A *Channel Adapter* is used to enable messaging for applications that do not have built-in messaging capabilities. For example, one could capture HTTP traffic and insert into the messaging infrastructure. If there are more than one messaging infrastructures running on a network, a *Messaging Bridge* can be used to send messages from one system to the other and vice versa.

When several applications need to exchange data, a *Message Bus* can be used to enable communication between all participating applications. New applications can be added easily to the bus, without affecting other applications that are already communicating over the bus and of course applications can be removed from the bus easily, either.
To summarize, one can see that channel patterns show various ways of sending messages from a sender to one or more receivers.

2.5.2 Messaging Endpoints

Endpoint patterns deal with various ways how applications can connect to message channels. A *Message Endpoint* is used for connecting an application to a message channel. A *Messaging Gateway* can be used if an application communicates over various message channels. The entire messaging related source code is put into the gateway where all communication between the application and the message channels happens. A *Messaging Mapper* can map the content of a received message to domain objects which are used by applications. A *Transactional Client* can be used when transactions between sender and receiver are needed. When using a *Polling Consumer*, a message receiver polls the message channel in regular intervals and looks if new message have arrived. The opposite of this is the *Event-Driven Consumer*. Here the consumer registers a callback method on the messaging infrastructure which gets called when a new message is available. *Competing Consumers* can be used when messages get delivered very fast and one consumer cannot handle the complete message load. So there are more consumers registered on a message channel but only one consumer receives and processes an incoming message. A *Message Dispatcher* can be used to coordinate message processing of more consumers on a message channel. When not all messages on a channel shall be received, a *Selective Consumer* can be used to filter out uninteresting messages. If a consumer does not want to miss any message a *Durable Subscriber* can be used. The messaging infrastructure stores messages until all registered consumers have received all messages. For filtering out duplicate messages on a message channel a *Idempotent Receiver* can be used. A *Service Activator* is used when an application wants to make services available to other applications. When using this pattern, a service can be invoked via messaging and non-messaging techniques.
Recapitulatory, endpoint patterns show a couple of different ways how applications can connect to and communicate over messaging channels.

2.5.3 Message Construction

Message construction pattern describe ways how messages can be constructed or a message flow between applications could be established.
A *Message* is something that is sent over a message channel from a sender to a receiver. A *Command Message* is a message containing a command that should be executed

at the receiver. A *Document Message* contains some content in which a consumer is interested. An *Event Message* informs a receiver about an event that occurred. *Request-Reply* can be used if the producer of the message also awaits a reply from the receiver. *Return Address* specifies the address where the receiver should send the reply to. *Correlation Identifier* can be used to mark that multiple messages belong together and therefore should be processed together. *Message Sequence* describe a sequence of message belonging together. The difference between *Correlation Identifier* and *Message Sequence* is that in the latter the ordering of messages is important. *Message Expiration* can be used to indicate that a message is not needed anymore if some time has passed by. This can be because there already exists an updated message with more actual information than the first one. *Format Indicator* can be used to indicate the format of the message content. This can be important if there are more versions of a content specification, that have to be parsed differently.

2.5.4 Message Routing

Routing patterns describe ways how messages can make their way from the producing endpoint through the messaging middleware until finally reaching the destination endpoint.
Pipes and Filters is an architectural concept where the message flow consists of pipes that transport messages and filters that do processing on incoming messages and pass messages to the next pipe. A *Message Router* can be used to route messages to endpoints which are waiting for incoming messages. A *Content-Based Router* can make its routing decisions based on the content of an incoming message. A *Message Filter* is used to filter out messages that should not be delivered to a specific endpoint. A *Dynamic Router* can change its routing rules during runtime through commands that are sent to the router. If a message shall be delivered to multiple consumers, a *Recipient List* can be used. Sometimes messages need to get split into individual messages that are sent to different endpoints. All replies can be aggregated into one message again. For achieving this, a *Splitter* and an *Aggregator* can be used. If ordering of the messages is important, but messages got disordered a *Resequencer* can be used to bring all messages back into order again. A *Composed Message Processor* is a combination of a *Splitter* which splits a message into parts, a *Router* routing the parts to different endpoints and an *Aggregator* aggregating all responses back into one message. Following scenario could be implemented using this pattern: it shall be validated that all order items in a message are in stock. The message can be split up into the order items and each item can be routed to a system that manages this item. A response from the stock system contains the availability of items. All responses are aggregated again into one response message for the shop system. A *Scatter-Gather* is another combination of patterns. A request is send to multiple systems which each response to the same reply channel. An *Aggregator* is used to collect the response and pass a response back

to calling system, depending on the collected messages. Imagine a shop system that only wants to buy from the cheapest vendor. A request is sent to all vendors and the aggregator only passes back the response of the cheapest vendor. If a message has to be routed to more than one system and the order of the systems is important, a *Routing Slip* can be used to configure the correct routing of a message. A *Process Manager* can be used when the flow of messages has to be in some order, but the order is not known at development time. The manager is able to inspect the content of a message and can send it to a specific endpoint according to the content. A *Message Broker* can be used to decouple the destination- from the sending endpoints. All messages will be passed to the message broker, which will then deliver the message to the correct endpoint.

2.5.5 Message Transformation

When applications use different message formats, transformation patterns can be used to transform messages in a way that the receiving application can interpret it.
A *Message Translator* translates the content of a message to a format expected by the receiving application. An *Envelope Wrapper* can be used if specific header information shall be added to a message, but the producing application cannot achieve this. The message is packed into an envelope that contains the necessary fields and then sent to the receiver which unpacks the message again. If a receiver needs additional information in the content of a message as the sender has put into, a *Content Enricher* can add content to a message. The opposite of this is a *Content Filter* where content can be cut out of a message. Sometimes parts of a message should not be seen by some systems on the way to the receiver. The *Claim Check* pattern cuts out information from a message and stores it somewhere. Instead of the content a placeholder is inserted to the message that can reference to the content later, when it shall be inserted into the message again. A *Normalizer* can be used to format messages to a standard data format used by the receiver. When using a *Canonical Data Model* all application adhere to the same message format when sending messages to other applications.

2.5.6 System Management

All previous patterns can be used when developing integration solutions. The last category of patterns is targeted for the operation and maintenance of integration solutions. System management patterns include possibilities to manage an integration solution. This also includes debugging and fixing errors with help of these patterns.
A *Control Bus* can be used for administration of components within a messaging infrastructure. For example all components in an EAI system could send status messages which are received by the *Control Bus*. If one needs to inspect a message on its way through the infrastructure, a *Detour* can be used to route a message to an endpoint where one can read and inspect the content of a message. A *Wire Tap* can also be used

to inspect messages on a channel. To debug and analyze the flow of messages through the system the *Message History* pattern might be helpful. A history field is added to a message, which is updated at every component where the message is processed. A related pattern to that is the *Message Store*. When a message is consumed at the last endpoint the history of the message is not available anymore. If one wants to inspect and analyze messages at a later time a *Message Store* can be introduced where messages are saved to. In a request-reply scenario one might want to reroute reply messages for inspection to some other endpoint as defined in the return-address field. For this a *Smart Proxy* is the pattern to use. When a message is received, the proxy replaces the return-address field with its own address. When the reply is received by the proxy it can send the message to any endpoint configured at the proxy. If some component has to be tested, a *Test Message* can be inserted into the messaging infrastructure for inspecting the behavior of a component. If there are messages on a channel that need to be deleted, a *Channel Purger* can be used to delete all messages on a channel.

2.6 Architectural Concepts

In this section some information on architectural approaches for software development and their relation to application integration are given. The concepts include service Oriented Architecture(SOA), event driven architecture(EDA) and the enterprise service bus(ESB).

2.6.1 Service Oriented Architecture (SOA)

Service oriented architecture is an architectural design approach where applications are built by invoking services that are provided by other applications. Services provided by one system are published to a directory, which can be browsed for services. A specification of a published service contains interface descriptions that specify what the service does, what it expects for providing the correct functionality, how to invoke the service and other information. Services can be found by software systems by specifying criteria descriptions they need. The registry tries to find a service according to the search criteria and passes the interface description of the service found back to the requesting system. After that it can invoke the service directly at the providing system. Thus the basic concept of a SOA is the Publish-Find-Bind paradigm which is described above. The ultimate goal of this concept is that systems can find services autonomously and replace services with better services as they are published. Thus the system behaves in an autonomous manner. What is slowing down the development of this vision is that there are not many public registry systems available. Another problem is trust and quality of service parameters for published services. There is no guarantee that the information provided by a service is correct or that an invocation is done at a

guaranteed speed that is acceptable for an application. This is still an area of research and it is not obvious now how these problems will be solved.
As this thesis is not about SOA, this short introduction should be enough for understanding the rest of the thesis. What will be shown next is the relation between SOA and EAI. Basically, what systems do when they are integrated is sharing of data, information and functionality. Data can be shared over many ways (files, databases, messages, etc.). When it comes to sharing functionality an application has to make some functions available to other systems. This is already a service for other systems which can be invoked. Thus the first step to a SOA is already done by sharing functionality. The second step is to implement a registry, allowing systems to dynamically find and bind required services.

2.6.1.1 SOA, Web Services and EAI

Misleadingly web services are sometimes used as a synonym for SOA. In fact, web services are only one possible technology for implementing a service oriented architecture. Many standards have been developed which should build the fundament for a widespread acceptance and usage of web services. For the registry component UDDI can be used, for describing interfaces of services WSDL is utilized. Communication between services can be done by the SOAP protocol. Around these three standards many other standards have been developed to support the growth of web services, usually referred as WS-* standards. The standards cover the topics transports (BEEP), messaging (SOAP, WS-Adressing, ...), description and discovery (UDDI, WSDL), reliability (WS-Reliable Messaging), transactions (WS-Atomic Transactions, WS-Coordination,...), security (WS-Security, WS-Trust, ...), business processes (BPEL) and management (WS-Manageability, WS-Distributed Management,...). A list of all web service standards was published by IBM and can be found online at [6].
Thus the WS-* standards would cover all topics which are needed for an EAI solution, SOAP can be used for messaging and can be extended by WS-Reliably Messaging to ensure that a message gets delivered. BPEL can be used to build business processes and for management of the EAI solution management standards can be used. One could now assume that web services are the tools for building EAI and nothing else is needed. Zhu discusses the power of web services for integration issues in [37]. Vinoski also wrote about web services for integration in [32]. Integration can be done with web services, but it is a very heavyweight approach. Web services standards are all based on XML and when implementing an EAI solution a lot of XML configuration files have to be written to get a working solution. The three main standards SOAP, WSDL and UDDI have been accepted by all vendors. All other standards have been developed to solve a specific problem in the web services domain. Unfortunately, for some problems not only one standard, but multiple competing standards were introduced by different vendors or interest groups. This leads to a situation where most standards are not

implemented in EAI products. Therefore finding compatible products implementing all needed standards is very hard and usually leads to a vendor lock-in. But as soon as standards are broadly accepted by all major vendors of EAI solutions the web service technology can be a reasonable approach for EAI solutions.
What was already said in the SOA introduction that not many public registries are available is also true for web services and UDDI, which also limits the widespread usage of web services. But inside a company the problems of trust and quality of service do not occur as they can be controlled by the company itself. Thus web services can widely be used for intra-organizational use. When using web services for inter-enterprise communication one can usually also trust business partners that provide services and can be used without any worries.

2.6.2 Event Driven Architecture (EDA)

Gartner first introduced the term event driven architecture in 2003 [26]. In an EDA applications produce events and by default do not care any further about what happens to the event. Other systems across the network can consume events and call some business functionality according to the received event. One advantage always mentioned in papers about EDA is that it behaves like in the real world, because business functions in an enterprise are always triggered by events which have to be handled the right way. Thus EDA ports the system of business events to computer systems.
Hohpe stated the key characteristics of an EDA [13]:

- Broadcast Communication: Not only one, but more systems can listen for events broadcast by any application.
- Timeliness: Events are published as they occur and not in any predefined time interval.
- Asynchrony: Publishing systems do not wait until any receiving system processes an event. It is just published and the producing application goes on with its execution without waiting for any reply.
- Fine Grained Events: Applications tend to publish simple individual events instead of aggregated ones.
- Ontology: Semantics of the receiving events has to be defined somehow, typically in some form of hierarchy.
- Complex Events Processing: Systems can understand the relationship of individual events and can handle these relations.

A difference between a SOA and an EDA is the exchange pattern. SOA typically is implement as a request-response exchange whereas EDA aims at asynchronity for

publishing events. An application producing events does not depend how an event is consumed or even not consumed at all. Thus EDA enables a real decoupling of systems. The usefulness of an EDA for EAI is that different applications can communicate via exchanging of events. Instead of calling services of other applications like in a SOA, the event driven approach aims at exchanging events. However an event could also trigger an application to call some business function, which implicitly leads to a SOA again.
One drawback worth mentioning when talking about EDAs is the different programming model used in many modern programming languages. The difference is the missing *Call Stack* which is used while programming with traditional languages like C, C++ or Java. This aspect is discussed by Hohpe in [13].

2.6.3 Enterprise Service Bus (ESB)

This subsection is an interesting part of the thesis as some of the the frameworks being used in the practical part call themselves an ESB. So the reader should know what an ESB is and what it is used for. The role of an ESB in a SOA and an EDA will be shown first. After that capabilities an ESB should provide will be described.
In figure 2.2 on the left side a typical SOA is shown. A client can access service A or service B. The drawback of this scenario is, if the binding of service B is changed or service B is replaced by some other service, the client has to be changed, too. When looking at the right side of the figure, the client only communicates with the ESB. When service B changes, the client does not have to be changed, because it actually does not even know about service B. It just sends an event to the ESB which is responsible for calling the correct service and passing the result back to the client. Thus when a service implementation is changed, the client does not have to be changed. So an ESB decouples client applications from service implementation. Marechaux states that an ESB combines and bridges together an SOA and an EDA [19]. This fact can already be seen in the little example described here. The client is responsible for providing business services to the user. When looking at a trading scenario, a user wants to place trade orders. This can be seen as an event for the system. The client publishes an event *PlaceTradeOrder*. It does not know who will handle the event, it just expects an answer that a trade has been placed or some error occurred while placing the trade. What happens now is that the ESB will consume the event and look for a registered service which is able to handle the event. The ESB has knowledge about all published services in an enterprise and matches the event to a service, which can place an order. It then invokes the service and sends back the answer to the client. Thus an ESB maps events to services, which was already stated above. This fact introduces service location transparency, sharing of services across the enterprise and the separation of business services from service implementations to the enterprise IT infrastructure.
At the moment there is no industry-agreed definition of what an ESB is and what the term actually ESB means. Mark Richards gave a talk about the role of an ESB in an

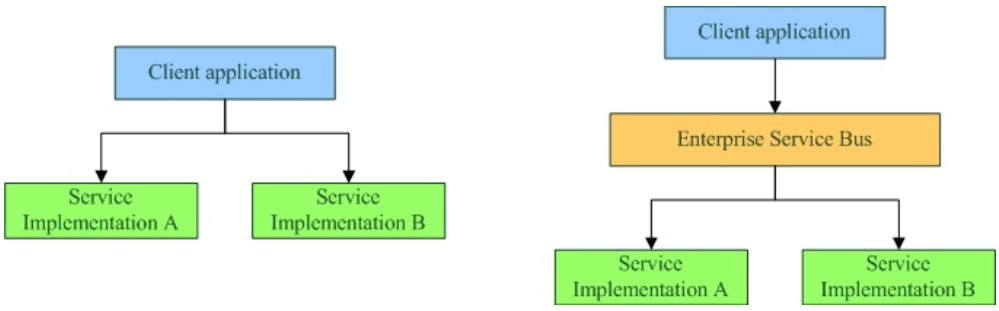

Figure 2.2: The role of an ESB in an SOA

SOA [23]. In the presentation four answers are given to the question of what an ESB actually is:

1. Is it a product?
2. Is it a pattern?
3. Is it an architectural component?
4. Is it a hardware component?

A typical answer could be that an ESB is part of all of them. Many products call themselves an ESB, other people say that ESB is pattern or an architectural component. The forth point may sound strange at first look, but IBM is offering a hardware component which provides many of the capabilities a typical ESB should provide, so it can be seen as an ESB.
Nick Allen, Program Manager at Microsoft asked for the definition of an ESB in his blog. Dubray summarized the replies of the question in [7]. What can be seen in the article is that different people have different opinions of what ESB is. Furthermore if one looks at ESB vendor specifications a lot of features of the vendor's ESB are given, but no clear definition of what an ESB is and if the vendor's features make up an ESB or not. Thus what can be seen again is that there is no concise definition of an ESB.
At this point, the presentation of Richards will be summarized to make the reader understand what an ESB is through telling what capabilities it should provide. Ten capabilities an ESB should or could provide are given in the presentation. Integration architects can look at them and find out if some of them are needed in their proposed project and afterwards look if there is a product available that provides the needed capabilities. Thus not all ten capabilities have to supported, to call a product an ESB; only a few of them, which are needed for a current problem are enough.
The capabilities are:

1. **Routing:** An ESB should be able to route messages between applications. Routing can scope from simple static routing up to complex rule-based routing.

2. **Message Transformation:** An ESB should be able to transform a message to a different format, the receiver of a message expects.

3. **Message Enhancement:** An ESB should be able to add/remove content to/from a message before delivering it to the receiver.

4. **Protocol Transformation:** An ESB should be able to talk to the sender via a specific protocol, but use a different protocol to forward an incoming message to the receiver.

5. **Service Mapping:** An ESB should be able to map business services to implementation services. In the example above the client calls a business service *PlaceTradeOrder*. The ESB is responsible for finding the correct implementation service for it.

6. **Message Processing:** An ESB should guarantee that received messages get delivered without loss of any message, even in the case of a system crash.

7. **Process Choreography:** An ESB should be able to coordinate multiple business services to process a request by a single service request. This is usually BPEL based. Each of the coordinated business services can also be called in a standalone fashion, but an ESB should also be able to coordinate them.

8. **Service Orchestration:** An ESB should also be able to coordinate multiple implementation services and not only business services as in Process Choreography.

9. **Transaction Management:** An ESB should provide a single unit of work for a business service request. To achieve this a framework is needed that provides the coordination of multiple resources across different services.

10. **Security:** An ESB should also provide security. As through an ESB all services are available to a whole enterprise, the ESB should protect services from unauthorized access.

When looking at an ESB from the EAI perspective it shows that it is a useful architectural concept because it bridges together the two integration approaches SOA and EDA. The ESB also offers a general communication middleware for applications which talk different protocols. It acts as a mediator between applications by transferring requests to required protocols. Combined with enterprise integration patterns an ESB is a good concept for building company wide EAI solutions.

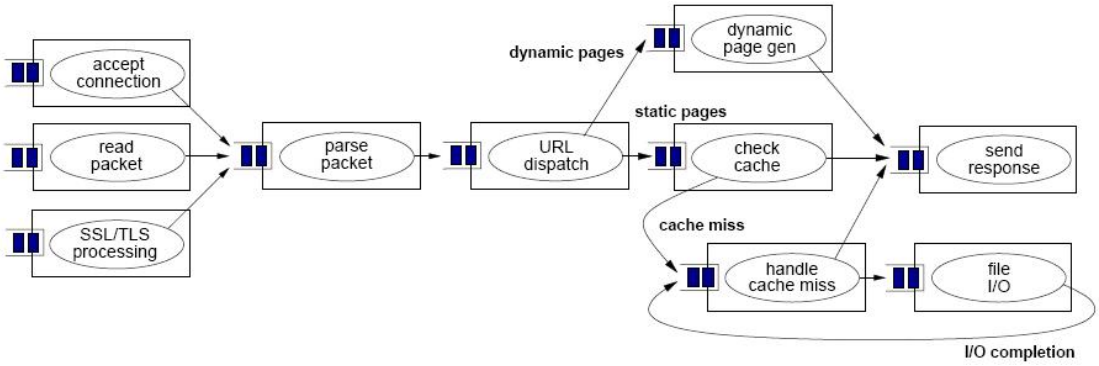

Figure 2.3: Overview of a HTTP server based on SEDA, taken out of [36]

2.7 Staged Event-Driven Architecture (SEDA)

Staged event-driven architecture (SEDA) is an architectural approach for building software systems, developed by Matthew Welsh while working on his dissertation [36]. All information given in this section are taken out of this work. SEDA is also described in [34, 35]. The title of the dissertation is „An Architecture for Highly Concurrent, Well-Conditioned Internet Services“, which already tells the goal of the SEDA approach. It provides an architectural guide for busy internet services according to massive concurrency and load conditioning demands.
In SEDA an application is divided into *stages* connected by *event queues*, resulting in a network structure. An example of a simple HTTP server developed using the SEDA approach can be seen in figure 2.3. The whole application is separated in stages and queues which connect the stages. Each stage is managed independently and can be run in sequence, parallel or in combination of both. Through the use of event queues is stage is individually load balanced.
The main aspects of the SEDA design are:

- „**efficient, event-driven concurrency:** To support massive degrees of concurrency, SEDA relies on event-driven techniques to represent multiple flows through the system. This design makes use of a small number of threads, rather than one thread per request. Nonblocking I/O primitives are used to eliminate common sources of blocking.“ [36]

- „**dynamic thread pooling:** To relax the rigid scheduling and nonblocking requirements for event driven concurrency, SEDA uses a set of thread pools, one per stage, to drive execution. This not only frees the application programmer from having to implement event scheduling (as the operating system handles scheduling

threads), but also allows event-handling code to block for brief periods of time, as additional threads can be allocated to a stage.“ [36]

- „**structured queues for code modularity and load management:** By partitioning an application into a set of stages with explicit queues between them, application designers can focus on the service logic and concurrency management for individual stages, plugging them together into a complete service later. Queues decouple the execution of each stage, allowing stages to be developed independently. Queues provide a point of control over the request stream in a service, as requests flowing across stages can be inspected and managed by the application. Likewise, admission control can be performed on a per-stage basis.“ [36]
- „**self-tuning resource management:** Rather than mandate a priori knowledge of application resource requirements and client load characteristics, SEDA makes use of feedback and control to automatically tune various resource usage parameters in the system. For example, the system determines the number of threads allocated to each stage based on perceived concurrency demands, rather than relying on a hard-coded value set by the programmer or administrator.“ [36]

2.7.1 Stages

The basic component within SEDA is a stage. A graphical description of it can be seen in figure 2.4. A stage consists of an *event handler*, an *incoming event queue* and a *thread pool*. Resource consumption, thread allocation, scheduling and admission control is managed by one or more controllers for each stage. A stage takes off a batch of incoming events from the event queue and invokes the event handler. The handler processes the events and dispatches outgoing events to event queues of other stages.
Usually a single event can be viewed independently from all other events arrived at the incoming queue. But it is also possible that more than one event belong together and should therefore be processed in common. This is the reason why the event handler cannot only take one event off the incoming queue but can also process multiple events together. This feature can increase throughput of a stage. The only function of the event handler is to process incoming events and dispatch events to outgoing queues. The event handler is not responsible and does not have any influence on threads within a stage, the input queue and other aspects of management and scheduling. This leads to a clear separation of concerns by separating the core business logic from thread management and scheduling.
Thread management is done by controllers. The amount of threads executing within a stage is based on an observation of the load and performance of a stage and is adjusted to optimal usage by the controller. SEDA offers various thread management strategies which are discussed in detail in [36].

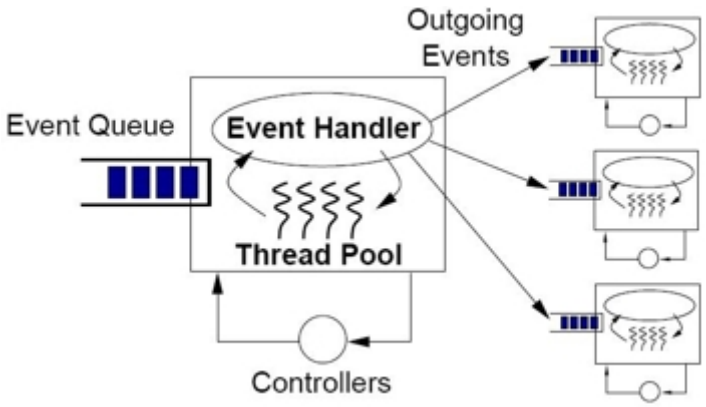

Figure 2.4: A SEDA stage, taken out of [36]

2.7.2 Event Queues

Queues are used for connecting stages in a SEDA architecture. For meeting constraints of a resource management policy queues in a SEDA architecture can reject to enqueue an event. If an event is not enqueued the queue signals an error to the stage that wanted to put the event onto it. Rejection is a sign of overloading for a stage which should lead to an adaption of services in some way.
Constructing the network of stages can be done statically or dynamically. When a static approach is used, all connections between stages are known at compile or load time. The dynamic approach allows stages to added and removed at runtime. Both approaches have advantages and disadvantages that have to weighed up against each other. A comparison of advantages and disadvantages of both approach will not be done here. This can rather be seen in [36].

2.7.3 SEDA and EAI

When looking at the explanations of SEDA one could not immediately find a connection between SEDA and EAI. SEDA is a architectural style for building software applications whereas EAI aims at integrating several software applications.
The connection is that most software systems built for implementing EAI solutions are built on the design principles of SEDA. When looking at the integration patterns, messages flow into components, which do some processing on them and afterwards dispatch messages again to one or more output queues. Thus a framework that offers an implementation of enterprise integration patterns could implement all patterns as SEDA stages and the incoming and outgoing queues as SEDA queues. In chapter 4 *Codehaus Mule* will be introduced, which is built using a SEDA approach.

3 Research Questions

3.1 Technical Research

When looking at an EAI solution one could find the following categories that can make up an integration solution. It does not mean that all parts listed below have to exist in an EAI solution, but the categories are some sort of points that should be considered while designing and developing an integration solution.

- Message Oriented Middleware (MOM)
- Enterprise Integration Pattern Implementation
- Rule Engine
- Workflow Engine
- Service Framework
- Management/Monitoring
- Web Container
- Application Container
- Data Transformation Engine
- Integrated Solution (ESB)

This list does not contain the term ESB. Instead it contains the item *Integrated Solution(ESB)*. The reason why this notation is used is the following: an explanation of an ESB through defining what capabilities it should provide was already given in section 2.6.3. There it was also noted that a system or framework does not need to comply with all capabilities, but only with those, which are needed in a specific situation to be called an ESB. When applying this thought to the list any item could be called an ESB without saying something wrong. Therefore the list contains the term *Integrated Solution(ESB)* instead. The meaning of the item is as follows: an integration of at least two of the items in the list is meant. Thus a framework that already provides a rule engine and a workflow engine can be called an *Integration Solution(ESB)* under this

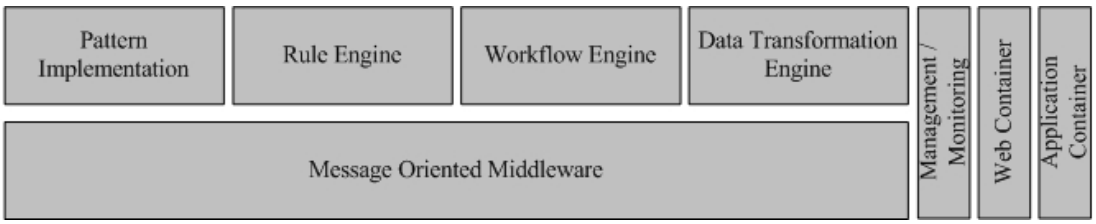

Figure 3.1: Graphical representation of categories

circumstances. This does not denote that it has to implement its own rule- or workflow engine, but uses already existing ones which are smoothly integrated into the complete framework.

A graphical representation of the list can be found in figure 3.1. The figure shows that at the lowest level some message oriented middleware has to exist that enables messaging for any kind of applications that need to participate in the integration solution. On the top of the messaging infrastructure other categories like pattern implementations, rule engines, workflow engines and data transformation engines can exist. These parts can make up an EAI solution. The adjacencies between parts do not mean that only adjacent parts can collaborate and cannot work with non-adjacent parts. All parts in that layer have to be coordinated to form a running and useful integration solution.

The categories management/monitoring, web- and application containers play a role on both levels. Management and monitoring must be done at all layers as it provides important runtime and system health information to administrators. Web- and application containers deal with runtime issues like where and how to host a solution. Thus they are needed at both layers, too. The categories services frameworks and integrated solution cannot be found in the figure. Service frameworks deal with making messaging services available for applications and can therefore be seen on the application side of an EAI solution. Another level where service frameworks can be situated is above the workflow engines where services are orchestrated to fulfill a complex business task. But again the service frameworks are situated at the client applications. What can be seen is that service frameworks play an important role for an EAI solution, but do not really fit into this graphical representation.

The figure shall provide the fundament for two research questions to be answered in this thesis. The first question is at which categories integration patterns come into play. It will be tried to find out at which parts they can be situated and implemented. In the figure there is already a category pattern implementation but patterns can be found and implemented in other parts of the figure, too. Thus a mapping from pattern categories introduced in section 2.5 to categories of an EAI solution shall be done. The second question that should be answered with the help of the figure is which frameworks can cover which categories. In the practical part a deeper look into some open source EAI

frameworks will be done. It will be tried to find out which patterns are supported and can be implemented and which parts of the figure are covered by the frameworks.
Another question that should be answered in the thesis is to find out architectural differences between the frameworks and work out best-practices for them. Furthermore it shall be investigated if there are circumstances where a specific framework should be preferred against others. Best practices shall include development issues and maintenance issues. Development issues mean how easy it is to use and implement patterns. Maintenance issues are for example the management of a running integration solution or updating running systems without stopping them.
To summarize, the technical research questions for this work are:

1. Classification and mapping of pattern categories to parts of an EAI solution
2. Support of integration patterns in open source frameworks
3. Classification and mapping of open source frameworks to parts of an EAI solution
4. Find out architectural differences and best practices for frameworks

For finding out these points a theoretical and a practical approach will be done. Theoretical research will mainly include documentation analysis to figure out which patterns and other features are said to be supported by a framework. In the practical part scenarios will be defined that cover integration patterns on the one side and various transport types on the other side. By implementing scenarios with the help of the frameworks their support for patterns and their classification shall be found out. Also best practices shall be discovered this way.

3.2 Business Research

It is always interesting for a company to know if it is beneficial for them to use open source software instead of commercial products. What shall be done in this thesis is a cost effectiveness study to figure out this point. However, the main part of thesis shall cover the technical research questions. The business part will not be the main research focus. Rather it should be a little amendment to the technical part. Thus fancy calculations can not be expected here.

4 Selected Standards & Frameworks for EAI

In this chapter some standards in the EAI and SOA domain are introduced in a nutshell. The introduction will give a basic understanding of them but not an extensive explanation. To get more information on them cited literature can be consulted. The second part of the chapter will give more information on parts of an EAI solution which were introduced in chapter 3. For each category some open source projects will be mentioned. The projects which were used during the practical part of the thesis will be explained with more details in the third part of this chapter.

4.1 Standards

This section introduces three standards that are settled in the EAI domain. The first one is Java Business Integration (JBI), the second is Service Component Architecture (SCA) and the third is Service Data Objects (SDO). SCA and SDO complement each other but each of them can also be used independent.

4.1.1 Java Business Integration

One of the most important standards in the EAI domain is Java Business Integration(JBI), elaborated by the Java community in the JSR-208 [29]. Steve Vinoski gives a good introduction to JBI [31], which acts as a reference for the introduction given in this thesis. The standard reuses preexisting specifications like WSDL and JMX and does not reinvent things that are already covered by these standards. Vinoski says in his article

> „JBI is an SOA: it treats its components as service providers and consumers.“ [31]

JBI helps enterprises to restructure their IT infrastructure towards a SOA. When taking a look at the expert group of the JBI specification one finds many of major commercial IT vendors like Novell, TIBCO, Sonic, Red Hat and many more. Thus the specification also has a good chance to get accepted by commercial vendors.

4.1.1.1 Architecture of JBI

The standard is based on the principles and approaches of web services. The architecture of JBI is very simple. It consists of a container and components that can be plugged into it. After components are deployed, they are able to communicate via the message router built into the container. The interaction of a component with other components is described via an abstract service model. The abstract model is situated above any particular transport protocol or message encoding format. To get work done the abstract model is not enough, therefore JBI also uses a concrete specification. The abstract model defines message types, abstract operations and service types or interfaces that group operations together. The concrete model specifies binding types for the protocol specification and endpoints where concrete communication details are specified. For the description of the abstract and concrete messaging models WSDL is used.
As already said above components live insided the JBI container. A component can either be a *Service Engine (SE)* or a *Binding Component (BC)*. A service inside the container is called a SE. When running outside the container, a service has to connect to the container through a BC. All communication between services is done via the *Normalized Message Router (NMR)*. A simple example that shows the difference between a service engine and a binding component is the following: if one wants to translate messages, some kind of XSLT or other transformation engine is needed. This component would be a SE as it offers the service of translating messages to other components. Theoretically the transformation engine is available for all transport protocols because all communication is done via the NMR. What has to be done to achieve this is to write binding components for the protocols one would like to use. If one wants to enable talking to the transformation engine via JMS, a binding component for JMS is needed. What is important to note is that only one BC for each protocol and not one for each SE is needed. Once a BC is implemented and deployed, all communication to any SE in the container can be done via the BC. This show the great advantage of the NMR.

4.1.1.2 The Normalized Message Router(NMR)

The NMR is responsible for carrying messages from the message producer to the message receiver, in other words between a service requestor and a service provider. Another service of the NMR is a discovery mechanism which enables consumers to find appropriate service providers. What is important to note is that the content of a message must be a well-formed XML document. Otherwise a message will be rejected by the NMR. Technically the exchange of messages happens via objects, called delivery channels, which offer methods for sending and accepting messages and for creating message exchange factories.

At least the four message exchange patterns based on the WSDL standard have to be supported by a JBI implementation.

- In-Only: A service consumer sends a message to a service provider, but the provider does not give any reply. As a consequence the provider cannot give any fault information back to the consumer, if any error occurred.
- Robust in-only: To overcome the drawback of in-only (no response in case of fault), in this exchange pattern the service provider sends a reply to the consumer if an error occurred.
- In-out: The service consumer sends a message to the service provider, which replies with a return message or a fault.
- In-optional-out: The service consumer sends a message to the service provider. The provider can response with an output message, a fault message or the provider can simply complete by setting the exchange status.

To send messages consumers and providers have to adhere to a two step process. The first step is to create a message exchange instance via the message factories. The second step is to invoke the methods send and receive on the created delivery channel. A message exchange does not only contain the content the sender put into the message but also some metadata and state information for the exchange.
Normalized in the NMR does not mean that all messages are translated to some *Canonical Data Model* but it means that the router treats the payload of a message as opaque data that is sends along to the receiver. It is obvious that the NMR has some contracts that have to be fulfilled by BCs and SEs for enabling communication via the NMR. But it does not care how components are implemented internally. Thus all components need to do, is to implement the interfaces of the NMR and are then able to exchange messages.

4.1.1.3 Management

The JBI container allows components to be plugged into it and also removed from it. Therefore it needs some mechanisms for managing the lifecycle, deployment, undeployment, starting and stopping of components. JBI make use of the JMX for this issues. The management beans provide methods for the points listed above.

4.1.1.4 Summary

JBI is a reasonable step towards standardization for EAI middleware systems. The goal of the specification is to move away from proprietary EAI systems to a vendor independent approach. With the definition of JBI a good fundament has been laid to

bring a standardization to integration solutions. In the open source community some JBI based projects are already available. Also some commercial products offer a JBI implementation (Oracle Fusion Middleware, TIBCO ActiveMatrix). The future will show if JBI will be accepted as the standard for EAI. Like with nearly all standards, the major vendors will decide if they support the standard or not, which will implicitly show if the standard will be successful.
JBI developers have done a good job while working on the specification. The reuse of WSDL and JMX is a good choice since they are already well-known and established standards. JBI provides a good base for implementing vendor independent EAI solutions. It will also be important to provide good development support for building JBI components and applications that use the components. There is already a collection of components available on the internet which can be downloaded and deployed into a JBI container.

4.1.2 Service Component Architecture & Service Data Objects

Service Component Architecture (SCA) and Service Data Objects (SDO) are two specifications that are settled in the SOA domain. SCA aims at the composition of services whereas SDO specifies a common data format for all services. At this point a brief introduction to both specifications is given. Furthermore Apache Tuscany[1], a framework that implements the specifications, is described shortly. An introduction to this area is also given by Alexander Schatten in [24] which serves as a reference for this introduction. Both specifications are supported by a number of major vendors, like BEA, IBM, Tibco, SAP and some more which gives them a good chance for a broad acceptance in the SOA domain.

4.1.2.1 Service Component Architecture (SCA)

David Chappel introduces SCA with these words:

> „What is an application? One way to think of it is as a set of software components working together. All of these components might be built using the same technology, or they might use different technologies. They might run inside the same operating system process, in different processes on the same machine, or across two or more connected machines. However an application is organized, two things are required: a way to create components and a mechanism for describing how those components work together.
>
> Service Component Architecture (SCA) defines a general approach to doing both of these things. Now owned by OASIS, SCA was originally created by a group of vendors, including BEA, IBM, Oracle, SAP, and others. The

[1] http://incubator.apache.org/tuscany/

> SCA specifications define how to create components and how to combine those components into complete applications. The components in an SCA application might be built with Java or other languages using SCA-defined programming models, or they might be built using other technologies, such as the Business Process Execution Language (BPEL) or the Spring Framework. Whatever component technology is used, SCA defines a common assembly mechanism to specify how those components are combined into applications.“ [5]

This short introduction gives the main characteristics of SCA. It mainly aims at the composition of services which are built platform independent and with different programming languages or frameworks. Thus it is possible that one component is a BPEL process and another is a Java application. With the help of SCA it can be specified how these two components can work together to provide a higher level business service or a complete application. The specification of components and the composition of them is defined in XML configuration files which have to be created.

4.1.2.2 Service Data Objects (SDO)

In SOA environment different services built on different platforms have to cooperate and exchange information. The Service Data Objects (SDO) specification is targeted at defining a data model description upon any platform. Thus a model can be found for the whole enterprise and be reused in a Java- as well as in a C++ application. SDO is called a *disconnected* model which means that it is only used for carrying data but is not connected to any data source. To achieve a connection *Data Access Services (DAS)* can be used, which is another specification. Other characteristics of SDO are that it is possible to define constraints and integrity conditions which can be validated. It is furthermore possible to embed a history to all data elements for reproducing which components made changes to which data.

4.1.2.3 Apache Tuscany

A framework that implements SCA, SDO and DAS is Apache Tuscany. It is currently in the incubator status of the Apache community. All three specifications are not dependent on each other and can therefore be used individually or together. Tuscany can help developers to either use any specification alone or in combination. It offers SCA support for Java, C++, Ruby and Phyton. For SDO, Tuscany offers support for Java, C++ and PHP developers, for DAS only Java and C++ are supported. It has to be said that not all components are ready to use but still have work in progress status. Although Tuscany is still under incubation status it is worth a look when implementing service oriented applications.

4.1.3 JBI versus SCA & SDO

On the ServiceMix and on the Open SOA project websites comparisons between JBI and SCA are provided [28] [8]. Either comparison comes to the conclusion that both standards are aimed at developing SOA applications, but are no direct competitors. SCAs target users are application developers whereas the target audience for the JBI standard are integration developers. Therefore the standards are no competitors but a combination of both standards within a company IT infrastructure might be useful but is not necessary.
SCA aims at developing applications from the ground up to the end with a fully SOA compatible model whereas JBI aims at integrating existing application into one solution and building a SOA infrastructure with the help of the integration approach.
However it is possible to use SCA as an integration technology for applications. When applications expose their functionality via services, they can be composed by SCA. Furthermore a data model defined with the help of the SDO specification can be used when integrating applications.
Although EAI can be done with SCA and SDO it will not be covered and discussed anymore. The approach for integration taken with SCA is a composition of services and components. The main focus of this thesis is a messaging approach for integration where components exchange messages and SCA could be one component in the complete integration solution. Also Tuscany will not be used for implementing scenarios as it is still in incubator status.

4.2 Frameworks

This section takes a closer look to the categories listed in chapter 3. For each category open source projects will be named. Frameworks that were used for implementing the practical part of the thesis will be discussed in more detail, whereas others are just named. The list of categories from chapter 3 is repeated here to bring it into mind again:

- Message Oriented Middleware (MOM)
- Enterprise Integration Pattern Implementation
- Rule Engine
- Workflow Engine
- Service Framework
- Management/Monitoring

- Web Container
- Application Container
- Data Transformation Engine
- Integrated Solution (ESB)

4.2.1 Message Oriented Middleware (MOM)

Messages need to be sent between different applications in an IT environment of a company. A message oriented middleware provides the necessary infrastructure for enabling messaging between applications. Thus when designing an EAI solution where applications must exchange messages a MOM is a very important part of the solution. A crucial requirement to a MOM is that in no case a message can get lost while traveling from the sender through the messaging infrastructure to the receiver. When a message is passed to the MOM it must be guaranteed that the message will be delivered even if the system crashes while delivering or some other failure happens. A message must even survive a restart of the MOM. When looking at the integration patterns this represents the *Guaranteed Delivery* pattern. A messaging infrastructure that does not fulfill this requirement should not be used in any productive environment, as unexpected errors due the loss of messages could happen.
Another requirement when thinking of a MOM is the ability to talk to different platforms. By this, technology platforms like Java, .NET or any host applications on the one side and operating systems like Windows, Unix or Linux derivates on the other side are meant. If a MOM only runs and works on one platform it will not find many supporters as the IT infrastructure in an enterprise usually is very heterogeneous and not build on one technology platform only. For fulfilling this requirement a MOM must be able to support many different protocols. For example an incoming JMS message is forwarded as an XMPP message. A MOM must therefore provide transformation and translation capabilities. When looking at the integration patterns, this requirement corresponds to the *Message Translator* pattern.
A deeper look at this pattern shows that the *Message Translator* addresses four levels of transformation. The first layer deals with data structures where structural mappings take place. The second layer in engaged with data types, where data type conversations are done. The third layer is responsible for data representation. Here conversion between different data formats take place, e.g. convert an XML format to an ASCII format. The forth layer covers transport type translation. This layer converts messages between different protocols, e.g. a JMS message to a HTTP message.
The question that arises now is if a MOM needs capabilities to support all four levels of the *Message Translator* pattern or only some of them? For sure it would be nice if a MOM could handle all levels, but this would already be too much functionality

for a MOM. It is enough for a messaging infrastructure to concentrate on its main functionality - messaging and not making any changes to incoming messages except converting them to other protocols. Thus the content transformation of a message should take place somewhere else but not in a MOM. For this purpose other frameworks exist that are specialized on offering such functionalities. But of course it is important that a MOM and other frameworks work together smoothly for providing more levels of transformation.
The third requirement for a MOM is scalability. This is a very important requirement, because without any support for this, it is in fact not usable in any productive environment. When more clients and applications are added to a messaging environment it must be ensured that functionalities are available which can handle the additional message load. Therefore a messaging infrastructure needs some capabilities for achieving scalability.
Another requirement which is related to scalability is a fail over mechanism. What happens if a MOM crashes for any reason? A MOM which is designed for productive use should have some mechanism included that avoids that users are not able to continue their work if the MOM crashes. For example this could be achieved by having more than one messaging brokers running, which are configured to take over the load of the counterpart if it crashes.
The last requirement for a MOM is security. Security does not only mean that messages can be encrypted. The more important aspect of security when dealing with a MOM is that access to specific queues can be denied or permitted for applications. As queues are publicly available to all systems there must be some mechanisms the exclude some systems from specific queues or topics.
The list of requirements provided here is not complete but only covers basic capabilities for a MOM. For specific projects more requirements can be found and it must be ensured that the used MOM covers all of them.
Various message brokers can be found in the open source community. Only a few of them here will be named here:

- Apache ActiveMQ[2]
- JBoss Messaging[3]
- OpenJMS[4]
- Proteus[5]

[2] http://activemq.apache.org
[3] http://labs.jboss.com/jbossmessaging
[4] http://openjms.sourceforge.net
[5] http://info-scape.com/proteus

A statement like „message broker A is the best and should be preferred“ cannot be done and would be incorrect. For finding a suitable MOM for an EAI solution a list of requirements has to be made and then the requirements have to be matched to the features of the available message brokers. After that a decision can be made for each individual project. For the implementation of the practical part, the *Apache ActiveMQ* message broker, so it will be described in more detail in section 4.3.

4.2.2 Enterprise Integration Pattern Implementation

Designing a good architecture for a software system is a difficult challenge. A good design must address the current requirements but should also be easily extendable for future requirements. A lot of patterns have been developed over the time to support finding accurate architectures for software systems. As it is already difficult to find an architecture for one software system only, it is even more difficult to find an architecture that integrates applications to solve new business requirements. To achieve this, one has to have knowledge about internals of all systems to figure out how they can be integrated. The next step after knowing how to connect applications, is finding a good solution for the message flow between them. To solve this problem patterns have been collected and documented which can be applied and reused in all architecture designs [14].
Thus when implementing an EAI solution patterns should be applied and play an important role in the design phase. As already stated in section 2.5 there are six categories of integration patterns. When thinking in a naive way, one could easily say that a pattern implementation framework should implement all patterns. But this is not necessary as some patterns are already addressed by a MOM, like the *Guaranteed Delivery* pattern or other categories of an EAI solution. Furthermore some patterns are only design issues where no implementation is needed.
When looking for frameworks that specialize on providing only pattern support not many are found, because in most cases routing or transformation capabilities are already a built-in functionality of a framework, like they are in Mule[6]. But there is one framework that specializes on providing a pattern implementation. This framework is Apache Camel[7], which is a subproject of ActiveMQ. Since release 5.0 of ActiveMQ, Camel is fully integrated into the message broker, so when one already gets a pattern implementation delivered with a MOM. More information on Camel will be given in section 4.3.
Another framework that provides mediation capabilities is Apache Synapse[8]. The difference between Camel and Synapse is that the latter is focused on web services environment, especially Axis2. Camel is designed to work with much more transport types

[6] http://mule.mulesource.org
[7] http://activemq.apache.org/camel
[8] http://synapse.apache.org/

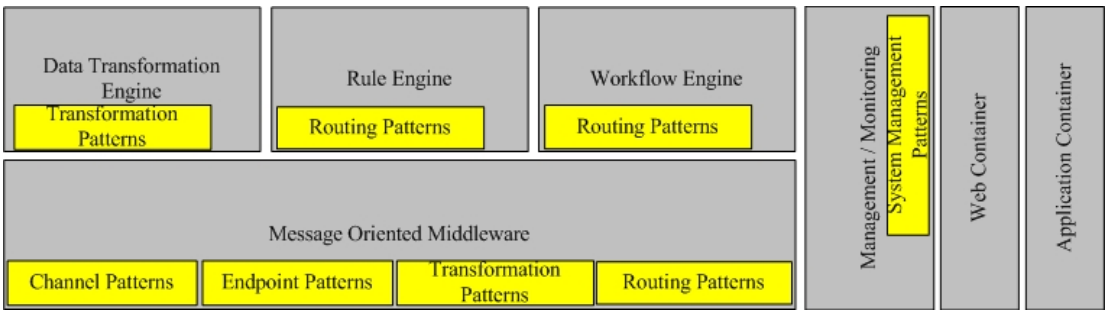

Figure 4.1: Classification of integration patterns to categories

and data formats.
One more alternative to Camel could be a rule engine to make routing decisions. Of course this is not the main goal of a rule engine as it should be used for expressing business rules, but they can be established to provide support for some routing patterns. Patterns do not only play a role in the pattern implementation category but can be found at all categories of an EAI solution, too. This is shown in figure 4.1 where a mapping from integration patterns to EAI categories is done. The item „Pattern Implementation“ was taken out of this figure and split up into the individual pattern categories. Then they were assigned to all other categories in the figure. When looking at the latest release of ActiveMQ patterns can already be found at the MOM level. Rule engines can also provide routing capabilities. Transformation engines can be used for implementing and using transformation patterns. Workflow engines can also support routing capabilities when messages have to be transfered between all participating parties of a workflow. System management patterns can be found in management tools where for example one can inspect the contents of a queue or topic.

4.2.3 Rule Engine

Rule engines are used for expressing business rules. The target users for such systems are not application developers, but domain experts. They can express their rules without any knowledge of programming language syntax or programming at all. Usually the expression of rules is more domain oriented to help domain experts do write their rules. The complexity of rules can vary from very simple to very complex rules. An argument against rule engines is that a software developer could just get the knowledge of the domain experts and implement business rules in the source code of an application. The drawback when hardcoding rules is that whenever rules change the application has to be changed, too. This means that the source code of has to be changed, recompiled, retested, reinstalled. This is not needed when using a rule engine to express business rules. Here only rules are changed and the rest of the software is not touched by anyone.

Thus when using a rule engine a lot of maintenance work for a software can be saved. This even gets more profitable if rules have to be adjusted very often.
What was said until now is targeted at one software system only. What is the relation of rule engines and EAI solutions? When using a rule engine in an EAI solution, it can for example be used for routing messages between endpoints. So a rule engine can provide support for some *Message Routing* patterns and are therefore very useful in EAI solutions. To achieve this, a rule engine must have the ability to interpret at least some properties of messages and apply them on rules to make routing decisions.
Another point where rule engines are useful in EAI solutions is that common business rules that need to be carried out by more than one application can be transferred to a rule engine and then exist only once in the enterprise. All applications can connect to the engine and execute common rules. When changes are necessary there is only one place where they must be carried out. No applications need to be changed as they all rely on the rule engine.
Another requirement for a rule engine is that there should be some visual editor for specifying rules. Essentially for non-technician users some visual support can be very useful.
Many rule engines can be found in the open source community. A list of some rule engines for Java was published by Java-Source.net [15].
Apache ServiceMix and Codehaus Mule offer support for Jboss Drools rule engine that can be used while building EAI solutions.

4.2.4 Workflow Engine

In a SOA services are published to a registry and can be used by other applications. An ideal scenario would be that new applications are built by reusing existing services only. When looking at an insurance company such services could be *create customer* or *rate customer*. When a customer wants to buy an insurance, the employee will have predefined steps from the creation of the customer until selling the insurance. This is called the workflow. If all tasks are provided as services the workflow can be build by using available services. For the execution an engine has to be used that can initialize and execute the process of selling an insurance. So a workflow engine can be an important part of an EAI solution as EAI targets to reuse existing services in a company IT infrastructure. There are several workflow engines available on the open source market. Just a few of them are listed here.

- ActiveBPEL[9]
- Apache ODE[10]

[9] http://www.active-endpoints.com
[10] http://ode.apache.org

- Enhydra Shark[11]
- JBoss jBPM[12]

A well known standard for workflows in the domain of web services is Business Process Execution Language(BPEL), which is an XML based standard and is used to orchestrate web services into one workflow. Another standard is Business Process Management(BPM). BPEL offers more functionality than BPM but is also more complex to use. A little comparison of both standards can be found in [22].

4.2.5 Service Framework

Services frameworks are the tool for bringing web services into an EAI solution. This can either be to call web services or to make existing functionalities of applications available through the use of web service technology. This means that a service that has been accessible via sending a JMS message to it can be made available via calling a web service. This is a very important part in an EAI solution as access to many old host systems has been made available trough writing a web service that operates on the host system but can be invoked via a web service call. Because web services are based on standards, interoperability should theoretically be no problem when working with web services. However in section 2.6.1.1 this problem was already described. In the open source community two frameworks have become the major players in the web service domain. These are:

- Apache CXF[13] - CXF has is the result of the merge of Celtix and XFire
- Apache Axis[14]

4.2.6 Management/Monitoring

One requirement that is important for all parts of an EAI solution is management and monitoring of all running components. Without such capabilities debugging a system and fixing errors is nearly impossible because one can never find out the real source of a failure. Instead of that only guesses and a trial and error approach must be used for finding and fixing errors. This is not a desirable way to correct errors because typically bugfixing is less time consuming when the source of an error is known instead of just guessing and verifying where the error could have occurred. As a software system is never free of errors this part becomes very important. Errors have to be fixed in very short time to get an integration running again in as less time as possible. Good

[11] http://www.enhydra.org/workflow/shark/index.html
[12] http://www.jboss.com/products/jbpm
[13] http://incubator.apache.org/cxf/
[14] http://ws.apache.org/axis2/

management and monitoring of an EAI solution can help to find and correct errors in short time.
Furthermore management also means that components can be started, stop, deployed and undeployed by the management tool. Thus some support for this should be available by all used frameworks in an EAI solution.

4.2.7 Web and Application Containers

Usually an enterprise has already some web- or application containers running in their IT infrastructure. For an EAI solution it is important that it can also connect to applications running in these containers. Another point that could be of interest is, if the developed integration solution can also run and be hosted in such a container. This way costs can be saved because the hardware would already exist in the company and an administrator for the container is also already available.
An example for a web container could be a Tomcat[15] server, examples for application containers are typical J2EE containers like Glassfish[16] or Geronimo[17] application server.

4.2.8 Data Transformation Engine

Data transformation is a very important requirement in every integration solution. Content and message translation has to be accomplished in nearly every EAI solution. This can either be simple translations of some fields, like a date field where a different date format is needed or changes in the ordering of fields in a message. One could even make protocol transformations with such an engine if both protocols are text based.
Usually XSLT will be used to achieve transformation as most exchanged messages contain XML content. Every framework that calls itself ESB comes with a built-in XSLT engine so that no other engine has to run in the integration solution. Some engines that are commonly used within frameworks are Xalan[18] or Saxon[19].

4.2.9 Integrated Solution (ESB)

When looking back one section at a data transformation engine, a simple engine running somewhere in the IT landscape could already be called an ESB as it fulfills the requirements message transformation, message enhancement and also protocol transformation. The Camel framework mentioned above can also be called an ESB as the requirements routing, message transformation, message enhancement and also protocol transformation are fulfilled. Thus already four out of ten requirements are supported by

[15] http://tomcat.apache.org
[16] https://glassfish.dev.java.net/
[17] http://geronimo.apache.org
[18] http://xalan.apache.org
[19] http://saxon.sourceforge.net

Camel. When looking further at ActiveMQ, message processing, security and transactions are also met. So already seven out of ten requirements are fulfilled by ActiveMQ. This already sounds like it is a pretty powerful ESB.
The meaning of this part of an EAI solution was already given in 3.1. What is understood under *Integrated Solution (ESB)* is a system that already integrates two or more parts that are described above. The advantage of the integration is that only one framework or container is needed that handles all communication between the different parts.
During development of the scenarios two open source frameworks that call themselves an ESB were used, which are:

- Codehaus Mule[20]
- Apache ServiceMix[21]

These two frameworks will be described in more detail in section 4.3. A very extensive comparison of the two frameworks is done by Rademakers and Dirksen in their book „Open-Source ESBs in Action“ [22]. There are also other other open source frameworks available, like the JBoss ESB[22] or OpenESB[23].
Mule and ServiceMix are very different frameworks. ServiceMix is an implementation of the Java Business Integration standard whereas Mule describes itself as an integration platform based on ideas from ESB architectures.
Like it was already stated in the MOM section, no framework or project can be announced as the best framework that fits for all possible needs. All requirements for a specific integration project have to be found out first and after that they can be matched with the features of each framework. After that a decision can be made which product is the best for the specific project. For a different project any other product could be better than the one decided for the first project.

4.3 More Details on Used Frameworks

In this section more technical details to the frameworks that have been used for developing scenarios for this thesis are given. The frameworks are *Apache ActiveMQ*, *Apache Camel*, *Apache ServiceMix* and *Codehaus Mule*. The main driver for the development of the first three projects is the company IONA[24]. They provide special releases of the projects for which they offer commercial support. New features of the projects are first

[20] http://mule.mulesource.org
[21] http://servicemix.apache.org
[22] http://labs.jboss.com/jbossesb
[23] https://open-esb.dev.java.net
[24] http://www.iona.com

implemented in their own releases and later passed to the Apache community. Mule is mainly developed by the company MuleSource which also offers a commercial release of the framework for which they offer support.

4.3.1 Apache ActiveMQ

When looking back at the requirements for a MOM, the first requirement was *Guaranteed Delivery*. ActiveMQ uses a message store for fulfilling this requirement. Furthermore ActiveMQ can not only enable messaging for Java clients via JMS but can also communicate with clients written in C, C++, C#, PHP, Phyton and many more. Additionally various transport protocols like VM, TCP, UDP, SSL, HTTP and also higher level protocols like REST, XMPP are offered. So the second and third requirement stated above are also fulfilled. The forth requirement was scalability. ActiveMQ comes with a couple of features that target this requirement: there can be more than one broker running and if one of them crashes or goes down the others will take over the message load until the broker starts working again. To achieve, this a fail over protocol can be configured at each broker. If more message brokers are running they can form a network of brokers that are connected to achieve scalability. Security features are also provided through different providers like JAAS or XML configuration files. Authorization can be done on queue or topic level, but also on message level. It is also possible to create a custom security provider and plug it into ActiveMQ via interceptors. For manageability and administration a web console is offered where queues and topics can be inspected.
ActiveMQ offers far more features for message producers and consumers than listed in this short introduction. To find out more about all features of it, visit the project website.
When mapping ActiveMQ to the figure of the EAI categories, it would cover the MOM and management/monitoring categories of the figure. As since the latest release Camel is also integrated into the message broker it could also cover the pattern implementation and data transformation categories.

4.3.2 Apache Camel

Apache Camel is an framework for implementing enterprise integration patterns defined by Hohpe and Woolf in [14]. To use the patterns, they 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 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 come from or can be written to. This means that one can easily read from a file in periodic intervals

with Camel or write messages to a file. One can also write to databases or to HTTP endpoints. For a full list of available transport components, see the project website. Furthermore Camel provide a built-in XSLT engine for transforming messages.
Camel can be used as routing and mediation engine in Apache ActiveMQ, Apache ServiceMix, Apache CXF and Apache Mina[25]. In the practical part of the thesis Camel is used in combination with ActiveMQ and ServiceMix. Camel does not pronounce itself as an ESB, rather it calls itself an integration pattern provider for the above mentioned frameworks. But when taking a look back to the definition and requirements of an ESB (see section 2.6.3), Camel can be called an ESB, too.
Using integration patterns in Camel is pretty easy and straightforward. All one has to do is to create a *CamelContext* in the Java, add all pattern rules to the context and finally start it by calling the *start* method. Alternatively, rules can be defined in XML. Since 5.0 release 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 makes them a good player in the EAI field. However when mixing ActiveMQ configuration code and Camel pattern rules keeping an overview of what parts belong to which framework might get harder as when using the separated.
What is worth noticing, when writing rules is that they are not executed each time a message is received, but only when the context is started. After that they are initialized and running. For example, it is important to know this when working with default initializations and String concetanations in Java where one wants to add a customized suffix to a Java String for each message. For example a String concetanation is shown in listing 4.1. In the example the receiving queue should have a defined prefix and a customized suffix for each message. However the concetanation is only done when the context is started and at this time the value for the suffix is evaluated and set. For all incoming messages the concetanations will not be done anymore. Listing 4.2 shows how the intended behavior of using a dynamic route can be achieved in Camel. The endpoint is evaluated and set as a header property of a message and afterwards referenced as the recipient of a message.
More examples of Camel routes will be shown in chapter 5 where scenarios are introduced and their implementation is shown.

Listing 4.1: Non working example for dynamic route

```
String destinationCity="";
from("activemq:queue:examplequeue").process(
   new Processor () {
      public void process(Exchange exchange) {
         Message in = exchange.getIn();
```

[25] http://mina.apache.org

```
         String city;

         //evalute value for city here
         //destinationCity is a variable defined outside
         //the route, so it can be used outside, too
         destinationCity = city;
   }})
.to("activemq:queue:examplequeue." + destinationCity);
```

Listing 4.2: Example of working example for dynamic route

```
from("activemq:queue:examplequeue")
.setHeader("endpointName",
   //prefix of queue name, dynamic part
   //will be appended by expression
   constant("activemq:receivequeue.").append(
      new Expression() {
         public String evaluate(Exchange exchange) {
         // value of name will be returned for endpoint name
         String name = "failure";
         Message in = exchange.getIn();
         /*
         * logic for evaluating variable name out of
         * variable in comes here
         */
         return name.toLowerCase();
      }
   }))
.recipientList(header("endpointName"));
```

4.3.3 Apache ServiceMix

Apache ServiceMix is an implementation of the JBI standard. As it is based on an open standard it should prevent from vendor lock-in. The first sentences on the website of ServiceMix are:

> „ServiceMix is lightweight and easily embeddable, has integrated Spring support and can be run at the edge of the network (inside a client or server), as a standalone ESB provider or as a service within another ESB. You can use ServiceMix in Java SE or a Java EE application server.
>
> ...
>
> ServiceMix uses ActiveMQ to provide remoting, clustering, reliability and distributed fail over.“

Figure 4.2 shows a high level architecture of ServiceMix. At the bottom of the figure some binding components of ServiceMix can be seen. At the top some service engines

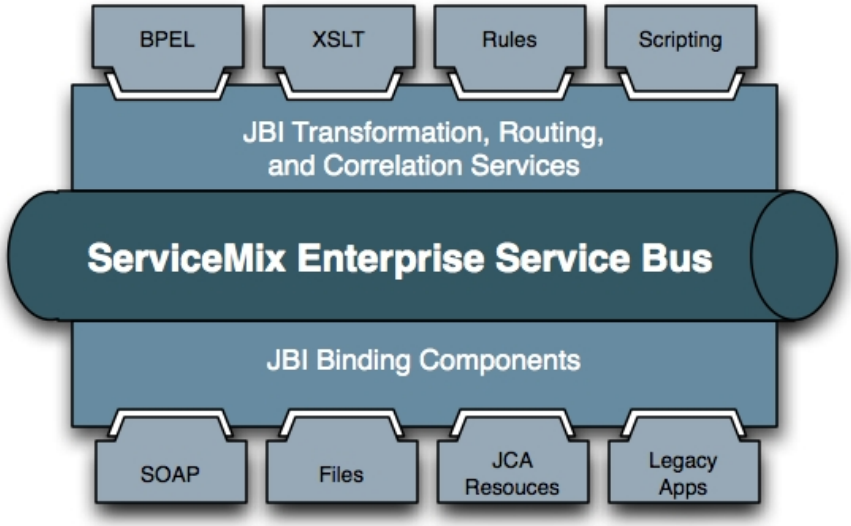

Figure 4.2: Architecture of ServiceMix taken from http://servicemix.apache.org

are displayed. In the middle of the two types of components the normalized message router is situated which handles all communication within the service bus.
ServiceMix already gets delivered with a lot of components that can immediately be used, like a Drools rule engine, JMS-, HTTP-, FTP-, file transport and more components. Also a Camel JBI component is delivered with ServiceMix. What is missing when taking a first look at the components, is a JDBC component to work with databases. A workaround for the missing JDBC component is to use the Drools component to connect to databases. There are also many JBI components available in the Java developer community[26] which can be deployed into ServiceMix.
An artifact than can be deployed to the ServiceMix container is called an *Service Assembly*. The assembly can contain several *Service Units*, which actually build the logic for the integration solution. A service unit is a part of the business logic that is needed for the complete integration solution. A solution could be sending JMS messages and using Camel for routing messages to the correct endpoint. For achieving this, one would create a service unit for sending and receiving JMS messages and one service unit for Camel that handles the routing logic. The two units are then packed into a service assembly and deployed to the ServiceMix container. While deploying, ServiceMix will take the JMS service unit and deploy it to its JMS component whereas the Camel service unit will be deployed to the Camel component. Thus the programming model for ServiceMix is to write individual service units which form the business logic for an integration solution. These are packed into a service assembly that will be deployed

[26] https://open-jbi-components.dev.java.net/

into the ServiceMix container.
For integration developers, the development of service units and service assemblies is fully integrated into Maven[27] where ServiceMix already provides archetypes for creating the needed artifacts.
ServiceMix can not only be run in a standalone mode, but can also be deployed into application- and web servers, like Geronimo, JBoss or a Tomcat server.
As ServiceMix is a JBI implementation management and administration can be done via JMX.
To summarize, the goal of ServiceMix is not to provide integration patterns, but to offer a container which hosts components and integration solutions. Of course a part of a solution can be patterns that are implemented but the aim of ServiceMix is to provide a higher level and standard architecture for EAI solutions.

4.3.4 Codehaus Mule

The first sentences in the description section of the Mule website are:

> „Mule is a messaging platform based on ideas from Enterprise Service Bus (ESB) architectures. An ESB works by acting as a sort of transit system for carrying data between applications within or outside your intranet.
>
> ...
>
> Mule, in fact, goes beyond the typical definition of an ESB. We prefer to define Mule as a light-weight messaging framework that contains a distributable object broker for managing communication between applications. The point of the object broker is to manage service components.“

Figure 4.3 shows what Mule intends to be. In the middle there are components (in Mule they are called UMO (Universal Message Object)). UMOs can receive and send messages via various transport types. Between UMOs and the transport types are a *Lifecycle Adapter* and a *Transformation Layer*. As the name suggest, the *Lifecycler Adapter* is responsible for managing the lifecycle of a UMO component. The *Transformation Layer* handles the transformation of messages before they are received or after they are sent by an UMO component. An UMO can be anything (JavaBean, component from another framework, etc.). It is basically the code that provide the business logic for the integration solution. There are no restrictions on an UMO except that it must provide a default constructor and of course it must be configured by Mule.
Mule uses an object broker which is responsible for managing communication and all components. It is built around the SEDA design pattern which was introduced in 2.7. Schumann et al. give an introduction on SEDA and how Mule is build around the SEDA architecture [27].

[27] maven.apache.org

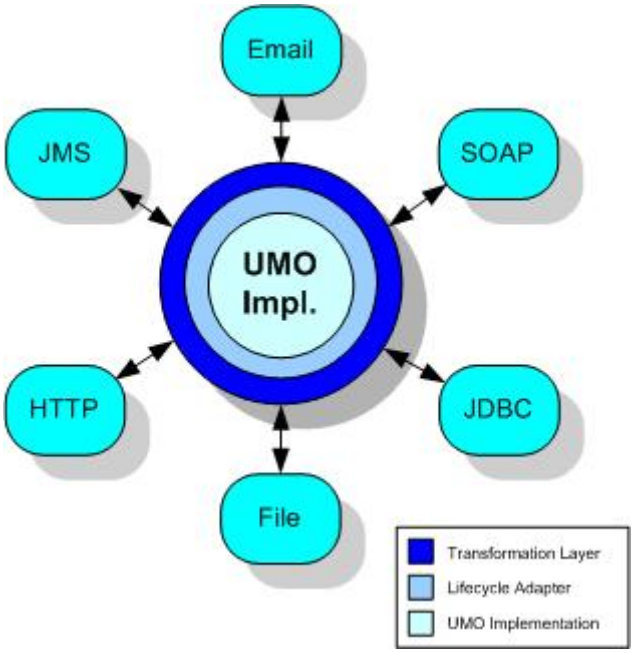

Figure 4.3: Mule UMO component, taken from http://mule.mulesource.org

There are several ways how Mule can run:

- Starting Mule from a script: Start scripts are available in the mule distribution directory. This is the easiest way to test integration solutions developed with Mule.
- Running Mule as a Windows Service or a Unix Daemon: This is the mode that should be used for running Mule in a production environment.
- Embedding Mule in a Java application: Mule can easily be embedded into a Java application so that it automatically starts when the application is started.
- Embedding Mule in a Web application: Like in a Java application Mule can also be embedded into a web application.

Mule offers support for many different transport protocols. Just a few examples are email, file, FTP, JMS and many more. Additionally transformers are offered for many protocols to easily convert the content of a message to a different format. If no transformer is available for a specific use case, implementing a custom transformer is not a big deal. Transformers in Mule represent most patterns that can be found in the *Message Translation* (see section 2.5.5) category. For all UMO components defined in

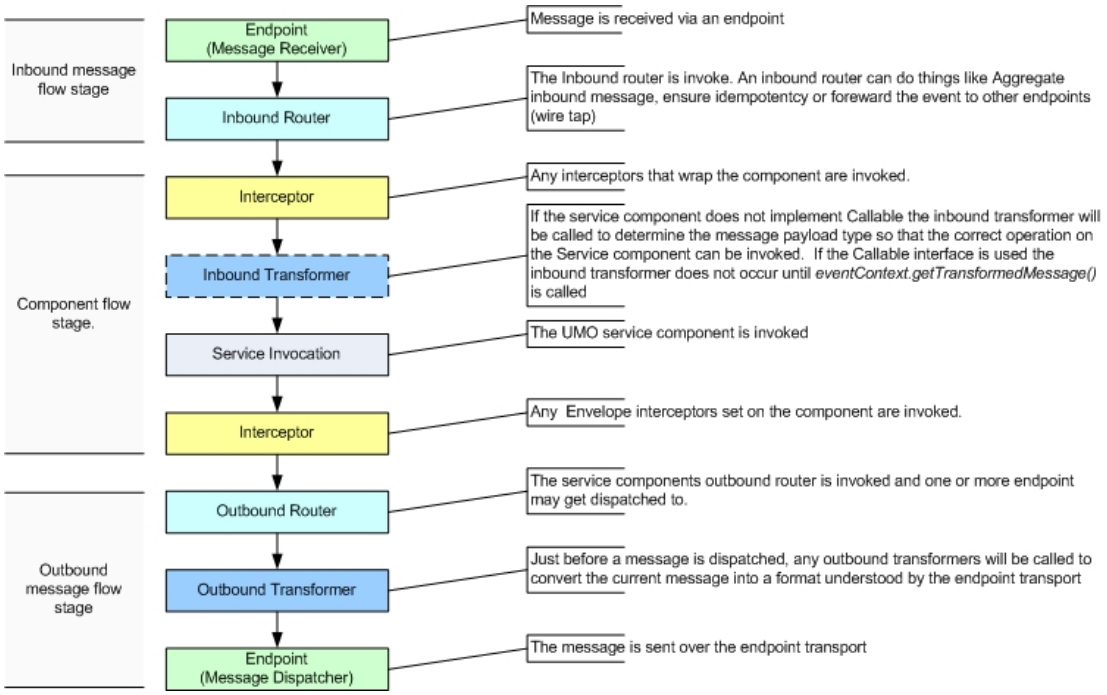

Figure 4.4: Mule message flow, taken from http://mule.mulesource.org

Mule inbound and outbound routers are configured which represent most of the *Message Routing* patterns (see section 2.5.4). Examples for routers will be described in more detail in chapter 5. The complete message flow to and from an UMO component is shown in figure 4.4.

A common field of application for Mule is shown in figure 4.5. Several applications need to be integrated into one EAI solution. Messages sent by applications will be received by Mule. Afterwards transformation, enrichment, filtering, etc. may be done done by UMO components within the Mule instance. At the end the final message will be sent to another external application. What can also be seen in the figure is that several Mule instances can communicate over network boundaries. This can be useful when introducing hierarchies of Mule instances, where each instance is responsible for enabling communication for some application groups only and the communication between groups is done with the help of an additional Mule instance.

The configuration of Mule is done in XML via a Mule or a Spring configuration file. This means when Spring is already used, all Mule elements can be configured through Spring, too. The usage also leads to the advantage that Mule objects can leverage Spring's AOP, transaction interceptors, DAO support, etc.

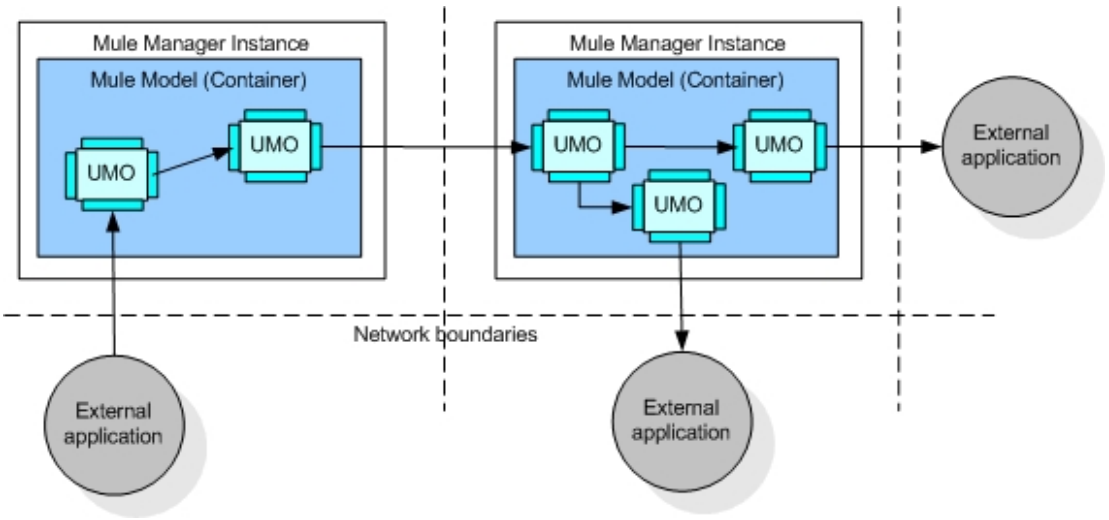

Figure 4.5: Mule, common field of application, taken from http://mule.mulesource.org

While working on this thesis the Mule community released a governance platform with an integrated registry and repository, called Mule Galaxy. It provides management capabilities for all artifacts, including lifecycle management, policy management, activity monitoring and some more. For this thesis Galaxy is out of scope and will not be considered anymore, but it is definitely worth looking at when designing and building an EAI solution with Mule, as such a powerful management component cannot be found very often in the open source domain.

5 Scenarios

This chapter will introduce scenarios that have been implemented using some of the before mentioned frameworks. The goal is to show how patterns like routing, transformation, aggregation, etc. can be applied and implemented. In addition it will be seen if it is easy and straightforward or exhausting to use patterns with the given frameworks. The scenarios are settled in different application domains. These are airport-, airline-, train- and hospital information systems. The domains were chosen because examples can be adjusted in both directions, either make them more simple or more complex until they fit into the target goal for a scenario. All scenarios are imaginary but could be found with little adjustments in the mentioned domains.
Each scenario has not been implemented with every framework, but all frameworks have been used extensively to find out qualities and attributes of them so that conclusions can be drawn.
For each scenario there will be one subsection that describes the situation, one that shows the modeling with enterprise integration patterns and modeling alternatives and one that gives interesting implementation details. A final subsection will discuss the findings of the scenario. The visual modeling will use a graphical notation that is also used by Hohpe and Woolf in [14].
The scenarios cover almost all routing and transformation patterns and some patterns in the other categories. Channel patterns like *Point-to-Point Channel*, *Publish-Subscribe Channel*, *Guaranteed Delivery* are nearly used in every scenario. Construction and endpoint already come with the used messaging protocol and do not need much support by the frameworks. 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 release versions of the used frameworks are Apache ActiveMQ 5.0, Apache Camel 1.2, Apache ServiceMix 3.2.1 and Codehaus Mule 1.4.3, which are all the latest stable releases at the time of implementing the scenarios for the thesis.

5.1 Communication between Air Control Agency (ACA) and Airline Information System

The first scenario shall show how to use a workflow engine in combination with Mule and ServiceMix. This represents the *Process Manager* pattern. Furthermore *Request-Reply*

is used as communication style.

5.1.1 Description

Before a flight can take place an airline must pass the flight plan to the ACA. The agency can approve or deny a submitted plan. The approval process includes checking details of the flight plan. The first check is if the airline is allowed to travel to countries which are given in the plan. The second step checks if the announced date and time of a flight are still available or already allocated by other flights. In the last check the flight route will be inspected.
If all checks are positive, the flight plan will be acknowledged, if not it will be denied.

5.1.2 Modeling

The approval process is an ideal candidate for modeling and executing it in a workflow engine. There are two options how it could be modeled. The first approach is to use a sequential process, where all steps are executed sequentially. An other options would be to use parallel execution of all process steps. When the steps do not rely on each other the parallel approach should be favored as all process steps can be executed individually. This leads to a faster execution of a process. However if there are dependencies between the steps of the process, they should be carried out in sequence. This process uses a shared object that is passed to each step for saving the outcome of a check. Therefore the sequential approach was preferred for this scenario.
Communication between the ACA system and the airline information system is handled in a *Request-Reply* fashion.

5.1.3 Implementation

Implementation support of the *Process Manager* pattern is available by Mule and ServiceMix. Mule offers an integrated jBPM process engine that can be utilized for building workflows. Furthermore a BPEL engine is also available for Mule. For ServiceMix an Apache Ode JBI component is available which can be deployed into the ServiceMix container and run BPEL processes. As a process- or workflow engine is not available for Camel the scenario was not implemented with it.

5.1.3.1 Mule

A graphical representation of the implementation can be seen in figure 5.1. For implementing the scenario with Mule, the jBPM process engine was used. An introduction to jBPM will not be given here as this would go beyond the scope of this work. Documentation and demonstration can be found online on the project website[1]. At this

[1] http://labs.jboss.com/jbossjbpm

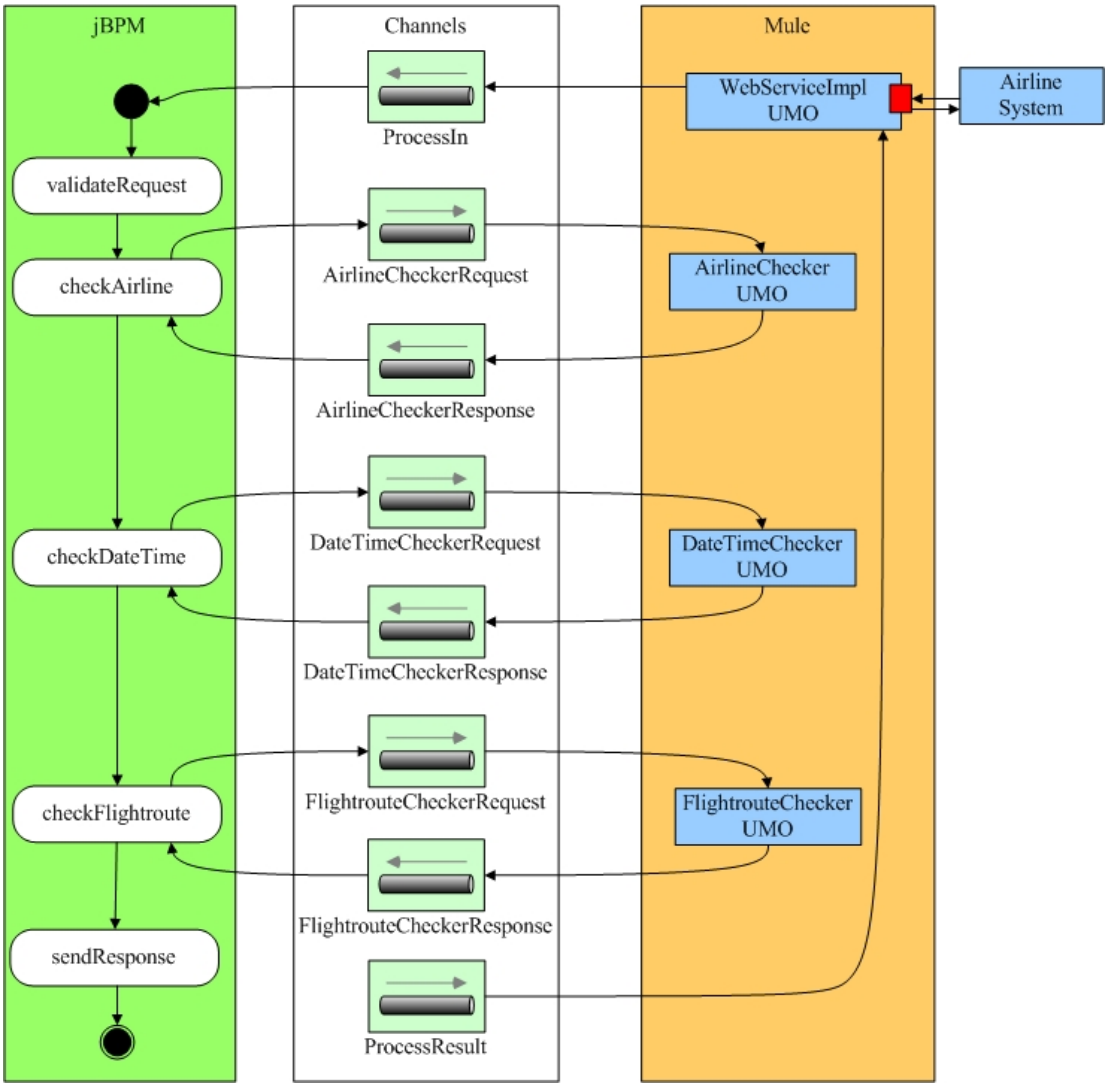

Figure 5.1: Cooperation between Mule and jBPM engine

point, only the integration of Mule and jBPM will be shown. The first thing needed when working with jBPM is to configure a connector and all necessary settings. The definition of a connector is shown in listing 5.1. At the definition a process id is set for identifying all running processes. The last property refers to a spring bean that configures all jBPM settings, like the name of a process configuration file, database settings and some more. Due the limited space, this bean will not be shown here. An example for it can be found in the Mule online documentation. At the bottom of the listing an endpoint configuration is shown. The name which is used in the address property maps to the name of a defined process. What has to be configured after declaring connectors and endpoints is a message exchange between Mule and the jBPM engine. To achieve this a UMO component has to be configured once for passing messages to the process and once for receiving message from the process and pass them along to the Mule environment. Listing 5.2 shows how messages can be passed to a process engine. The endpoint for the engine is configured as outbound-router for the UMO component. All sources for messages sent to the engine are configured as endpoints for the inbound-router. The UMO component needs methods for handling all incoming messages. The methods can be simple „pass-through“ methods where nothing is done on an incoming messages or can perform additional operations on a message. The same UMO also handles all outgoing messages from the process engine, which is shown in listing 5.3. Now the jBPM endpoint is configured as inbound-router and all receivers are configured as outbound routers. The last step that has to be achieved to enable communication is to send messages from the jBPM engine. For this an jBPM action is supplied by Mule, which can be seen in listing 5.4. To ensure that an incoming message to the process engine comes from the correct endpoint Mule also offers an action for validating the message source. This is shown in listing 5.5.

Listing 5.1: jBPM connector in Mule

```
<!-- jBPM connector -->
<connector name="jBpmConnector" className="org.mule.providers.bpm.
    ProcessConnector">
   <properties>
      <!-- This field in LoanQuoteRequest holds the unique process id. -->
      <property name="processIdField" value="requestId" />
      <!-- jBpm itself is configured by a series of Spring beans at the end
          of this file. -->
      <spring-property name="bpms">
         <ref local="jbpm" />
      </spring-property>
   </properties>
</connector>
...
<global-endpoints>
   <endpoint name="ProcessEngine" address="bpm://AnnounceFlight" type="
```

```
       senderAndReceiver"/>
</global-endpoints>
```

Listing 5.2: Passing messages to a jBPM engine

```
<!-- Messages sent to the process engine -->
<mule-descriptor name="ToBPMS" implementation="eip.reqrep.mule.bpm.
    StartProcess">
   <inbound-router>
      <global-endpoint name="ProcessIn"/>

      <global-endpoint name="AirlineCheckerResponse" />
      <global-endpoint name="DateTimeCheckerResponse" />
      <global-endpoint name="RouteCheckerResponse" />
   </inbound-router>
   <outbound-router>
      <router className="org.mule.routing.outbound.FilteringOutboundRouter">
         <global-endpoint name="ProcessEngine" synchronous="false" />
      </router>
   </outbound-router>
</mule-descriptor>
```

Listing 5.3: Receiving messages from a jBPM engine

```
<!-- Messages generated from the process engine -->
<mule-descriptor name="FromBPMS" implementation="eip.reqrep.mule.bpm.
    StartProcess">
   <inbound-router>
      <global-endpoint name="ProcessEngine" />
   </inbound-router>
   <outbound-router>
      <router className="org.mule.routing.outbound.EndpointSelector">
         <global-endpoint name="ProcessResult"/>
         <global-endpoint name="AirlineCheckerRequest" />
         <global-endpoint name="DateTimeCheckerRequest" />
         <global-endpoint name="RouteCheckerRequest" />
      </router>
   </outbound-router>
</mule-descriptor>
```

Listing 5.4: Sending messages from a jBPM engine

```
<action class="org.mule.providers.bpm.jbpm.actions.SendMuleEvent">
   <payloadSource>flightplanRequest</payloadSource>
   <endpoint>ProcessResult</endpoint>
   <synchronous>true</synchronous>
</action>
```

Listing 5.5: Validator for message source in jBPM

```
<action class="org.mule.providers.bpm.jbpm.actions.ValidateMessageSource">
   <expectedSource>ProcessIn</expectedSource>
</action>
```

The listings in this section show the environment for integrating a jBPM process into Mule. What is not shown here is the process definition as this is specific to jBPM and is not scope of this work. For developing process definitions a graphical eclipse[2] plugin is available that helps building a process definition.
For achieving a request-reply behavior a web service was used. A Mule component is exposed as a web service which receives requests that trigger the process engine. Listing 5.6 shows how to configure an UMO component to make it available as a web service. To pass the result back to the caller, a simple return-statement is used by the UMO which implements the service.

Listing 5.6: Expose UMO as web service

```
<!-- expose UMO component as webservice -->
<mule-descriptor name="submitFlightPlanService"
   inboundEndpoint="axis:http://localhost:81/services"
   implementation="eip.reqrep.mule.service.SubmitServiceImpl">
</mule-descriptor>
```

5.1.3.2 ServiceMix

As already mentioned, an Apache Ode BPEL engine is available for ServiceMix. After downloading from the project website, the JBI component can be deployed to the ServiceMix container.
For deploying BPEL processes a couple of steps have to be done. All relevant process file have to be placed in a service-unit folder. Furthermore an Ode deployment descriptor has to be placed in the same folder. After that all contents must be zipped to an archive and put into a service-assembly folder. The service-assembly folder has to contain a JBI description file. The last step is to zip the service-assembly folder and put it into the deploy directory of ServiceMix. After that the process can be run inside ServiceMix.

5.1.4 Discussion

This scenario showed how to use a *Process Manager* pattern with a jBPM process engine in Mule. Furthermore *Request-Reply* was used for communicating the result of a process back to the invoker. The implementation was uncomplex with Mule as it already delivers all required components which only have to be configured correctly and enough online documentation is provided.

[2] http://www.eclipse.org

For ServiceMix only a guide for building processes was provided due the lack of BPEL knowledge and experience of the author. The deployment of the Ode engine to the used version of ServiceMix also caused some problems because of some classloading issues. To deploy the engine to ServiceMix, the JBI service assembly of Ode has to be changed and enhanced a bit to resolve the problems. A guide for making the correct changes can be found in the ServiceMix mailing lists. After that, the deployment does not produce any errors and BPEL processes can be deployed into it.

5.2 Communication withing Railway Information Systems

This scenario is taken from a railway domain. It shows simple splitting and routing of messages between systems.

5.2.1 Description

A train makes its way from the departure to the destination railway station. While traveling it can stop at several stations. When the train departs, the departing railway station is in possession of the travel plan. The plan includes the identifier of the train, the time of departure and the travel route. The route consists of all stops and the destination of a train with additional time information for each stop. This information has to be sent to all stations where the train stops, so that they can prepare the arrival of the train like working the switches to the right rails or displaying arrival information on displays. But each station shall only get the information which is necessary for the specific station and not the complete travel information of the train. So the message has to be split and then the divided parts have to be sent to the appropriate stations.

5.2.2 Modeling

Modeling and message flow of the scenario can be seen in figure 5.2. The yellow part of the message represents the header information that has to be passed to all stations. The white parts of the message represents the information for one stop of a train. A recipient list forwards the message to a message router and to a claim check component. The router is responsible for sending the message to the correct destination station according to the content of the message. At the claim Check component the header of a message is filtered out. After that the message passes a splitter that splits the message into parts which represent one stop of the train. Subsequently the header of the message that was cut out before is inserted again. The final step is a content-based router that passes message to the correct final system according to the station name of the stop. The main reason why the claim check pattern is used in this scenario is that if the header would be in the message content when it passes the splitter, the first split message would contain the header, whereas the second message would not contain it. But as all

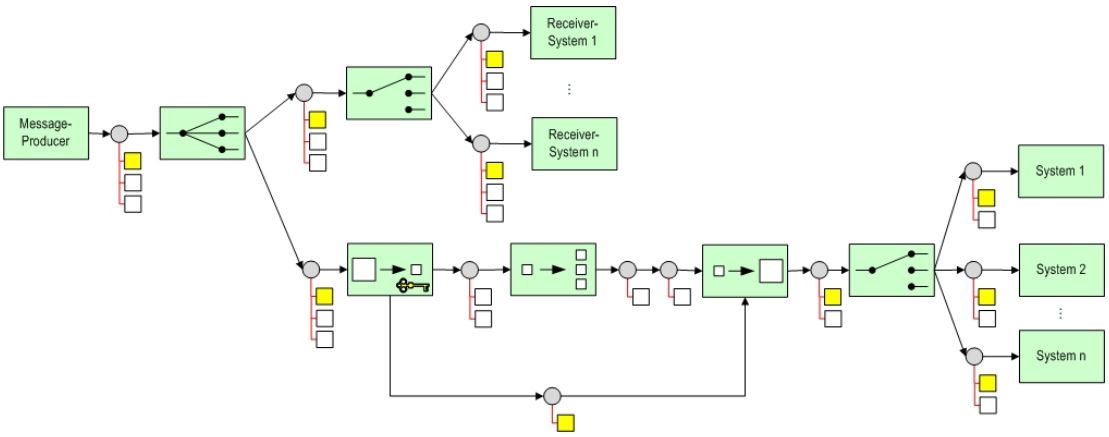

Figure 5.2: Trainsplitter message flow

messages need to have the correct header information the claim check was used. An alternative to this could be to use a message translator that translates the content of a message in a format where the header information is inserted before each stop. After that a splitter can be used to split the message into parts for each stop.
The message that is encoded in XML format. A sample message can be seen in listing 5.7.

Listing 5.7: Example of a train schedule plan

```
<?xml version="1.0" standalone="yes"?>
<trainSchedule>
   <trainId>IC-4574</trainId>
   <departure>
      <station id='ViennaSouth' name='ViennaSouth'></station>
      <date>31.12.2008</date>
      <time>0943</time>
   </departure>
   <destination>
      <station id='NeusiedlSee' name='NeusiedlSee'></station>
      <date>31.12.2008</date>
      <time>1030</time>
   </destination>
   <stops>
      <station id='Gramatneusiedl' name='Gramatneusiedl'>
         <arrival>
            <date>31.12.2008</date>
            <time>0958</time>
         </arrival>
         <departure>
            <date>31.12.2008</date>
```

```
            <time>1000</time>
         </departure>
      </station>

      <!-- insert more stations here -->

   </stops>
</trainSchedule>
```

For this scenario not many mentionable alternatives regarding to patterns exist. One alternative could be to replace the recipient list with a publish-subscribe channel. But this could lead to security issues as anyone can connect to the channel and listen for messages on it. Instead of using content-based routers one could use a recipient list or a publish-subscribe channel and filter out messages that are not addressed to the receiver. But this would lead to a higher message load that are sent to a channel and then filtered out which increases unnecessary traffic. A router is definitely the better choice in the given situation.

5.2.3 Implementation

The scenario was implemented with Camel and ServiceMix as splitting and dynamic routing can be achieved with them very easily. For Mule splitting logic is also not complex but requires more XML configuration. Splitting and dynamic routing in Mule is shown in a subsequent scenario and therefore this scenario was not implemented with Mule.

5.2.3.1 Camel

The scenario has been implemented with Camel and ActiveMQ as JMS message broker. The interesting parts of the scenario are the support for the claim check and the dynamic content-based router. Camel offers a *Processor* class that can operate on the content of a message. With the help of that class the header of the message can be stored and inserted again later. Listing 5.8 shows how the text of the header is stored into a variable. In listing 5.9 the same method is used to paste the content to the message again.

Listing 5.8: Example Processor for Camel

```
from("activemq:queue:camel.train.plan.queue")
.process( new Processor()  {
   public void process(Exchange exchange) {
      Message in = exchange.getIn();
      // the variable preamble is defined outside the rule
      String text = (String)in.getBody();
      //from the value of text the header can be evaluated
      preamble = ... ; //evaluate here
```

```
    }
})
```

Listing 5.9: Content enricher in Camel

```
from("activemq:queue:camel.train.plan.queue.preambleenricher")
.process(new Processor () {
   public void process(Exchange exchange) {
      Message in = exchange.getIn();
      in.setBody(preamble + in.getBody(String.class) + "\n</trainSchedule>")
          ;
   }
}).to("activemq:queue:camel.train.plan.queue.router");
```

For setting the destination of a message dynamically, the *Expression* class can be used. The method *evaluate* can operate on the content of a message and return the required parts which represent the destination. Afterwards this is set as a header property of the message and then used for routing the message to the correct destination.

Listing 5.10: Dynamic content-based router for Camel

```
from("activemq:queue:camel.train.plan.queue.router")
.setHeader("endpointName",
   constant("activemq:queue:camel.train.plan.queue.").append(
      new Expression() {
         public String evaluate(Exchange exchange) {
            Message in = exchange.getIn();
            String text = (String)in.getBody();
            String value;

            //evaluate content for variable value here
            return value;
         }
      }
   )
)
.recipientList(header("endpointName"));
```

5.2.3.2 ServiceMix

This scenario can be implemented with ServiceMix in two possible variants. The first one is to simply reuse all Camel rules which are already shown in the previous subsection. All that has to be done to achieve this, is to configure ActiveMQ in the *CamelContext* service-unit file. This can be seen in listing 5.11.
A more ServiceMix-like approach is to configure service-unit which receive messages from and send messages to JMS queues. A consumer which receives messages from a JMS queue and a provider which sends message to a JMS queue are shown in listing

5.12. The *service* and *endpoint* attributes are used for identifying a service by the container. The attribute *targetService* indicates the name of the component where an incoming request should be routed to. *destinationName* is the name of the JMS queue where message shall be received from. The JMS provider contains an attribute *destinationChooser* which implements the logic for choosing the correct destination queue. This bean has to be implemented as it contains custom logic. It must implement the *DestinationChooser* interface of ServiceMix. The implementation is shown in listing 5.13. The *chooseDestination* method has access to the message and can evaluate the value for the destination queue out of it. This way a dynamic router can be implemented in ServiceMix.

Listing 5.11: Enabling JMS for the Camel JBI component

```
<bean id="activemq" class="org.apache.camel.component.jms.JmsComponent">
   <property name="connectionFactory">
      <bean class="org.apache.activemq.ActiveMQConnectionFactory">
         <property name="brokerURL" value="tcp://localhost:61616"/>
      </bean>
   </property>
</bean>
```

Listing 5.12: JMS consumer and provider in ServiceMix

```
<jms:consumer service="splitter:receiveTravelplan"
   endpoint="endpoint"
   targetService="splitter:dynamicSender"
   targetEndpoint="endpoint"
   destinationName="camel.train.plan.queue"
   connectionFactory="#connectionFactory" />

<jms:provider service="splitter:dynamicSender"
   endpoint="endpoint"
   destinationChooser="#myDestinationChooser"
   connectionFactory="#connectionFactory" />

<bean id="connectionFactory" class="org.apache.activemq.
    ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616" />
</bean>

<bean id="myDestinationChooser" class="eip.splitter.servicemix.
    MyDestinationChooser" />
```

Listing 5.13: DestinationChooser in ServiceMix

```
public class MyDestinationChooser implements DestinationChooser {

   public Object chooseDestination(MessageExchange exchange, Object message)
          {
```

```
      NormalizedMessage nMessage = (NormalizedMessage)message;
      SourceTransformer sourceTransformer = new SourceTransformer();
      try       {

         String message = sourceTransformer.contentToString(nMessage);
         String destination ="";

         //evaluate content of variable destination out of variable message
             here

         return destination;
      }
      catch (Exception e) {
         e.printStackTrace();
      }
      return null;
   }
}
```

5.2.4 Discussion

This scenario showed how simple splitting and dynamic routing can be achieved with Camel and ServiceMix. Splitting was done with simple XPath statements in Camel and ServiceMix. For a dynamic router, Camel offers the *Expression* class which can inspect the content of a message and return a value according to it. In ServiceMix all Camel configurations can immediately be reused and deployed. Another way for achieving dynamic routing in ServiceMix is to implement the *DestinationChooser* interface of ServiceMix and configure it on JMS endpoints where routing shall be done.
The implementation of the scenario in Camel and ServiceMix was straightforward for both when it is known which techniques of the frameworks to use for implementing a dynamic router. Especially the *DestinationChooser* interface in ServiceMix could provide more documentation from the project team. However when one knows about it, it can be used very easily.

5.3 Communication between Air Control Agency (ACA) and other Airport Systems

5.3.1 Description

An airplane transmits status information about its flight in regular intervals to the ACA. The message contains information like the flight number, current position, destination, estimated time to land, flight route etc. The status information is inserted into the

information system of the agency and shared with other systems at the airport. The agency system will create a status message for each airplane and publish the message for systems which are interested in the status information of an airplane.
An airport information system can listen for messages to display the estimated time to land on their information displays in terminals or other information sources (internet, sms service, etc.). Before messages are received, a content filter is established to filter out details of the message which are not relevant for the airport information system. This way only the flight number, city of departure and estimated time to land remain in the message, which is enough information for displaying on info screens.
Another system connected to the channel is the ground service system. This system is only interested in status messages of airplanes which will arrive in near future for coordinating all ground services for the arriving airplane. The system will generate an *unload* message and will send it to other systems which are responsible for unloading a plane. This can differ according to the type of flight. The type can be a passenger-, cargo-, combination of cargo- and passenger-, private- or an extraordinary flight (e.g. visiting of a state president). According to the type of flight, appropriate systems have to be informed by the ground service system for operating the unloading of a plane
Potential other systems which can also be interested in status information messages are taxi and bus systems which want to get information on arriving flights or a SMS gateway system which will send SMS messages to people who have registered for a specific flight.

5.3.2 Modeling

The modeling of the message flow of the scenario can be seen in figure 5.3. The agency uses a publish-subscribe channel where all status messages are published and all interested systems can connect to. For the ground service, taxi and bus systems only flights which will arrive in near future are of interest. Therefore a message filter drops out unnecessary messages. Additionally a content filter is applied to filter out content of the message which is not interesting for specific systems (e.g. the flight route might not be interesting for a taxi system). A content filter is also used for filtering out message content before the airport information system receives a message.
When the ground service system receives a message it creates an unload message and has to send it to the correct target system which is responsible for unloading a plane. To achieve this a content-based router is used for delivering messages to the correct system. The router should also configure a route to an invalid message channel in case that a message cannot be delivered.
For all published messages, a message expiration should be used to sort out old messages automatically. By using this, it can be ensured that no outdated flight status information is received by any system.
An alternative to the publish-subscribe channel for status messages could be to use two

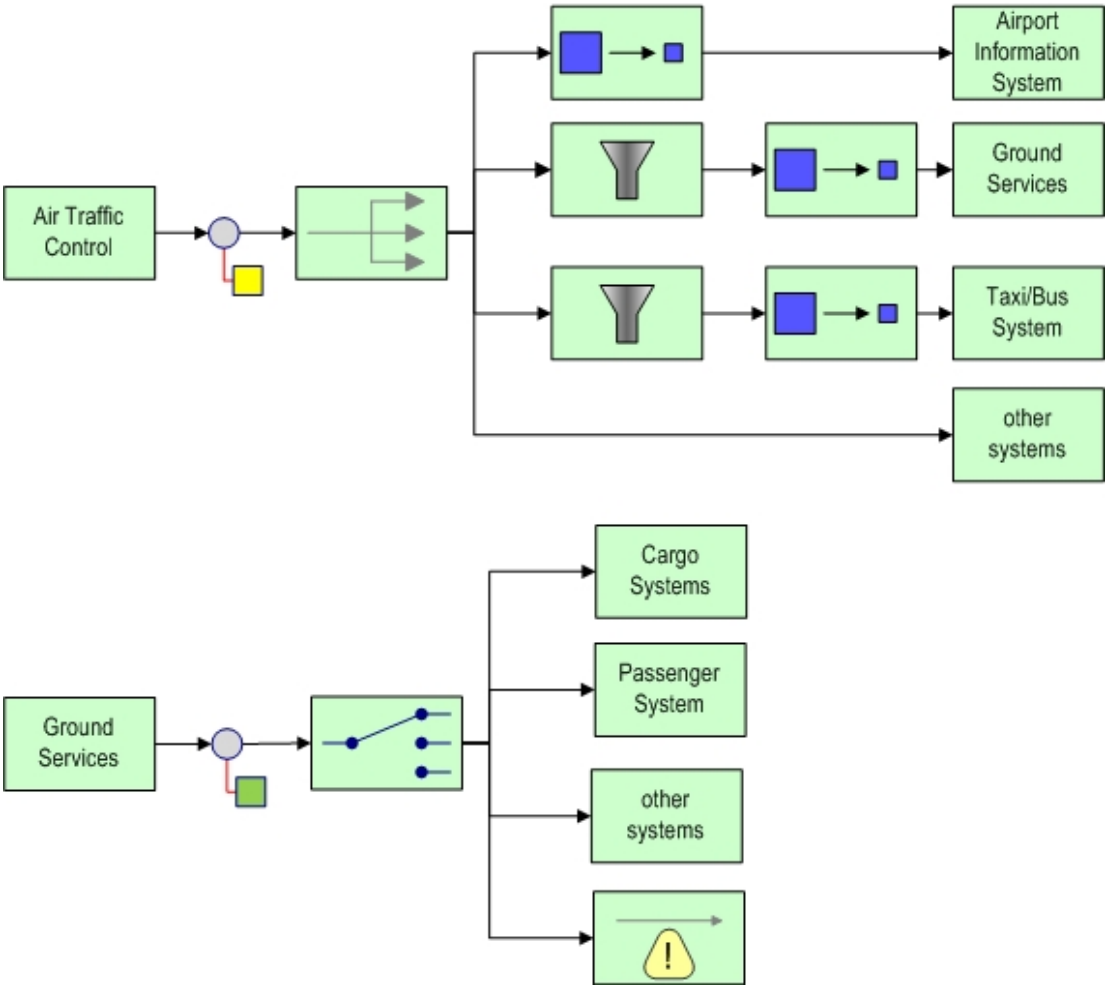

Figure 5.3: Communication between systems on airport

different channels. One point-to-point channel for the airport information system which publishes status information from all airplanes. A second publish-subscribe channel can be used for publishing information of arriving flights, to which interested systems can connect to. This approach can also improve security because a taxi system should not see the same information on flights like an airport information systems can see. In the first approach where all systems are connected to one publish-subscribe channel all systems can have access to the same messages.
As a second alternative, a recipient list could be used to forward messages to interested systems. The advantage of using a publish-subscribe channel is extensibility. No changes to the messaging infrastructure have to be made if additional systems want to receive status messages. Whereas when using a recipient list, new systems have to be added to the list manually.
Instead of a content-based router a publish-subscribe channel can be used for publishing unload messages. Each participating system can be implemented as a selective consumer so that only messages which are interesting for the specific system are received. Instead of a selective consumer a message filter could also be used for filtering out messages.

5.3.3 Implementation

The scenario has been implemented with Camel, ServiceMix and Mule. For the implementation of this scenario not only patterns were of interest but also different transport types for messages, like SMTP, XMPP or file. However only Mule was able to write messages to all three of them. Camel and ServiceMix lack in support for XMPP and ServiceMix does not even support SMTP in the used version. Therefore all messages were written to files in ServiceMix.

5.3.3.1 Mule

The implementation of the scenario in Mule was focused on the message filter and different transport types. A message filter in Mule can be configured at the inbound-router for a component. An example is shown in listing 5.14. A class which implements the logic for the filter has to be specified. The component has to implement the *UMOFilter* interface. It provides a method *accept* which has to return *true* if a message should be accepted and *false* if it should be filtered out.
For achieving content-based routing, the ground control systems sets a property *type* of a message, which is used for making routing decisions. If the type is set to „cargo“ a message is sent to the cargo endpoint which is a XMPP message. If the type is set to „passenger“ an email is sent to inform the passenger system. The type can also be set to „mixed“ which means that a message is sent to the cargo and the passenger endpoint. The last possibility for the type is „other“ which writes the contents of a message to a file. Listing 5.15 shows the implementation of a content-based router in Mule. What is

important for the type „mixed“ is the flag *matchAll* which is set to true. The flag means that for a message all routers are checked. If it would be set to false the execution would stop at the first matching router. The *filter* is used to make content-based decisions for routing. What can also be seen in the listing is the usage of *transformers* which act as a message translator for protocols. In the example a *StringToEmailMessage* transformer is used. Transformers have to be configured in the configuration file of Mule. The used transformers for this scenario can be seen in listing 5.16. One can notice that except of one, all used transformers are already delivered with Mule and can immediately be used. Finally listing 5.17 shows the configuration of an XMPP endpoint in Mule. For every used endpoint an appropriate connector has to be defined in Mule. Listing 5.18 shows the configuration of a JMS and a XMPP connector in Mule.

Listing 5.14: Message filter in Mule

```
<inbound-router>
   <global-endpoint name="statusTopic"/>
      <router className="org.mule.routing.inbound.SelectiveConsumer">
         <filter className="eip.status.mule.transformer.GroundControlFilter"
             />
      </router>
</inbound-router>
```

Listing 5.15: Content-based Router in Mule

```
<outbound-router matchAll="true">
   <router className="org.mule.routing.outbound.FilteringOutboundRouter">
      <global-endpoint name="cargoQueue"/>
      <filter expression="type=cargo"
         className="org.mule.routing.filters.MessagePropertyFilter"/>
   </router>

   <router className="org.mule.routing.outbound.FilteringOutboundRouter">
      <!-- passenger endpoint -->
      <endpoint address="smtp://da_eip_test%40gmx.at:datest2007@mail.gmx.net
          ?address=${email.toAddress}"
         transformers="StringToEmailMessage" >
         <properties>
            <property name="fromAddress" value="${email.fromAddress}"/>
            <property name="subject" value="${email.subject}"/>
          </properties>
      </endpoint>
      <filter expression="type=passenger"
         className="org.mule.routing.filters.MessagePropertyFilter"/>
   </router>

   <router className="org.mule.routing.outbound.FilteringOutboundRouter">
   <!--passenger endpoint-->
      <endpoint address="smtp://da_eip_test%40gmx.at:datest2007@mail.gmx.net
          ?address=${email.toAddress}"
```

```
          transformers="StringToEmailMessage" >
         <properties>
            <property name="fromAddress" value="${email.fromAddress}"/>
            <property name="subject" value="${email.subject}"/>
         </properties>
      </endpoint>
      <filter expression="type=mixed"
         className="org.mule.routing.filters.MessagePropertyFilter"/>
   </router>

   <router className="org.mule.routing.outbound.FilteringOutboundRouter">
      <global-endpoint name="cargoQueue"/>
      <filter expression="type=mixed"
         className="org.mule.routing.filters.MessagePropertyFilter"/>
   </router>

   <router className="org.mule.routing.outbound.FilteringOutboundRouter">
   <!-- default system listening for files -->
      <endpoint address="file://./output">
         <properties>
            <property name="outputPattern" value="OutputGCC_File.txt" />
            <property name="outputAppend" value="true"/>
         </properties>
      </endpoint>
      <filter expression="type=other"
         className="org.mule.routing.filters.MessagePropertyFilter"/>
   </router>
</outbound-router>
```

Listing 5.16: Transformers in Mule

```
<transformers>
   <transformer name="ContentFilter" className="eip.status.mule.transformer.
       MessageContentFilter" returnClass="java.lang.String" />
   <transformer name="JabberTransformer" className="org.mule.providers.xmpp.
       transformers.ObjectToXmppPacket"/>
   <transformer name="JmsTransformer" className="org.mule.providers.jms.
       transformers.JMSMessageToObject"/>
   <transformer name="StringToEmailMessage" className="org.mule.providers.
       email.transformers.StringToEmailMessage"  returnClass="javax.mail.
       Message" />
</transformers>
```

Listing 5.17: XMPP endpoint in Mule

```
<endpoint address="xmpp://rthullner:datest@jabber.org:5222/
    robertthullner@jabber.eu" transformers="JmsTransformer␣JabberTransformer"
     connector="jabberConnector"/>
```

Listing 5.18: Connector definitions in Mule

```
<connector name="jmsConnector" className="org.mule.providers.jms.
    JmsConnector">
   <properties>
      <property name="specification" value="1.1"/>
      <property name="connectionFactoryJndiName" value="ConnectionFactory"/>
      <property name="jndiInitialFactory" value="org.apache.activemq.jndi.
          ActiveMQInitialContextFactory"/>
      <map name="connectionFactoryProperties">
         <property name="brokerURL" value="tcp://localhost:61616"/>
      </map>
   </properties>
</connector>

<connector name="jabberConnector" className="org.mule.providers.xmpp.
    XmppConnector"/>
```

5.3.3.2 Camel

The implementation of this scenario requires a message filter and a content-based router. A filter in Camel is shown in listing 5.19. The *filter* component in a rule filters out all message where the specified predicate evaluates to *false*. The *Processor* class of Camel was used to evaluate the filter criteria and setting a property to the message which can be used for filtering.
The second part of the scenario in Camel is a straightforward usage of a content-based router. An example for that will be shown in one of the following scenarios. What is shown in listing 5.20 is the configuration of a mail component in Camel. After the configuration the mail component can be used in Camel routes (see listing 5.21).

Listing 5.19: Message filter in Camel

```
from("statusInfo:topic:camel.flight.status.topic")
.process( new Processor () {
   public void process(Exchange exchange) throws Exception {
      boolean accept;
      //evaluate value for accept here
      exchange.getIn().setHeader("accept", accept);
   }
})
.filter(header("accept").isEqualTo(true))
.to("statusInfo:queue:camel.flight.status.topic.filtered");
```

Listing 5.20: Configuration of mail component in Camel

```
MailComponent mail = new MailComponent();
context.addComponent("mail", mail);
```

```
final MailEndpoint ep = (MailEndpoint) context.getEndpoint("mail://44078249
    @mail.gmx.net:25?password=datest2007");
ep.getConfiguration().setProtocol("smtp");
ep.getConfiguration().setFrom("da_eip_test@gmx.at");
ep.getConfiguration().setHost("mail.gmx.net");
ep.getConfiguration().setPassword("datest2007");
ep.getConfiguration().setPort(25);
ep.getConfiguration().setUsername("44078249");
ep.getConfiguration().setDestination("rthullner@gmx.at");
```

Listing 5.21: Usage of mail component in Camel

```
from("activemq:queue:camel.flight.unload.queue")
.setHeader("subject", constant("Unload_Instruction"))
.to(ep)
```

5.3.3.3 ServiceMix

ServiceMix does not offer a message filter pattern in its EIP component. Therefore the filter of Camel was reused in a Camel JBI component. For routing messages, the content-based router of the ServiceMix EIP component was used, which can be seen in listing 5.22. The listing also shows how to use a default destination which maps to the invalid message channel pattern. ServiceMix did not offer support for writing messages to SMTP or XMPP, so all message were simply written to files which are placed in different subfolders to show that routing worked.

Listing 5.22: Content-based router in ServiceMix

```
<eip:content-based-router service="status:routerFromGCC" endpoint="endpoint"
    >
   <eip:rules>
      <eip:routing-rule>
         <eip:predicate>
            <eip:xpath-predicate xpath="/unloadInstruction/transporttype='
                cargo'" namespaceContext="#nsContext" />
         </eip:predicate>
         <eip:target>
            <eip:exchange-target service="status:fileWriterCargo" />
         </eip:target>
      </eip:routing-rule>
      ...
      <eip:routing-rule>
      <!-- there is no predicate, so this is the default destination -->
         <eip:target>
            <eip:exchange-target service="status:fileWriterOther" />
         </eip:target>
      </eip:routing-rule>
```

```
   </eip:rules>
</eip:content-based-router>
```

5.3.4 Discussion

The scenario showed content-based routing in Camel, ServiceMix and Mule. The configuration in Mule is not complex, but when more endpoints and expressions are used, it can get very large really soon. It is important to keep an overview of all configured routing decisions. In ServiceMix the configuration of a filter is also done via XML but the code does not look as overloaded as in Mule.
A custom *Message Filter* is supported by Mule with a provided interface which can be implemented for achieving customized behavior.
In Camel a filter can be applied by providing a *filter* statement which is evaluated on each message. The statement takes a *Predicate* which evaluates to true or false for making a decision if a messages is filtered out or not. The Camel filter was reused in ServiceMix as it does not support the pattern with its EIP component.
Furthermore it was shown how different transport types can be configured in Mule. Not all used transport types could used in Camel and ServiceMix as they are buggy in the used release, for example the XMPP transport.

5.4 Communication between Air Control Agency (ACA) and Weather Information System

5.4.1 Description

The ACA information system must provide real time weather data to other systems around the airport. These systems include airline and airport information systems as well as other systems which are interested in weather information around the airport.
Various sensors (temperature, humidity, etc.) sense weather information from the surrounding area. This partial information is aggregated to a complete message and passed to the weather information system. The systems does some enhancements on the information, like adding time information and then sends it to the ACA system.

5.4.2 Modeling

A modeling of the scenario can be seen in figure 5.4. The workflow is triggered by sending a message to a publish-subscribe channel. All weather information sources will listen for messages on this channel. Each information source will generate a message which will be passed to an aggregator which combines all parts into one message. Afterwards

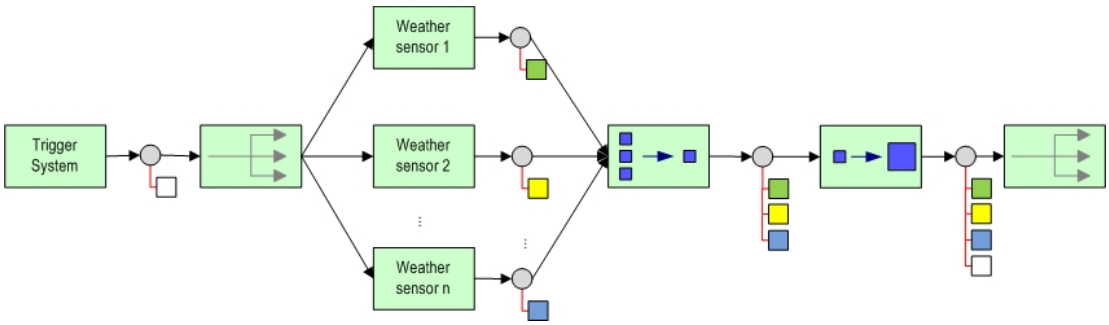

Figure 5.4: Weather Information system

a content enricher can add additional information to the message. The resulting message is published to a publish-subscribe channel where interested systems can connect to and wait for messages.
An alternative modeling would be to use a scatter-gather patterns for this scenario. When using this pattern, one component triggers the first message. All weather sources will generate a reply and send it back to the component which initially triggered the workflow. Afterwards the receiving component can aggregate all responses and enrich the content of a message before publishing the resulting message again.
Another alternative could be to use a recipient list instead of the publish-subscribe channel. But in this case the extensibility is more limited as when using a publish-subscribe channel. When a recipient list is used, the configuration has to be changed each time a new weather information source is added to the system. This does not have to be done in the first modeling approach.

5.4.3 Implementation

For implementing this scenario only Mule was used. The reason is that Mule offer more possibilities for aggregation of messages. Camel only has one methodology to achieve aggregation, which is shown in the last scenario. For Mule one possibility is shown in this scenario and another one in the last scenario. For ServiceMix the scenario could not be implemented due the weak aggregation support.

5.4.3.1 Mule

The scenario has been implemented with Mule in the first two modeling alternatives. The first alternative was used to test the aggregator support of Mule. For the second alternative different transport channels have been integrated, all sending their reply to one UMO component which is responsible for aggregating and further publishing.

Mule offers various ways to support the aggregator patterns. For this scenario a *respons-router* component that collects all responses is used. An example of that is shown in listing 5.23. What can be seen is that the outbound router of the trigger component contains a *reply-to* field, indicating where all replies will be sent to. After that the *response-router* is configured. The business logic of the aggregator is implemented in the *WeatherAggregator* component, which has to extend the *ResponseCorrelationAggregator* class of Mule. The other two components shown in the listing will listen for messages on the publish-subscribe channel used by the first component. What is important for these two components is that they not have an *outbound-router* configured. This is already done by the *reply-to* field and the *response-router* in the first component.

Listing 5.23: Aggregator in Mule

```
<mule-descriptor name="WeatherInfoBroker" implementation="eip.weather.mule.
    WeatherInfoBroker">
...
   <outbound-router>
      <router className="org.mule.routing.outbound.OutboundPassThroughRouter
          ">
         <global-endpoint name="WeatherInfoRequestTopic"/>
            <reply-to address="WeatherInfoReplies"/>
      </router>
   </outbound-router>

   <response-router timeout="1000000">
      <global-endpoint name="WeatherInfoReplies"  />
      <router className="eip.weather.mule.aggregator.WeatherAggregator"/>
   </response-router>
</mule-descriptor>

<mule-descriptor name="humidityProducer" implementation="eip.weather.mule.
    HumidityProducerMule">
   <inbound-router>
      <global-endpoint name="WeatherInfoRequestTopic" />
   </inbound-router>
</mule-descriptor>

<mule-descriptor name="temperatureProducer" implementation="eip.weather.mule
    .TemperatureProducerMule">
   <inbound-router>
      <global-endpoint name="WeatherInfoRequestTopic"/>
   </inbound-router>
</mule-descriptor>
```

5.4.4 Discussion

This scenario showed how to use a *response-router* which aggregates all responses into one message can be implemented with Mule. An abstract class is provided by Mule that has to be subclassed for providing custom aggregation logic. Mule also provides a good documentation for implementing and configuring all necessary components.

5.5 Communication within Hospital Information Systems

5.5.1 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 fulfill 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 only containing the patient 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.

5.5.2 Modeling

The modeling can be seen in figure 5.5. The message produced by the nurse consists of three parts. The parts are split and sent to a router who 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 parts 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 outcomings 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.
Modeling alternatives are rare for this scenario, as security and privacy of data are important for this scenario 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.

5.5.3 Implementation

All important parts of this scenario were implemented with Camel, ServiceMix and Mule to show the support for splitting and adjacent aggregation. One type of response aggregator in Mule was already described in the previous example. This scenario will introduce another type of aggregator of Mule which can be used for aggregating messages according to their correlation identifier.

5.5.3.1 Camel

The implementation of this scenario using Camel was very straightforward and did not bring many troubles. For the communication JMS messages with XML content were used. Listing 5.24 shows how the message can be split, routed and aggregated again with Camel. For splitting and routing XPath statements are used. What can be seen in the routes is that a message can be send to more than one endpoint which corresponds to a recipient list. After that an aggregator is used to aggregate messages according to their JMSCorrelationID which is set by the nurse system before sending a message. The class *MyAggregationStrategy* defines how the messages should be aggregated (e.g. content of latest message to be inserted at the beginning or the end of the already existing message, ...). The usage of a XSLT message transformer in Camel can be seen in listing 5.25. It is important to note that the body of the message has to be converted to a string before it is passed to the XSLT engine.

Listing 5.24: Splitter, router and aggregator in Camel

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

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

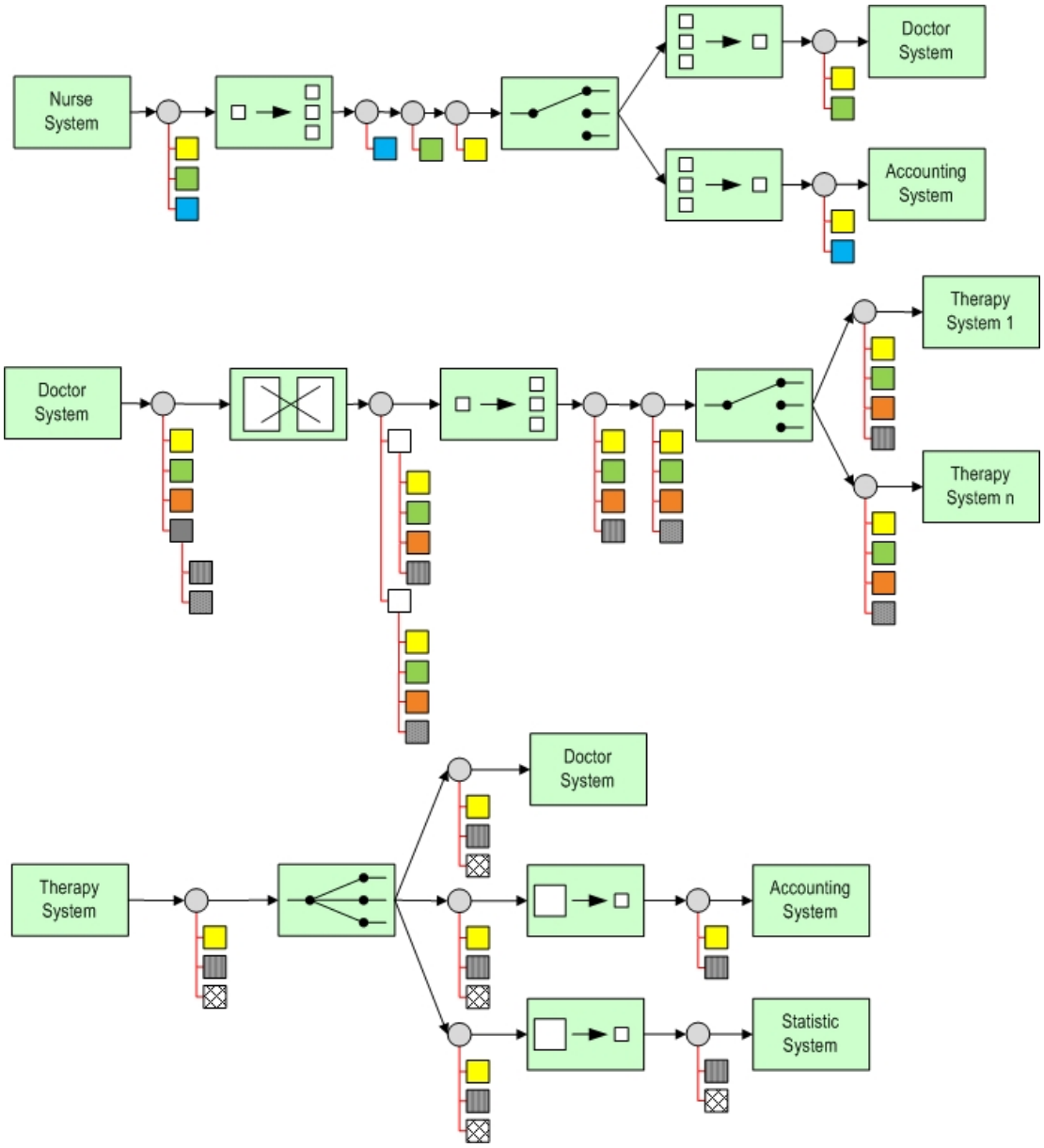

Figure 5.5: Hospital message flow

```
//route parts to different endpoints
from("hospital:camel.hospital.nurse.splitted")
.choice()
   .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");

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

Listing 5.25: XSLT transformer in Camel

```
File styleSheet = new File("xslt\\transFurtherTherapies.xslt");
XsltBuilder xsltBuilder = XsltBuilder.xslt(styleSheet);

from("hospital:camel.hospital.doctor.aggregator.therapy")
.convertBodyTo(String.class)
.process(xsltBuilder)
.to("hospital:camel.hospital.doctor.aggregator.therapy.xslt");
```

5.5.3.2 Mule

In this scenario another type of aggregator in Mule can be used. Mule offers an abstract implementation of a *CorrelationAggregator* which can be extended for implementing a custom aggregator. Like in the previous scenario the usage has to be specified in the configuration file. The difference is that this time it has to be configured at the inbound-router section of a component. Listing 5.26 shows the configuration of the aggregator. The class *CustomAggregator* provides the business logic and has to extend the abstract class *CorrelationAggregator* of Mule. Splitting a message into parts and routing them to the correct aggregator is also straightforward in Mule. Configuration can be done at the outbound-router section of a component with a combination of *FilteringXmlMessageSplitter* and a *JXPathFilter*. Listing 5.27 shows the appropriate

configuration for the accounting system. The same router configuration has to be done for the doctor system only changing the endpoint addresses and XPath statements.

Listing 5.26: Correlation aggregator in Mule

```
<inbound-router>
   <endpoint name="doctorAggregator" address="vm://hospital.doctor" />
   <router className="eip.hospital2.mule.util.CustomAggregator"/>
</inbound-router>
```

Listing 5.27: Splitting, filtering and routing in Mule

```
<outbound-router matchAll="true">
   <router className="org.mule.routing.outbound.FilteringXmlMessageSplitter"
       >
      <endpoint address="vm://hospital.accounting" >
         <filter expression="count(/patient)=1" className="org.mule.routing.
             filters.xml.JXPathFilter"/>
      </endpoint>
      <endpoint address="vm://hospital.accounting" >
         <filter expression="count(/paymentInformation)=1" className="org.
             mule.routing.filters.xml.JXPathFilter"/>
      </endpoint>
      <properties>
         <property name="splitExpression" value="/document/*"/>
      </properties>
   </router>
</outbound-router>
```

5.5.3.3 ServiceMix

The implementation of the scenario in ServiceMix was not as straightforward as with Camel or Mule. When using the Camel JBI component for ServiceMix the splitting of a message could not be achieved. The reason for this is that JBI has a restriction that only well-formed XML messages are allowed to be transfered on the bus. But after splitting the XML header is lost and therefore not accepted by ServiceMix anymore. This can be solved by using the splitter component of the ServiceMix EIP component. This splitter automatically inserts the XML header to a message again. By using this component the splitting could be done. Listing 5.28 shows how the splitter can be defined for the EIP component of ServiceMix. But after splitting is done, the aggregator makes some troubles, too. The properties set for aggregation of messages did not get copied after splitting the message and therefore the aggregator could not get its work done. To overcome this, message translations can be used before splitting the message. With an XSLT transformer the contents of a message are transformed, so that after splitting the correct parts of a message are grouped together and do not need to get aggregated again. The transformation is done after the nurse system sends a message.

For carrying out transformation in ServiceMix a *pipeline* pattern can be used which passes a message to an XSLT engine before it is sent to the next service-unit. What has also be done is to configure a service-unit that is responsible for the transformation. A simple configuration for that can be seen in listing 5.29. The attribute *resource* specifies the XSLT file to use for the transformation. The configuration of the *pipeline* pattern which invokes this service can be seen in listing 5.30. The dynamic router for therapy system was implemented as in the train splitting scenario with the help of the *destinationChooser*. When going a detour with an XSLT transformer instead of an aggregator the scenario can be implemented with ServiceMix, however an aggregator would be more convenient to use.

Listing 5.28: Splitter in EIP component of ServiceMix

```
<eip:xpath-splitter service="hospital:xpathSplitter" endpoint="endpoint"
     xpath="/document/*" namespaceContext="#nsContext">
   <eip:target>
      <eip:exchange-target service="hospital:routing" />
   </eip:target>
</eip:xpath-splitter>
```

Listing 5.29: XSLT transformer service-unit in ServiceMix

```
<saxon:xslt service="hospital:xslt" endpoint="endpoint"
   result="string"
   resource="classpath:hospitalTransform.xslt" />
```

Listing 5.30: Pipeline pattern in ServiceMix

```
<eip:pipeline service="hospital:pipeline"
   endpoint="endpoint">
   <eip:transformer>
      <eip:exchange-target service="hospital:xslt" />
   </eip:transformer>
   <eip:target>
      <eip:exchange-target service="hospital:xPathSplitter" />
   </eip:target>
</eip:pipeline>
```

5.5.4 Discussion

The scenario showed splitting and aggregation support in Camel and Mule, where a *CorrelationAggregator* was used. As aggregation is buggy in ServiceMix a workaround with XSLT transformations was shown. Implementing the scenario with either framework could be achieved without many troubles. As this scenario was the largest of all it could be seen that Camel offers the most clarity for defining all rules as they are configured with the help of the Java DSL which provides a very clear and straightforward

declaration of rules. Like in all other scenarios the configuration of Mule can get very large and difficult to read as the configuration grows. In ServiceMix the configuration is split up in many parts according to the service engine where a specific part of the configuration will be deployed to. It is important to keep an overview of all parts of a configuration, which can also get complex. This will be explained in more detail in section 6.4.

6 Results

In this chapter answers to the research question in chapter 3 are given. The first part shows the support of enterprise integration patterns of the four used frameworks in the practical part. For each pattern category a table and explanations are provided that show if a pattern is available in a specific framework. The second part of this chapter shows how the frameworks support categories of an EAI solution which were also introduced in chapter 3. The third section will explain how the used frameworks can help building service oriented and event driven architectures. Furthermore some software engineering aspects of the frameworks are given. The last part gives some business aspects when using open source products instead of commercial products.

6.1 Frameworks and Enterprise Integration Patterns

This section describes the support of integration patterns by the used frameworks. The following abbreviations are used in the tables.

- s = supported - pattern is supported by a framework
- ns = not supported - pattern is not supported by a framework
- oos = out of scope - not expected to be supported by a framework
- di = design issue - just a design issue when designing an EAI solution - no support by frameworks is needed

For each pattern some explanation is given in the text and should be read as it contains more information than given in the tables. The entries in the tables have been found by studying documentation, gained experience from implementing scenarios and review by email communication with Tijs Rademakers[1].

6.1.1 Messaging Channels

Table 6.1 shows the message channel patterns and their support by the used frameworks. The *Message Channel* pattern is supported by all frameworks. One could say that the used message broker is responsible for providing all message channels and therefore ActiveMQ is the framework that mainly relates to this pattern. But when also considering

[1] author of the book „Open Source ESBs in Action“ [22]

files, SMTP, XMPP as message channels ActiveMQ cannot cover all these protocols. All other frameworks provide methods for communicating with such components and therefore all frameworks support this pattern.
The patterns *Point-to-Point Channel* and *Publish-Subscribe Channel* are are supported by all frameworks. However, they rely on the underlying messaging protocol. JMS provides the components *Queue* and *Topic* for making these two patterns available. When using an in-VM messaging a *Publish-Subscribe Channel* is not supported. Thus the support of these pattern do not only rely on the frameworks but also on the used messaging protocol.
Also *Datatype Channel* and *Invalid Message Channel* are special uses of a *Message Channel*. But in this case it does not rely on the used messaging protocol but on the contracts made by the designer of an EAI solution. It is therefore primarily a design issue for a channel. If the contract requires that only only messages of a specific datatype are allowed on a channel then this implicitly leads to a *Datatype Channel* pattern. However Mule provides explicit support for this pattern where it can be specified which Java objects are allowed on a channel. A router that forwards all messages that do not meet any routing rule to a default channel or an application which puts invalid messages to a default channel, lead to an *Invalid Message Channel*.
Dead Letter Channel is a pattern that should be supported by the messaging infrastructure. The difference between an *Invalid Message Channel* and a *Dead Letter Channel* is that in the first case a message cannot be delivered because a receiver has rejected a message or a router could not find a routing rule. In the second case the messaging infrastructure cannot even find a receiver and therefore a message cannot get out of the messaging channel and the messaging infrastructure. Thus the message broker has to provide some mechanisms for sorting out such messages. All frameworks offer support for a *Dead Letter Channel* by specifying timeouts and retry counts.
Guaranteed Delivery is a pattern that should be provided by the message broker. Thus all messages received via a message broker meet this pattern, as the broker uses a datastore for storing messages.
The *Channel Adapter* pattern deals with making messaging available to system which do not have any messaging capabilities. For example a file on a file system or on a ftp server is not available for messaging. But Camel, ServiceMix or Mule can make the file available through the messaging system. Of course this is only a single example and does not deal with making legacy system available for messaging. But writing and parsing files can be done by nearly all system which can be seen as *Channel Adapter* pattern. When designing an EAI solution it has to be specified which adapters are necessary and then checked if the messaging system can support the integration of the adapter. For ActiveMQ this pattern is out of scope as ActiveMQ comes into play as soon as a message exists but this patterns is situated one step before a message is created.
A *Messaging Bridge* is something that cannot be supported out of the blue. It has to be known which protocols have to be bridged and a support for all possibilities cannot

Pattern Name	ActiveMQ	Camel	ServiceMix	Mule
Message Channel	s	s	s	s
Point-to-Point Channel	s	s	s	s
Publish-Subscribe Channel	s	s	s	s
Datatype Channel	di	di	di	di / s
Invalid Message Channel	di	di	di	di
Dead Letter Channel	s	s	s	s
Guaranteed Delivery	s	oos	oos	oos
Channel Adapter	oos	s	s	s
Messaging Bridge	s	s	s	s
Message Bus	s	s	s	s

Table 6.1: Mapping of Message Channel Patterns to Frameworks

be given. Nevertheless all frameworks offer some functionalities for implementing a *Messaging Bridge* that can bridge specific protocols. Therefore one could state that all frameworks provide support for this pattern.
The last pattern in this category is the *Message Bus*. When using this pattern all applications are connected to a bus where they put their messages onto. New applications can easily be connected to or disconnected from the bus without affecting other applications. Thus the bus helps to decouple all participating applications. The goal of all frameworks is to decouple applications. Because of that, all frameworks themselves can already be called a *Message Bus* as all messaging will be done via the frameworks and therefore they act as a bus for all participating applications.

6.1.2 Messaging Endpoints

A mapping from endpoint patterns to frameworks is shown in table 6.2
A *Message Endpoint* must be available to connect an application to a messaging infrastructure. Endpoints are mainly provided by a message broker but can also be connectors to SMTP, POP, file, etc. All frameworks provide mechanisms to make messaging available for many types of protocols. Therefore they all provide support for this pattern.
The *Messaging Gateway* deals with the separation of messaging code and other business logic code in an application. It is a pattern that has to be considered when building applications. The scope of the frameworks is not building, but integrating applications and therefore this pattern is out of scope for all of them.
A *Messaging Mapper* maps contents of domain objects to contents of a message. Camel and Mule provide support for this pattern. In both framework custom transformers can be written and applied when sending and receiving messages. For ActiveMQ such

a mapper is out of scope as it mainly deals with providing messaging infrastructure. ServiceMix can also offer support for this pattern through the Camel component, but has no built-in mechanism for supporting it. Therefore it is marked as not supported.
All frameworks offer support for *Transactional Clients* by providing mechanisms like XA or Spring transactions.
Polling Consumer and *Event-Driven Consumer* are supported by all frameworks, either. When using polling a channel is checked in regular intervals for new messages. When the event-driven approach is used, a callback is registered on a channel that will be called by the messaging infrastructure when a new message arrives.
The *Message Dispatcher* pattern is not supported out of the box but can be implemented using either a *Selective Consumer* or a *Message Router*. When multiple consumers listen for messages, each of them could have a select criteria to only consume some of the messages on a channel. This leads to some simple form of load balancing. On the other side a router could route messages to different consumers and achieving load balancing again. Therefore this pattern is marked as supported for all frameworks.
Competing Consumer is a type of load balancing where more than one client are listening on a queue and taking off messages, but one message is only processed by one client. While this client is busy, other clients take off messages from the channel. When using JMS messaging, a message only gets delivered to one consumer on a queue. Thus all frameworks can support this pattern when using JMS messaging.
The *Selective Consumer* pattern depends on the used messaging protocols. When only using JMS, the API provides mechanisms to provide a selection criteria by a message consumer. For other protocols this might not be available. What could be done instead of a *Selective Consumer* is a *Message Filter* which is provided by all frameworks.
Durable Subscriber also depends on the underlying messaging protocols. JMS supports subscriptions to queues or topics ensuring that a consumer cannot miss any message. For other protocols like SMTP this is not available.
An *Idempotent Receiver* is supported by Camel and Mule. For ActiveMQ this pattern is out of scope as it does not deal with contents of a message. ServiceMix can use the Camel component to get support for this pattern.
The *Service Activator* pattern deals with making services of an application available to other applications. This pattern is situated in the application architecture but out of scope for integration frameworks as it has to be applied before integration can take place.

6.1.3 Message Constructions

Table 6.3 shows the mapping from message construction patterns to the four frameworks. Some patterns in this category do not relate to the used frameworks as construction of messages is settled more in the design phase of an EAI solution where contracts for exchanged messages are specified.

Pattern Name	ActiveMQ	Camel	ServiceMix	Mule
Message Endpoint	s	s	s	s
Messaging Gateway	oos	oos	oos	oos
Messaging Mapper	oos	s	ns	s
Transactional Client	s	s	s	s
Polling Consumer	s	s	s	s
Event-Driven Consumer	s	s	s	s
Competing Consumers	s	s	s	s
Message Dispatcher	s	s	s	s
Selective Consumer	oos	s	s	s
Durable Subscriber	s	s	s	s
Idempotent Receiver	oos	s	s	s
Service Activator	oos	oos	oos	oos

Table 6.2: Mapping of Messaging Endpoint Patterns to Frameworks

A *Message* has to be supported by all frameworks. If this would not be supported the framework could not be used as they all deal with messaging. The patterns *Command Message*, *Document Message* and *Event Messages* are special usages of the *Message* pattern. They specify the content of a message. If it is an event that has to be communicated or a document that should be shared is not relevant for the frameworks. Thus these patterns are just design issues for an EAI solution and are therefore not related to the frameworks.

The *Request-Reply* pattern also is a design issue. Applications need to know what they have to do after receiving a message. If a reply is needed the receiving application has to achieve that, but none of the frameworks.

The patterns *Return Address*, *Correlation Identifier* and *Format Indicator* are a kind of metadata of a message. When only JMS messaging, there are already property fields available in the JMS API for these fields. But all frameworks offer capabilities for setting property values of message where a return address or correlation id should be set as property of a message. Therefore these patterns are supported by all frameworks

Message Sequence deals with splitting large messages into sequences and send them individually. Like with the patterns above this behavior can be achieved by setting properties in the header of a message. Therefore support is provided by all four frameworks.

Message Expiration deals with a time to live for messages. All frameworks support some timeout mechanisms where messages are deleted when a certain time has passed by.

Pattern Name	ActiveMQ	Camel	ServiceMix	Mule
Message	s	s	s	s
Command Message	di	di	di	di
Document Message	di	di	di	di
Event Message	di	di	di	di
Request-Reply	di	di	di	di
Return Address	s	s	s	s
Correlation Identifier	s	s	s	s
Message Sequence	s	s	s	s
Message Expiration	s	s	s	s
Format Indicator	oos	s	s	s

Table 6.3: Mapping of Message Construction Patterns to Frameworks

6.1.4 Message Routing

A mapping from routing patterns to frameworks can be seen in table 6.4. Most of these patterns are out of scope for ActiveMQ as it is just a message broker which provide infrastructure for messaging and has no capabilities for inspecting any content of messages and doing routing based on it. But as Camel is fully integrated into ActiveMQ everything said about Camel is also true for ActiveMQ. But for this comparison the two frameworks are split into two parts. As most patterns are out of scope for ActiveMQ, whenever the term *three frameworks* is used Camel, ServiceMix and Mule are meant.

Pipes and Filters is an architectural style for building EAI solutions and is therefore not related to any framework, but of course the it can be implemented with the help of all frameworks.

Message Router and *Content-Based Router* are supported by Camel, ServiceMix and Mule. ServiceMix can support the patterns through their own integration patterns component and the Camel component. All frameworks are offering static routers as well as routing based on the content of a message.

The *Message Filter* pattern is also supported by Camel, ServiceMix and Mule which can make content-based decisions.

Per definition a *Dynamic Router* is in possession of a rule base which includes routing rules and can be changed by a control channel. Such a pattern is not supported by any of the frameworks. However implementing a rule base for a router is not a big deal. For example a database where one can easily update any entries could be used as a rule base. Therefore this pattern is marked as design issue, as it is a special case of a router.

A *Recipient List* and a *Resequencer* are supported by all three frameworks, whereas the *Routing Slip* pattern is only supported by Mule and ServiceMix.

A *Splitter* is supported by all three frameworks but has some limitations in ServiceMix.

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	di	di	di
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 6.4: Mapping of Message Routing Patterns to Frameworks

The Camel component for ServiceMix is not free of bugs and therefore the pattern cannot be used by the Camel JBI component. It was already said that in ServiceMix messages always have to be well-formed XML content. But after splitting of an XML message, the XML header information is not copied to all split messages and ServiceMix throws an exception because of that. When using the pattern component of ServiceMix splitting works fine.
An *Aggregator* is also supported by all three frameworks. However the usage of this pattern in ServiceMix is not unproblematic. There are problems getting it running which in fact could not be done during this thesis. Mule offers abstract base classes where one can implement all required business logic.
A *Process Manager* is supported by Mule and ServiceMix. Mule has a built-in jBPM engine to run business processes and also provides support for some BPEL engines. An Apache Ode JBI component is available which can be deployed to ServiceMix. BPEL processes can be deployed to this engine. Camel does not offer any support for process management.
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 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.

6.1.5 Message Transformation

Table 6.5 shows all transformation patterns and their support by the four frameworks. Like it was already stated at the routing patterns, also the transformation patterns are

Pattern Name	ActiveMQ	Camel	ServiceMix	Mule
Message Translator	s	s	s	s
Envelope Wrapper	oos	s	s	s
Content Enricher	oos	s	s	s
Content Filter	oos	s	s	s
Claim Check	oos	s	s	s
Normalizer	oos	s	s	s
Canonical Data Model	di	di	di	di

Table 6.5: Mapping of Message Transformation Patterns to Frameworks

out of scope for ActiveMQ as it only is a message broker.
A *Message Translator* is supported by all frameworks, even by ActiveMQ as it has some support for translating protocols of messages. The other frameworks additionally offer some XSLT transformations on the content of a message.
The *Envelope Wrapper* pattern can be seen as special use case of a translator. It puts an envelope around the origin message and then sends the message. This is a special form of translation and is therefore supported by Camel, ServiceMix and Mule. Also the *Normalizer* is a special use case of a translator which means support is provided by all three frameworks.
Content Enricher and *Content Filter* are supported by all three frameworks. For ServiceMix one has to be careful when changing the content of a message as it has to be a well-formed XML document after filtering or enriching.
The *Claim Check* pattern is a combination of other patterns. Therefore the pattern is supported by all frameworks, but some coding has to be done to achieve the support.
The *Canonical Data Model* is not related to the frameworks as it is a design issue while making contracts for the EAI solution. The pattern means that all messages adhere to a common format. Perhaps this pattern would be better settled in the message construction than in the message transformation category.

6.1.6 System Management

Table 6.6 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 on 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 informations to the control bus telling it that is still alive. All frameworks provide support for administration through JMX. This can al-

ready 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 to provide the functionality of a *Detour*. Again this is only a design issue. For ActiveMQ this pattern is out of scope as it deals with routing issues.

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, ServiceMix and Mule.

A *Message History* deals with adding an history element to messages traveling through the messaging infrastructure. By doing this, each component can analyze 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 scope for the frameworks. Another way could be to use a *Content Enricher* after each component that inserts a history element to 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 a database. 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 design issue. As ActiveMQ needs to ensure *Guaranteed Deliver* it already uses a *Message Store* for persisting all messages.

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 to 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 behavior 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 of the message 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 design issue. For ActiveMQ it is mapped as out of scope as the framework provides mechanisms for all messages and do not care if messages are used for testing or not.

The *Channel Purger* is used to delete all messages from a channel. ActiveMQ has support for this pattern through its administration console. For Camel, ServiceMix and Mule this pattern is out of scope as it should be situated directly in the message broker.

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 6.6: Mapping of System Management Patterns to Frameworks

6.1.7 Summary

What can be seen in the tables of this section is that ActiveMQ mainly provides support for messaging channel and endpoint patterns. The target patterns of Mule and Camel are routing and transformation patterns for which they provide nearly full support. ServiceMix also offers support for routing patterns but is constrained to XML message content which limits some patterns, like *Splitter* and *Aggregator*. System management patterns are mainly combinations of patterns or need some additional implementation effort which has to be done. 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 easily be done.

6.2 Frameworks and EAI categories

In chapter 3 categories which can make up an EAI solution were listed. This section provides a classification of ActiveMQ, Camel, ServiceMix and Mule to EAI categories. Figure 6.1 shows a mapping from categories to the four frameworks. It can be seen that ActiveMQ basically provides a message oriented middleware. Additionally it can offer management and monitoring capabilities. Camel mainly is a pattern implementation and also offers a data transformation engine. It is a spring based framework, therefore the category application container is also mapped to Camel. ServiceMix provides support for all EAI categories. For pattern implementation ServiceMix offers its own pattern component and the Camel JBI component. Furthermore a Drools-, an Apache Ode- and a Saxxon component are available for ServiceMix which map to the categories rule-, workflow- and data transformation engine. For management and monitoring JMX is used. ServiceMix also offers integrated spring support and is deployable

in a Geronimo application server. It is important to state that all components which are available in ServiceMix are not defined by the JBI standard. The standard only specifies the environment that has to be provided for all components. This includes installation, deployment, control and monitoring of components but not any functionality of them. Furthermore it is not said that any workflow- or rule engine has to exist in any JBI environment. In figure 6.2 the architecture of a JBI environment is shown. All components which are available in ServiceMix are represented by either binding components or service engines in this figure. Thus ServiceMix is an implementation of the JBI standard that already delivers many components which can immediately be used. Like ServiceMix, Mule also supports all EAI categories. For workflows jBPM or BPEL engines are available as well as Drools which can be integrated into Mule. Furthermore a XSLT transformation is also available. As well as the other frameworks Mule offers management and monitoring via JMX. It can also be integrated with a JBoss or a Geronimo server. At last Mule also offers spring support. What is important in the figure is the striped representation of a message oriented middleware. This indicates that Mule does not only provide one message broker but abstracts from it and can be used with various brokers that can be selected independently for each project.

6.3 Frameworks and SOA, EDA

Many strategies for enterprise IT architecture have the goal to migrate their systems to a SOA. This is done because organizations expect benefits when migrating the IT landscape. Anne Thomas Manes gave a presentation about the business values of a SOA, where management benefits are explained [18]. In this section some points how the introduced frameworks can help to build a SOA in an enterprise are given.
Usually web services are transported over HTTP protocol in a request reply fashion. With the help of the Apache CXF service framework it is possible to bridge web service invocation over a JMS protocol. The usage of JMS introduces real asynchronity for web service calls and decouples provider and requestor of a service. ActiveMQ can be used as message broker in the infrastructure and enables the bridging of SOAP messages. Additionally one can also use Camel for routing web service invocations to any desired endpoint which shows that this approach leads to asynchronity and loose coupled systems in a SOA.
Camel, ServiceMix and Mule can all be used for implementing SOAs as they can decouple applications. An application send a message which is processed by any framework and passed to a service provider which can handle the message. The configuration of the frameworks knows where service providers can be found and will send all requests to the appropriate provider. An advantage of decoupling is that any service can be replaced without affecting other applications. Only the configuration of the framework has to be changed to route requests to new services. All frameworks offer a reasonable

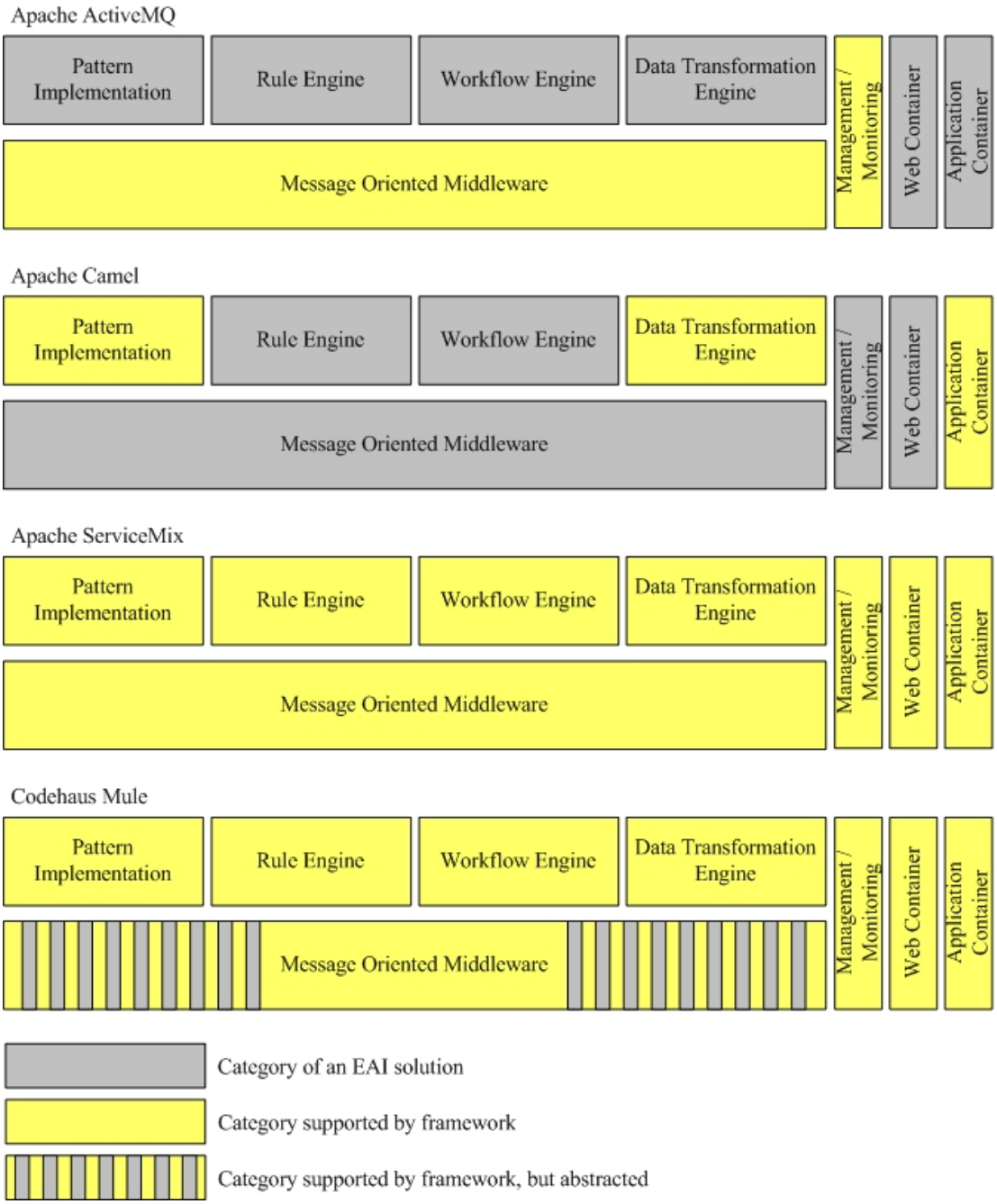

Figure 6.1: Classification of frameworks to EAI categories

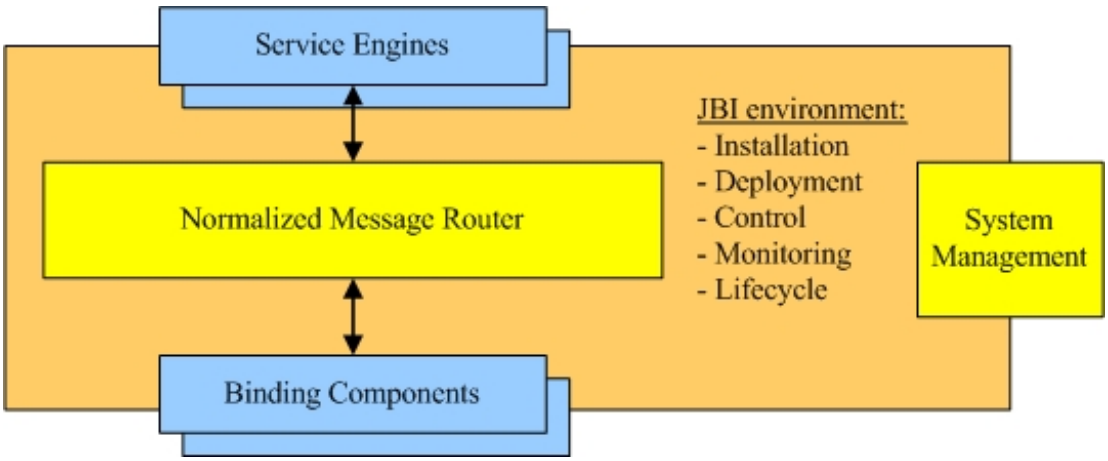

Figure 6.2: Architecture of JBI

amount of transport protocols for communicating with applications and bridging of protocols. For example a framework can be used to expose legacy applications as services. A file transport can be used for communicating with the legacy system but any other protocol can be used for invoking the service at the framework. An application will send a request which is handled by a framework and passed to the legacy system. The calling application will not know where the service implementation is located. It will just send a request to the framework and await a reply.

It is also possible for the frameworks to expose existing services to new protocols. For example a request that has always been sent over a JMS message can also be sent via a HTTP message and also get routed to the correct service implementation. Now a service is available using two different transport protocols which ensures extensibility of the complete integration solution.

A request can be seen as an event in an event-driven architecture, thus frameworks bridge together the approaches of SOAs and EDAs.

6.4 Software Engineering Aspects

In this section some software engineering aspects of frameworks are considered. This is done because it is not only relevant that frameworks provide a lot of functionalities, but the functionalities have to be used by developers to implement a solution. Therefore it should be known how easy or difficult it is to use functionalities of a framework.

6.4.1 Development & Configuration

6.4.1.1 ActiveMQ

For ActiveMQ not much development support is needed, it justs needs to be started and after that applications can connect to the broker and send and receive messages. Everything that has to be done is to include the needed libraries to an application. Thus from the prospect of a developer not much has to be done to work with ActiveMQ. But what has to be done is to configure ActiveMQ for specific purposes. Configuration is done by editing the *activemq.xml* configuration file. For all configuration options explanations are given on the ActiveMQ project website. Therefore the configuration can be done very easily and quickly.

6.4.1.2 Camel

As already said in chapter 4 there are two possibilities to implement patterns using Camel. One can either use XML or a Java DSL for configuring rules. Both options are documented well on the project website. Java could be preferred when an IDE is available which supports code completion. When using Java one can decide how pattern implementations are structured. One can therefore group related patterns together in one function to keep an overview. Camel leaves developers with the choice how the code is structured and does not have any constraints. For example it is also possible to place routing code directly after messaging. An advantage of this could be that a developer immediately recognizes what will happen to a message after it is sent. On the other side one has to consider that routing and mediation code is distributed over many Java classes.

6.4.1.3 ServiceMix

ServiceMix offers Maven archetypes for creating service units and service assemblies for all components delivered with the distribution. After running the appropriate commands all necessary files for a service unit are created and just have to be edited by developers. This simplifies the initial effort for a developer as it is ensured that no required files are forgotten to be created.
One drawback which results in the component architecture of ServiceMix is that for each service engine which shall be used in an EAI solution one service unit has to be created. For each service unit a folder structure is created in the main project folder, which contains all required configuration files. The main configuration file for a service unit could be found in the following directory: *mainfolder/service-unit-folder/src/main/ressources*. Imagine five service units for a project which results in five service unit folders, each containing the needed configuration file in the third subfolder. One can see that this leads to an extensive folder structure. Keeping the overview about

this is not simple especially when projects are getting larger. Furthermore nearly all configuration files are named *xbean.xml*. When some or all of them are opened in an editor it is also not easy to find the correct file for editing immediately. This can lead to errors very fast when making changes to a wrong configuration file. Additionally all maven configuration files have to be maintained by a developer, too. On the one side Maven archetypes help creating the required folder structure and configuration files but keeping an overview about all files can get very hard for larger projects. Some specialized tool support for managing all configuration files would be a really good step towards simplification of development for ServiceMix.

6.4.1.4 Mule

Configuration in Mule is done in an XML configuration file. There all connectors, transformers, used patterns, etc. are configured. Example configurations are found on the Mule project website and therefore configuration is straightforward. However the configuration file can get large very fast and keeping an overview about one large XML file is a challenging task. Some XML editor with code completion and code checking might be the first step for simplifying configuration. While developing scenarios of this thesis Mule 1.4.3 was used as it was the latest stable version. However in the finishing phase of this thesis version 2.0 of Mule was released. This version has already done a step towards simplification of configuration files where not as much XML is needed as in version 1.4.3. This is also stated as an argument for Mule 2.0 in the Mule community. But as all implementation work was already finished at the time of the release and the complete thesis also was in a finishing state, the scenarios did not get re-implemented with Mule 2.0.
The second part of development in Mule are UMO components. Support for this is given implicitly as their are just POJOs which can be written in the preferred development environment.

6.4.1.5 Summary

Development support is an important factor when choosing an appropriate framework. Camel places no constraints on development and a developer can decide its favorite way how and where patterns are implemented. ServiceMix and Mule are not as flexible as Camel. The configuration in ServiceMix is done via various configuration files where one can lose the overview very fast. Mule configuration is the opposite of this. In Mule all configuration is done in one XML file which can get large in short time. Keeping the overview of large XML files is also not an easy task. But as already mentioned, this will be simplified in the next version of Mule. Thus ServiceMix and Mule are some kind of extreme examples for types of configuration. For each type one can find strengths and weaknesses. It also depends very much on the developer who can prefer one style

over the other. But for both frameworks it has to be said that development support by providing tools for making configurations would ease development very much.
The highest form of development support would be to provide graphical editors where it is possible to plug in pattern components and generate source code or configuration files out of it. But this vision is not supported by any framework.

6.4.2 Maintenance

Maintenance is a very important part for an EAI solution as it has to be ensured that message do not queue up because of a bottleneck. For fixing errors in short time, management tools can help to locate the source of an error.

6.4.2.1 ActiveMQ

ActiveMQ offers a web console on which size of queues and topics can be evaluated. It is also possible to delete, purge or send messages to channels. It is also possible to list all durable subscribers for channels. Additionally all information can be exported to an XML file that can be processed by other tools.
Another way for monitoring and managing a running ActiveMQ instance is to use a JMX console as ActiveMQ offers JMX support. A third option is to use command line tools provided by ActiveMQ to get information about running broker instances. A last possibility for monitoring and managing JMS queues is to use HermesJMS [2]
One drawback of ActiveMQ is that the configuration cannot get updated while the broker is running. If the configuration file gets altered the broker has to be stopped and started again for the configuration changes to take place.

6.4.2.2 Camel

Camel offers a visualisation plugin where one can generate a graphic out of a context. One drawback of the visualisation is that it only is a static diagram.
It is possible to add new routes to a context whenever a context is running. Thus adding new application and new routes is possible without stopping anything. But of course the routes for newly added applications have to be in another Java file than already existing routes. What is not possible with Camel is to update and remove routes from a context. To achieve this, the context has to be stopped, altered and started again. This implies that while the context is stopped, all application cannot communicate via Camel anymore. This can be weakened as messages will not get lost when using persistent messaging but the delivery gets delayed for some time. For applications where no timing issues exist this might not be a problem but for time-dependent applications this is not a desirable situation. In the Camel community a request has been made that

[2] http://www.hermesjms.com

rule updates can be done while a context is running but it is not sure when this feature will be added to a release.

6.4.2.3 ServiceMix

A big advantage of the JBI standard is that it defines a real container where solutions are deployed to. It is possible to update running configuration without the need to stop some components or the complete container. As ServiceMix is a JBI implementation it is not necessary to stop any deployed EAI solution for carrying out updates. Updating can be achieved by using provided Maven archetypes.
Management and monitoring of the container can be done via JMX. ServiceMix also offers a visualisation plugin to create static diagrams out of a configuration, like in Camel.

6.4.2.4 Mule

Like ActiveMQ and ServiceMix, JMX can also be used to manage components in Mule. Additionally Mule offers a governance framework, called Mule Galaxy, for managing configurations and activity monitoring.
Furthermore Mule offers a configuration visualizer for generating diagrams out of configuration files.
One drawback of Mule is that a configuration cannot get updated while it is running. This means when updates have to be carried out the configuration has to be stopped first. After doing all updates the configuration can get started again. However dynamic updates are supported by the enterprise edition of Mule which is not available for free. Like it was already said for Camel, in a time-dependent environment stopping a configuration for doing updates is not a desirable solution.

6.4.2.5 Summary

What can be seen is that all frameworks offer support for JMX management and monitoring. Also all frameworks provide some kind of visualisation of a context or a configuration. What is only supported by ServiceMix are dynamic updates to running solutions. This is definitely a great advantage of ServiceMix. For all other frameworks, a running solution has to be stopped for carrying out updates and then started again. It depends on an EAI project if this behavior can be accepted or not.

6.4.3 Documentation

For all frameworks it is important to offer a good documentation to support developers. Without documentation the best frameworks or tools are useless as nobody knows how they can be used.

This section will not be divided into subsections as the quality of all documentations is almost equal. All frameworks offer an online documentation which gives most important information needed for getting started and implementing first examples. There are also tutorials and cookbooks available on the project websites. However only all basic topics are covered and more detailed information is often not found in the documentation. Additionally the structure of the documentation could be improved by all projects.
All frameworks offer support through mailing lists which is really good. For most cases an answer is given within some hours. One can also search the lists archives, where a problem solution can be found quite often.
To summarize, the documentation of all frameworks is improvable but all basic information can be found. What would also be nice but is offered by none of the frameworks, is an offline documentation which would also introduce a better structure to the documentation.
An alternative to the minor documentation for ServiceMix and Mule is a book about these two frameworks, written by Rademakers and Dirksen [22] which provides a really good introduction and coding examples to both frameworks.

6.5 Business Aspects

Aaron Weiss has written an interesting article about the total costs of ownership of commercial and open source software [33]. He mainly brings three points to be considered when making a decision between commercial and open source software.
The first point is *Upfront Costs and Internal Synergy*. Internal synergy means the level of success an open source project has. If a project is successful it will find more developers which can make the project better and even more successful and therefore make it attractive to a wider range of users. A project that has a high internal synergy is the Apache web server. The argument that initial license savings are eliminated by the follow-up labor costs can be weakened when using open source software with high internal synergy but gets stronger for projects with lower internal synergy.
The second point in the article is *Labor vs. Licenses*. At this point the author brings an interesting comparison:

> „Commercial software is ostensibly designed to meet the needs of customers, which may include administrative facilities. In contrast, open source software is typically designed to meet the needs of a task.“ [33]

The argument is that the usage of open source software is investment in human capital of a company. It might be even the case that initial license costs are less expensive that this investment. But over time this might change. A commercial software vendor has to bring in revenues which can lead to license renewal fees and enforced update programs. The vendor could stop the support for a specific license which automatically

forces customers to update the license and pay fees for that. This cannot happen when using open source software. Here the costs are fully controlled by the company and not by any software vendors. The author concludes with this statement:

> „Open source software demands only an investment in human capital, a cost that can be controlled by the organization itself rather than a third party.“ [33]

The third point that is mentioned in the article is *Scalability*. Commercial licenses are sometimes based on the number of users or the number of processors on which they run. This even gets worse if more processors are used in one computer. There is no difference for an administrator to administrate a single-CPU or a four-CPU computer. But to use the software four licenses are needed instead of one.
Thus a growing company must always consider growing license costs. When using open source software the costs scale with the growth of the company.
The article concludes with the following sentences:

> „The crux of the comparison between commercial and open source software is where the better human capital investment lies in a third-party vendor, or in your own experts.
>
> For short-term or task-specific or turnkey solutions, TCO may lean in favor of commercial vendors. For long-term customizable and adaptable solutions, TCO may lean in favor of open source products, and keeping long-term cost control in-house.“ [33]

What has to be done now is to find out the internal synergy of the used open source frameworks which determines the cost range of internal labor costs. I would not rate the internal synergy of all four projects as high as the Apache web server as all frameworks are still not free of errors and documentation is not complete. Thus an organization must assume that investments in human capital or commercial support have to be done when using one of the frameworks. Like Aaron Weiss said, an organization has to decide if investments should be done in own human capital or in vendors human capital. When thinking that EAI solutions are important projects which are vital for a company to run all their systems and are usually very long-term projects, I would think that an investment in own human capital should be preferred.
Another point that has to be mentioned in this context is when open source software is used and investments in own human capital are done the outcoming solution will be highly adaptable for all specific needs and requirements. As the source code can be changed new requirements that might not be provided by any commercial vendor can be implemented by the company itself. This could advantage as an enterprise can react on changes in market situations faster as when they rely on any vendor to implement new requirements. This can lead to significant business advantages.

7 Conclusion & Future Work

7.1 Conclusion

In chapters 4 and 6 a lot of information were given to ActiveMQ, Camel, ServiceMix and Mule. Chapter 5 showed some examples how patterns can be implemented with the help of the frameworks. This can provide a basis for making a decision which can privilege one framework against the others. ActiveMQ can be taken out of a comparison as it mainly is a message broker on which all other frameworks can rely on. Thus support for routing and transformation patterns are out of scope for it. But since the latest release of ActiveMQ, Camel is fully integrated into the broker, which means that all rules can be configured directly in ActiveMQ configuration files. This combination makes them a good player in the EAI field. However for further explanations ActiveMQ and Camel are treated independently from each other.
Camel and ServiceMix use ActiveMQ as message broker whereas Mule abstracts from it and can work with several message brokers. Thus the main decision has to be done between Camel, ServiceMix and Mule. If for some reason ActiveMQ should not be used in an organization a decision favoring Mule can be taken as it does not depend on it. If ActiveMQ can be used, more facts have to be included in the decision making process. All frameworks offer support for a reasonable amount of integration patterns. Section 6.1 gives details about all supported patterns by each framework. Thus one can identify all needed patterns and look up in the tables if they are supported by a framework. This can favor a decision for one framework which supports all required patterns. If there are requirements for more than just patterns, like workflow engines Camel cannot be used anymore. For this purpose a decision between ServiceMix and Mule must be made. ServiceMix and Mule have both advantages and disadvantages that have to be compared for finding a decision. ServiceMix is completely based on a standard which reduces vendor lock-in. Also a lot of third party JBI components are available which can all be deployed into the ServiceMix container. If further components are needed one can implement custom components for specific needs, which also have to be based on the JBI standard and can therefore be reused, too. These points argue for ServiceMix. However it is limited to XML content in messages. Thus when more than just XML messages, like transferring objects, are required ServiceMix has limitations. However an XML serialization of objects could be used to transfer them via ServiceMix which weakens the limitation a little bit. Mule does not constrain message contents which means that also objects can be transfered. One drawback of Mule is that it cannot

handle dynamic configuration updates which is supported by ServiceMix. Dynamic configuration is only available in a commercial release of Mule.

Mule and Camel are built upon ideas from developers which favor their approach for EAI. On the other side ServiceMix is fully built upon the JBI standard which downsizes vendor lock-in as everybody can write standard compliant components. When additional components for Camel or Mule are needed it requires a lot more implementation effort as one has to know how to integrate new components into the existing framework. From this perspective ServiceMix is situated one level higher as Camel and Mule.

What can also be used are combinations of frameworks, for example using Camel inside ServiceMix for pattern support. It was already announced that the Camel JBI component is not free of bugs but for some patterns in can be used without any troubles. Now one further question could arise. When using Camel and ServiceMix in combination one can connect to many transport types, like file, web services, etc. with the help of both frameworks. This was also shown in scenario 5.2 where JMS message are sent directly by the Camel JBI component. However, if possible ServiceMix should be preferred to connect to common transports as the connection to external services can be made available for all components on the bus and not only for one component.

The decision for one framework is definitely a very hard. One has to consider many points when evaluating frameworks. From the point of pattern support one could say that the most simple framework that supports all required integration patterns should be used. From that point of view a decision between Camel and Mule has to be made as development and pattern support can be achieved much easier and faster as in ServiceMix. An objective decision is a very hard to do. It rather depends on preferences of integration architects and developers. If one prefers writing XML configuration instead of specifying patterns in Java classes one would prefer Mule. When the opposite is preferred Camel is the framework of choice. However if updates on running solutions have to be made both frameworks fail and a decision towards ServiceMix has to be done.

If the company strategy is building an EAI solution based on a standard, Camel and Mule are disqualified as they do not implement a standard and ServiceMix is the choice without having any competitors.

Thus it can be seen that a general statement that one framework is the best cannot be done. Instead all requirements have to be collected and mapped to the points given in this thesis for finding a framework that fits to all needs.

7.2 Future Work

This section contains thoughts complementing this thesis, which can be done as in future work in another thesis or practical work.

7.2.1 Improve Development Support

What has been seen at the software engineering aspects development support and documentation could be improved for all four frameworks. A convenient feature would be some graphical user interface where one can plug in all required patterns and configuration code is generated out of it. The business logic for such a tool would be rather simple as there is no need for complex algorithms but just generating configuration statements. The implementation of a good graphical interface takes more programming effort than the business logic for such a tool. Nevertheless, this would really ease development and reduce development time and chances for making defective configurations. Perhaps one could start with implementing basic patterns like a message router, content filter and -enricher, a splitter and an aggregator. A prototype for such a tool could be implemented in a future practical or thesis. When it is defined for which framework a it will be implemented, one can also expect support by the project developers as such a tool would help to gain popularity for their framework.

7.2.2 More Investigation on Frameworks

This thesis focused on enterprise integration patterns and their support by some open source frameworks in the EAI domain. Patterns are important for designing and implementing good and extensible integration projects. However, patterns are only one aspect when building EAI solutions for organizations. Other aspects like transactions, security, scalability, etc. have not been considered in this thesis but are an important criteria when evaluating different EAI frameworks. All four frameworks considered in this thesis, offer some support for such capabilities but need further evaluation, which can be done as future work.

7.2.3 Find Architectural Best Practices for Location of Service Implemenations

All frameworks used in the thesis can be called an ESB. Especially ServiceMix and Mule have support for the requirements stated in section 2.6.3. One goal of an ESB is to decouple service consumers and service providers which can be done with Mule and ServiceMix. A goad of decoupling is that a client application does not contain much business function code anymore, but business logic functionalities are achieved by calling services. These are provided by any other application which the client may not even know. When bringing this concept to an extreme, a client application could only provide a user interface and all functionalities of the application are achieved by calling some service. This can be compared to host systems where terminals were used as user interfaces and all processing was done on a host system. Of course the difference between a host and a service oriented approach is that services are not located on one system only but are distributed across a network. But from a clients points of view the

behavior is the same. Thus the way gone in business applications was starting at host systems using a thin client over to rich client applications and are now moving back to thin clients with a SOA in the background again. The interesting question to be clarified here is which parts of business logic code should be left at the client application and which parts should be implemented as services and invoked by a client. What has to be found is a good balance between services which are reused by many clients and local business logic code. One aspect that has to be considered in this context are timing issues. Making a service call will always take longer than calling a local method. Thus a first statement could be to leave time critical business functions at the client and move others to shared services. But this is only one aspect that has to be considered. A future work in this context could be done to find out more points that have to be observed.

7.2.4 Implementing Scenarios Using a Business Process Modeling Approach

The scenarios implemented in the practical part all used messaging approaches for implementing business task. Simple events were exchanged as messages for triggering some functionality of participating applications. Another approach for implementing all scenarios would be to model them as business processes and implement and execute them using a business process engine like jBPM or BPEL. It would be interesting to see, a comparison of both approaches in regards to enterprise integration patterns.

7.2.5 Implementing Scenarios Using a SCA & SDO Approach

Another approach that can be used for implementing some scenarios is using SCA and SDO. Both specification were introduced in chapter 4 but were not considered anymore. However like BPM, it would be interesting to see a comparison between SCA and SDO and the messaging approach used in this work. The first step could be to find a data model and describe it with SDO. Afterwards a composition of exposed services can be achieved by using SCA. This can be done in a future practical or thesis.

8 Summary

This final chapter will summarize the thesis in a very short manner and give all major points which were covered.
The thesis started with a short introduction to enterprise integration application, including various types and approaches for EAI. Furthermore enterprise integration patterns were introduced and their usefulness in integration projects was discussed.
After that four open source frameworks which are settled in the EAI domain were introduced and their support for integration patterns was examined. The frameworks are Apache ActiveMQ, Apache Camel, Apache ServiceMix and Codehaus Mule. In chapter 5 scenarios, which have been implemented during this thesis, were shown. In the scenarios one can see how integration patterns are supported and can be implemented with the introduced frameworks. The scenarios mainly show the usage of routing and transformation patterns. In chapter 6 tables which included all patterns and their support by frameworks are given. These tables can provide a first decision support for finding an appropriate framework with regard to integration patterns.
Furthermore categories which can make up an EAI solution were listed and discussed in this thesis. In chapter 6 it was also discussed which categories are provided by the used frameworks. These mappings can provide a second decision support for finding an appropriate framework.
The last technical decision support was given in chapter 6 in form of software engineering aspects. These included development support, maintenance possibilities and documentation of the four frameworks.
Thus three issues for finding an EAI framework were shown and discussed in this thesis. These can provide a fundament for finding an appropriate framework. However, as it was mentioned in chapter 7 not only these, but more points have to be considered in the decision making process.
Additionally to the technical research questions it was also analyzed if the usage of open source software can bring benefits to an enterprise. The bottom line was that investments in human capital of a company instead of investments in license costs have to be done. But for long time projects a total cost of ownership calculation will favor investments in own human capital against license costs. Another point that was mentioned in this context was that the agility of an enterprise gets higher if it can extend open source software to specific needs or market conditions what cannot be done when using commercial software.

Bibliography

[1] Alexander, Christopher ; Ishikawa, Sara ; Silverstein, Murray ; Jacobson, Max ; Fiksdahl-King, Ingrid ; Angel, Shlomo: *A pattern language*. New York : Oxford Univ. Press, 1977

[2] Buschmann, Frank ; Henney, Kevlin ; Schmidt, Douglas: *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing (Wiley Software Patterns Series)*. John Wiley & Sons, 2007

[3] Buschmann, Frank ; Henney, Kevlin ; Schmidt, Douglas: *Pattern Oriented Software Architecture: On Patterns and Pattern Languages (Wiley Software Patterns Series)*. John Wiley & Sons, 2007

[4] Buschmann, Frank ; Meunier, Regine ; Rohnert, Hans ; Sommerlad, Peter ; Stal, Michael: *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., 1996

[5] Chappell, David: *Introducing SCA*. July 2007

[6] developerworks, IBM: *Standards and Web services*. http://www.ibm.com/developerworks/webservices/standards. – [Online; accessed 01-March-2008]

[7] Dubray, Jean-Jacques: *What does the term ESB actually mean?* http://www.infoq.com/news/2007/09/what-does-the-term-esb-mean. September 2007. – [Online; accessed 01-March-2008]

[8] Edwards, Mike: *Relationship of SCA and JBI*. http://www.osoa.org/display/Main/Relationship%20of%20SCA%20and%20JBI. March 2007. – [Online; accessed 11-March-2008]

[9] Fowler, Martin: *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002

[10] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995

[11] Gelernter, David: Generative communication in Linda. In: *ACM Trans. Program. Lang. Syst.* 7 (1985), Nr. 1, S. 80–112. – ISSN 0164-0925

[12] Gleghorn, Rodney: Enterprise Application Integration: A Manager's Perspective. In: *IT Professional* 7 (2005), Nr. 6, S. 17–23. – ISSN 1520-9202

[13] Hohpe, Gregor: *Programming Without a Call Stack - Event-driven Architectures*. http://www.eaipatterns.com. 2006. – [Online; accessed 01-March-2008]

[14] Hohpe, Gregor ; Woolf, Bobby: *Enterprise Integration Patterns*. Addison Wesley, 2004

[15] Java-Source.net: *Open Source Rule Engines in Java*. http://java-source.net/open-source/rule-engines. – [Online; accessed 12-March-2008]

[16] Kaye, Doug: *Loosely Coupled: The Missing Pieces of Web Services*. RDS Press, 2003. – ISBN 1881378241

[17] Kircher, Michael ; Jain, Prashant: *Pattern-Oriented Software Architecture: Patterns for Resource Management*. John Wiley & Sons, 2004

[18] Manes, Anne T.: *The Business Value of SOA*. http://www.infoq.com/presentations/anne-thomas-manes-business-soa. February 2008. – [Online; accessed 22-March-2008]

[19] Marechaux, Jean-Louis: *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]

[20] Martin, Daniel ; Wutke, Daniel ; Scheibler, Thorsten ; Leymann, Frank: An EAI Pattern-Based Comparison of Spaces and Messaging. In: *EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*. Washington, DC, USA : IEEE Computer Society, 2007, S. 511

[21] Group, Object M.: *BPMN: Business Process Modelling Notation Specification*. http://www.bpmn.org. 2006. – [Online; accessed 01-March-2008]

[22] Rademakers, Tijs ; Dirksen, Jos: *Open-Source ESBs in Action*. Manning Publications Co, 2007

[23] Richards, Mark: *The Role of the Enterprise Service Bus*. http://www.infoq.com/presentations/Enterprise-Service-Bus. October 2006. – [Online; accessed 01-March-2008]

[24] Schatten, Alexander: Service Komposition und Datenaustausch mit Apache Tuscany. In: *Infoweek.ch* (2008). – [in German]

[25] Schmidt, Douglas C. ; Rohnert, Hans ; Stal, Michael ; Schultz, Dieter: *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. New York, NY, USA : John Wiley & Sons, Inc., 2000

[26] Schulte, Roy W.: *The Growing Role of Events in Enterprise Applications*. http://www.gartner.com/DisplayDocument?doc_cd=116129. 2003. – [Online; accessed 01-March-2008]

[27] Schumann, Rune ; Bjornstad, Rune P.: *Using SEDA to Ensure Service Availability*. http://www.infoq.com/articles/SEDA-Mule. – [Online; accessed 01-March-2008]

[28] Strachan, James: *How does ServiceMix compare to Tuscany or SCA*. http://servicemix.apache.org/how-does-servicemix-compare-to-tuscany-or-sca.html. March 2007. – [Online; accessed 11-March-2008]

[29] Ten-Hove, Ron ; Walker, Peter: *Java Business Integration (JBI), final release*. http://jcp.org/en/jsr/detail?id=208. August 2005. – [Online; accessed 10-March-2008]

[30] Umapathy, Karthikeyan ; Purao, Sandeep: Exploring Alternatives for Representing and Accessing Design Knowledge About Enterprise Integration. In: *ER* Bd. 4801, Springer, 2007, S. 470–484. – ISBN 978-3-540-75562-3

[31] Vinoski, Steve: Java Business Integration. In: *IEEE Internet Computing* 9 (July-Aug. 2005), Nr. 4, S. 89–91. – ISSN 1089-7801

[32] Vinoski, Steve: Integration with Web services. In: *IEEE Internet Computing* 7 (Nov.-Dec. 2003), Nr. 6, S. 75–77. – ISSN 1089-7801

[33] Weiss, Aaron: Real World Open Source: The TCO Question. In: *ServerWatch* (2005). – [Online; accessed 01-March-2008]

[34] Welsh, Matt ; Culler, David: Adaptive overload control for busy internet servers. In: *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*. Berkeley, CA, USA : USENIX Association, 2003, S. 4–4

[35] Welsh, Matt ; Culler, David E. ; Brewer, Eric A.: SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In: *Symposium on Operating Systems Principles*, 2001, S. 230–243

[36] Welsh, Matthew D.: *An architecture for highly concurrent, well-conditioned internet services*, University of California, Berkeley, Dissertation, 2002. – Chair-David Culler

[37] Zhu, Jun: Web services provide the power to integrate. In: *Power and Energy Magazine, IEEE* 1 (Nov.-Dec. 2003), Nr. 6, S. 40–49. – ISSN 1540-7977

List of Figures

List of Tables

Listings