Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart

Studienarbeit Nr. 2127

Realization of EAI Patterns with Apache Camel

Pascal Kolb

Studiengang:	Informatik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Thorsten Scheibler
begonnen am:	26.10.2007
beendet am:	26.04.2008
CR-Klassifikation	D.2.11, D.3, H.4.1

Table of Contents

Table of Listings ..... **vii**

1 Introduction **1**

1.1 Task Description *1*

1.2 Structure of this thesis *2*

2 Apache Camel Fundamentals **3**

2.1 Introduction into Apache Camel *3*

2.2 Apache Camel's Architecture *4*

2.2.1 Camel Components and Endpoints 4

2.2.2 Camel Exchange and Message 6

2.2.3 Camel Type Converter 6

2.2.4 Camel Processor 7

2.2.5 Camel Context 7

2.2.6 Camel Domain Specific Language / Fluent API 8

2.2.7 Camel Routes 8

2.3 A simple Camel example *9*

3 EAI Patterns in Apache Camel **11**

3.1 Messaging Channels *12*

3.1.1 Point-to-Point Channel 12

3.1.2 Publish-Subscribe Channel 13

3.1.3 Datatype Channel 13

3.1.4 Invalid Message Channel 13

3.1.5 Dead Letter Channel 13

3.1.6 Guaranteed Delivery 14

3.1.7 Channel Adapter 14

3.1.8 Messaging Bridge 15

3.1.9 Message Bus 15

3.2 Message Construction *15*

3.2.1 Command / Document / Event Message 16

3.2.2 Request-Reply 16

3.2.3 Return Address 17

3.2.4 Message Expiration 17

3.2.5 Correlation Identifier / Message Sequence / Format Indicator 18

3.3 Pipes and Filters *18*

3.4 Message Routing *20*
3.4.1 Content-Based Router 20
3.4.2 Message Filter 21
3.4.3 Dynamic Router 21
3.4.4 Recipient List 22
3.4.5 Splitter 23
3.4.6 Aggregator 24
3.4.7 Composed Message Processor / Scatter-Gather 27

3.5 Message Transformation *27*
3.5.1 Envelope Wrapper 27
3.5.2 Content Enricher 28
3.5.3 Content Filter 28
3.5.4 Claim Check 29
3.5.5 Normalizer 30

3.6 Messaging Endpoint *30*
3.6.1 Messaging Gateway 30
3.6.2 Messaging Mapper 31
3.6.3 Polling Consumer 31
3.6.4 Event-Driven Consumer 32
3.6.5 Competing Consumers 32
3.6.6 Message Dispatcher 33
3.6.7 Selective Consumer 34
3.6.8 Durable Subscriber 34
3.6.9 Idempotent Receiver 35
3.6.10 Service Activator 35

4 Implementation 37

4.1 Concept *37*

4.2 Technologies used *38*

4.3 The EAI-2-Camel Framework *39*
4.3.1 Helper Classes 40
4.3.2 Message Producer Template 41
4.3.3 Web Service Endpoint 41
4.3.4 Web Service Provider 42
4.3.5 Message Translator 43
4.3.6 Content Enricher 43
4.3.7 Recipient List 44
4.3.8 Aggregator 45

4.4 Generation of Apache Camel based Java-code *47*
4.4.1 Parameterization of the EAI patterns 47
4.4.2 Integration into the Eclipse based Editor 48
4.4.3 Generation of the Code 48
4.4.4 Required Libraries 49

4.5 Example Messaging System *49*

Bibliography **51**

Appendix **55**

A.1 Camel example using built-in Pipeline *55*
A.1.1 Java Code 55
A.1.2 Output 58
A.1.3 Discussion 60

Table of Listings

Listing 1 : Configuring Endpoints 5
Listing 2 : Simple Camel example 10
Listing 3 : Dead Letter Channel for non-messaging-system-components 14
Listing 4 : A Messaging Bridge in Apache Camel 15
Listing 5 : Request-Reply in Apache Camel 17
Listing 6 : Message Expiration for JMS-Endpoints in Apache Camel 17
Listing 7 : Apache Camel's approach to the Pipes-and-Filters pattern 19
Listing 8 : Pattern conform approach to the Pipes-and-Filters pattern 20
Listing 9 : Fixed route in Apache Camel 20
Listing 10 : Content-Based Router in Apache Camel 21
Listing 11 : Message Filter in Apache Camel 21
Listing 12 : Possible implementation of the Dynamic Router pattern 22
Listing 13 : Recipient List in Apache Camel 23
Listing 14 : Splitter in Apache Camel 24
Listing 15 : Apache Camel's Aggregator with a custom Aggregation Collection 26
Listing 16 : Message Translator in Apache Camel 27
Listing 17 : Possible Content Enricher implementation in Apache Camel 28
Listing 18 : Content filtering using an XPath expression in Apache Camel 29
Listing 19 : Possible Claim Check implementation in Apache Camel 29
Listing 20 : Normalizer in Apache Camel 30
Listing 21 : Polling Consumer in Apache Camel 31
Listing 22 : Event-Driven Consumer in Apache Camel 32
Listing 23 : Competing Consumer using the JMS-Component in Apache Camel 32
Listing 24 : Message Dispatcher using a Content-Based Router 34
Listing 25 : Selective Consumer using Apache Camel's JMS-Component 34
Listing 26 : Durable Subscriber using Apache Camel's JMS-Component 35
Listing 27 : Apache Camel's Idempotent Receiver 35
Listing 28 : The Service Activator pattern in Apache Camel 36
Listing 29 : Using the Message Producer Template of the EAI-2-Camel Framework 41
Listing 30 : Retrieving the data to enrich in an XSLT-stylesheet via a special URL 44
Listing 31 : Retrieving the data to enrich in an XSLT-stylesheet via an XSLT parameter 44

1 Introduction

Enterprise application systems rarely live in isolation as creating a single huge application that runs a complete business is almost impossible. Thus enterprises typically run hundreds or even thousands of different application systems. Because most of these application systems typically need to work together to support common business processes and to share data across application systems, these application systems need to be integrated. Enterprise application integration needs to provide efficient, reliable and secure data exchange between multiple enterprise applications.

An important aspect of application integration is to keep the integrated application system independent. This is important in order to enable the different application systems to evolve without affecting each other and thus making the overall solution more tolerant to changes. This is often referred to as Loose Coupling. Loose Coupling can be achieved by using asynchronous communication, for example by using messaging.

Because integrating enterprise applications is a challenging topic some frequently recurring problems and their solution has been described in form of patterns. Patterns are a technique to document expert knowledge and experience. The book "Enterprise Integration Patterns" by Gregor Hohpe and Booby Woolf [HW03] describes a collection of patterns residing in the domain of enterprise application integration using messaging.

The main objective of this work is to analyze the implementation of these Enterprise Integration Patterns in Apache Camel [Cam].

1.1 Task Description

In this work the Apache Camel platform shall be evaluated regarding the correct implementation of the Enterprise Integration Patterns. Missing patterns shall be supplemented and the platform shall be extended by appropriate mechanisms if required. Thereby an application programming interface (API) shall be developed, that enables the modeling of integration scenarios based on the enterprise integration patterns. This API shall also enable the generation of executable code. The results of this work shall be demonstrated by a prototype and a concrete example.

In the work of [Dru07] a tool for the graphical modeling of integration scenarios has already been developed. This tool enables the generation of WS-BPEL [AAA+07] processes from the modeled integration scenario. The solution developed as part of this work is based on this tool. The tool is thereby extended with the ability to generate executable Java-code.

1.2 Structure of this thesis

This thesis is structured as follows:

In *chapter 2* the work introduces to the Apache Camel platform and summarizes its architecture. It is important to understand Apache Camel's architecture as this is a requirement to understand the evaluation of Apache Camel and its implementation of the enterprise integration patterns.

In *chapter 3* the different enterprise integration patterns described in Hohpe's and Woolf's book are evaluated regarding their implementation in Apache Camel. The evaluation of the patterns is structured in groups similar to their occurrence in Hohpe's and Woolf's book.

In *chapter 4* the results from *chapter 3* are used to extend the modeling tool which was developed as part of the work from [Dru07]. The tool is extended with the ability to create executable Java code from the modeled integration scenario.

2 Apache Camel Fundamentals

2.1 Introduction to Apache Camel

This chapter introduces to Apache Camel [Cam]. The Apache Camel version discussed in this work and used for the implementation described in chapter 4 below is Apache Camel release 1.2.0, the most recent release to date. The Apache Camel project is a relatively young project – it was started in early 2007 – and is thus lacking good documentation. For this reason, this chapter introduces to Camel's architecture so the reader doesn't need to crawl through the projects little and unstructured documentation or Camel's API specification, to be able to follow the evaluation of Camel regarding its implementation of the Enterprise Integration Patterns [HW03] in chapter 3 below.

Apache Camel is a routing and mediation engine which implements the Enterprise Integration Patterns from [HW03]. It provides a framework which can be used to integrate different applications using the Enterprise Integration Patterns and it can be used as a routing and mediation engine for the following Apache projects:

- Apache ActiveMQ, an open source message broker [Act]
- Apache CXF, an open source web service framework (JAX-WS) [Cxf]
- Apache MINA, a networking framework [Min]
- Apache ServiceMix, an open source ESB and JBI container [Ser]

The Camel engine can run as a standalone Java-Application or can be deployed as a JBI component into Apache ServiceMix. Since ActiveMQ 5.0 [Act] it is also possible to run the Camel Engine in ActiveMQ's broker process. For details about the integration of Camel in ServiceMix and ActiveMQ see the respective project pages' documentation. To configure the routing and mediation rules Camel supports two approaches. One approach is to configure Camel via Spring [Spr] based XML Configuration files. The other approach is to configure Camel via Camel's Java based Domain Specific Language (DSL), where the configuration is done by chained Java method calls in a fashion that the produced Java code is easily human readable, thus Camel's DSL is also called "Fluent API" (See Listing 2 below for an example of the Fluent API). With either approach the configuration is facilitated trough smart completion of the routing rules by either the XML or the Java editor. Due to the time constraints of this work, and since the Apache Camel developers recommend using the Java DSL in preference over the Spring based XML configuration - because it is more expressive and offers maximum smart completion by the IDE – the focus of this work is on the configuration of Camel via its Java Domain Specific Language.

2.2 Apache Camel's Architecture

Figure 1 below gives an overview of Camel's architecture. The main building blocks of Camel are *Components*, *Endpoints* and *Processors*. *Endpoints* and *Processors* are wired together to *Routes*. *Routes* represent the routing and mediation rules and are added to a *Camel Context* which is a container for the whole configuration. The following paragraphs explain these building blocks in more detail.

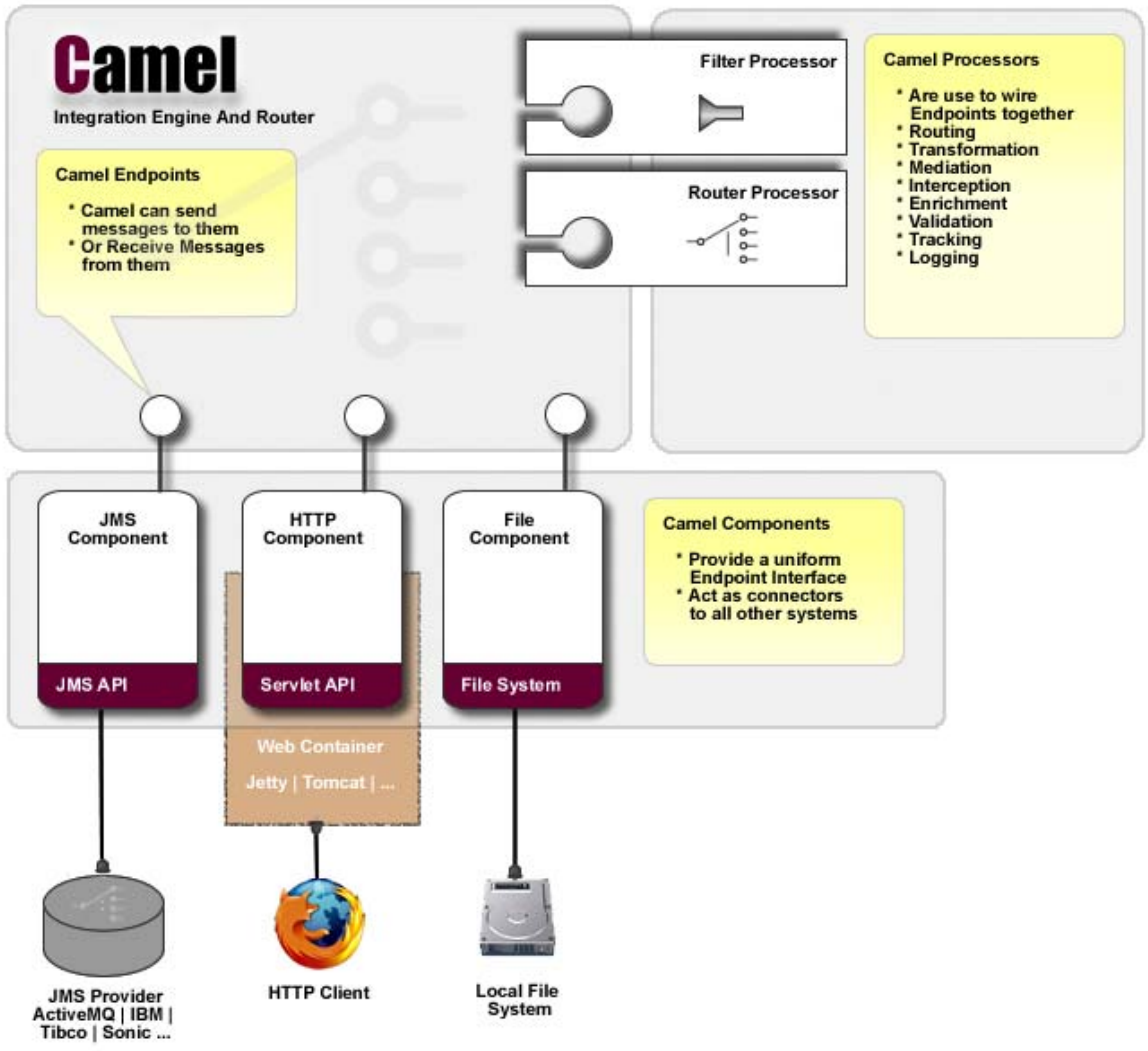


Figure 1: Apache Camel's Architecture [Cam]

2.2.1 Camel Components and Endpoints

To be able to integrate with different applications on different platforms, Apache Camel supports multiple so called *Components*, where each of these *Components* enables Apache Camel to send

and receive messages using a certain protocol or middleware for example a message broker. Some common Components are:

- The *JMS-Component* which enables Apache Camel to connect to any JMS provider.
- The *File-Component* which enables Camel to read files from a file system or to write messages as files to a file system.
- The *Direct-Component* where a message producer directly, i.e. synchronously, invokes the message consumers.
- The *SEDA-Component* which implements in-memory blocking queues to be able to produce and consume messages asynchronously.

For a list of the currently supported components refer to Camel's project page [Cam]. A *Component* acts as connector to other systems or middleware and is essentially a factory for Camel *Endpoints*, to which messages can be sent and from which messages can be received. In that sense Camel *Endpoints* represent a channel as in the *Channel* pattern from [HW03]. A Camel *Endpoint* created by Camel's *JMS Component* for example would represent a JMS queue or topic. Camel uses URIs to identify *Endpoints* and lazily creates them using the appropriate *Component* if they don't already exist. This makes it simple to configure routes, as *Endpoints* don't have to be instantiated explicitly. The URI-scheme of the *Endpoint-URI* identifies the component which is responsible for creating the *Endpoint*. Camel auto-discovers *Components* by their scheme. Therefore a *Component* has to have a configuration file in the proper directory[1], which has to reside somewhere in the class path. Alternatively the *Component* can be registered explicitly with the *Camel Context*. *Components* with a complex configuration are usually registered explicitly with the *Camel Context* as seen with the *JMS Component* ('`myComponent`') in Listing 2 below.

Components always implement the `Component`[2] interface and are usually not used directly to create *Endpoint* instances. Instead the *Endpoint* instances are created implicitly when an URI is used in a *Route* or when an *Endpoint* for an URI is requested from the *Camel Context*. *Endpoints* always implement the `Endpoint`[3] interface and are used as interface for sending and receiving messages. They are preconfigured by the *Component* they are instantiated by, but can also be configured directly and used in routing rules instead of URIs like in Listing 1 below.

```
1  SomeEndpointClass myEndpoint
2      = (SomeEndpointClass) camelContext.getEndpoint("myEndpointURI");
3  myEndpoint.setSomething("aValue");
4
5  from(myEndpoint).to("anotherEndpointURI")
```

Listing 1: Configuring Endpoints

[1] `META-INF/services/org/apache/camel/component/`
[2] `org.apache.camel.Component`
[3] `org.apache.camel.Endpoint`

The sending and receiving of messages to and from *Endpoints* is done via `Producer`[1] and `Consumer`[2] instances created by an *Endpoint*. A *Consumer* consumes messages from an Endpoint's underlying channel. Camel supports the *Event-driven Consumer* pattern by using the `createConsumer(Processor)` method, which creates a `Consumer` instance, invoking the supplied `Processor`[3] on message arrival. Camel also supports the *Polling Consumer* pattern by using the `createPollingConsumer()` method, which creates a `PollingConsumer`[4] that can be polled for new messages in a blocking or in a non-blocking way.

A *Producer* adds messages to an *Endpoint's* underlying channel and is implemented in Camel trough the `Producer`[3] interface which offers methods for the creation of *Exchanges* (see 2.2.2 *Camel Exchange and Message* below) and a method for the sending of these *Exchanges*. It is important to note that the sending of *Exchanges* is done via the `process()` method of the `Processor`[3] interface. This allows the usage of a *Producer* as a *Processor* which processes messages by adding them to the *Endpoint's* underlying channel and which also sets the *Exchange's* output message to the received response message in case of in/out exchange patterns, like request/reply exchanges for example.

2.2.2 Camel Exchange and Message

To support various message exchange patterns, like request/reply message exchanges or event messages, Camel offers the `Exchange`[5] interface which basically contains an input, output and a fault message, depending on the `ExchangePattern`[6]. An Exchange can also be – and usually is – extended by a *Component* to expose the underlying transport semantics of that *Component*.

The input, output or fault messages contained in an *Exchange* implement the `Message`[7] interface representing the *Message* pattern and basically containing a message body and a collection of message headers. The message body can contain any Java object, but for almost all *Components* the message body needs to contain a serializable[8] object, as the message will leave the sender's address space or even the machine it originates from.

2.2.3 Camel Type Converter

Because some *Endpoints* always return messages of a certain type – the `FileEndpoint`[9] for example always returns an instance of `File`[10] as message body – and because it's quite common to

[1] `org.apache.camel.Producer`
[2] `org.apache.camel.Consumer`
[3] `org.apache.camel.Processor`
[4] `org.apache.camel.PollingConsumer`
[5] `org.apache.camel.Exchange`
[6] `org.apache.camel.ExchangePattern`
[7] `org.apache.camel.Message`
[8] i.e. implement the `java.io.Serializable` interface
[9] `org.apache.camel.component.file.FileEndpoint`
[10] `java.io.File`

need to convert a message body to and from certain types, Camel offers an extensible type conversion strategy via so called *Type Converters*.

Camel provides the method `getBody(Class)` in the `Message`[1] interface, which returns the message payload converted to the given type. The conversion is done by the implementation of the `TypeConverter`[2] interface which is associated with the current `CamelContext`[3]. Camel's default implementation of the converter is the `DefaultTypeConverter`[4] class which discovers type converters by searching trough defined[5] packages for classes and methods which are annotated with the `@Converter`[6] annotation. To support common conversions, Camel provides already some built-in type converters.

2.2.4 Camel Processor

Another building block of Apache Camel, besides the *Component*, is the *Processor* which, as the name indicates, processes messages. A *Processor* is handed over an *Exchange* (i.e. a message) by the Camel engine, and after processing it, Camel sends the *Exchange* to a channel or invokes the following *Processor*. This seems to follow the *Pipes-and-Filters* pattern, where the pipes are the channels represented by the Camel *Endpoints* and the filters are the Camel *Processors*. And indeed, if applied in a proper way, it implements the *Pipes-and-Filters* pattern. But it is also possible to chain Camel *Processors* directly where one *Processor* directly invokes the next, which violates the *Pipes-and-Filters* pattern. This is discussed in detail in chapter 3.3 below. A Camel *Processor* can also perform routing by deciding where to send a message to and thus can also be used to implement the *Routing* patterns.

Camel *Processors* are *Event-Driven Consumers* (as in [HW03]) and implement the `Processor`[7] interface. Most patterns implemented in Apache Camel are *Event-Driven Consumers* as this enables the Camel container to manage pooling and threading. One of the few pattern implementations in Camel which are not *Event-Driven Consumers* is the *Aggregator* pattern which implements the *Polling Consumer* pattern.

2.2.5 Camel Context

A *Camel Context* is the container for a specific integration scenario and represents that scenarios routing rule base. It implements the `CamelContext`[8] interface and contains a collection of all *Component* instances used in the integration scenario and all specified Camel *Routes* of the inte-

[1] `org.apache.camel.Message`
[2] `org.apache.camel.TypeConverter`
[3] `org.apache.camel.CamelContext`
[4] `org.apache.camel.impl.converter.DefaultTypeConverter`
[5] The packages to be searched have to be defined in the following file:
`META-INF/services/org/apache/camel/TypeConverter`
[6] `org.apache.camel.Converter`
[7] `org.apache.camel.Processor`
[8] `org.apache.camel.CamelContext`

gration scenario. The *Routes* define the processing used on inbound message exchanges from a specific *Endpoint*. To add *Routes* to a *Camel Context* using the Java DSL aka Fluent API, Camel provides the `RouteBuilder`[1] class, which can be used, as in the example in listing 2 below, to create *Routes*.

Camel provides a simple `Service`[2] interface for managing Camel's lifecycle. The interface provides two methods, the `start()` and the `stop()` method. Most *Components* and *Endpoints* implement this interface and it's the *Camel Context's* task to start all the necessary services when the Camel engine is started. The *Camel Context* itself implements the `Service` interface which is used to start the Camel engine as seen in the example in listing 2 below.

Camel also supports a registry plug-in strategy which allows Camel to retrieve Camel *Processors* or *Endpoints* for example from a registry. The `Registry`[3] is associated with the *Camel Context* and can be replaced by any registry provider implementing this interface. Camel offers built-in support for JNDI as a registry.

2.2.6 Camel Domain Specific Language / Fluent API

As mentioned before, Camel offers a Java Domain Specific Language (DSL) also known as Fluent API which allows the writing of routing rules in an easily human readable format supported by the Java development environment's smart completion capabilities. Camel's DSL is very expressive as it allows to mix own code with the DSL by using custom *Processors*, *Expressions* and *Predicates* as it will be demonstrated later.

Expressions are used in Camel's DSL to configure dynamic rules, where they offer evaluation of message exchanges. The *Recipient List* Pattern for instance uses an *Expression* for getting the recipient list contained in a message. Camel comes with already built-in support for a variety of expression languages like XPath [CD99] for example, which is implemented in the `XPathBuilder`[4] class.

In addition to *Expressions*, Camel offers also *Predicates* in its DSL to check if a message exchange has a certain property. As for the *Expressions*, Camel also offers built-in support for predicate languages, one of them is XPath implemented in the `XPathBuilder` class. *Message Filters*, for example, use *Predicates* to decide whether a message should be forwarded or filtered.

2.2.7 Camel Routes

Routes are rules to configure how Camel processes incoming messages. A *Route* is a chain of Camel *Processors*, implemented by the `Pipeline`[5] class, where the result of one *Processor* is the

[1] `org.apache.camel.builder.RouteBuilder`
[2] `org.apache.camel.Service`
[3] `org.apache.camel.spi.Registry`
[4] `org.apache.camel.builder.xml.XPathBuilder`
[5] `org.apache.camel.processor.Pipeline`

input of the following *Processor* and so on till the last *Processor* is reached and the `Pipeline` returns from its `process(…)` method invocation. The first element of a route is always a *Consumer* of an *Endpoint* which invokes the *Pipeline*. It is important to know that the *Pipeline* returns only after the whole route – i.e. the whole processing chain – has been processed, which means that the *Consumer* of an *Endpoint* is only done processing an *Exchange* after the message has been processed by the whole route. This is important first in order to process request-response message exchanges which are returned to the consumer. And second to ensure that a message is only successfully consumed after the processing by the last processor, which is usually an *Endpoint's Producer*, is finished.

2.3 A simple Camel example

The code in listing 2 below shows a simple example of Apache Camel, where messages are received from the JMS queue `'test.queue'`, processed by the Processor `'myProcessor'` and put in the `'testFolder'`. As mentioned before the `CamelContext` is the container of our integration scenario, so our example starts with creating a `CamelContext` (line 1), then adds a *Component* to it (line 9), then adds *Routes* to it (line 11) and finally the `CamelContext` is executed for 10 minutes (lines 36-38).

In this example two *Components* are used: the `'myComponent'`-*Component* which is explicitly registered with the *Camel Context* and the `'file'`-*Component* which is auto-discovered. One can also see how *Endpoints* are referenced by their URIs and created implicitly by Camel. Lines 29 to 31 demonstrate how the usage of Camel's DSL aka Fluent API creates human readable and easy to understand code. The *Processor* in this example is defined in lines 16 to 25 and doesn't alter the messages it processes, but simply prints the message bodies to the console. The *Processor's* custom implementation also shows how the type conversion mentioned above is used to get a `String` representation of the message body (line 22).

```
CamelContext context = new DefaultCamelContext();

// explicitly add a JMS Component which connects to the
// ActiveMQ server 'myServer' on port 61616:
ConnectionFactory connectionFactory
    = new ActiveMQConnectionFactory("tcp://myServer:61616");
Component myComponent
    = JmsComponent.jmsComponentAutoAcknowledge(connectionFactory);
context.addComponent("myComponent", myComponent);

context.addRoutes(new RouteBuilder() {

    public void configure() {

        // create a message processor:
        Processor myProcessor = new Processor() {

            public void process(Exchange e) {
                // get the inbound request message:
                Message inboundMessage = e.getIn();
                System.out.println("Processing message: "
                          + inboundMessage.getBody(String.class));
            }

        };

        // take messages from the JMS-Queue 'test.queue', process
        // them and put them into the folder 'testFolder':
        from("myComponent:queue:test.queue")
            .process(myProcessor)
            .to("file://testFolder");
    }

});

context.start();
Thread.sleep(600000); // wait for 10 minutes
context.stop();
```

Listing 2: Simple Camel example

3 EAI Patterns in Apache Camel

In this chapter Apache Camel is evaluated regarding which of the Enterprise Integration Patterns [HW03] are implemented in Camel, to what extent they are implemented and if the implementation is correct – according to their definition in Hohpe's and Woolf's book on Enterprise Integration Patterns [HW03].

Because the Camel Project is relatively young and thus has only a rudimentary and incomplete documentation, the evaluation has mostly been done by reading Camel's API specification in form of JavaDocs and by reading through Camel's implementation itself. Also it has to be mentioned that the evaluation is based on the latest Camel release at the time of this work, which is release 1.2.0. This release contains several bugs and is also missing functionality that is implied by the specification. This fact reflects in the following sub-chapters in form of comments to Camel's implementation. At the time of the creation of this work the developing of release 1.3.0 was in progress where some bugs mentioned in this work are already fixed. This is ignored for the evaluation of the patterns in this work, as the evaluation needed to reflect one defined version of Camel for which release 1.2.0 was chosen, as the most recent release. Further, it is important to mention that Camel's documentation doesn't refer to a specific release of Apache Camel. Rather the documentation is updated continuously without mentioning which release of Camel is actually documented. This is even true for Camel's API documentation available online, but the JavaDoc documentation can also be generated from the source code available online by oneself, resulting in a consistent API specification.

The evaluation of the different patterns is following the order of their appearance in the book of Hohpe and Woolf [HW03]. The only difference is that the more generic patterns that form their own sub-chapter containing their several sub-patterns, like the *Message Channel*, the *Message*, the *Pipes-and-Filters*, the *Message Router*, the *Message Translator* and the *Message Endpoint* are not discussed in an introductory chapter like in the case of Hohpe's and Woolf's book, but in their respective subchapter.

Due to the limited time frame of this work not all patterns could be evaluated. The patterns that have been skipped have been chosen because they were either good documented in Camel's documentation or they were advanced patterns of their respective category. Among the patterns that are not evaluated are also all *System Management* patterns because they are advanced patterns that describe a different aspect of a messaging solution and thus can be evaluated separately. The following patterns are not evaluated either:

- The *Resequencer* pattern which is implemented in Camel and has a comparably good documentation on the project website.
- The *Routing Slip* pattern which is not implemented at all in Apache Camel.

- The *Process Manager* pattern which goes beyond the scope of Apache Camel and is covered by workflow engines like the one developed in the Apache ODE project [Ode].
- The *Message Broker* which is an architectural pattern that is opposed to the *Pipes-and-Filters* architecture and is the architectural pattern, the graphical editor and BPEL generator of the diploma thesis from Druckenmüller on Parameterization of EAI Patterns [Dru07], which will be extended in this work, is based on.
- The *Canonical Data Model* pattern which is not implementable in Apache Camel, but is rather a pattern that should be followed by the architect of a messaging solution.
- The *Transactional Client* pattern which is implemented in Camel using Spring transactions and is a complex pattern that needs a deep evaluation

3.1 Messaging Channels

The *Message Channel* pattern is not explicitly implemented in Apache Camel instead the message channels are provided by the underlying middleware, the operating system or are implemented completely in Camel depending on the kind of channel. What Camel does is providing a common interface to address and access channels using *Endpoints*, which are usually identified and addressed by their URI. As explained in chapter 2.2 Camel supports different *Components* for integrating with different systems. These *Components* create *Endpoints* which all implement the `Endpoint`[1] interface, providing a unified interface for accessing the channels of the underlying infrastructure. This can be a channel, implemented by a JMS queue and accessed through the `JmsEndpoint`[2], as well as a channel implemented as a directory in a file system, accessed through the `FileEndpoint`[3].

3.1.1 Point-to-Point Channel

As mentioned above, the *Messaging Channels* are not explicitly implemented in Apache Camel but are provided by *Components*. A common *Component* that enables Apache Camel to use many different message queuing systems via JMS [HBS+] is the *JMS Component*. The *JMS Component* provides *Point-to-Point Channels* as `Endpoints`[1] on JMS-Queues. These *Endpoints* are referenced via URIs of the following format:

[1] `org.apache.camel.Endpoint`
[2] `org.apache.camel.component.jms.JmsEndpoint`
[3] `org.apache.camel.component.file.FileEndpoint`

```
"jms:queue:myQueueNameA"
```

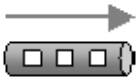
Another *Component* that offers *Point-to-Point Channels* is Camel's *SEDA Component* which offers access to in-memory queues.

3.1.2 Publish-Subscribe Channel

Like for the *Point-to-Point Channel* also the *Publish-Subscribe Channel* is provided by Camel *Components*. The *JMS Component* provides *Publish-Subscribe Channels* as `Endpoints`[1] on JMS-Topics [HBS+]. The *Direct-Component* also implements a *Publish-Subscribe Channel*, as all *Consumers* registered on a *Direct-Endpoint* receive all messages sent to that *Endpoint*. The JMS-Topics are addressed using *Endpoint*-URIs of the following format:

```
"jms:topic:myTopicNameA"
```

3.1.3 Datatype Channel



The *Datatype Channel* is a conceptual pattern which advices to use separate channels for messages with separate data types. This pattern is not implemented in Apache Camel, but of course Camel can be used to implement the pattern using a separate channel for each message type.

3.1.4 Invalid Message Channel

Like the *Datatype Channel*, the *Invalid Message Channel* is a conceptual pattern. It advices message receivers to put messages they can't process on a dedicated channel: the *Invalid Message Channel*. It can be implemented by using any channel available through Camel's *Components* to which `Processors`[2] send their messages they can't process.

3.1.5 Dead Letter Channel

The *Dead Letter Channel* pattern can generally not be implemented in Apache Camel as this is a channel where the underlying messaging system shall put messages it can't deliver. But as Apache

[1] `org.apache.camel.Endpoint`
[2] `org.apache.camel.Processor`

Camel can work with several different *Components* whereof most of them don't use an underlying messaging system, it can make sense to use an error handler that Apache Camel provides which is called `DeadLetterChannel`[1] and that can be used to handle errors which occur while a `Processor` is processing a message. In fact Camel's `DeadLetterChannel` is the default error handler and is configured not to send messages to a particular *Dead-Letter Channel*, but to try to redeliver the messages six times, waiting for one second between each attempt.

The `DeadLetterChannel` is particularly useful to handle errors of `Producers`[2] that don't use an underlying messaging system like the `FileProducer`[3] which could raise an exception because it can't deliver a message into a folder because of missing write permissions. Listing 3 below shows how to set up a `DeadLetterChannel`Fehler! Textmarke nicht definiert. and configure a redelivery policy that retries five times to redeliver messages while increasing the delay between the delivery attempts exponentially, starting from 0.5 seconds.

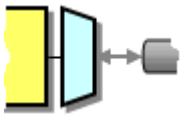
```
1  from("myComponent:endpointA")
2     .process(myCustomProcessor)
3     .errorHandler(
4        deadLetterChannel("myComponent:myDeadLetterChannel")
5           .maximumRedeliveries(5)
6           .initialRedeliveryDelay(500)
7           .useExponentialBackOff())
8     .to("file:/path/to/a/directory/without/writing/permission");
```

Listing 3: Dead Letter Channel for non-messaging-system-components

3.1.6 Guaranteed Delivery

The *Guaranteed Deliver* pattern is not implemented in Apache Camel either, but is provided by *Components* like the *JMS-Component* that uses persistent delivery by default, the *File-Component* that stores messages on a file-system and the *JPA-Component* which uses a database table as a message queue.

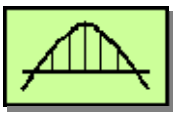
3.1.7 Channel Adapter



The *Channel Adapter* pattern is not implemented in Apache Camel, the contrary is the case: one can use Camel to implement a *Channel Adapter* for applications that are written in Java by writing code that sends messages to Camel *Endpoints*.

[1] `org.apache.camel.processor.DeadLetterChannel`
[2] `org.apache.camel.Producer`
[3] `org.apache.camel.component.file.FileProducer`

3.1.8 Messaging Bridge

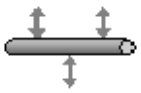


Apache Camel can also be used to implement a *Messaging Bridge*, as Camel supports many different *Components* and can also use the *JMS-Component* to connect to most messaging systems. A *Messaging Bridge* can be implemented by creating simple routing rules that connect the different channels of the different messaging systems or even channels which are not based on messaging systems. Listing 4 below shows how the rules for such a messaging bridge would look like. If necessary one could even add some custom `Processors`[1] to the routes, that transform the messages to fit the requirements of the receiving messaging infrastructure.

```
1 from("myJmsComponentA:endpointX").to("myJMSComponentB:endpointX");
2 from("myJmsComponentA:endpointY").to("myJMSComponentB:endpointY");
3 from("myJmsComponentA:endpointZ").to("file:/my/path/Z");
```

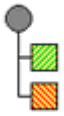
Listing 4: A Messaging Bridge in Apache Camel

3.1.9 Message Bus



Camel doesn't implement the *Message Bus* pattern, rather it can be used to implement a message bus using an underlying messaging system as common communication infrastructure and implementing the message routing and message transformation using the various patterns which are implemented in Apache Camel. As mentioned above Camel can also be used to implement *Channel Adapters* for those application systems that are implemented in Java.

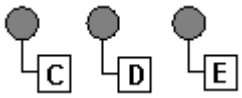
3.2 Message Construction



The *Message* pattern is implemented in Camel by the `Message`[2] interface. In Camel messages are always wrapped by the `Exchange`[3] interface to offer support for different message exchange patterns like request-reply or one-way messaging. The simplest exchange pattern which is also often used is the `InOnly`[4] pattern for one way messaging. In this case the message can be obtained by the `getIn()` method of the `Exchange` interface.

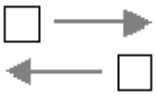
[1] `org.apache.camel.Processor`
[2] `org.apache.camel.Message`
[3] `org.apache.camel.Exchange`
[4] `org.apache.camel.ExchangePattern.InOnly`

3.2.1 Command / Document / Event Message



As Hohpe and Woolf indicate in their book [HW03] there are no special message types for a *Command Message*, *Document Message* or an *Event Message*. Rather the content of a message makes it what it is, i.e. a *Command Message* is a message that contains a command. Thus Camel doesn't implement these patterns; it is rather the architect's liability to create and use messages according to the patterns.

3.2.2 Request-Reply



The *Request-Reply* pattern is theoretically implemented by Camel's `Exchange`[1] interface which is an abstraction of a message exchange like a request and its corresponding reply message. To use the *Request-Reply* pattern the `Exchange`, which contains the request message, has to be created using Camel's `InOut`[2] exchange pattern. Usually this is done by invoking the `Endpoints`[3] `createExchange(ExchangePattern.InOut)` method. Exchanges, created using the `InOut` exchange pattern should then contain the request message as the `Exchange's In`[4] message and the receiver then writes its reply message to the `Exchange's Out`[5] message.

At the time of the creation of this work almost no *Component* supports the `InOut`[2] exchange pattern and thus the *Request-Reply* pattern. The *JMS-Component* doesn't implement the `InOut` exchange pattern, neither does the *SEDA Component*. Then again the *Direct Component* does implement it, as it synchronously passes the `Exchange` instance to the receiver. To add support for the `InOut` exchange pattern to a *Component*'s `Producer`[6], that sends a message and would receive the reply asynchronously, the Producer has to add a *Return Address* (see 3.2.3 Return Address) to the message's header and listen on the corresponding channel. In case of the *JMS-Component* this could be implemented using the `JMSReplyTo` header specifying a temporary JMS-destination [HBS+] the reply is expected on. While waiting for the reply message the `Producer` has to block after sending the request message. To support also asynchronous callback as described by Hohpe and Woolf [HW03] a Producer could also implement the `AsyncProcessor`[7] interface which offers such a callback mechanism.

Listing 5 below demonstrates how the Request-Reply pattern is used in Apache Camel. As mentioned before this example only works with *Components* that support the `InOut` exchange pattern.

---

[1] `org.apache.camel.Exchange`
[2] `org.apache.camel.ExchangePattern.InOut`
[3] `org.apache.camel.Endpoint`
[4] Via the `org.apache.camel.Exchange.getIn()` method
[5] Via the `org.apache.camel.Exchange.getOut()` method
[6] `org.apache.camel.Producer`
[7] `org.apache.camel.AsyncProcessor`

```
1 Endpoint myEndpoint = context.getEndpoint("myComponent:endpointA");
2 Producer myProducer = myEndpoint.createProducer();
3 Exchange myExchange = myEndpoint.createExchange(ExchangePattern.InOut);
4 Message request = myExchange.getIn();
5 request.setBody("my request body"); // create the request
6 myProducer.process(myExchange); // send the request
7 Message response = myExchange.getOut(); // get the response
```

Listing 5: Request-Reply in Apache Camel

3.2.3 Return Address

As mentioned above (Chapter 3.2.2) the Return Address is implicitly implemented by Apache Camel's Endpoint's `Producers`[1]. If message exchanges are sent using the `InOut`[2] exchange pattern which corresponds to a request-reply pair, then the Endpoint's `Producer` has to take care of adding a return address to the message header, listen on the channel corresponding to the return address and return the received reply message via the `Exchange`[3] interface.

3.2.4 Message Expiration

The *Message Expiration* pattern is not implemented in Apache Camel but may be provided by a *Component's* underlying messaging system. For example the *JMS-Component* provides message expiration which can be set either as default for all *Endpoints* of a *JMS-Component* via its `setTimeToLive(…)` method or per *Endpoint* via the *Endpoint-URI* as in listing 6 below.

To correctly implement the *Message Expiration* pattern the *Component's* `Consumer`[4] should also discard expired messages (or put them on the dead-letter queue) it received. This is currently not the case for Camel's *JMS-Component*.

```
....to("jms:queue:myDestination?timeToLive=5000");
```

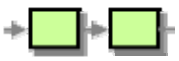
Listing 6: Message Expiration for JMS-Endpoints in Apache Camel

[1] `org.apache.camel.Producer`
[2] `org.apache.camel.ExchangePattern.InOut`
[3] `org.apache.camel.Exchange`
[4] `org.apache.camel.Consumer`

3.2.5 Correlation Identifier / Message Sequence / Format Indicator

The *Correlation Identifier*, *Message Sequence* and the *Format Indicator* pattern are not implemented in Apache Camel, but they can be implemented by setting appropriate headers on the producer side and evaluating them on the consumer side. Headers are name-value pairs, which are added to messages using the `setHeader(…)` method of the `Message`[1] interface.

3.3 Pipes and Filters



Camel supports the *Pipes-and-Filters* pattern using two different approaches, whereas the approach explained in Camel's documentation does not correctly implement the *Pipes-and-Filters* pattern. The approach according to Camel's documentation makes use of the `Pipeline`[2] class which is offered by Camel and which is a Camel *Processor* delegating the processing of message exchanges to a chain of sub processors. Weather the *Processors* in the *Pipeline* are processing messages as request-response, i.e. using the *Exchange's* out-message, or modify the original message, i.e. the in-message, doesn't matter, as Camel's *Pipeline* handles this accordingly.

Camel's Java DSL also supports the usage of the `Pipeline`[2] class via the `pipeline(…)` method as used in listing 7 below. In contrast to the `Pipeline` class which supports a list of `Processors`[3] as members, the Java DSL's `pipeline(…)` method supports only `Endpoints`[4] whose *Producers* are used as member *Processors* in the *Pipeline* (see chapter 2.2 Camel Components and Endpoints for details about *Producers*). But this is no drawback, as Camel's *Direct Component* can be used (as seen in listing 7 below) to invoke the *Filter Processors*. Camel's *Direct Component* is implemented such, that *Producers* on *Direct Endpoints* synchronously invoke the *Consumers* of the according *Endpoint*. Camel's documentation suggest exactly this implementation - using the `Pipeline` class in conjunction with *Direct Endpoints* as seen in listing 7 below - but this implements the *Pipes-and-Filters* pattern only in the broader sense as the pipe is only implemented as a synchronous method call and is thus lacking the concept of a queue which would decouple the processing of one filter and its predecessor / successor. Whether this is an in-memory queue or an entire message oriented middleware wouldn't matter for the correct implementation of this pattern. Tightly coupled like that, only one message can be processed by the pipeline at the same time instead of being able to process several messages in parallel, each message being in another stage. This could be compensated by configuring Camel to use several threads for executing the *Pipeline*. But this would still not offer the same semantics as queue based pipes would offer and which is intended by the *Pipes-and-Filters* pattern.

[1] `org.apache.camel.Message`
[2] `org.apache.camel.processor.Pipeline`
[3] `org.apache.camel.Processor`
[4] `org.apache.camel.Endpoint`

```
1  // create the routes for the filters:
2  from("direct:endpointFilter1").process(myFilter1Processor);
3  from("direct:endpointFilter2").process(myFilter2Processor);
4  from("direct:endpointFilter3").process(myFilter3Processor);
5
6  // create the pipeline route:
7  from("direct:endpointA").pipeline("direct:endpointFilter1",
8                                    "direct:endpointFilter2",
9                                    "direct:endpointFilter3",
10                                   "direct:endpointB");
```

Listing 7: Apache Camel's approach to the Pipes-and-Filters pattern

The intuitive approach of using an asynchronous *Component* like Camel's *SEDA Component* instead of a synchronous one like Camel's *Direct Component* doesn't result in the intended goal. This is because Camel's asynchronous *Components* are lacking the support of `InOut` message exchanges. Using Camel's current implementation of asynchronous Components, the processing by an asynchronous `Producer`[1] would only add the message to the *Endpoint's* channel and return without awaiting a response which should then be added to the *Exchange's* out-message. This would make the `Pipeline`[2] call the next `Processor`[3] with the unmodified message which in turn would only add the message to a channel. This results in a processing where neither the resulting message of one filter is passed to the next filter nor the filters are executed consecutively but possibly in parallel. This wouldn't have anything to do with the *Pipes-and-Filters* pattern anymore. In the appendix an example can be found where this is demonstrated (A.1 Camel example using built-in Pipeline).

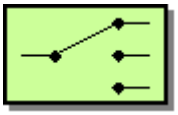
The second approach is simple and exactly implements the *Pipes-and-Filters* architecture. It just consists in using several routing rules, one for each filter in the *Pipes-and-Filters* chain. Only two things have to be taken into account using this approach. First the *Endpoints* used have to be implemented by an asynchronous *Component*, because otherwise the same drawback, as in the previous approach, would apply. Namely the lack of the decoupling of the filters by a concept like a queue would result in an implementation not strictly following the *Pipes-and-Filters* pattern. Second all *Processors* must read received messages from an *Endpoint* and send to another *Endpoint*, i.e. two *Processors* must never be chained directly together. Using this approach the intended pipeline from the example in listing 7 above would look as seen in listing 8 below.

[1] `org.apache.camel.Producer`
[2] `org.apache.camel.processor.Pipeline`
[3] `org.apache.camel.Processor`

```
1  // create the pipeline routes:
2  from("seda:endpointA").process(myFilter1Processor)
3      .to("seda:endpointFilter2");
4  from("seda:endpointFilter2").process(myFilter2Processor)
5      .to("seda:endpointFilter3");
6  from("seda:endpointFilter3").process(myFilter3Processor)
7      .to("seda:endpointB");
```

Listing 8: Pattern conform approach to the Pipes-and-Filters pattern

3.4 Message Routing

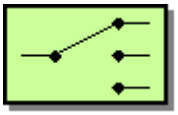


Camel supports most *Message Routing* patterns – they are described in the following subchapters. The most trivial router, which is a fixed router that forwards all messages to a fixed output channel, is implemented in Apache Camel by simple usage of the Java DSL as seen in listing 9 below. This fixed route can be useful if the decoupling of two *Endpoints* is required, for example to reserve the possibility to replace the simple route through a more sophisticated *Router* later, or if a *Messaging Bridge* is to be implemented.

```
from("myComponent:endpointA").to("myComponent:endpointB");
```

Listing 9: Fixed route in Apache Camel

3.4.1 Content-Based Router



Apache Camel supports the *Content-Based Router* pattern by the use of its `ChoiceProcessor`[1], which is implemented by managing a list of *Filter Processors*. A `FilterProcessor`[2] is a wrapper class for *Processors* and also contains a `Predicate`[3] that is used to evaluate *Exchanges*. The *Filter Processor* delegates the message processing to its wrapped *Processor* only if its *Predicate* applies to that message. The *Choice Processor* performs the routing by iterating trough its list of *Filter Processors* and choosing the first *Processor* in the list whose *Predicate* matches the message subject to routing. If none of the *Predicates* in the list matches, the *Choice Processor* delegates the processing to its associated *Otherwise Processor*.

Even though the *Choice Processor* uses a list of *Filter Processors* for its implementation, it correctly implements the *Content-Based Router* pattern as it just uses the *Filter Processor's Predicate* as a container for *Processor-Predicate* pairs. Its implementation performs predictive routing as required

[1] `org.apache.camel.processor.ChoiceProcessor`
[2] `org.apache.camel.processor.FilterProcessor`
[3] `org.apache.camel.Predicate`

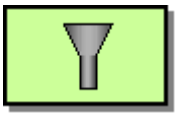
by the *Content-Based Router* pattern in contrast to a reactive filtering approach by the use of several *Message Filters*. The only critique to Camel's implementation is that it discards messages if none of the *Predicates* match and no *Otherwise Processor* is defined, instead of putting those messages on a *Dead Letter Channel* or handling them somehow.

Camel also supports the *Choice Processor* via its Java DSL as demonstrated in listing 10 below.

```
1  from("myComponent:endpointA").choice()
2      .when(myPredicate1).to("myComponent:endpointB")
3      .when(myPredicate2).to("myComponent:endpointC")
4      .when(myPredicate3).to("myComponent:endpointD")
5      .otherwise().to("myComponent:endpointE");
```

Listing 10: Content-Based Router in Apache Camel

3.4.2 Message Filter



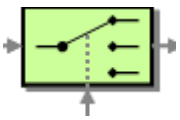
The *Message Filter* is implemented in Apache Camel using the `FilterProcessor`[1] class. As mentioned above the *Filter Processor* processes messages only if the associated `Predicate`[2] matches a message and discards it otherwise. The *Filter Processor* is also integrated in Camel's Java DSL using the `filter(…)` methods as demonstrated in listing 11 below.

Camel's *Filter Processor* supports only stateless filtering. If stateful filtering is required, as for the filtering of duplicate messages, the *Filter Processor* would have to be extended. But in this case the Java DSL would have to be extended too, to support the stateful *Message Filter* or it would have to be wired to the *Endpoints* without the use of the Java DSL.

```
6  from("myComponent:endpointA").filter(myFilterPredicate)
7                                .to("myComponent:endpointB");
```

Listing 11: Message Filter in Apache Camel

3.4.3 Dynamic Router



The *Dynamic Router* pattern is not implemented at all in Apache Camel. It can be implemented though by extending the `ChoiceProcessor`[3] class by adding a sub-processor for handling control messages and by adding a dynamic rule base which could be persisted for example by using the Java Persistence API (JPA). Even though such a `DynamicRouterProcessor` would not be sup-

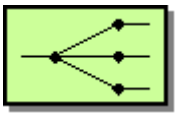
[1] `org.apache.camel.processor.FilterProcessor`
[2] `org.apache.camel.Predicate`
[3] `org.apache.camel.processor.ChoiceProcessor`

ported directly in Camel's Java DSL it could still be used in the DSL without having to modify the DSL. Listing 12 below demonstrates how the use of an extended `ChoiceProcessor`[1] class (named `DynamicRouterProcessor` in the listing), which persists its rule base via the JPA and uses a unique rule-base-name to identify its associated rule base, could look like. In the listing the constructor takes *Endpoint-URIs* as parameters for the control message channel and the otherwise channel to support easy usage in Camel's Java DSL. But it should be noted that for an implementation to be consistent with Camel's architecture, such a `DynamicRouterProcessor` class should also have a constructor that takes `Processor`[2] instances as parameters instead of *Endpoint-URIs*.

```
1 from("myComponent:endpointA").process(
2    new DynamicRouterProcessor("myComponent:controlEndpoint",
3                               "rulebase-test1",
4                               "myComponent:otherwiseEndpoint")
5 );
```

Listing 12: Possible implementation of the Dynamic Router pattern

3.4.4 Recipient List



Apache Camel supports the Recipient List Pattern by a simple class, the `RecipientList`[3] class, which implements the *Recipient List* by using an *Expression* which evaluates the message exchanges and outputs a list of recipients as *Endpoint-URIs*. The *Expression* can be created using any of Camel's built-in `Expression`[4] classes, for example by the `XPathExpression`[5] class, or can be a custom `Expression` class. The only constraint such an *Expression* has to fulfill is that it has to return the list in one of the following formats:

- As a `Collection`[6] of `Endpoints`[7] or of *String* representations of *Endpoint URIs*
- As an *Array* of `Endpoints` or of *String* representations of *Endpoint URIs*
- As a `NodeList`[8]
- As a single `Endpoint` or a single *String* representation of an *Endpoint URI*

Camel's *Recipient List* implementation iterates through the list gained from the *Expression* and sends a copy of the message exchange to each channel - represented by an *Endpoint* - in the list. The sending of the messages is implemented sequentially, depending on the *Endpoints*. A parallel implementation could be considered, which is not available in Apache Camel.

---

[1] `org.apache.camel.processor.ChoiceProcessor`
[2] `org.apache.camel.Processor`
[3] `org.apache.camel.processor.RecipientList`
[4] `org.apache.camel.Expression`
[5] `org.apache.camel.model.language.XPathExpression`
[6] `java.util.Collection`
[7] `org.apache.camel.Endpoint`
[8] `org.w3c.dom.NodeList`

In case the *Endpoint's Producers* modify the *Exchange* while processing, like it can be the case if the *Direct Component* is used, then the resulting *Exchange* of letting a *Recipient List* process a message, would be the result returned by the last Endpoint in the list. Even if this possibility is implemented, it wouldn't be of any use to process the result returned by a *Recipient List* and is thus usually not done.

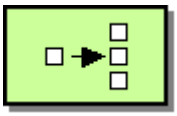
Camel's Java DSL also supports this pattern by using the `recipientList` method as demonstrated in listing 13 below. Note that for the reasons just mentioned it is not common to append the `to(…)` expression to the routing rule seen in listing 13 below.

```
from("myComponent:endpointA").recipientList(myExpression);
```

Listing 13: Recipient List in Apache Camel

A *Dynamic Recipient List* as described by [HW03] is not implemented in Apache Camel, but the existing *Recipient List* could be made dynamic by implementing a custom *Expression*, which uses a persisted rule base that can be modified through control messages to evaluate a message exchange - similar to the *Dynamic Router* described in chapter 3.4.3 above.

3.4.5 Splitter



The *Splitter* pattern is implemented in Apache Camel by the `Splitter`[1] class, using an `Expression`[2] to get a list of the split message parts which are then processed by the *Splitters* child *Processor*. Camel's `Splitter` also supports an `AggregationStrategy`[3] which it uses to aggregate the results of the processing of the different message parts. This feature is not part of the *Splitter* pattern from [HW03], but can be useful in conjunction with `InOut`[4] Exchanges where the Splitter returns the aggregated message as response to a message sent to the *Splitter*. This is particularly useful for a simple implementation of the *Scatter-Gather* pattern. As the actual splitting is done by an `Expression`, the `Splitter` class can be used to implement *Iterating Splitters* as well as *Static Splitters* (see [HW03]). The message parts split by the *Expression* are then all processed by the *Splitter's* child *Processor*, which means, in case of *Producers* as child *Processors*, that all message parts are added to the *Producers* corresponding *Message Channel*. Similar to the *Recipient List's Expression*, also the *Expression* of the *Splitter* needs to return a list of message parts which has to be of one of the following formats:

- A `Collection`[5] of message bodies (i.e. arbitrary *Java Objects*)
- An *Array* of message bodies

[1] `org.apache.camel.processor.Splitter`
[2] `org.apache.camel.Expression`
[3] `org.apache.camel.processor.aggregate.AggregationStrategy`
[4] `org.apache.camel.ExchangePattern.InOut`
[5] `java.util.Collection`

- A `NodeList`[1]
- A single message body

The *Splitter* copies all headers from the source message to the split message parts, thus also including possible correlation identifiers, and also adds two new headers:

- The `SPLIT_SIZE`[2] header which indicates into how many parts the original message was split.
- The `SPLIT_COUNTER`[3] header which is a sequence number ranging from zero to `(SPLIT_SIZE`[2] `- 1)`, and indicates the order of the message parts, originating from the same source message.

The fact that the message headers are copied and that the two mentioned headers are added, can simplify the task of the *Aggregator* if used in conjunction with the *Splitter* pattern to implement the *Scatter-Gather* pattern.

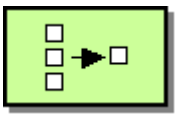
Like the *Recipient List*, the *Splitter* processes the message parts in sequence, but especially if complex message *Processors* (i.e. long running ones) are used, it is desirable that all message parts are processed in parallel. Parallel processing is not implemented in Camel's *Splitter*, but can be achieved using *Producers* of *Endpoints* as *Processors* that run quickly to decouple the message *Processors* from the *Splitter*. For example the *SEDA-Component* that uses in-memory queues or the *JMS-Component* that just sends a message using its underlying messaging middleware could be used. And despite the parallel processing, the *Pipes-and-Filters* architecture requires anyway the decoupling of the *Splitter* from its following *Processors* (i.e. filters).

Like the other patterns also the *Splitter* is integrated in Camel's Java DSL and can be used with either a custom *Expression* or with an *Expression* created by Camel's built-in expression languages. Listing 14 below shows how the *Splitter* is used with Camel's Java DSL.

```
1  from("myComponent:endpointA")
2      .splitter(mySplitterExpression)
3      .to("myComponent:endpointB");
```

Listing 14: Splitter in Apache Camel

3.4.6 Aggregator



The *Aggregator* pattern is not fully implemented in Apache Camel, but Camel can be extended to fully support the *Aggregator* pattern. Camel's *Aggregator* is implemented in the `Aggregator`[4] class which is, unlike most other pattern implementations by Camel, a *Polling Consumer* that thus runs in its own thread. Camel's *Aggregator* uses a *Correlation Expression* to correlate the incoming

[1] `org.w3c.dom.NodeList`
[2] `org.apache.camel.processor.Splitter.SPLIT_SIZE` = "org.apache.camel.splitSize"
[3] `org.apache.camel.processor.Splitter.SPLIT_COUNTER` = "org.apache.camel.splitCounter"
[4] `org.apache.camel.processor.Aggregator`

messages and thus for being able to decide which messages belong together and should be aggregated. The *Correlation Expression* can be any `Expression`[1] and needs to return an arbitrary `Object`[2] that uniquely identifies the group of messages to aggregate. This *Correlation Identifier* `Object` returned needs to be equal, in the sense of the `Object's equals(…)` method, to all other `Objects` returned for messages that belong together. Camel's *Aggregator* performs the actual aggregation of the messages through the `AggregationStrategy`[3] interface whose `aggregate(…)` method needs to be implemented to supply the *Aggregator* with an aggregation strategy. The aggregation strategy is used by invoking the `aggregate(…)` method with two parameters each time a new message arrives. One parameter is the current aggregate (i.e. the *Exchange*[4] which represents the messages with a certain *Correlation Identifier* that have been aggregated till now) and the other is the newly arrived `Exchange`[4]. The `aggregate(…)` method then returns the new aggregate.

The third property of an *Aggregator*, besides the message correlation and the aggregation strategy, is the completeness condition which is not implemented in Camel's *Aggregator*. Camel aggregates messages till a defined amount of messages arrived or till a defined amount of time passed. But as the message counter and the timeout are common to all arriving messages, instead of having separate ones for each aggregate, this doesn't even implement a basic completeness condition because the decision whether an aggregate is complete, does neither depend on how many messages arrived for an aggregate, nor on how long it is ago that the first message arrived for an aggregate. Thus from an aggregate's point of view the completion is somehow random.

To implement a completeness condition the `Aggregator`[5] can be extended by using a custom `AggregationCollection`[6]. The *Aggregator* uses the *Aggregation Collection* to store his aggregates, and iterates through the collection after the defined amount of messages arrived or after the defined amount of time passed, sending all aggregates of the collection to its sub processor. The *Aggregation Collection* thereby makes use of the *Correlation Identifier Expression* and the *Aggregation Strategy* mentioned above.

One way to add a completeness condition to the `AggregationCollection` could be to extend the `AggregationCollection` and to expose only aggregates that are complete, according to a completeness condition, which could be implemented using Camel's `Predicates`[7]. Using this approach only the messages exposed to the Aggregator and thus only the complete aggregates would be forwarded to the Aggregator's child `Processor`[8].

Another shortcoming of Camel's *Aggregator* implementation is that it doesn't support a persistent state, namely it doesn't include a persistent implementation of the `AggregationCollection`. This shortcoming could also be fixed by developing a custom implementation of the

[1] `org.apache.camel.Expression`
[2] `java.lang.Object`
[3] `org.apache.camel.processor.aggregate.AggregationStrategy`
[4] `org.apache.camel.Exchange`
[5] `org.apache.camel.processor.Aggregator`
[6] `org.apache.camel.processor.aggregate.AggregationCollection`
[7] `org.apache.camel.Predicate`
[8] `org.apache.camel.Processor`

`AggregationCollection`[1] which stores its state in a database, for example via the *Java Persistence API* [DK06].

Because Camel's implementation of the Aggregator doesn't need prior knowledge of a new aggregate - it simply creates a new aggregate if an aggregate with the current correlation identifier doesn't exist yet – it implements a so called *self-starting Aggregator*. Another variant of the *Aggregator* where the *Aggregator* needs prior knowledge of the aggregates is called an *initialized Aggregator*. Such an *Aggregator* could be initialized for example by a *Splitter* or a *Recipient List* through a separate channel as proposed by [HW03]. Such an implementation could again be realized in Camel by a custom `AggregationCollection` implementation which listens on an additional channel for initialization messages.

Camel supports the *Aggregation* in its DSL by using the `aggregator(…)` methods, but these methods don't support a custom `AggregationCollection` and thus can't be used for an implementation that correctly implements the *Aggregator* pattern. For using the *Aggregator* pattern in Camel's routing rules either the Java DSL needs to be adapted to support custom `AggregationCollections` or the Aggregator class can be used directly for creating a route, although this approach produces less readable code. Listing 15 below shows how to create a routing rule using the latter approach and using a custom `AggregationCollection` implementation which also supports a correlation identifier `Predicate`[2]. Note that, as mentioned before, Camel's *Aggregator* is a *Polling Consumer* and thus needs to run in its own thread. For that reason the *Aggregator* has to be started through its `start()` method, provided by the `Service`[3] interface, after the *Camel Context* has been started.

```
1  AggregationCollection myAggregationCollection
2     = new MyAggregationCollection(myCorrelationExpression,
3                                   myAggregationStrategy,
4                                   myCompletnessConditionPredicate);
5
6  Aggregator myAggregator
7     = new Aggregator(getEndpoint("myComponent:endpointA"),
8                      getEndpoint("myComponent:endpointB")
9                         .createProducer(),
10                     myAggregationCollection);
```

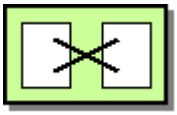
Listing 15: Apache Camel's Aggregator with a custom Aggregation Collection

3.4.7 Composed Message Processor / Scatter-Gather

The *Composed Message Processor* and the *Scatter-Gather* pattern and also a combination of both can be implemented by combining the *Splitter, Router, Aggregator, Recipient List* and *Publish-Subscribe Channel* pattern as described in Hohpe's and Woolf's book [HW03].

[1] `org.apache.camel.processor.aggregate.AggregationCollection`
[2] `org.apache.camel.Predicate`
[3] `org.apache.camel.Service`

3.5 Message Transformation



Message Transformation, and thus the *Message Translator* pattern, is supported by Apache Camel in several different ways and for several different levels of transformation. Transport-level transformation is implicitly done by Apache Camel's *Components*. Data representation level transformation is supported by Apache Camel's *Type Converter* (see chapter 2.2.3 Camel Type Converter above) for simple data type transformation and by Camel's `DataFormat`[1] interface, which offers a plug-in strategy for the marshalling and unmarshalling of messages.

Data type and data structure level transformations are supported in Camel by using custom `Processors`[2] which allow the transformation of messages using arbitrary Java code or by using an *Expression* whose result replaces the old message body. Listing 16 below demonstrates both approaches.

```
1  // translation by a custom Processor:
2  from("myComponent:endpointA1")
3     .process(myCustomTranslatorProcessor)
4     .to("myComponent:endpointB1");
5
6  // translation by an Expression:
7  from("myComponent:endpointA2")
8     .setBody(myTranslatorExpression)
9     .to("myComponent:endpointB2");
```

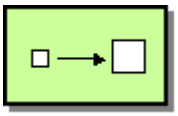
Listing 16: Message Translator in Apache Camel

3.5.1 Envelope Wrapper

The *Envelope Wrapper* pattern is implemented in Apache Camel by its *Components*. If a messaging system or a transport protocol requires a special message format, it's the corresponding *Component's* task to wrap the message so that the message is compliant to the infrastructure the *Component* interfaces to. For example the JMS-Component wraps a Camel message in a JMS message and also adds the Camel message headers as JMS properties to the JMS message. If application level wrapping of messages is needed it can be implemented by the same techniques described in chapter 3.5 above.

[1] `org.apache.camel.spi.DataFormat`
[2] `org.apache.camel.Processor`

3.5.2 Content Enricher

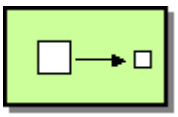


The *Content Enricher* pattern is not implemented in Apache Camel. The best way to add a *Content Enricher* to Camel is to create a custom `Processor`[1] which is able of collecting the required data needed for the enriching of the messages. For example the custom *Processor* could connect to a database system or invoke a web service to get the required data. By using the latter approach a custom *Processor* could be implemented that is generic enough to be reusable. Such a *Processor* could invoke a web service by sending incoming messages to the web service which then returns the required information that can then be used in an XSL transformation together with the original message to generate the enriched message. Listing 17 below shows how such an implementation could be integrated in Camel's Java DSL

```
1  Processor myContentEnricher
2     = new MyContentEnricher(myWebserviceWsdlUrl,
3                             myServiceName,
4                             myPortName,
5                             myXsltUrl);
6
7  from("myComponent:endpointA")
8     .process(myContentEnricher)
9     .to("myComponent:endpointB");
```

Listing 17: Possible Content Enricher implementation in Apache Camel

3.5.3 Content Filter



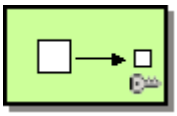
The *Content Filter* is not explicitly implemented in Apache Camel but the same general message transformation techniques described in chapter 3.5 above can be used to implement a *Content Filter*. Particularly the use of an `Expression`[2] to filter content of a message provides a simple approach to a *Content Filter* where the *Expression* would return the filtered content. For example Camel's built-in `XPathBuilder`[3] could be used to filter message content according to an XPath expression. Listing 18 below demonstrates such a scenario in Camel's Java DSL.

```
10 from("myComponent:endpointA")
11    .setBody(XPathBuilder.xpath("/my/xpath/expression"))
12    .to("myComponent:endpointB");
```

Listing 18: Content filtering using an XPath expression in Apache Camel

[1] `org.apache.camel.Processor`
[2] `org.apache.camel.Expression`
[3] `org.apache.camel.builder.xml.XPathBuilder`

3.5.4 Claim Check



Apache Camel doesn't implement the *Claim Check* pattern, but can be extended to support this pattern. Unlike for the *Content Filter* pattern a custom `Processor`[1] implementation should be used, as it would be bad design to develop a custom `Expression`[2] that has side effects like the storage of the filtered data. A `Processor`, filtering the content, would need to store the filtered content persistently to be able to retrieve it later, for example it could store the filtered content in a database which could be accessed using the Java Persistence API [DK06]. Also a unique key to identify the filtered content is needed, that has to be either already present in the message or generated.

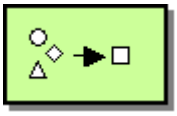
A general implementation of the *Claim Check* could use three `Expressions` to identify the removed content, the content to keep and the unique key, which also can be generated. A less general implementation that knows about the format of the messages could be implemented using only one `Expression` and compute all other necessary information. For example, one that can only handle XML data and uses an XPath *Expression* to identify the content to filter, and where the filtered message content is computed and the unique key is generated.

Such general implementations would require each instance to have its own storage, especially in the case of generated keys, so that the filtered content of the different instances is not mixed. Listing 19 below shows how such a general implementation of the *Claim Check* pattern, which handles XML messages, uses an XPath expression to identify the data to filter and identifies its persistent storage via a unique string, could be used in Camel's Java DSL.

```
1  Processor myClaimCheck
2     = new MyClaimCheck ("/my/xpath/expression",
3                         "myClaimCheck1");
4
5  from("myComponent:endpointA1")
6     .process(myClaimCheck)
7     .to("myComponent:endpointB1");
```

Listing 19: Possible Claim Check implementation in Apache Camel

3.5.5 Normalizer



The *Normalizer* is not implemented in Apache Camel, but as it is a composed pattern consisting of a *Content-Based Router* and several *Message Translators*, it can be implemented by using Camel's

[1] `org.apache.camel.Processor`
[2] `org.apache.camel.Expression`

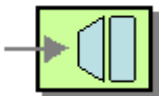
implementation of those patterns. Because the *Normalizer* is a separate pattern and thus its components can and should be tightly coupled, one could renounce the usage of pipes, i.e. of channels, (see chapter 3.3 Pipes and Filters above) between the *Content-Based Router* and the *Message Translators*. A *Normalizer* would then be configured in Camel's Java DSL as demonstrated in listing 20 below.

```
1  from("myComponent:endpointA").choice()
2     .when(myMessageFormat1Predicate)
3        .process(myXformProcessor1).to("myComponent:endpointB")
4     .when(myMessageFormat2Predicate)
5        .setBody(myXformExpression2).to("myComponent:endpointB")
6     .when(myMessageFormat3Predicate)
7        .process(myXformProcessor3).to("myComponent:endpointB")
8     .when(myMessageFormat4Predicate)
9        .setBody(myXformExpression4).to("myComponent:endpointB")
10    .otherwise()
11       .to("myComponent:invalidMessageChannel");
```

Listing 20: Normalizer in Apache Camel

3.6 Messaging Endpoint

3.6.1 Messaging Gateway



Apache Camel itself provides a low-level *Messaging Gateway* as it abstracts from the infrastructure and messaging systems which Apache Camel's *Components* interface with. Camel offers general messaging semantics through generic interfaces that can be used independently from the underlying messaging infrastructure. In case of the *JMS-Component* Camel simply uses the JMS-API [HBS+], which is also a low-level *Messaging Gateway*.

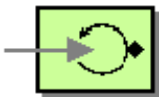
Camel's documentation mentions that the *Messaging Gateway* is supported using Camel's *Bean-Component* which offers the `ProxyHelper`[1] class that can generate proxy-classes to invoke methods on remote Java-Beans (i.e. remote Java classes). But as the proxy offers only synchronous remote invocation via messaging, and as the generated proxy class doesn't perform any error handling but exposes the application to messaging specific exceptions, this approach is only suitable to implement very basic *Messaging Gateways*. But Apache Camel can be used to implement higher-level *Messaging Gateways* that hide the messaging semantics from an application, by developing a domain specific API that uses the Camel API to send messages. Both approaches are only possible of course, if the application, we want to provide a gateway for, is written in Java.

[1] `org.apache.camel.component.bean.ProxyHelper`

3.6.2 Messaging Mapper

The *Messaging Mapper* pattern is not implemented in Apache Camel, but Camel can be used to implement a *Messaging Mapper* if the target application is a Java application by providing a unified interface to different messaging infrastructures. For example the *Messaging Mapper* could implement Camel's `Processor`[1] interface that handles incoming message exchanges and map them to application specific events.

3.6.3 Polling Consumer

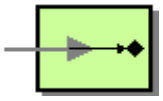


The Polling Consumer pattern is implemented in Apache Camel by the `PollingConsumer`[2] interface which offers blocking as well as non-blocking methods to poll for new messages. `PollingConsumer`[3] instances are obtained using the `createPollingConsumer()` method of the `Endpoint`[4] interface as Camel's Java DSL doesn't offer support for *Polling Consumers* yet. Listing 21 below demonstrates how a polling consumer that polls for new messages in a blocking way is implemented in Camel. Note that the `PollingConsumer` on endpoints created by Camel's *JMS-Component* has a bug that makes the blocking `receive()` method, used in the example below, act like a non-blocking receive, which means that the method might return immediately with a null value as result.

```
1  Endpoint endpoint = context.getEndpoint("myComponent:endpointA");
2  PollingConsumer pollingConsumer = endpoint.createPollingConsumer();
3  while(true) {
4      Exchange exchange = pollingConsumer.receive();
5      // process message here
6  }
```

Listing 21: Polling Consumer in Apache Camel

3.6.4 Event-Driven Consumer



The *Event-Driven Consumer* pattern is implemented in Apache Camel's `Processor`[5] interface and is the consumer model used when routes are created using Camel's Java DSL. Each time a new message arrives the `Consumer`[6] invokes the `process(…)` method of the `Processor` registered with the `Consumer`. The *Event-Driven Consumer* is the recommended consumer model in Apache

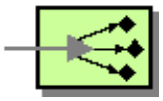
[1] `org.apache.camel.Processor`
[2] `org.apache.camel.PollingConsumer`
[3] `org.apache.camel.PollingConsumer`
[4] `org.apache.camel.Endpoint`
[5] `org.apache.camel.Processor`
[6] `org.apache.camel.Consumer`

Camel because Camel offers a `ThreadProcessor` which allows pooling and management of the threads that are used to process messages by a `Processor`. But note that although Camel's `Consumers`[1] use the `Processor` interface to notify the attached message processors of incoming messages and thus seem to make these message processors *Event-Driven Consumers* they internally implement the *Polling Consumer* pattern. This is for example true for the *File-Component* and the *SEDA-Component* but not for the *JMS-Component*. Listing 22 below demonstrates how an *Event-Driven Consumer* is implemented in Camel.

```
1 from("myComponent:endpointA").process(new Processor() {
2     public void process(Exchange exchange) {
3         // process message here
4     }
5 });
```

Listing 22: Event-Driven Consumer in Apache Camel

3.6.5 Competing Consumers

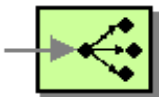


Competing Consumers can be implemented in Apache Camel by simply creating two routes that have the same endpoint, which has to be an endpoint on a *Point-to-Point Channel*, as their incoming channel. The *JMS-Component* also supports the *Competing Consumer* pattern using a single route if the incoming *JMS-Endpoint* is configured to support concurrent consuming, as done for the endpoint in listing 23 below.

```
6 from("jms:queue:myQueueA?concurrentConsumers=5").process(myProcessor);
```

Listing 23: Competing Consumer using the JMS-Component in Apache Camel

3.6.6 Message Dispatcher



The *Message Dispatcher* pattern is not implemented in Apache Camel, but could be implemented by developing a custom `Processor`[2] that delegates the message processing to some child `Processors` that run in their own thread. The pattern can also be implemented by combining a `ChoiceProcessor`[3], which basically implements a *Content Based Router* and a `ThreadProcessor`[4] which enables asynchronous processing of its child `Processors` using a thread pool. Because a *Consumer* is only done processing a message when the whole route has

[1] Consumers are created via `org.apache.camel.Endpoint.createConsumer(Processor processor)`
[2] `org.apache.camel.Processor`
[3] `org.apache.camel.processor.ChoiceProcessor`
[4] `org.apache.camel.processor.ThreadProcessor`

been processed (see chapter 2.2.7 above for details) the Message Dispatcher can only be implemented if the incoming *Endpoint* support *Competing Consumers*, as the *JMS-Components Endpoints* do, if configured appropriately. Note that using asynchronous processing doesn't change this behavior - it just prevents the *Consumer's* thread from blocking. All this is also true for a custom implementation of a *Message Dispatcher* as a custom `Processor`, as long as the *Processor* behaves like intended by Camel's architecture. This means it should return only after the `Processor`, it dispatched to, returned too. But if needed, that rule could be violated and the custom `Processor`[1] could return from the `process(…)` method without waiting for the completion of its child processor. The drawback would be that `InOut`[2] message exchanges wouldn't be supported by such a *Route* and that the *Dispatcher* would successfully consume messages before the processing is complete and thus before the message has been added to the following channel. This could cause messages to get lost in case the *Dispatcher* crashes.

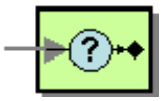
Another approach could be to implement a custom *Message Dispatcher* as a *Polling Consumer*. Here the restrictions just mentioned don't apply, as the *Polling Consumer* doesn't support `InOut` exchanges anyway and the messages are immediately consumed anyway by the *Polling consumer*, unless an acknowledgement mechanism like the one of JMS is used. Thus such a *Dispatcher* could delegate the processing to its child `Processors` in separate threads.

Listing 24 below demonstrates how a Message Dispatcher can be implemented in Apache Camel using a *Content-Based Router* and an *Endpoint* that allows *Competing Consumers*, as mentioned before.

```
1  from("jms:queue:myDestination?concurrentConsumers=5").thread(5).choice()
2      .when(header("type").isEqualTo("A"))
3          .process(myProcessorForTypeA)
4      .when(header("type").isEqualTo("B"))
5          .process(myProcessorForTypeB)
6      .when(header("type").isEqualTo("C"))
7          .process(myProcessorForTypeC)
8      .otherwise().to("jms:queue:myInvalidMessageQueue");
```

Listing 24: Message Dispatcher using a Content-Based Router

3.6.7 Selective Consumer



The *Selective Consumer* pattern is not implemented in Apache Camel and it can't be implemented without support by the underlying messaging system. If the used messaging system and the corresponding Camel *Component* support the *Selective Consumer* pattern, as this is the case for the *JMS-Component*, then the *Endpoint* has just to be configured accordingly - usually via its URI. If the messaging system doesn't support the *Selective Consumer* pattern but supports message peek-

[1] `org.apache.camel.Processor`
[2] `org.apache.camel.ExchangePattern.InOut`

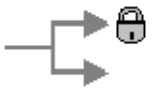
ing, i.e. the receiving of a message without to consume it, the pattern could be implemented by extending a *Component* with the availability to peek for messages that match some *Expression* and to deliver and consume only those. If the *Message Channel* is a *Publish-Subscribe Channel* the *Selective Consumer* can also be implemented without support by the messaging system, by using a *Message Filter*. If the messaging system doesn't support selective consumers nor message peeking and the channel is not a *Publish-Subscribe Channel*, then a *Message Dispatcher* could be considered instead of a *Selective Consumer*.

As JMS supports the *Selective Consumer* pattern by using a selector, which is a SQL92 predicate on message headers – called properties in JMS. And as the *JMS-Component* supports this feature, the *Selective Consumer* pattern can be implemented on *JMS-Endpoints* as demonstrated in listing 25 below. Note that the selector has to be escaped for the endpoint URI to be valid.

```
1 from("jms:queue:myDestination?selector=type%3D%27A%27")
2    .process(myProcessor); // selector: type='A'
```

Listing 25: Selective Consumer using Apache Camel's JMS-Component

3.6.8 Durable Subscriber



Like the *Selective Consumer*, the *Durable Subscriber* is a messaging system specific pattern and is implemented for example in Camel's *JMS-Component*. Listing 26 below demonstrates how an *Endpoint* on a JMS topic can be made a *Durable Subscriber*. Note that, when using JMS directly, a unique client ID and a unique subscription name must be provided to register as a *Durable Subscriber*. If a *Component* doesn't support *Durable Subscribers*, Camel can also be used to implement a *Durable Subscriber* by creating a route from a *Publish-Subscribe* channel to a *Point-to-Point Channel*, from which the client, that needs a durable subscription, can receive its messages.

```
3 from("jms:queue:myDestination?clientId=myClientId1&durableSubscriptionNam
4 e=mySubscriptionName1").process(myProcessor);
```

Listing 26: Durable Subscriber using Apache Camel's JMS-Component

3.6.9 Idempotent Receiver

The *Idempotent Receiver* pattern is implemented in Camel by the `IdempotentConsumer`[1] class, which uses an `Expression`[2] to extract or generate a unique identifier for an incoming message exchange and an instance of the `MessageIdRepository`[3] interface to store the unique identifiers

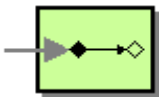
[1] `org.apache.camel.processor.idempotent.IdempotentConsumer`
[2] `org.apache.camel.Expression`
[3] `org.apache.camel.processor.idempotent.MessageIdRepository`

for later comparison with incoming messages. A simple *Expression* could just return a message's unique message ID which is often generated by the messaging infrastructure. Using these two general interfaces Camel's *Idempotent Receiver* can be adapted to fit the current requirements. The implementation of the `MessageIdRepository` interface, that Camel offers, is an in-memory based implementation that also allows the setting of a maximum cache size. If persistent storage of the receive message identifiers is required, a custom implementation of the `MessageIdRepository` interface storing the values can be used. Listing 27 below shows how Camel's *Idempotent Receiver* can be configured using Camel's Java DSL. In this example the message's `"MyMessageId"`-header is used as unique identifier and Camel's in-memory `MessageIdRepository` with a maximum capacity of 100 entries is used for storing these identifiers.

```
1  from("myComponent:endpointA")
2     .idempotentConsumer(header("MyMessageId"),
3                         MemoryMessageIdRepository
4                            .memoryMessageIdRepository(100))
5     .process(myProcessor);
```

Listing 27: Apache Camel's Idempotent Receiver

3.6.10 Service Activator



The Service Activator pattern is implemented by Camel's *Bean-Component* and Camel's `BeanProcessor`[1]. They both enable the processing of messages by some arbitrary Java-Bean. The *Bean-Component* can be used to send messages to some *Bean-Endpoint*, where Camel queries the JNDI-context associated with the *Camel Context* for the bean. In case of the `BeanProcessor` a bean instance or the bean's class are passed on construction of the `BeanProcessor`. In both cases the method to invoke is discovered using several ways. One possibility is to specify the method to invoke in the message header or in the *Bean-Endpoint's* URI. As last resort Camel tries to find an appropriate method according to the message type, using introspection. For details about the method discovery see Camel's documentation [Cam]. Listing 28 below demonstrates the usage of the *Bean-Component*, specifying the method to invoke in the *Endpoint*-URI, and the usage of the `BeanProcessor` via Camel's Java DSL. Note that for the *Bean-Component* to work, the bean has to be stored in *Camel Context's* JNDI-context under the name `"myService"`.

[1] `org.apache.camel.component.bean.BeanProcessor`

```
1  from("myComponent:endpointA")
2     .to("bean:myService?methodName=myMethod")
3     .to("myComponent:endpointB"); // using the Bean-Component
4
5  from("myComponent:endpointC")
6     .bean(new MyService(), "myMethod")
7     .to("myComponent:endpointD"); // using the BeanProcessor
```

Listing 28: The Service Activator pattern in Apache Camel

4 Implementation

In chapter 3 the Enterprise Application Integration (EAI) Patterns from [HW03] were evaluated regarding if and how they are implemented in Apache Camel. In this part of the work the results from chapter 3 are used to extend the Eclipse based editor which was created as part of the diploma thesis from [Dru07]. This Eclipse based editor was developed as an Eclipse plug-in which enables the graphical modeling of messaging systems. The graphical editor allows combining the different EAI patterns to model a messaging system that implements the *Pipes-and-Filters* architecture. This messaging system model is then used to generate a WS-BPEL [AAA+07] process that implements the messaging system modeled. The generated WS-BPEL process uses web services to provide most of the functionality offered by the different patterns and in turn offers the execution of that process as web service.

The goal of extending the Eclipse plug-in, developed by Druckenmüller [Dru07], is to add the ability to generate Apache Camel based Java-code out of the messaging system modeled. This Java-code shall implement a standalone application that uses an underlying message oriented middleware to implement the *Message Channels* and that uses Apache Camel to implement the routing and the transformation logic as well as the message endpoints.

Like the generated WS-BPEL [AAA+07] process, also the generated Camel based application shall use web services to support the various EAI pattern implementations. Although the modeled messaging system wouldn't need to offer its functionality as web service, because it could interact with the outside world using messaging, the solution developed in this work shall also be able to provide its functionality as a web service. The reason the solution has to be able to do so, is that the generated Java application should have the same semantics and interfaces to the outside world as the WS-BPEL process generated by the Eclipse plug-in.

4.1 Concept

Apache Camel's implementation of the Enterprise Integration Patterns, evaluated in chapter 3 above, relies in most cases on some sort of custom Java-code. For example, often the parameters needed by a pattern require a custom implementation of some interface. Patterns that are not or not completely implemented in Camel also often require a custom `Processor`[1] implementation. The implementation from [Dru07] uses some common web standards based on XML [BPS+06] to implement the identified parameters of the different EAI patterns. These common web standards are XPath [CD99], XSLT [Cla99] and web services [BL07] which are widely used in [Dru07] to implement some logic required by the patterns.

[1] `org.apache.camel.Processor`

To fill the gap between the Java-code centric pattern implementation of Camel and the web technology based implementation of [Dru07] a framework is being developed. This framework consists of classes that implement the different EAI patterns or support Camels implementation of the patterns. These classes are all based on Camel and aim to extend Camel in a consistent and reusable fashion, while using web based technology like XPath, XSLT and web services to implement the patterns. Using these technologies adds the constraint, that all processed messages have to be XML messages. But as XML is a widely accepted standard this is no drawback in the sense of compatibility with other solutions.

By using this framework, the Java-code generation process, which is the actual goal of extending the Eclipse plug-in, is reduced to the generation of Java-code that instantiates the classes of the framework corresponding to the patterns. The classes are instantiated using the parameters of the corresponding pattern and are connected by building Apache Camel *Routes*.

The extension of the Eclipse plug-in from [Dru07] is done by adding a custom so called ActionDelegate[1] to the plug-in which iterates through the modeled messaging system and thereby generates the appropriate Java-code. Because the time constraints on this work don't allow the implementation of all EAI patterns, the code generation is being implemented in an extensible fashion to allow easy completion by future works. The code generation being done by an Eclipse plug-in and Eclipse being also a Java development environment just calls for an implementation that generates the Java-code using Eclipse's Java Development Tools [Jdt]. That way a Java project is created inside Eclipse containing a package with the generated Java-code. Being generated as a Java project in Eclipse, the messaging system can conveniently be modified and build inside of Eclipse.

4.2 Used Technologies

The implementation, done in this work, uses several technologies. These technologies were used together with Apache Camel to implement the framework mentioned above as well as to implement the code generator. In the following these technologies – except Apache Camel, which was described in its own chapter – are described:

JAX-WS 2.0

The Java API for XML-Based Web Services (JAX-WS) [CHM06] offers a standardized interface for invoking and providing web services using the Java programming language. JAX-WS offers a high-level and a low-level API, where the high-level API can be used to comfortably invoke or provide a web service by binding a Java interface to a web service defined in a WSDL file [BL07]. The high-level API handles things like type conversion and enables the programmer to use a web service through a simple method call, or to provide a web service by simply implementing a Java method. The low-level API provided by JAX-WS enables the programmer to work on the message level. Here a web service can be invoked by creating either a complete SOAP message [ML07] or by just

[1] `org.eclipse.ui.IWorkbenchWindowActionDelegate`

providing the message payload (i.e. the message body) and letting JAX-WS handle the message headers. The low-level invocation of web services is offered by JAX-WS's `Service`[1] interface. Providing a web service using JAX-WS's low-level API enables the implementer to work on the message-level in the same way as for the low level invocation. This means that the web service provider can either work with the whole message or just with the message payload. Such a low level web service provider needs to implement JAX-WS's `Provider`[2] interface.

JAXP 1.4

The Java API for XML Processing (JAXP) offers several APIs for the validation, parsing, transformation and extraction of content from XML documents. In the framework, implemented as part of this work, XML transformation using XSLT [Cla99] stylesheets is done using JAXP's XML transformation API located in the javax.xml.transform package. This is the main technology used to implement the *Message Translator* pattern and is also used frequently in this framework in conjunction with the invocation of web services. Another JAXP API frequently used in this framework is JAXP's XPath API which is found in the javax.xml.xpath package. This API is used to evaluate XPath expressions [CD99] to select a specific portion of an XML document.

Eclipse Visual Editor 1.2.0

The Eclipse Visual Editor [Vep] is an Eclipse plug-in that provides graphical user interface for the implementation of graphical Java applications. In this work it is used to implement graphical dialogs for the interaction with the user during the generation of the Apache Camel based Java-code.

Eclipse JDT 3.2.2

The Eclipse Java Development Tools (JDT) provide the Eclipse Platform [Ecl] with the capability to serve as an integrated development environment (IDE) for Java applications. The JDT Core, which is the part of the JDT used for this work, provides the infrastructure of the Java IDE by providing a model for the Java project tree and an API to modify this model. For the implementation of the Java code generator the JDT Core API is used to create and manipulate the java classes and packages needed for the generated system to be able to run. Also the required libraries needed to run the generated messaging system are added to the Java project's classpath using the JDT Core API.

4.3 The EAI-2-Camel Framework

The EAI-2-Camel framework was implemented to fill the gap between the Java-code centric pattern implementation of Camel and the web technology based implementation of [Dru07]. It consists of several classes, either implementing one of the EAI patterns or supporting the implementation of the EAI patterns.

[1] `javax.xml.ws.Service`
[2] `javax.xml.ws.Provider`

4.3.1 Helper Classes

The framework contains some helper classes that are used to support the implementation of the patterns in the framework. One of these helper classes is the `Eai2CamelNamespaceContext`[1] which is a simple implementation of JAXP's `NamespaceContext`[2] interface. This interface is used by JAXP to resolve namespace prefixes used for example in XPath expressions.

The second helper class is the `WebServiceFilter`[3] class that is the basic class which is extended by all patterns that invoke a web service in their implementation. The `WebServiceFilter` implements Camel's `Processor`[4] interface and is thus an *Event-Driven Consumer*. It uses JAX-WS's low-level API to invoke a web service by sending the incoming messages as the message body of the web service invocation. Because a web service invocation can take some time, the web service is invoked asynchronously. It is using JAX-WS's callback mechanism to get notification about the completion of the invocation. For the `Processor` not to block while the web service is invoked, the `WebServiceFilter` also implements Camel's `AsyncProcessor`[5] interface, which enables Camel to invoke the `WebServiceFilter` asynchronously. This enables the complete asynchronous processing of messages. The `WebServiceFilter` also uses JAXP's message translator to translate the response message of the web service invocation to a different Java type, required by the implementation. Here the identity transformation is done, but classes extending the `WebServiceFilter` might choose to transform according to an XSLT stylesheet.

Despite the `WebServiceFilter` being used to implement the EAI patterns that are using web services for their implementation, it also implements a part of the "External Service" pattern introduced by [Dru07]. This "External Service" pattern is basically a filter in the *Pipes-and-Filters* architecture that is implemented by a web service, i.e. the whole message is processed by the web service. In addition to the invocation this "External Service" pattern also allows to provide a web service to the outside. This is not implemented by the `WebServiceFilter` but by the *Web Service Provider* mentioned below.

The framework also contains the `Eai2CamelContext`[6] class which extends Apache Camel's default implementation of the *Camel Context*. This extended *Camel Context* allows adding services, i.e. classes implementing Camel's `Service`[7] interface, to the Camel Context. These Services' lifecycle will then be managed by the *Camel Context* together with its own lifecycle. Particularly the *Camel Context* will start the services after it was started itself and will stop them before stopping itself.

[1] `de.unistuttgart.iaas.framework.eai2camel.Eai2CamelNamespaceContext`
[2] `javax.xml.namespace.NamespaceContext`
[3] `de.unistuttgart.iaas.framework.eai2camel.WebServiceFilter`
[4] `org.apache.camel.Processor`
[5] `org.apache.camel.AsyncProcessor`
[6] `de.unistuttgart.iaas.framework.eai2camel.Eai2CamelContext`
[7] `org.apache.camel.Service`

4.3.2 Message Producer Template

The framework offers a message producer template that facilitates the creation of a message producer. It is provided by the abstract class `MessageProducerTemplate`[1] which extends Camel's `CamelTemplate`[2] and implements the `Service`[3] interface enabling Camel to manage its lifecycle. Classes extending the `MessageProducerTemplate` need to implement the `run()` method from the `Runnable` interface which is invoked in a separate thread when the *Camel Context* is started. The implementation of the *run()* method must regularly, in particular before producing messages, check if the Camel Context is still running by using the `isRunAllowed()` method. The advantage of this message producer template is that it provides several easy to use methods to send messages to some *Endpoint*. The example in listing 29 below demonstrates an implementation of the `run()` method which sends messages to the default *Endpoint*, passed to the constructor of this `MessageProducerTemplate` instance.

```
1  public void run() {
2      while (this.isRunAllowed()) { // check if Camel Context is running
3          try {
4              this.sendBody("some message content");
5              Thread.sleep(5000); // try to sleep 5 seconds
6          } catch (InterruptedException e) { } // can be ignored
7      }
8  }
```

Listing 29: Using the Message Producer Template of the EAI-2-Camel Framework

4.3.3 Web Service Endpoint

The EAI-2-Camel framework also adds a new *Endpoint* to Apache Camel that can be used to invoke web services. The *Web Service Endpoint* works in a similar way as the `WebServiceFilter` class mentioned in chapter 4.3.1 above. It also uses JAX-WS's `Service` class to invoke the web service asynchronously and JAXP to transform the response message to a required type. The Web Service Endpoint is implemented using three classes, one for the *Endpoint*, one for the *Consumer* and one for the *Producer*. The implementation is located in the following package: de.unistuttgart.iaas.framework.eai2camel.component.ws.

Note that this is just the implementation of an *Endpoint* not of a complete *Component*. Thus the use of URI's to create *Web Service Endpoints* is not supported. A future work could create a *Web Service Component* which would enable Apache Camel to use web services as a channel. This could be useful for example to perform transformation on the wire by a web service or to implement a *Message Channel* that provides *Guaranteed Delivery* using a web service. This *Component* could also add the ability to provide a web service to the outside world which is implemented us-

[1] `de.unistuttgart.iaas.framework.eai2camel.endpoint.MessageProducerTemplate`
[2] `org.apache.camel.CamelTemplate`
[3] `org.apache.camel.Service`

ing a Camel *Route*. To support this also the *Web Service Endpoint* of this work would need to be extended.

4.3.4 Web Service Provider

The EAI-2-Camel framework enables Apache Camel to offer a web service which is implemented by processing a message through a Camel Route. As the *Web Service Provider* offers a web service to the outside it is a message consumer from the invoker's point of view. But to Apache Camel the *Web Service Provider* is in first place a message producer, as it implements the web service by producing a message and sending it to some *Endpoint*. On the other hand the *Web Service Provider* supports sending a response message to a received request, thus it is also a consumer of messages from Camel's point of view. This is because the response, which will be sent to the invoker of the web service, is received and consumed by some *Endpoint*.

The *Web Service Provider* is implemented by the `WebServiceProvider`[1] class of the framework. This class supports the implementation of a web service which sends the request message to an *Endpoint*, and expects the response message on a different *Endpoint* to forward the response to the web service invoker. The *Web Service Provider* is intentionally not implemented using Camel's ability to process messages as *InOut-Exchanges* due to the incomplete implementation mentioned in chapter 3.2.2 (Request-Reply pattern). The `WebServiceProvider` implements Camel's `Processor`[2] interface to act as an *Event-Driven Consumer* on the *Endpoint* the *Web Service Provider* expects the reply message. When the web service is invoked the `WebServiceProvider` packs the request message in a Camel *Exchange* and sends it to the *Endpoint* passed in the constructor. Because JAX-WS 2.0 does only support synchronous web service implementations the `WebServiceProvider` blocks till it receives the answer via its `process(…)` method of the `Processor` interface. This is not an ideal implementation, but JAX-WS 2.1, which supports asynchronous implementations of web services, is not implemented yet by all JAX-WS providers. And even with JAX-WS 2.1 the web service client stays connected during the execution of the web service, weather executed synchronously or asynchronously. This is because messaging-style web service invocation is not supported by JAX-WS yet.

The `WebServiceProvider` implementation also supports the implementation of a web service where the invoker doesn't expect any response. In this case the web service just creates a Camel *Exchange*, sends it to the *Endpoint* and returns.

To allow Camel to manage its lifecycle, which means providing or not providing the web service, the `WebServiceProvider` implements Camel's `Service`[3] interface.

[1] `de.unistuttgart.iaas.framework.eai2camel.endpoint.WebServiceProvider`
[2] `org.apache.camel.Processor`
[3] `org.apache.camel.Service`

4.3.5 Message Translator

To support the *Message Translator* pattern from [HW03] using web technologies the `MessageTranslator`[1] class was implemented in the EAI-2-Camel framework. This class is an *Event-Driven Consumer* and thus implements Camel's `Processor`[2] interface. It supports the transformation by either a web service or by an XSLT stylesheet. Which of those technologies are used for the transformation depends on which constructor is used to instantiate the `MessageTranslator`.

If the transformation by a web service is chosen, the `MessageTranslator` simply forwards its incoming messages and expects the translated message as response. The web service invocation is done by extending the `WebServiceFilter`[3] class which already provides that functionality.

In case of transformation using XSLT, the `MessageTranslator` uses JAXP's Transform-API[4] to perform the XSLT transformation using the supplied XSLT stylesheet.

4.3.6 Content Enricher

The *Content Enricher* is a special case of a *Message Translator*, thus the *Content Enricher* implementation of the EAI-2-Camel framework extends the `MessageTranslator`[5] class and reuses part of it. The *Content Enricher*, which is implemented in the `ContentEnricher`[6] class, provides its functionality using different approaches. What approach is used is determined according to which constructor is used to instantiate the `ContentEnricher`.

One option offered is to delegate the whole content enriching process to a web service. In that case the whole message is send to a web service which enriches the message and returns the resulting message. This approach doesn't differ at all from a web service based transformation implemented by the `MessageTranslator` or from the general processing of a message by a web service implemented by the `WebServiceFilter`[3] class.

Another option is to send the whole incoming message to a web service which returns the data needed to enrich that message. This returned data is then used to enrich the data using an XSLT stylesheet. For the XSLT processing to be able to use the data which is subject of the enrichment it needs to be made available somehow to the XSLT processor. In order to do this, two approaches have been implemented in the `ContentEnricher`. One approach provides a custom `URIResolver`[7] to the XSLT processor that resolves a special URL to a document containing the data needed. Using this approach a XSLT stylesheet can simply access the data by loading a docu-

[1] `de.unistuttgart.iaas.framework.eai2camel.transformation.MessageTranslator`
[2] `org.apache.camel.Processor`
[3] `de.unistuttgart.iaas.framework.eai2camel.WebServiceFilter`
[4] `javax.xml.transform`
[5] `de.unistuttgart.iaas.framework.eai2camel.transformation.MessageTranslator`
[6] `de.unistuttgart.iaas.framework.eai2camel.transformation.ContentEnricher`
[7] `javax.xml.transform.URIResolver`

ment residing on that special URL. This is demonstrated in the XSLT snippet in listing 30 below where you can also see the special URL used.

```
1 <xsl:copy-of select="document('http://contentenricher.transformation
2                       .eai2camel.framework.iaas.unistuttgart.de')" />
```

Listing 30: Retrieving the data to enrich in an XSLT-stylesheet via a special URL

Alternatively the data returned from the web service is also made available to the XSLT processor using a XSLT parameter. Using this approach a XSLT stylesheet could reference to this data by declaring the parameter "NewData" and using the XSLT variable "$NewData" that contains the data returned by the web service. This is demonstrated in the XSLT snippet in listing 31 below.

```
3 <xsl:param name="NewData" />
4 ...
5 <xsl:copy-of select="$NewData" />
```

Listing 31: Retrieving the data to enrich in an XSLT-stylesheet via an XSLT parameter

The two options of using the `ContentEnricher` just mentioned have a drawback in common. Namely they require a web service which is able to handle input of a message type specific to the input message type of the *Content Enricher*. Thus the third approach enables to use a more generic web service which accepts a parameter - like a primary key in some database table for example – and returns the corresponding data. For the *Content Enricher* being able to send that message an XPath expression is used to identify the parameter in the message payload. The XPath expression is evaluated using JAXP's XPath-API[1]. To be able to resolve the namespace prefixes used in such an XPath expression the constructor of the `ContentEnricher` also provides the possibility to supply a `NamespaceContext`[2].

4.3.7 Recipient List

Like suggested in the evaluation of the *Recipient List* pattern (see 3.4.4 Recipient List), a custom *Camel Expression* is developed as part of the EAI-2-Camel framework. This `Expression`[3] creates a list of recipients depending on the message received and is implemented in the `RecipientListExpression`[4] class. The `RecipientListExpression` supports three possibilities to create the list of recipients.

One possibility is to send the message which is subject to the evaluation to a web service that returns a list of recipients. The `RecipientListExpression` doesn't require the web service to return a list of recipients of a specific XML-Schema [FW04]. It just requires it to return a message

[1] `javax.xml.xpath`
[2] `javax.xml.namespace.NamespaceContext`
[3] `org.apache.camel.Expression`
[4] `de.unistuttgart.iaas.framework.eai2camel.routing.RecipientListExpression`

whose payload contains an arbitrary XML root element which contains one or more arbitrary child elements. The string values of these child elements are then mapped to Apache Camel *Endpoint-URIs*. The web service implementation could be deemed to return Camel *Endpoint-URIs* directly, but this would tightly couple the web service to Apache Camel. Thus the indirection of using a mapping between the channel identifiers returned by the web service and the Camel *Endpoint-URIs* is introduced. This mapping can be passed to the constructor as a `Map`[1] instance or can be created by multiple calls to the `setMapping(…)` method of the `RecipientListExpression`. The web service is invoked using the JAX-WS API as described for the `WebBasedFilter` above. The only difference is that the web service is invoked synchronously because the processing can't continue while the *Expression* is evaluated. Particularly Apache Camel doesn't offer an asynchronous version of the `Expression`[2] interface.

Another possibility is to extract the recipient list from the message which is subject to routing. Therefore the message needs to contain already a list of the recipients. This list could for example be added to the message using a *Content Enricher*. Like for the web service based approach, the recipient list contained in the message needs not to be of a specific XML-Schema. It just needs to have one root element with several child elements whose string values are mapped to Camel *Endpoint-URIs* using the same mapping mechanism a described before. This prevents a tight coupling between the component that adds the recipient to the message and the *Recipient List* implementation. The recipient list contained in the message payload is identified using an XPath expression. This XPath expression is evaluated using JAXP's XPath-API and, like described for the Content Enricher (4.3.6), a `NamespaceContext`[3] is needed to resolve the namespace prefixes used in the XPath expression.

The third possibility is to determine to which recipient to send a message to, based on a set of Camel *Predicates*. Each *Predicate* is associated with a Camel *Endpoint-URI* and if a message meets a *Predicate*, that message is sent to the corresponding *Endpoint-URI*. This implementation resembles Camel's implementation of the `ChoiceProcessor`, i.e. the *Content-Based Router*. The only difference is that a copy of the message is sent to all recipients, not only to the one with the first matching *Predicate*.

4.3.8 Aggregator

The Aggregator pattern is implemented in the EAI-2-Camel framework as suggested in the evaluation of the pattern in chapter 3.4.6 above. This means it is implemented using a custom `AggregationCollection`[4] that hides the aggregates to Camel's `Aggregator`[5] till the aggregate is complete. This custom `AggregationCollection` is implemented in the framework's `ExtendedAggregationCollection`[6] class. As mentioned by [HW03] and in the evaluation of

[1] `java.util.Map`
[2] `org.apache.camel.Expression`
[3] `javax.xml.namespace.NamespaceContext`
[4] `org.apache.camel.processor.aggregate.AggregationCollection`
[5] `org.apache.camel.processor.Aggregator`
[6] `de.unistuttgart.iaas.framework.eai2camel.routing.aggregator.ExtendedAggregationCollection`

the *Aggregator* pattern (chapter 3.4.6), an implementation of the *Aggregator* pattern needs to specify three properties: A correlation between the messages to aggregate, a completeness condition to decide when an aggregate is complete and an aggregation algorithm that computes the aggregated message. Apache Camel offers interfaces for the implementation of the aggregation algorithm and of the correlation identifier as well as for the completeness condition. These interfaces are used by the `ExtendedAggregationCollection` implementation of this framework.

The aggregation algorithm is implemented in Apache Camel by classes implementing the `AggregationStrategy`[1] interface. This interface implements an iterative aggregation approach. Each time a new message in form of an `Exchange`[2] arrives, the `AggregationStrategy's aggregate(…)` method is invoked that shall then compute a new `Exchange` or modify the existing one to represent the aggregation of the messages arrived till then. This iterative approach saves storage as the aggregate is kept small by performing the aggregation each time a message arrives. But some aggregation strategies might require performing the aggregation after all messages of an aggregate arrived. To accommodate for those aggregation strategies and to support the `ExtendedAggregationCollection's` implementation of aggregate timeouts, all aggregates are wrapped in an `AggregateExchange`[3]. The `AggregateExchange` transparently offers all methods of the wrapped `Exchange` and adds some useful functionality that can be used to implement an aggregation strategy. Among this functionality is the ability to store a list of the messages which belong to this aggregate. It also stores the amount of messages that were aggregated in this aggregate, which can be useful when implementing a completeness condition based on the amount of aggregated messages.

The EAI-2-Camel framework developed in this work implements already two aggregation strategies. One of them is the `AggregationStrategyBest`[4] that performs the aggregation by choosing the "best" message as the aggregated message. The best message is selected according to a value contained in the message which is identified using an XPath expression. The messages are then aggregated by comparing this designated value of an incoming message with the current aggregate and choosing the highest or the lowest value. The comparison is done numerically if possible or lexicographically otherwise. The other aggregation strategy implemented in the frame work is the `AggregationStrategyMostRecent`. This strategy uses the most recently arrived message as the aggregated message.

The `ExtendedAggregationCollection` uses a Camel *Predicate* to implement the completeness condition. This completeness condition `Predicate`[5] can use all information available through the `AggregateExchange` interface. Especially it can use the message count or a possible message history stored there. Three completeness `Predicates` are already implemented in the framework. Two of them are simple `Predicates` that never respectively always hold. The `CompletnessPredicateNever`[6] is particularly useful in conjunction with the timeout mechan-

[1] `org.apache.camel.processor.aggregate.AggregationStrategy`
[2] `org.apache.camel.Exchange`
[3] `de.unistuttgart.iaas.framework.eai2camel.routing.aggregator.AggregateExchange`
[4] `de.unistuttgart.iaas.framework.eai2camel.routing.aggregator.AggregationStrategyBest`
[5] `org.apache.camel.Predicate`
[6] `de.unistuttgart.iaas.framework.eai2camel.routing.aggregator.CompletnessPredicateNever`

ism implemented in the `ExtendedAggregationCollection`. This timeout mechanism allows overriding a completeness condition after a certain amount of time passed, and thus marking the exchange as complete. This functionality is not implemented in a completeness condition as the completeness condition is evaluated each time a message arrives for a certain aggregate. If a possible timeout would only be checked on incoming messages, then the timeout might never be detected if no message is arriving anymore for an aggregate. This would render the timeout mechanism useless, thus it is implemented directly in the `ExtendedAggregationCollection`, supported by the `AggregateExchange`. The third completeness condition implemented in the framework is the one implemented in the `CompletnessPredicateMessageCount`[1]. This simple completeness conditions applies to aggregates that aggregated a certain amount of messages.

To correlate an incoming message to an aggregate, i.e. to the messages it should be aggregated with, the `ExtendedAggregationCollection` uses a Camel *Expression*. As already discussed in chapter 3 (3.4.6 Aggregator) this *Expression* needs to return an identifier that can be compared to the one returned by other messages. In conjunction with this framework the `XPathExpression`[2] of Apache Camel can be used to identify a value in the message's XML body as correlation identifier.

4.4 Generation of Apache Camel based Java-code

This section describes how the actual generation of Apache Camel based Java-code is done. It describes how the generator-code is integrated in the graphical editor implemented by [Dru07], and how the messaging system model is traversed to generate the code.

4.4.1 Parameterization of the EAI patterns

In the work of [Dru07] the EAI patterns were analyzed regarding what parameters are needed to use the patterns. These parameters were first analyzed theoretically and then used for the implementation of the Eclipse based editor implemented by [Dru07]. To implement the modeled messaging systems as Camel based Java applications the parameters stored in the messaging system model, developed by [Dru07], had to be used. But for some patterns the supplied parameters were not sufficient to generate a solution that executes this messaging system. The reasons the parameters didn't suffice were mostly not because the theoretical parameterization was lacking a parameter, but because some parameters in the implementation of the editor didn't contain deployment specific information - which was a requirement to the editor design. In particular the patterns that use a web service had only the WSDL port-type and the WSDL operation specified. But to actually invoke the web-service without using any discovery mechanism, the concrete WSDL service and the concrete WSDL port, the operation shall be invoked on, needs to be known.

[1] `de.unistuttgart.iaas.framework.eai2camel.routing.aggregator.CompletnessPredicateMessageCount`

[2] `org.apache.camel.model.language.XPathExpression`

Only the *Aggregator* pattern was lacking a parameter that is not a deployment specific parameter. Namely in the work of [Dru07] the *Aggregator* pattern is lacking a correlation identifier. The reason this parameter was omitted is probably because the implementation of this work was targeted to WS-BPEL. A WS-BPEL process is instance based, i.e. the data flowing through a WS-BPEL process always belongs to a certain process instance. Thus the *Aggregator* implementation in WS-BPEL doesn't need to correlate the "messages" to an aggregate as they are already correlated by the process instance they belong to.

To get the needed additional parameters for the web service, a graphical dialog was created in this work. This dialog is implemented in the `Eai2CamelWebServiceConfigurationDialog`[1] class and is shown for each pattern, needing the information during the code generation. This is not really a user-friendly way of collecting the required parameters, and could be improved by future works. The same approach is used to collect the required correlation identifier for the *Aggregator*, where the dialog is implemented in the `Eai2CamelAggregatorConfigurationDialog`[2] class.

4.4.2 Integration into the Eclipse based Editor

The integration into the Eclipse based Editor from [Dru07] is being done in a minimal invasive fashion. The generator integrates into the editor by implementing a so called `ActionDelegate`[3]. This provides an additional action in form of a button in Eclipse's toolbar and an additional entry in the menu bar. All classes invoked by the `ActionDelegate` are independent of the Eclipse based editor implementation. They just rely on the model[4] used to store the messaging system.

4.4.3 Generation of the Code

The generation starts when the `GenerateCamelActionDelegate`[5] is invoked. It checks the messaging system for errors in the same way the corresponding `ActionDelegate` of the WS-BPEL generating part does. Then it creates a `CamelWriter`[6] instance and invokes its `write()` method. This method adds the Java-Nature to the project the current messaging model is located in. This means it makes the project a java project. Then it adds the libraries that are required to run the generated messaging solution to the Java project. These libraries are listed in section 4.4.4 below. After the Java project has been initialized a package under the `eai2camel.generatedSystems`[7] is created and the created package is populated with some templates.

The actual messaging system is then generated by traversing the messaging system model in a breath-first manner, starting from a node that is an interface to the outside world. Usually this is an "External Service" pattern that has only an outgoing channel connected to it. The reason for travers-

[1] `de.unistuttgart.iaas.eaiparam.editors.dialogs.Eai2CamelWebServiceConfigurationDialog`
[2] de.unistuttgart.iaas.eaiparam.editors.dialogs.Eai2CamelAggregatorConfigurationDialog
[3] `org.eclipse.ui.IWorkbenchWindowActionDelegate`
[4] `de.unistuttgart.iaas.eaiparam.model.*`
[5] `de.unistuttgart.iaas.eaiparam.actions.GenerateCamelActionDelegate`
[6] `de.unistuttgart.iaas.eaiparam.actions.CamelWriter`
[7] `de.unistuttgart.iaas.eaiparam.eai2camel.generatedSystems`

ing the messaging system using breath-first search is to generate a messaging system that is human readable. Using this approach the patterns appear in the generated source code in the order of the message flow.

While traversing the messaging system a generator method for each filter (in the sense of the *Pipes-and-Filters* architecture) of the messaging system is invoked. These generator methods are implemented in a separate class, the `CamelFilterPatternGenerator`[1] class. To create an extensible solution the appropriate generator method for a filter doesn't need to be invoked directly. Instead the `CamelFilterPatternGenerator` offers a `getGeneratorMethod(…)` method which uses introspection to discover the appropriate generator method for the current filter. This enables the implementation of additional patterns by simply adding an appropriate generator-method to the `CamelFilterPatternGenerator` class.

4.4.4 Required Libraries

The generated messaging system requires some Java libraries to be able to run. These libraries and the version used while developing the EAI-2-Camel generator are listed in the following:

- Apache Camel 1.2.0 [Cam] is needed as the generated Java application is based on this Apache Camel.
- The Spring Framework 2.5 [Spr] is a framework that is used by Apache Camel and is thus required.
- The EAI-2-Camel framework, which was developed as part of this work and contains the actual implementation of the EAI patterns, is also needed.
- Apache ActiveMQ 4.1.1 [Act] which is required as it is used as JMS provider for the generated messaging systems.
- Apache CXF 2.0.3 [Cxf] which is used as JAX-WS provider and is thus needed to invoke the web services the different patterns use.

4.5 Example Messaging System

To demonstrate the Java generator and the EAI-2-Camel framework developed in this work an example messaging system has been created. This created example implements the Loan-Broker scenario, where a messaging system is modeled that provides a web service. This web service implements the Loan-Broker scenario by sending the loan quote request through a chain of EAI patterns and returning the resulting message to the web service client. Figure 2 shows the modeled messaging system.

[1] `de.unistuttgart.iaas.eaiparam.actions.CamelFilterPatternGenerator`

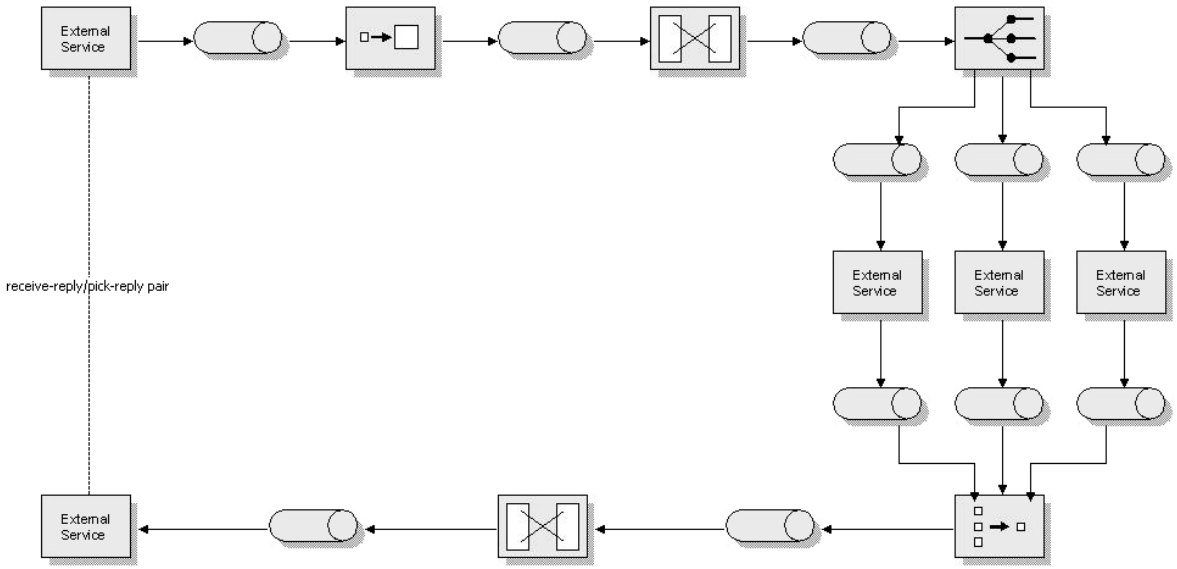


Figure 2: The Loan-Broker messaging system

The modeled messaging system starts with a pair of "External Service" patterns that provide the web service. Incoming request messages are sent to a *Content Enricher* that enriches the request with a credit score by asking a credit bureau for the customer's credit standing. Then the list of banks that fit the customer profile is computed by an XSLT stylesheet and added to the message. Here, a *Message Translator* is used, because the computation is done by an XSLT stylesheet, but actually the creation of the recipient list implements the *Content Enricher* pattern. After this step the message is distributed to the banks according to the recipient list contained in the message, whereupon the responses received from the banks are aggregated to a single message by selecting the best loan offer. Finally the aggregated message is translated to fit the web service's response message type, and then sent as response to the web service client.

To be able to run the generated messaging system implementation of this example, all necessary web services and XSLT stylesheets were implemented as part of this work.

Bibliography

[AAA+07] Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, Alejandro Guízar, Neelakantan Kartha, Canyang Kevin Liu, Rania Khalaf, Dieter König, Mike Marin, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri and Alex Yiu. *Web Services Business Process Execution Language*, Version 2.0. Comitee Specification, April 2007. Available electronically at:
http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf
Accessed on: 07.04.2008

[Act] *Apache ActiveMQ*, Version 4.1.1. Available electronically at:
http://activemq.apache.org/
Accessed on: 02.04.2008

[BL07] David Booth and Canyang Kevin Liu. *Web Services Description Language (WSDL)*, Version: 2.0. W3C Recommendation, June 2007. Available electronically at:
http://www.w3.org/TR/wsdl20-primer/
Accessed on: 13.04.2008

[BO06] Rahul Biswas, Ed Ort: *The Java Persistence API - A Simpler Programming Model for Entity Persistence*. Sun Microsystems Inc., 2006. Available electronically at:
http://java.sun.com/developer/technicalArticles/J2EE/jpa/
Accessed on: 07.04.2008

[BPS+06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler and François Yergeau. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. W3C Recommendation, September 2006. Available electronically at:
http://www.w3.org/TR/xml/
Accessed on: 13.04.2008

[Cam] *Apache Camel*, Version: 1.2.0. Available electronically at:
http://activemq.apache.org/camel/index.html
Accessed on: 02.04.2008

[CD99] James Clark and Steve DeRose. *XML Path Language (XPath)*, Version: 1.0. W3C Recommendation, November 1999. Available electronically at:
http://www.w3.org/TR/xpath/
Accessed on: 07.04.2008

[CHM06] Roberto Chinnici, Marc Hadley and Rajiv Mordani. *The Java API for XML-Based Web Services (JAX-WS) 2.0*. Specification, April 2006. Available electronically at:
http://jcp.org/aboutJava/communityprocess/final/jsr224/index.html
Accessed on: 13.04.2008

[Cla99] James Clark. *XSL Transformations (XSLT)*, Version: 1.0. W3C Recommendation, November 1999. Available electronically at:
http://www.w3.org/TR/xslt/
Accessed on: 13.04.2008

[Cxf] *Apache CXF*, Version 2.0.3. Available electronically at:
http://incubator.apache.org/cxf/
Accessed on: 02.04.2008

[DK06] Linda DeMichiel and Michael Keith. *Enterprise JavaBeans,Version: 3.0*. Specification, May 2006. Available electronically at:
http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html
Accessed on: 07.04.2008

[Dru07] Bettina Druckenmüller. *Parametrisierung von EAI Patterns*. Diplomarbeit Nr. 2583, Universität Stuttgart, 2007.

[Ecl] *Eclipse Development Platform*, Version: 3.2.2. Available electronically at:
http://www.eclipse.org/
Accessed on: 22.11.2007

[FW04] David C. Fallside and Priscilla Walmsley. *XML Schema, Second Edition*. W3C Recommendation, October 2004. Available electronically at:
http://www.w3.org/TR/xmlschema-0/
Accessed on: 13.04.2008

[HBS+] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli and Kate Stout. *Java Message Service Specification Final Release 1.1*. Specification, April 2002. Available electronically at:
http://jcp.org/aboutJava/communityprocess/final/jsr914/index.html
Accessed on: 07.04.2008

[HW03] Gregor Hohpe, Bobby Woolf: *Enterprise Integration Patterns*, Addison-Wesley Professional, 2003, ISBN 0-321-20068-3

[Jdt] *Eclipse: Java Development Tools (JDT)*, Version: 3.2.2. Available electronically at:
http://www.eclipse.org/jdt/
Accessed on: 13.04.2008

[Min] *Apache MINA*. Available electronically at:
http://mina.apache.org/
Accessed on: 02.04.2008

[ML07] Nilo Mitra and Yves Lafon. *SOAP*, Version: 1.2. W3C Recommendation, April 2007. Available electronically at:
http://www.w3.org/TR/soap12-part0/
Accessed on: 13.04.2008

[Ode] *Apache ODE*. Available electronically at:
http://ode.apache.org/
Accessed on: 18.04.2008

[Ser] *Apache ServiceMix*. Available electronically at:
http://servicemix.apache.org/
Accessed on: 02.04.2008

[Spr] *Spring Framework*, Version 2.5. Available electronically at:
http://www.springframework.org/
Accessed on: 04.04.2008

[Vep] *Eclipse: Visual Editor Project*, Version: 1.2. Available electronically at:
http://www.eclipse.org/vep/
Accessed on: 13.04.2008

[Wal06] Norman Walsh. *Java API for XML Processing*. Specification, August 2006. Available electronically at:
http://jcp.org/aboutJava/communityprocess/mrel/jsr206/
Accessed on: 13.04.2008

Appendix

A.1 Camel example using built-in Pipeline

This example demonstrates the differences discussed in chapter 3.3 using Camel's *Pipeline* with synchronous or asynchronous Components, where the asynchronous Component doesn't support `InOut`[1] message *Exchanges*. The example configures two different `Pipelines`[2], one starting from the *Endpoint* "`seda:endpointA`" using the *SEDA-Component* which implements in-memory queues and is thus asynchronous and one starting from the *Endpoint* "`direct:endpointA`" using the *Direct-Component* whose *Producer* directly invokes its *Consumer* and thus is synchronous. When executed, this example sends five test messages to each of the two `Pipelines` and prints the messages received as output from the `Pipeline` to the console including the receiving thread's ID. Also the two Filter Processors print a message to the console on processing, including their thread's ID. The second *Filter Processor* also waits for 0.5 seconds to provoke the parallel processing of the different test messages. The Java code of this example is complete and ready to be compiled. After the code you can find a possible output and a discussion of the output.

A.1.1 Java Code

```
import org.apache.camel.CamelContext;
import org.apache.camel.CamelTemplate;
import org.apache.camel.Exchange;
import org.apache.camel.Message;
import org.apache.camel.Processor;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.impl.DefaultCamelContext;

public class PipesAndFiltersExample {

   public static void main(String[] args) {
      try {
         // create the camel context:
         CamelContext context = new DefaultCamelContext();

         // add the routes:
         context.addRoutes(new RouteBuilder() {
             public void configure() {
```

[1] `org.apache.camel.ExchangePattern.InOut`
[2] `org.apache.camel.processor.Pipeline`

```
// create the message processor for filter 1:
Processor myFilter1Processor = new Processor() {
    public void process(Exchange e) {
        // get the inbound request message:
        Message inboundMessage = e.getIn();
        System.out.println("      Filter 1 ["
           + Thread.currentThread().getId()
           + "] processing:"
           + inboundMessage.getBody(String.class)
                .split("\n")[0]);
        // create the new body:
        String responseBody
           = inboundMessage.getBody(String.class)
           + " – message processed by filter 1"
           + "\n";
        // set the outbound response message:
        Message outboundMessage = e.getOut();
        outboundMessage.setBody(responseBody);
    }
};

// create the message processor for filter 2:
Processor myFilter2Processor = new Processor() {
    public void process(Exchange e) {
        // get the inbound request message:
        Message inboundMessage = e.getIn();
        try {
           // try to sleep 0.5 seconds:
           Thread.sleep(500);
        } catch (InterruptedException ex) {}
        System.out.println("      Filter 2 ["
           + Thread.currentThread().getId()
           + "] processing:"
           + inboundMessage.getBody(String.class)
                .split("\n")[0]);
        // create the new body:
        String responseBody
           = inboundMessage.getBody(String.class)
           + " - message processed by filter 2"
           + "\n";
        // set the outbound response message:
        Message outboundMessage = e.getOut();
        outboundMessage.setBody(responseBody);
    }
};

// create the message processor for endpoint B:
Processor myEndpointBProcessor
   = new Processor() {
    public void process(Exchange e) {
        // get the inbound request message:
        Message inboundMessage = e.getIn();
        System.out.println("Receiving message ["
```

```
                        + Thread.currentThread().getId()
                        + "]: \n"
                        + inboundMessage
                            .getBody(String.class));
                }
            };


            // create the routes for async-processing:
            from("seda:endpointFilter1")
                .process(myFilter1Processor);
            from("seda:endpointFilter2")
                .process(myFilter2Processor);
            from("seda:endpointB")
                .process(myEndpointBProcessor);

            // create the pipeline for async-processing:
           from("seda:endpointA")
               .pipeline("seda:endpointFilter1",
                         "seda:endpointFilter2",
                         "seda:endpointB");



            // create the routes for sync-processing:
            from("direct:endpointFilter1")
                .process(myFilter1Processor);
            from("direct:endpointFilter2")
                .process(myFilter2Processor);
            from("direct:endpointB")
                .process(myEndpointBProcessor);

            // create the pipeline for sync-processing:
           from("direct:endpointA")
               .pipeline("direct:endpointFilter1",
                         "direct:endpointFilter2",
                         "direct:endpointB");
        }
});

// create a camel template for sending messages:
CamelTemplate camelTemplate = new CamelTemplate(context);

// start the camel context
context.start();

// send messages to async pipeline:
System.out.println(
      "#### Example with asynchronous endpoints: ####"
      + "\n\n");
for (int i = 1; i <= 5; i++) {
   camelTemplate.sendBody("seda:endpointA",
```

```
                                    "message body #" + i + "\n");
            }

            Thread.sleep(3000); // wait for 3 seconds

            // send messages to sync pipeline:
            System.out.println("\n\n\n\n"
                  + "#### Example with synchronous endpoints: ####"
                  + "\n\n");;
            for (int i = 1; i <= 5; i++) {
               camelTemplate.sendBody("direct:endpointA",
                                    "message body #" + i + "\n");
            }

            Thread.sleep(5000); // wait for 5 seconds
            context.stop();

         } catch (Exception e) {
            // this is an example -> don't handle exceptions:
            e.printStackTrace();
         }
      }

}
```

A.1.2 Output

```
#### Example with asynchronous endpoints: ####


      Filter 1 [8] processing:message body #1
Receiving message [10]:
message body #1

      Filter 1 [8] processing:message body #2
      Filter 1 [8] processing:message body #3
      Filter 1 [8] processing:message body #4
      Filter 1 [8] processing:message body #5
Receiving message [10]:
message body #2

Receiving message [10]:
message body #3

Receiving message [10]:
message body #4

Receiving message [10]:
message body #5
```

```
      Filter 2 [9] processing:message body #1
      Filter 2 [9] processing:message body #2
      Filter 2 [9] processing:message body #3
      Filter 2 [9] processing:message body #4
      Filter 2 [9] processing:message body #5



#### Example with synchronous endpoints: ####


      Filter 1 [1] processing:message body #1
      Filter 2 [1] processing:message body #1
Receiving message [1]:
message body #1
 – message processed by filter 1
 - message processed by filter 2

      Filter 1 [1] processing:message body #2
      Filter 2 [1] processing:message body #2
Receiving message [1]:
message body #2
 – message processed by filter 1
 - message processed by filter 2

      Filter 1 [1] processing:message body #3
      Filter 2 [1] processing:message body #3
Receiving message [1]:
message body #3
 – message processed by filter 1
 - message processed by filter 2

      Filter 1 [1] processing:message body #4
      Filter 2 [1] processing:message body #4
Receiving message [1]:
message body #4
 – message processed by filter 1
 - message processed by filter 2

      Filter 1 [1] processing:message body #5
      Filter 2 [1] processing:message body #5
Receiving message [1]:
message body #5
 – message processed by filter 1
 - message processed by filter 2
```

A.1.3 Discussion

The output shows two things: First you can see the problem of using asynchronous Components that don't implement the *InOut* message exchange pattern with Camel's *Pipeline*, which was men-

tioned in chapter 3.3, in the first part of the results (#### Example with asynchronous endpoints ####). You can see that the messages leave the Pipeline unaltered due to the fact that the Filters output is not reused by the following Filter, neither it is received by the final Endpoint (for an explanation on why this is, see chapter 3.3). Also you can see that the test messages are received by the final Endpoint even before they are processed by the second Filter Processor (this is also discussed in chapter 3.3). Second you can see in the second part of the results (#### Example with synchronous endpoints ####) that due to the *Direct-Component*, which is synchronous, there is only one message in the *Pipeline* at each point in time (for a discussion see chapter 3.3).

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Stuttgart, 24.04.2008 _____

(Pascal Kolb)