

Enterprise Integration Patterns Exemplified in Java Business Integration

Christoph Hartmann

Seminar Subject-specific English for SST 2006
Hasso-Plattner-Institute for Software Systems Engineering

christoph.hartmann@hpi.uni-potsdam.de

Abstract

For software developers it has become becoming increasingly difficult to ignore design patterns as discussed by numerous books. This paper examines the question whether design patterns are as useful as often argued by the academia. Are design patterns useful in real world scenarios? Do design patterns increase the time of development? Or might they be time-consuming without real value? This paper analyzes the benefit of design patterns in a case study. This attempt is based on Java Business Integration and the Enterprise Integration Patterns. To understand the use of these design patterns, this paper deals with brief descriptions of Service-oriented Architectures, the Enterprise Service Bus and Java Business Integration.

Keywords: Design Patterns, Enterprise Integration Patterns, SOA, ESB, JBI, Web Services

1. Introduction

In general, enterprise applications tend to become very complex. They hold a huge amount of data and the data access should be accomplished in real time. Such applications need to be scalable and have to fit to the corporate structure. Increasing requirements demand the reuse of software at a large scale. To get a general idea of the software structure, sufficient documentation and a software architect are required. Software engineering has become a task of integrating existing software parts. Today's software complexity requires at least one person who has a broad overview. When building a house, an architect might have the architectural overview. Comparably the architecture for each application plays a crucial role in the process of software engineering. Abstract design patterns may suit to describe a solution for the implementation of applications.

In the following sections, this paper will explain design patterns as well as Java Business Integration and aims to demonstrate an example of the use of design patterns, which are mapped to the Java Business Integration specification. This approach will not invent new design patterns, but instead use existing patterns which have been introduced by Hohpe [HW04]. The next section deals with what is understood when talking about "design patterns".

2. Basics

2.1. Design Patterns

In recent years, there has been an increasing interest in design patterns. The notion design patterns is well-known in software engineering since Erich Gamma et al. published the book "Design Patterns - Elements of Object-Oriented Software" [ea95]. At the end of the twenty-first century, this view is further supported by Martin Fowler, who wrote the book "Patterns of Enterprise Application Architecture" [Fow02], and by Gregor Hohpe and Bobby Woolf. The latter ones composed the publication "Enterprise Integration Patterns - Designing, Building and Deploying Messaging Solutions" [HW04].

Firstly, it is necessary to clarify what is exactly meant by the term "design patterns". Surprisingly, the definition for pattern is not specific to software engineering. The term "pattern" is derived from Christopher Alexander, an architect for building complexes. Alexander published the book "A Pattern Language: Towns/Buildings/Construction" in 1977. He defined the term "patterns" as follows:

“Each pattern describes a problem which occurs over and over again in our environment, and then describes a core of the solutions to that problem, in such a way that you can use that solution a million times over, without ever doing it the same way twice”¹

A pattern makes a solution reusable due to its independence from a specific implementation. As a result of using design patterns, software developers might have a common understanding of a problem and this leads to an effective way for easier communication. According to Gamma et al. [ea95]², a pattern is generally classified into four essential elements: *pattern name*, *problem*, *solution* and *consequences*. The *pattern name* should be short, meaningful and easy to bear in mind. The associated *problem* describes specific design problems and probably a list of preconditions, which must be met before applying the pattern. As a consequence of this problem, the *solution* describes a template which could be applied if this described problem occurs. However, the solution provided is not intended to describe a concrete implementation. A complete pattern description includes a list of *consequences* to support architects to evaluate design alternatives, estimate costs or benefits.

Furthermore, design patterns are not always suitable. Thus, it might happen that software engineers might try to solve less complex problems by using no suitable patterns. Therefore, it could be concluded that the original problem is transformed to a more difficult one. A further disadvantage might be that the use of design patterns could lead to reduced creativity in programming due to the frequent use of design patterns. Finally, design patterns are not scientifically proved. Nevertheless, they are used in practice. The more a design pattern is used, the more it is considered to be valuable. Therefore, design patterns cannot be invented on a theoretical basis. They need to be found in real world scenarios.

2.2. Integration

Integration is an act of combining two or more things in such a way that they work together. In this respect, integration is connected to enterprise applications and information technology. Enterprise applications store a huge amount of persistent data which are accessed simultaneously by multiple users.

Usually, enterprises use more than one enterprise application. For instance, the SAP enterprise resource planning system is linked to specialized software for production. However, the connection between different applications often becomes highly complex. Therefore, many business and technical problems appear quite frequently.

According to Hohpe [HW04]³, the integration of enterprise applications overcome four main challenges. First of all, an *unreliable network* connection, which is caused by the failure of computers, routers or network cable, could emerge. Secondly, the communication over a network is *slower* than a local method call. Thirdly, by connecting two or more applications, an interface is required for communication. In general, two applications are *different* and do not base on the same operating system, programming language or data format. Finally, every vendor develops its application differently. Application *changes*, for instance after an upgrade, *are inevitable* and, as a consequence, existing interfaces may not work properly any longer.

However, changes to legacy systems are not possible in most cases. Missing documentation and interface descriptions as well as the few integrations standards (such as XML, XSL or Web Services) aggravate the task of enterprise application integration. For these problems, four integration approaches were found in the past. In early stages, *file transfer* was used to exchange data between multiple applications. Reader and writer could not access a file at the same time. Additionally, the reader does not get information when a data change occurs. Hence, the reader has to read the data periodically. The next step in the development was the *shared database*. The database separates the data access from the storage of the data. Compared to the file transfer, the shared database allows multiple readers since the data is stored in a common database. Though, the shared database did not send data change events. The next development was the *remote procedure call* (RPC). To increase scalability, layers in a software need to be fine-grained. As mentioned above, the file transfer and the shared database only separate the code from the stored data. This allows the separation of code functions. Each application exposes some procedures running on different computers. All the remote software parts build up a whole system. Unfortunately,

¹ [ea77] p. X

² [ea95] p. 3

³ [HW04] p. XXIX

RPC has the problem that if one part of the software crashes the whole system does not function anymore.

Finally, each application is connected to a common messaging system. This fourth solution for integration problems is called *messaging*. Applications exchange data by using messages. Due to the use of messaging systems the applications are independent of a platform and a programming language. Consequently, it is easier to communicate in distributed systems. In addition, messaging is asynchronous. The receiver does not need to be online when the sender is sending the message. The messaging system stores the message until the receiver is ready to communicate.

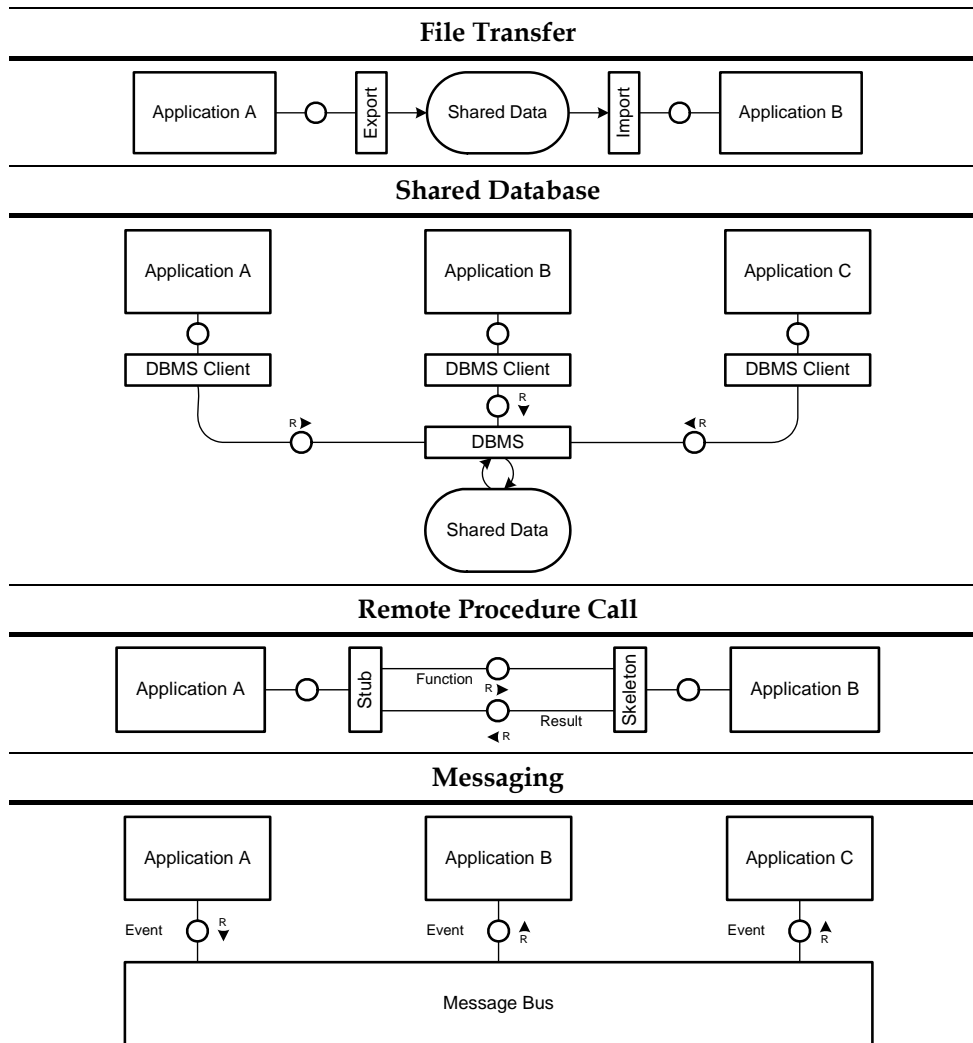


Figure 1: Integration approaches

Messaging consists of five main steps: *Create, Send, Deliver, Receive, Process*. The sending application creates a message and sends the message to the message bus. The bus stores the message and forwards it to the receiver. The receiver acknowledges the receipt and processes the message. The following figure illustrates this point.

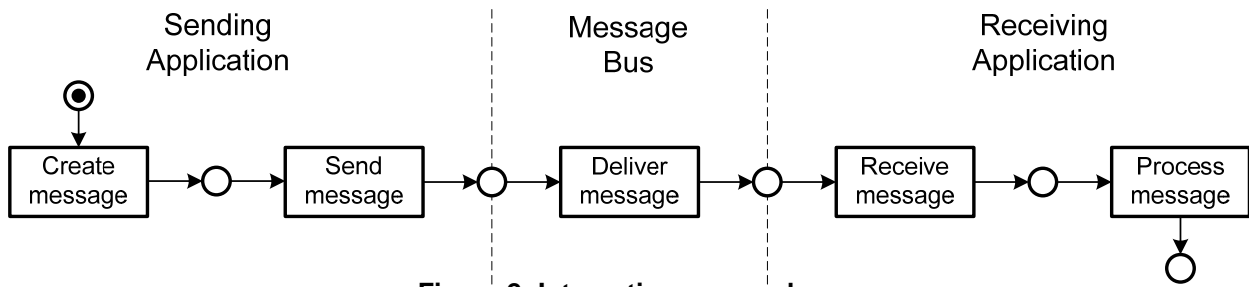


Figure 2: Integration approaches

To conclude, Hohpe [HW04] identifies *remote communication, variable timing, throttling, reliable communication, disconnected operation, mediation, thread management, platform/language integration* and *asynchronous communication* as benefits of a messaging approach.

2.3. Service-oriented Architecture

The hype of Service-oriented Architecture (SOA) emerged in the last decade. Although this kind of architecture was considered to be new, it had similarities with existing technologies such as Enterprise Application Integration (EAI). Various definitions of the term SOA have been suggested, no common definition exists, though. One definition by Natis and Schulte is "Service-oriented architecture (SOA) is a client/server software design approach in which an application consists of software services and software service consumers (also known as clients or service requesters). SOA differs from the more general client/server model in its definitive emphasis on loose coupling between software components, and in its use of separately standing interfaces."⁴

This paper understands the term "service" in the field of software engineering as reusable domain specific functions located in a "service repository". They are synchronously or asynchronously callable. "Modules" can be seen as the ancestors of services. A service is shared enterprise-wide and it is more abstract than a module. In combination with SOA in an enterprise, services are considered as Web Services in most cases. A Web Service is a piece of software available over the Internet and utilizing XML as message format.

Due to this paper cannot cover the wide range of SOA [Erl05], [DKD04] and [EP06] are recommended for further reading.

2.4. Enterprise Service Bus

An Enterprise Service Bus (ESB) is an infrastructure used for composite SOA applications. It combines *messaging, data transformation, reliable routing* and *Web Services*. Due to the massive use of messages the ESB is based on Message-oriented Middleware (MOM). The following picture serves to the main idea. The services and the client are connected to the message bus. Every piece of exchanged information is routed over the message bus. The services are added via adaptors to the message bus.

In contrast to EAI, an ESB is based on XML and all related standards (e.g. XSD, SOAP, WSDL). To run business processes, a message bus includes a process manager in most cases. For further reading [Cha04] is recommended.

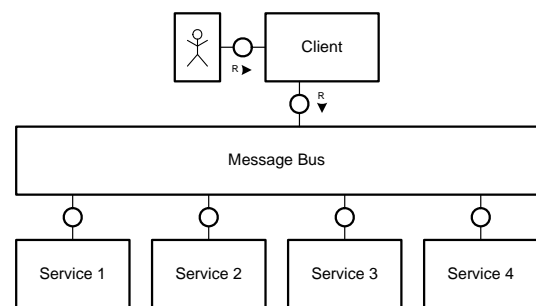


Figure 3: Enterprise Service Bus

⁴ [NS03]

2.5. Java Business Integration

The Java Business Integration (JBI) is a specification of the Java Community Process (JCP)⁵ with the identification JSR⁶ 208. JBI defines an ESB implementation in Java and specifies four core elements: *packaging*, *message routing*, *component life cycle* and *management*. Every message sent from the message bus is required to be in this normalized form. JBI uses XML as message exchange format.

JBI has an extensible design by making use of pluggable integration of components. The definition of the package mechanism is necessary to reuse a JBI-based application in different JBI-based ESBs. For developers this has the advantage that the application runs on JBI compliant software from different vendors. The most important types of components are *service engines* (SE) and *binding components* (BC). A service engine contains business logic. A binding component handles protocol-specific functionality such as FTP, HTTP or SOAP and should not contain business logic. Binding components allow the reuse of protocol handling in different service engines. Additionally, two component types exist. A *shared library* (SL) contains functionality that is often used in different JBI components. To deploy JBI applications to customers, a composite component is required. For that reason, a *service assembly* is capable of containing multiple service engines, binding components or service libraries.

After this brief introduction, I will focus on the JBI communication process. On the basis of this process, a trial for applying enterprise application patterns will be carried out. The JBI specification defines four core message exchange patterns: *in-only*, *robust in-only*, *request-response* and *in-optional out*. The request-response is the most significant one because it includes the activities of the others.

In the following, all communication between the two parties, namely a service provider and a service consumer, will be explained. The service consumer invokes a request on delivery channel to the message router. Every JBI component has exactly one delivery channel to the message router. If the message is not normalized, a binding component has to transform the bound message (e.g. FTP⁷) into an XML message. The XML message in turn is encapsulated in a JBI *Message Exchange*. After the message router receives the Message Exchange it will be put on the delivery channel of the target JBI component. The target component processes the Message Exchange and a response message is returned through the message router. The message router is only able to handle normalized JBI Message Exchange and therefore the router is named *Normalized Message Router*. The process flow is shown in the figure 5 below. For further reading "Introduction into the Java Business Integration" [har06] and "JBI Specification" [mic] is recommended.

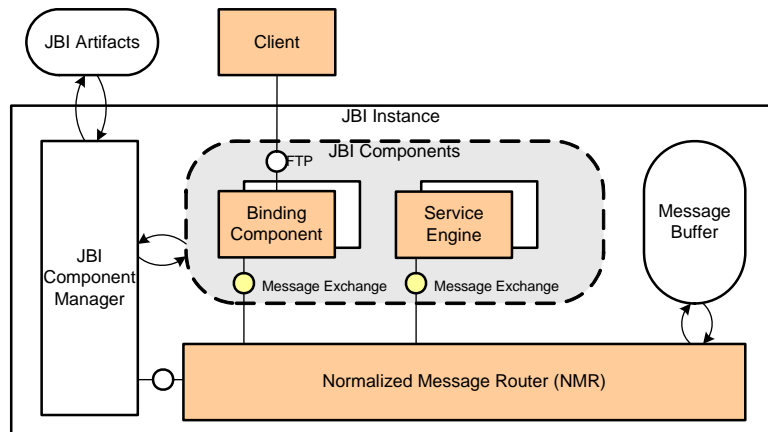


Figure 4: Java Business Integration Architecture

⁵ <http://www.jcp.org>

⁶ JSR - Java Specification Request. This is a document submitted to JCP to propose the development of a new specification or significant revision to an existing specification (source: <http://www.jcp.org>)

⁷ FTP - File Transfer Protocol is used to transfer files. (<http://www.ietf.org/rfc/rfc0959.txt>)

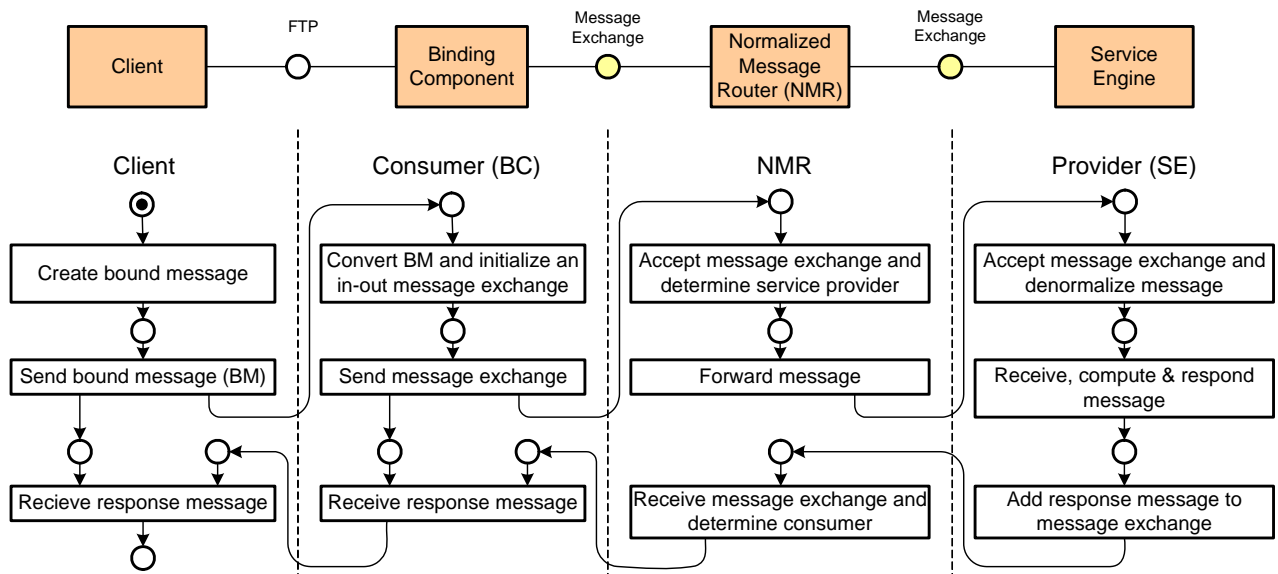


Figure 5: Request-response exchange pattern

3. Analysis

As a basis for the following analysis, the communication process described above is used. A suitable pattern for every action in a request-response communication is identified. The analysis is divided into three parts. Firstly, the focus will lie on the communication between the client and the JBI component. Secondly, a closer look at the communication between two JBI components follows. Finally, the execution within a service engine will be described.

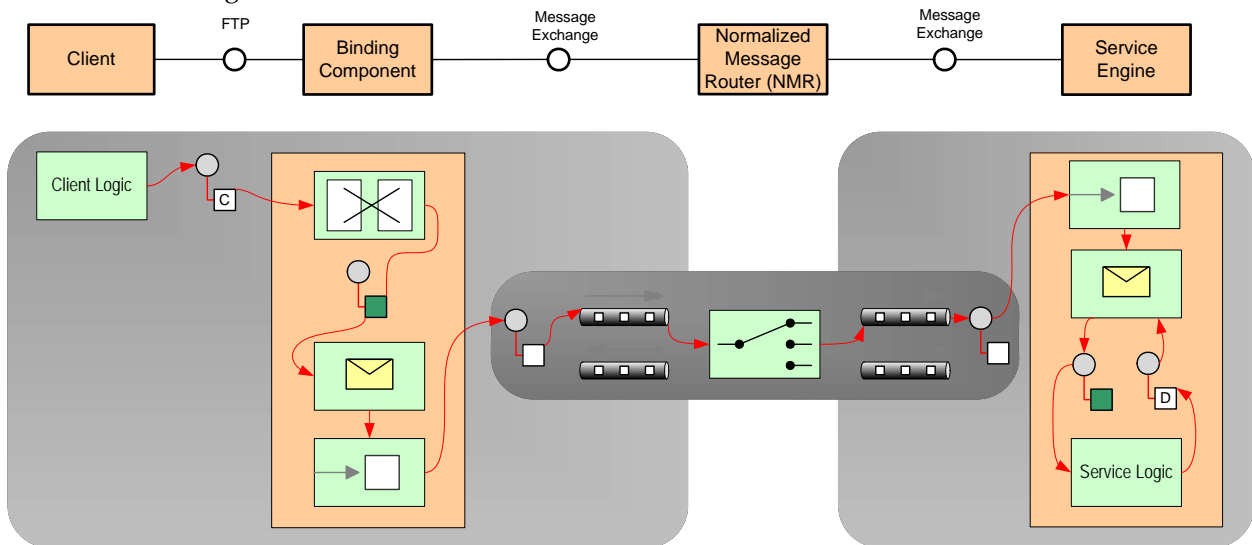


Figure 6: Enterprise integration patterns applied for Java Business Integration

3.1. Communication Client - JBI Component

The client sends a message to the binding component. The sent message is directly mapped to the *Message* (66)⁸ pattern. In general, a message consists of two parts. The header contains information necessary for the messaging software, whereas the body holds the content. In the given case study the client sends a FTP message. The more fine-grained pattern for a FTP message is a *Command Message* (145). An FTP message is a bound message and needs to be transformed into a normalized message. This transformation from the FTP message into a XML message is accomplished by a *Message Translator* (85).

⁸ Numbers at the design pattern name indicate the page number in [HW04]

This pattern is often used with legacy applications in which the data format cannot be changed easily. By using a translator, an interaction with external business partners via a standardized data format is possible.

For the transformed message the *Document Message* (147) pattern fits. A document message pattern passes data and lets the receiver decide how to deal with the data. A successful transmission is important for messaging. To enhance reliability, a guaranteed delivery or message expiration pattern may be considered. JBI specifies that a XML message is packed into a Message Exchange. A suitable pattern for doing this is the *Envelope Wrapper* (330). This is a specialized *Message Translator* (85) which takes the raw message and transforms it into a message format that complies with the messaging system. The envelope contains additional information such as security and endpoint information that is required for message delivery. At the destination an un-wrapper reverses the modification by removing the header and all additional information.

To connect to the messaging system, a *Message Endpoint* (95) is used to send and receive messages. The endpoint patterns encapsulate the messaging system from the rest of the application. This pattern helps to change the messaging system without affecting the application. Ideally, the message endpoint source code could be rewritten by leaving the rest unchanged. To complete the communication with the messaging system, the *Message Channel* (60) delivers the given message from the endpoint to the messaging system. The message endpoint is only the interface to the messaging system. Since JBI only sends normalized messages, a *Datatype Channel* (111) helps to implement this feature. A *Datatype Channel* (111) accepts only one type of data. The separation between *Message Endpoint* (95) and *Datatype Channel* (111) helps exchanging parts of the messaging system with less effort. Now, the message is delivered to the message router.

3.2. Communication JBI Component – Router – JBI Component

The Normalized Message Router receives its messages over a channel. The *Message Router* (78) pattern decouples sender and receiver of a message. The key benefit of the message router is that the decision criteria maintain in a single location. The surrounded components are not aware of the existence of the message router.

The router sends the Message Exchange through the *Datatype Channel* (111) to the service engine. There, the message will be processed. However, the message router needs knowledge about all possible destination channels. In critical business processes this might be a bottleneck. Multiple usage of message routers is suggested to increase performance.

A *Message Bus* (137) is a combination of *Datatype Channel* (111) and *Message Router* (78). The bus allows different systems to communicate through a shared set of interfaces and serves as a universal connector. So, a message bus forms a simple useful Service-oriented Architecture. Such solutions are difficult to understand due to loose coupling. A message history inspection will be helpful in such cases.

JBI does not specify a realization of the message bus. Thus, a refinement of the patterns is not possible. The next section deals with the process of a message within a service engine.

3.3. Execution within a JBI Component

After receiving the normalized message the *Envelope Wrapper* (330) unwraps the message and extracts the raw message. Then, the raw message can be processed with business logic for instance BPEL⁹ in a SOA. For the implementation of a BPEL engine, the pattern *Process Manager* (312) is particularly suitable. The manager executes any sequence of steps, either sequential or in parallel. The result of the process will be sent back as mentioned in the sections 3.2. Due to the fact that any service engine contains different business logic, a closer look is not possible. In most cases enterprise integration patterns are not appropriate for this task.

⁹ BPEL – Business Process Execution Language. With BPEL a business process can be described as Web Services (http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel)

All applied pattern can only serves as a suggestion. Every pattern could be replaced with more fine-grained patterns. The purpose of this paper was to show that patterns are useful for common programming jobs in practice. The case study has illustrated that parts of the specification can be successfully described with patterns.

4. Conclusion

This paper has investigated whether patterns could help in real world scenarios. These findings support the argument that patterns can help decrease development time by improving the efficiency of communication between developers.

Research has shown that the communication process of JBI can be described with design patterns. With these patterns, it is easier to understand how an implementation with patterns is possible. As a result, the task of implementing JBI is dividable into a set of smaller tasks. This is necessary in order to implement a large application in team work.

Although the design patterns are useful for software engineering, the main problem of these patterns remains its granularity. The use of fine-grained patterns is questionable. Moreover, the identification of patterns can be very difficult and time-consuming. Concluding, these findings suggest a use of patterns in large systems, but developers should remain open-minded for other solutions as well.

However, the scope of this paper underlies some restrictions. This paper could only discuss integration patterns, although other design patterns such as object-oriented patterns, enterprise application architecture patterns or SOA patterns exist.

References

- [AZ05] P. Avgeriou and U. Zdun. *Architectural patterns revisited—a pattern language*. 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), Irsee, Germany, July, 2005.
- [BD05] S. Baker and S. Dobson. *Comparing service-oriented and distributed object architectures*. Proceedings of the International Symposium on Distributed Objects and Applications, LNCS. Springer Verlag, 2005.
- [BDtH05] A. Barros, M. Dumas, and A.H.M. ter Hofstede. *Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection*. 2005.
- [Bus03] Christoph Bussler. *B2B Integration*. Springer, 2003.
- [Cha04] David A. Chappell. *Enterprise Service Bus - Theory in Practice*. O'Reilly, 2004.
- [Dic98] Alan Dickman. *Designing Applications with MSMQ*. Addison-Wesley, 1998.
- [DKD04] K. Banke D. Krafzig and D.Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices*. Prentice Hall, 2004.
- [ea77] Alexander et al. *A Pattern Language*. Oxford, 1977.
- [ea95] Gamma et al. *Design Pattern - Elements of Reusable Object-Orientated Software*. Addison-Wesley, 1995.
- [ea98] William J. Brown et. al. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1998.
- [ea03] Deepak Alur et al. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2003.
- [EP06] Hugh Taylor Eric Pullier. *Understanding Enterprise SOA*. Manning Publications, 2006.
- [Erl05] Thomas Erl. *Service-Oriented Architecture*. Pearson Education, 2005.
- [Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.
- [Har06] Ch. Hartmann. *Einführung in Java Business Integration*. Seminar System Modeling, 2006.
- [HW04] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions*. Pearson Education, 2004.
- [Jur01] Matjaz B. Juric. *Professional J2EE EAI*. Wrox Press, 2001.
- [Kee05a] Martin Keen. *Patterns: Extended Enterprise SOA and Web Services*. IBM redbooks, 2005.
- [Kee05b] Martin Keen. *Patterns: Implementing an SOA Using an Enterprise Service Bus*. IBM redbooks, 2005.
- [Kee05c] Martin Keen. *Patterns: Integrating Enterprise Service Buses in a Service-Oriented Architecture*. IBM redbooks, 2005.
- [Lin99] David S. Linthicum. *Enterprise Application Integration*. Addison-Wesley, 1999.
- [Lin03] David S. Linthicum. *Next Generation Application Integration*. Addison-Wesley, 2003.
- [Mic] Sun Microsystems. *JBI Specification JSR 208*. <http://jcp.org/en/jsr/detail?id=208>.
- [MNN04] J. Mendling, G. Neumann, and M. Nuttgens. *A Comparison of XML Interchange Formats for Business Process Modelling*. Proceedings of EMISA, pages 129–140, 2004.
- [NS03] Natis, Yefim V.; Schulte, Roy W.: Introduction to Service-Oriented Architecture. Gartner Research, ID Number: SPA-19-5971, 2003-04-14.
- [Ste] A. Sterkin. *Teaching Design Patterns*.
- [Wei] M. Weiss. *Patterns for Web Applications*. Proc. PLOP 2003.