Interface Construction, Deployment and Operation – a Mystery Solved

Alexander Hagemann
Hamburger Hafen und Logistik AG
Bei St. Annen 1
20457 Hamburg, Germany
Email: hagemann@hhla.de

Gerrit Krepinsky
Hamburger Hafen und Logistik AG
Bei St. Annen 1
20457 Hamburg, Germany
Email: krepinsky@hhla.de

Christian Wolf
Hamburger Hafen und Logistik AG
Bei St. Annen 1
20457 Hamburg, Germany
Email: wolf@hhla.de

Abstract—**The increasing digitalization pressure within the industry resulted in a continuously growing demand on IT supported business processes over the past decades. This pressure changed singular mainframe applications into large, distributed application landscapes usually operated in a 24/7 hours mode. Simultaneously, this enforced increased support demands to the application management as well as further application integration requirements during development. Therefore, decoupled, robust and supervise able applications are required. Since the behavior of these applications is determined solely by their communication behavior on interfaces, it becomes apparent that interfaces are of overall significance within such distributed application landscapes. But in our experience, interfaces usually do not get the required attention during design, construction, deployment and operation, which is in contrast to their importance. Instead, only technical reports like, e.g., syntactical descriptions, are usually given and important functional as well as operational aspects have been omitted. This leads to unstable and unnecessary complex interface implementations threatening the 24/7 hours mode of operation. To address the aforementioned issues, this paper contributes a new comprehensive approach on interface design, construction, deployment and operation for distributed application landscapes. This includes guidelines for a functional interface design and interface migration patterns for deploying application interfaces into a 24/7 hours running environment.**

Keywords–interface; interface aspects; communication requirements; communication services; messaging; communication error handling; business process; interface design; interface versioning; interface migration; interface operation.

I. Introduction

In recent years, growing business demands enforced an increasing information technology (IT) support of many business processes. To rule the resulting functional complexity within the IT, several applications are usually necessary. A direct consequence of this fragmentation is the distribution of business processes over applications, which have to communicate with each other in order to fulfill the requirements of the business processes. This communication requires well defined interfaces between applications due to functional [1] as well as technical reasons.

Generally, the design of application interfaces is a difficult and critical task [1], [2], [3], since the behavior of applications belonging to the class of reactive systems, i.e., applications responding continuously to the environment, is determined by their interfaces only [4]. Consequently, badly designed interfaces may lead to functional misbehavior and may propagate internal application problems directly to communication partners [5], [6].

Wrong assumptions during the interface design phase about functional domain behavior and communication technologies can have devastating effects with respect to interface operations and the integration of further applications into the application landscape [7], [8]. Furthermore, interfaces have relatively long life cycles and their implementations are usually costly to modify. A change of an interface specification always requires either its backward compatibility to previous interface versions, or a change of all applications implementing the interface.

Operating a large application landscape in a 24/7 hours mode imposes additional challenges concerning interface deployment and operation because all applications implementing a new interface must be launched into an already running environment [8]. Once this has been done, interface supervision and communication error detection by the application management are also necessary to enable the agreed operational availability of the application landscape [9].

To overcome the above mentioned restrictions, this paper gives an overall overview on all aspects concerning interfaces. It extends the approach for functional interface design and interface transition into operation presented in [1] by adding further important aspects like communication technology selection and transmission protocol definition, including error handling and runtime supervision. Further examples are given presenting an implementation of the proposed transmission protocol and illustrating the functional interface design approach in more detail.

Beginning with a review on related work in Section II, Section III presents some interface theory and resulting aspects necessary to be considered for a successful interface design. By introducing typical requirements enabling a founded selection of an appropriate communication technology in Section IV, Section V deals with a transmission protocol offering further communication services thus enabling a reliable and robust communication. Further details on error handling and communication supervision are described in Section VI, followed by a conrete implementation example of the transmission protocol in Section VII.

Thereafter, Section VIII gives an introduction and comparison of different design approaches for the construction of a functional interface specification, followed by an industrial example of a concrete interface design in Section IX. Finally, Section X deals with the interface launch into an already running application landscape.

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

II. Related work

The increased distribution and complexity of business functionalities enforces IT personal responsible for complex application landscapes to use well-designed integration solutions. Due to the ever increasing dependencies between applications on functional, semantic, technical and operational levels, a holistic integration approach is necessary to successfully manage changes within application landscapes [8]. Thus, the design of interfaces becomes the key for successful application integration.

Within the literature, a lot of information exists regarding different aspects of interfaces like performance, reliability, routing etc. Typically, these documents either deal with technical protocols only and omit functional interface properties, like, e.g., the Internet Protocol (IP) [10], the Hypertext Transfer Protocol (HTTP) [11], File Transfer Protocol (FTP) [12], Java Messaging Service (JMS) [13], Remote Method Invocation (RMI) [14], Advanced Message Queueing Protocol (AMQP) [15] and the Blink Protocol [16], or are bound to specific functional domains like the Financial Information eXchange [17], the FIX Adapted for Streaming [18] or the various United Nations Electronic Data Interchange for Administration, Commerce and Transport (UN/EDIFACT) protocols [19]. But none of them gives explicit guidelines for an interface design. Other common approaches like service oriented architectures (SOA) [20] or the representational state transfer (REST) [21] protocol represent rather general architectural styles. Both are more suitable giving architectural guidelines for application design, than for the construction of concrete interfaces.

More specific work on interface design has been done by Henning [2] and Bloch [3] defining principals and high level processes for good interface designs. Both authors argue that it is hard to design good interfaces due to the required understanding of the underlying functional context. Additional aspects on interface design have been introduced by Iridon [22] to decouple application implementations from domain models [7], using a canonical model, as well as Bonati et al. [8] for supporting different interface versions in parallel during operations.

However, none of these works gives a holistic view on interface design, construction, deployment and operation as presented in this paper. This includes specific guidelines for a functional interface design as well as interface migration patterns for deploying application interfaces into an environment operating 24/7.

III. Interface basics

In order to design an appropriate interface, the general structure of an interface must be considered. Once this has been done, it will become obvious which information must be provided to define an interface. Drilling down into an interface, which is located in the application layer in the Open Systems Interconnection model (OSI model) [23], [24], typically, a three layered structure, as shown in Figure 1, becomes visible. Each of these layers has a dedicated important purpose that can be summarized as follows:

- *functional layer*: this topmost layer is responsible for the functional semantics of the information exchanged. Using the analog of natural speech, the functional layer defines the meaning of words spoken.

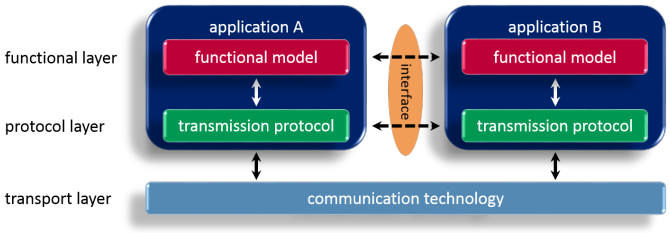

Figure 1. The different layers of an interface. Dotted arrows denote virtual connections within each layer. The communication takes part using the connections denoted by solid arrows. Note that the functional and the protocol layers are located within the application layer of the OSI model, while the transport layer typically encapsulates all layers below the application layer [23], [24].

- *protocol layer*: Within this layer the transmission protocol used to exchange the information is defined. Similar to natural speech, the protocol layer represents the language spoken, e.g., English.
- *transport layer*: Here, the necessary physical transportation of the information is carried out. This layer correlates to the signal transfer using sound waves in a manner similar to natural speech.

Each of these layers communicates virtually with its counterpart located at the other application. Therefore, a layer physically passes the information to its underlying layers until the information is physically transported to the other application. At this point, the information is passed upwards up to the corresponding layer. Only if both sides within one layer use identical functional models or transmission protocols, respectively, communication will take place. Otherwise, the communication is broken.

Given the layered structure of an interface, different aspects arise, which must be considered during the design, implementation, integration and operational phases of an interface life cycle. These aspects focus on different issues and enable the development of robust interfaces. All aspects are independent with respect to each other, focusing on a specific property an interface must satisfy.

A. Functional aspect

While two or more applications are communicating with each other over an interface, the applications assume different functional roles, called *server* and *client*, respectively. An application is called *server* with respect to an interface if it is responsible for the business objects, business events and related business functions that are exposed to other applications through this interface. If business objects and business functions of different business processes are affected, the necessary messages may be combined into a single interface.

Providing an interface is equivalent to defining an interface contract [8] that must be signed by an application in order to communicate with the *server*. The interface may support synchronous and asynchronous communication as well as message flows in both directions, i.e., sending and receiving messages.

Note that this definition deviates slightly from the commonly used client-server definition where the *server* offers a service

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

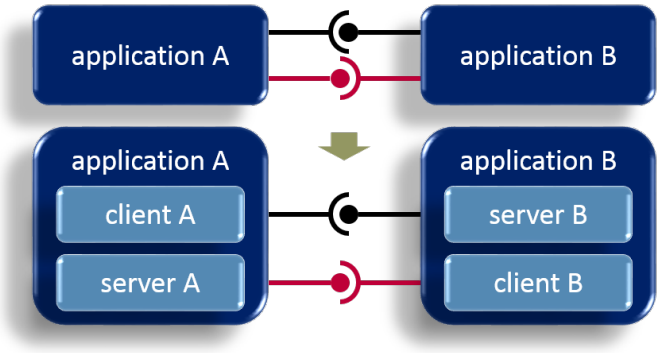

Figure 2. Applications assuming multiple roles simultaneously, i.e., one part of the application serves as a *server* while another part serves as a *client*, as indicated in the top part of the figure. Note that both supported interfaces are not identical.

which can be accessed by *clients* via a synchronous request-reply communication protocol only [25]. Because synchronous communications couples *server* and *client* tightly at runtime, asynchronously based communication should generally be preferred, avoiding these disadvantages [9].

A *client* is an application consuming an interface provided by a *server*. Despite the fact that the interface contract is initially defined by the *server*, a common agreement on the contract is made when the *client* connects to the *server*. Thereafter, none of the participating applications may change the interface contract without agreement of the other party.

Often an application assumes multiple roles with respect to different interfaces concurrently, i.e., the application can be *server* and *client* simultaneously, see Figure 2. It is important to emphasize that this behavior is valid with respect to different interfaces only while for a single interface, the roles of the participating applications are always unambiguous.

B. Semantical aspect

The semantical aspect focuses on the kind of information that may be exposed by the *server* via an interface. Generally, any internal implementation detail of the *server*, i.e., the server model, must never be exposed on an interface. Instead, the information exposed must always be tied to the underlying business processes, thus binding the interface implementation to the domain model [7], [8].

Integration within an IT application landscape requires the decoupling of business and software design due to different responsibilities. In other words the domain model and its related implementations in the *server* and *client* usually have different life cycles, which must be decoupled to reduce the dependencies between business and software development. Therefore, an integration model, linked to the domain model, should be used on interfaces [9], [22], which decouples their implementations from the domain model. Additionally, the integration model conceals all internal application models and details thus decoupling *server* and *client* from each other, as shown in Figure 3.

An interface itself consists of a set of messages, containing *business objects* and *business events* only [26]. This set of exposed information is naturally restricted due to the responsibility of the *server*, i.e., only the *business objects* or *business events* the *server* is responsible for may be communicated via the interface.

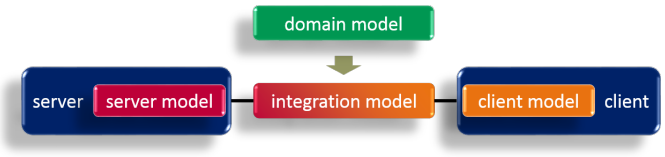

Figure 3. Decoupling the domain model from its implementation using an integration model. Note that different *server* and *client* implementation models are also decoupled in this way [22].

C. Technical aspect

Exchanging information via an interface using *messaging* [9], i.e., by sending and receiving messages, results in specific communication styles depending on the thread behavior of the sender. The resulting communication style should be indicated on the interface using different message types as follows:

- *asynchronous communication*: a message of type *Notification* is sent and the thread resumes execution.
- *synchronous communication*: a message of type *Request* is sent and the thread execution is stopped until a message of type *Reply* has been received.
- *communication supervision*: a message of type *Error* is used to asynchronously report communication errors, that cannot be handeled by the application itself, to the application management. Introducing *Error* messages as an own type emphasizes their semantical difference to *Notifications*.

This set of message types leads to a technical view of an interface. Depending on the functional role of an application, not all message types may be send by all applications. Instead, valid message types sent by an application depend on its functional role with respect to the interface considered.

As depicted in Figure 4, a *server* may send *Notification* and *Reply* messages only. The former message type represents a business event communicated to the environment of the *server*. The *Reply* answers a synchronous *Request* of the *client*. A *server* may not send a *Request* to a *client* due to the functional responsibility of the requested data; doing this would reverse the roles of *server* and *client* and therefore, this *Request* must be part of another interface provided by the former *client*.

The *client* may send *Notifications* and *Requests* to the *server*. Both message types represent accesses to services provided by the *server*. A *Reply* may not be send by a *client* to a *server* because a *client* has no functional responsibility with respect to the interface considered. Again, doing this would reverse the roles of *client* and *server* between both applications. The fourth message type, the it Error message, may be send by all applications in case of communication problems only. This message type is by no means part of a functional interface and send to dedicated communication channels only. In fact, *Error* messages are part of the interface to the application management for supervising ongoing IT operations.

D. Dynamical aspect

An important aspect of an interface is its dynamic behavior describing all valid message sequences on the interface. Since

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

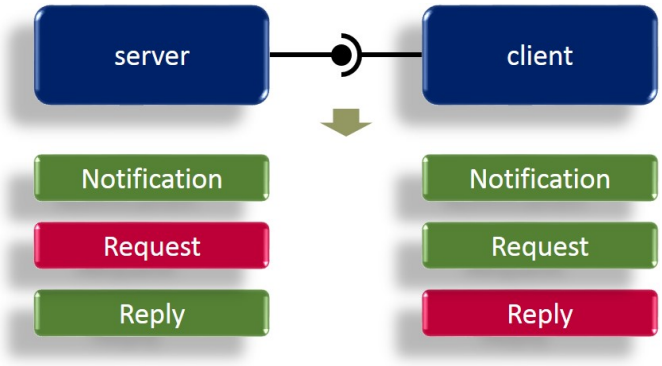

Figure 4. Representation of the message types that a *server* or a *client* may send. Message types depicted in green are allowed to send while red message types may only be received. The *Error* message type has been omitted due to its pure technical nature.

all messages received are processed within a specific context inside the application, there exist important constraints with respect to the message sequence. Thus, a message received out of the defined sequence will not be processed by the application, instead this will result in an error.

Consequently, the dynamic behavior must be described using an appropriate description. Using sequence diagrams of the Unified Modeling Language (UML) [27] is not sufficient for this case, since they describe specific communication examples only. Especially runtime problems, e.g., race conditions, cannot be described holistically using sequence diagrams. Instead, it is strongly recommended to use finite state machines [28], which allow a complete description of the dynamic behavior, as shown in the example of an interface between a container device and a terminal control system depicted in Figure 5.

E. Nonfunctional aspect

Interfaces must be able to provide enough context to execute the desired parts of the business processes within an application since they separate the application from its environment and determine its behavior [4]. Besides the functional and semantical aspects imposed on an interface, this context contains nonfunctional design aspects also, covering robustness, performance and understandability. An application

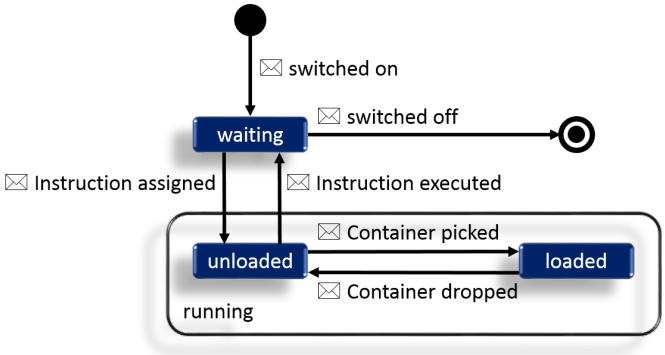

Figure 5. Example of a simplified state machine describing the dynamic behavior of an interface.

will correctly execute the business processes only if these nonfunctional aspects are fully satisfied.

The *robustness* of an interface is crucial with respect to the stability of the overall application landscape. Poorly designed interfaces may propagate internal application errors during runtime, thus causing severe damage within other applications of the application landscape [2], [5].

Robustness of interfaces is achieved by

- *functional coherence*, i.e., assigning a unique functional responsibility to each application according to the underlying domain model thus avoiding inconsistent states of business objects within the application landscape,
- a *non-transactional behavior* between applications [29] thus dropping complex and time consuming coordination problems resulting from the underlying commit protocols [30],
- *integrated content constraints*, i.e., using functional appropriate range restrictions of field values, and
- an *independent field structure* where the content of fields must not depend on the content of other fields within the same message.

Obviously, interfaces must satisfy the required *performance* too. Otherwise, business processes will not work correctly since required business functions may not be executed in time [4]. Using

- a *minimalised interface design*, i.e., a design containing a minimal set of messages only without imposing undue inconvenience for the clients [2] and
- a purely *asynchronous communication* style thus technically decoupling the *client* from the *server* at runtime [31]

covers performance issues at design time already.

Furthermore, well designed interfaces must have a strong and documented relation to the underlying business context thus being easier to learn, remember and use correctly [2], [7]. This will enhance the interface cost efficiency over time due to the enhanced acceptance of the interface within the development teams. This *understandability* can be achieved through

- a functional consistent *message naming schema*, where message names must be functionally meaningful and short to support a clear and understandable integration model on the interface and
- its binding to the application model using an appropriate adapter, see Figure 6 for details.

F. Operational aspect

Usually, each interface requires the usage of a specific infrastructure depending on the used communication technology, e.g., a web server in case of REST over HTTP or a JMS server. To ensure the correct usage of an interface the required infrastructure, its deployment and the communication channel topology must be defined also. The latter one defines the general communication structure, i.e., broadcast or point-to-point communication [9].

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

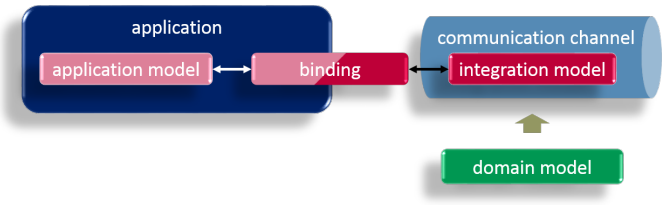

Figure 6. Binding of the integration model to a given application model. Note that the integration model is aligned to the domain model. The required adapter, implementing this binding, can either be a part of the application itself or be realised as a seperate component.

G. Interface components

Given the different aspects presented so far, each of them describing a different important issue with respect to interfaces, the necessary components for a complete interface specification can be derived:

- *message description*: Syntactical descriptions of all messages exchanged over the interface.
- *semantic description*: The meaning of messages on the interface must be specified, i.e., their functional behavior within the comprehensive business process. This description must include the meaning of all individual message fields.
- *dynamic description*: The dynamic behavior of the interface must be fully specified. This specification includes all possible message sequences and the behavior of the applications in case of errors.
- *infrastructure description*: A description of the necessary infrastructure must be provided.
- *quantity description*: The non-functional performance requirements for the interface must be described.

It is important to notice that an interface specification is a signed bilateral contract, which may be changed by mutual agreement of all participating parties only [8]. This contract is represented by the set of artifacts as described above, so none of the artifacts given there may be missed.

IV. Communication technology

In order to enable communication between applications an appropriate communication technology is necessary. Communication technologies like, e.g., HTTP, FTP, JMS or RMI, have specific transmission properties and often require specific middleware components to fulfill common communication tasks, like, e.g., routing or address resolution [23]. Specific requirements, derived from the typical 24/7 hours mode of operation, guide the selection of appropriate communication technologies.

A. Communication requirements

Large application landscapes often require a set of different communication technologies to satisfy all requirements of IT operations and to fulfill the underlying service level agreements. In order to keep the IT operation costs low, only a few and proven mainstream technologies should be considered for the complete application landscape [8]. Suitable communication technologies typically satisfy an appropriate subset of the following requirements depending on communication properties defined by the enterprise itself.

1) Coupling: The degree of coupling between applications is one of the most important requirements concerning communication technologies. Since modern application landscapes typically involve applications from a wide set of different software vendors, a loose application coupling is usually required [8], since applications

- use different software technologies, e.g., programming languages,
- assume different runtime behaviors, e.g., the number of threads or processes used by an application is completely transparent and independent from all other applications and
- have different life cycles, i.e., they should be replaceable during runtime if they implement the same interface contract without imposing technical interferences to other applications [9].

Loose coupling is directly supported by the communication technology if an asynchronous communication style is used [9], [22]. Furthermore, this requirement usually has a direct impact on maintenance and support costs of an application landscape.

2) Stability: In modern application landscapes things will happen during runtime that cannot be foreseen at development time [5]. Introducing stability into the application landscape reduces the effects of such faults and consequently lowers support costs. Stability is gained by preventing the propagation of failures over application boundaries. A common pattern to achieve this goal is a decoupling of applications during runtime [5], [6], i.e., all applications must technically run completely independ from each other.

3) Flexibility: Applications and middleware can be deployed to different physical or logical machines without having more effort than configuration costs [32], i.e., IP adresses, port numbers etc. are specified in appropriate configuration files. Therefore, the required middleware components of the communication technology have to be flexible enough in order to ensure this arbitrary application distribution. Note that this flexibility usually has a direct impact on maintenance and support costs for application landscapes.

4) Reliability: To minimize support costs for application landscapes the communication must be reliable, i.e., the communication technology has to be able to guarantee the delivery of information. Note that this guarantee usually does not comprise a guaranteed time slot within the information will be delivered to the receiver. Additionally, the requested guarantee of delivery holds for common problems like network breakdown, system breakdown, disc crash etc. only but usually not for disaster scenarios destroying the complete infrastructure. Failing to deliver information to the receiver leads to communication faults that must be cleared by the application management. Consequently, this requirement has a direct impact on the amount of fault clearings.

5) Performance: The communication technology has to be fast enough to handle the predicted information throughput thus guaranteeing the performance specified in the underlying service level agreements. It must also be able to deal with information burst situations during communication. Furthermore, while communicating with another application, the sender should not be delayed by the communication technology, i.e.,

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

sending an information is completely decoupled from its transmission. Note that more specific performance requirements depend on individual application requirements and therefore cannot be specified here.

6) Monitoring: To supervise the communication between applications, access points should exist in the communication lines [9]. Using these access points, the application management can

- monitor occurred communication errors,
- monitor the load of the communication channels,
- directly read all information exchanged by the applications for debugging purposes and
- insert test information into the communication channels.

This supports a reduction of the fault clearing time thus reducing the support costs for the application landscape. Additionally, new applications must be integrable into existing IT monitoring tools without further adjustment. In case an application has a technical problem, e.g., a database crash, it should be possible to communicate the problem to the application management using special communication channels [9].

7) Costs: The application communication should be standardized in order to minimize development and operational costs for application integration [8]. It should also be possible to encapsulate most parts of the communication resulting in a minimal code invasion of the application software [9].

In order to minimize the efforts during system integration tests, access to the communication technology should be easily replaceable by mocks [22]. This will minimize software license costs because middleware components of the communication technology must not be used on the various environments used during development and integration testing stages [32].

8) Technical security: If something goes wrong, security has to provide a degree of confidence that the application landscape can remain in a state of normal operation [33]. Communication technology, being one part that ensures the technical security of the application landscape, has to protect the authenticity and integration of all messages exchanged [33]. Additionally, an automated communication recovery is typically required after an application or communication failure in order to minimize the failure recovery time.

B. Selecting a communication technology

Adequate communication technologies have to be selected depending on the communication requirements to be satisfied. These requirements are usually determined by given service level agreements. For example, requirements like *Coupling* and *Stability* typically exclude the use of synchronous communication technologies like RMI [14] or HTTP [11]. Other requirements, e.g., *Performance*, either demand very specialized communication technologies like the Blink Protocol [16] or prevent specific technologies such as shared databases [9].

In contrast, using communication technologies explicitly supporting asynchronous messaging as integration style like, e.g., JMS [13] or AMQP [15] commonly satisfy all requirements. Since messaging is considered to be the best application integration approach [9], it is usually best suited for most integration purposes.

V. Transmission protocol

While the selected communication technology enables exchange of information between applications, further communication services like, e.g.,

- robust communication,
- support of synchronous and asynchronous communication in parallel,
- business process logging,
- support of enterprise integration patterns [9],
- automated reconnect behavior and
- ongoing communication supervision

are typically required in order to support a 24/7 hours maintenance of an application landscape. These communication services are typically not fully provided by the communication technology; instead they must be implemented at the application layer of the OSI model [23], [24]. Consequently, an implementation of these communication services requires an additional transmission protocol layered on top of the communication technology, see Figure 1 for details.

This section introduces such a transmission protocol based on messaging to provide the above mentioned communication services. Using messaging to transmit information between applications requires sending and receiving of structured messages using specific communication channels identified by *destination* names. This messages are exchanged using communication channels addressed by unique destination names. A message consists of a header containing some necessary meta-information to safely transmit the message between applications and a body containing the functional payload data [9].

A. Message header

The meta-information within the header is organized in a set of fields and represents an important part of the transmission protocol. Consequently, a message sender has to use these header fields as described in the following.

1) Message ID: This field contains a unique identification of the message. It is typically provided by the messaging infrastructure itself, see, e.g., [13].

2) Message type: Each message send by an application must belong to one of the four message types *Notification*, *Request*, *Reply* or *Error*, as described in Section III.

3) Message reply to: Generally, using messaging leads to an asynchronous communication behavior between applications. In order to implement synchronous communication, i.e., the sender stops and resumes processing after receiving the reply message, the *Request-Reply* and *Return Address* patterns [9] can be used. A request message, which is always of message type *Request*, has to use this field, where the name of the reply destination must be inserted. The receiver of the request will send its answer to the given reply destination specified in this field.

4) Message correlation ID: In case of synchronous communication, the requester can identify the answer, which always has the message type *Reply*, to his *Request* using the correlation ID. According to the *Correlation Identifier* pattern [9], the sender of the reply must include the original message ID of the request as correlation ID in the reply. The field must

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

not be used in all other cases. Note that the sender of the request must store the message ID of the request in order to compare it to the correlation ID of the *Reply*.

5) Message expiration: This field is used for the *Message Expiration* pattern [9]. If the sender specifies a *time to live* for a message, the messaging infrastructure will deliver the message to the receiver only if it is within the *time to live*. If the *time to live* is expired and the message couldn't be delivered to the receiver, the message will be destroyed in the messaging infrastructure. In case of *Requests*, which usually have a timeout [5], the *time to live* should always be set equal to the timeout, which will decrease the load for the message receiver.

6) Message name: The field message name contains the name of the message. Message names should be meaningful and unique.

7) Message sender: The field message sender contains the name of the sender of the message. This name must be unique within the application landscape and should be as precise as possible. For example, a naming schema could follow an inverse URL like naming style where single parts of the name are separated by capital letters.

8) Error text: The field error text is used for *Error* messages only and contains the exception text, see Section VI for further details.

9) Message version: In case of distributed applications problems always occur while operating different interface versions in parallel. In order to deal with future changes of an interface, the *Format indicator* pattern [9] is used, where each message carries its corresponding interface version number in this field. All applications should have an external editable list of version numbers, called *compatibility list*, for each implemented interface that defines all valid interface versions. The receiver of a message compares the version of a received message with its *compatibility list* and calculates the compatibility state. The following compatibility states exist:

- *true*, if the version of the message is contained in the *compatibility list*
- *false*, in all other cases.

If the compatibility state is true, the message will be processed, otherwise the message will be redirected to an error message channel, as described in detail in Section VI.

Note that a standard downward compatibility behavior as given by some communication technologies like, e.g., HTTP [11] cannot be used at the protocol layer because functional semantics may change significantly between different interface versions thus leading to an incorrect functional behavior. Using an explicit *compatibility list* avoids these semantical problems.

10) Sequence number: Using the patterns *Resequencer* and *Message Sequence* [9], the receiver can get a stream of out-of-sequence messages back into the original sequence. Therefore, the sequence number field can be used, containing a unique sequence number. Each sender generates his own stream of unique sequence numbers; they need not to be unique across different applications. If the range of numbers is exceeded, the sequence will start with the lowest number again.

Note that messaging infrastructures typically guarantee to keep the sequence of all messages sent via a **single** destination only. There is **no** guarantee, however, if multiple destinations between sender and receiver are used in parallel to transmit messages.

11) Trace ID: The field trace ID can be used in distributed systems to trace an activity over individual application boundaries. A common usage of the trace ID is, e.g., for logging purposes within the business processes.

B. Message body

The body of a message contains the functional payload, i.e., its content is the reason why this message has been sent. The content must be a valid structure fulfilling the following issues:

- In order to avoid technical dependencies based on, e.g., compiler issues, only text messages should be used.
- The content of the body should follow a formal document structure like, e.g., the Extensible Markup Language (XML) [34].
- This structure must allow a complete formal syntactical validation of the message, e.g., the XML Schema Definition Language (XSD) [35] in case of XML.
- Only the Universal Time Coordinated (UTC) format may be used for content concerning time. The time format used must follow the international norm ISO 8601 including the time zone designator [36].
- Binary data must be converted using, e.g., a Base64 encoding [37].

For the required encoding, i.e., the transformation between a byte sequence and a unicode string, to transmit a text document, usage of the Universal Character Set Transformation Format (UTF) UTF-8 or UTF-16 is strongly recommended.

VI. Communication error handling

During interface operations, failures may occur within the communication of applications. These communication problems can be categorized into *connection problems*, resulting from infrastructure issues in the transport layer, or *communication problems*, resulting from message computation failures in the functional and protocol layers, respectively. Each failure category requires its own error handling procedures, which are described in the following subsections.

A. Connection problems

Connection problems usually result from issues within the communication technology, preventing an application from sending or receiving information and thus interrupting its communication. Consequently, connection errors always represent severe problems, which require immediate activities by the application management. Since the application management can solve connection errors at runtime, applications should support their activities through an automated reconnect behavior to the messaging infrastructure as follows.

Connection failures can occur at any point in time and are signaled by the communication technology to the application. Once this has been done, a configurable timer, called *connection timeout*, must be started by the application. After this *connection timeout* expires, the application must signal the connection error to the application management and disconnect itself from the infrastructure of the communication technology,

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

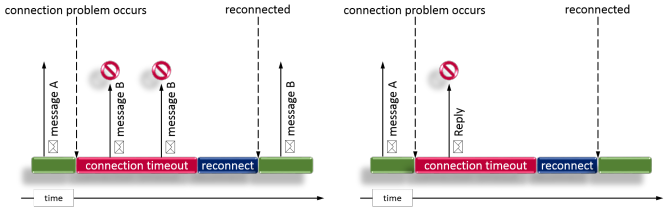

Figure 7. The left figure shows the behavior for *Notification* and *Error* messages, here commonly denoted as messages. Once the connection is broken, the connection timeout starts. If the connection remains broken the application disconnects from the middleware infrastructure, once the connection timeout has been reached. After reestablishing the connection, the message is sent. For *Reply* messages only, as shown in the right figure, nothing happens after the first attempt to send the *Reply* message.

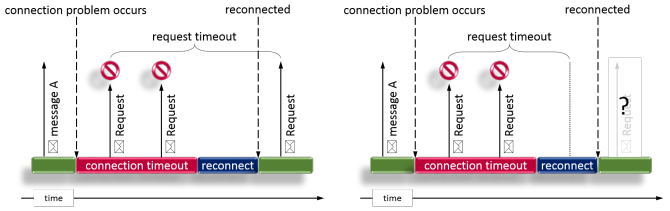

Figure 8. Timeout behavior in case of a *Request*. If the connection is reestablished within the request timeout time, see left figure, the message is sent after reestablishing the connection. If the connection cannot be reestablished within the request timeout, the application decides if the *Request* message will be dropped or resend, as shown in the right figure.

followed by a reconnect attempt to it. The advantage of this behavior, which shall be repeated forever until the connection has been successfully reestablished, is, that changes within the network infrastructure may be executed during application runtime.

The *connection timeout* must be stopped immediately if the connection error has been solved within the time period spanned by the *connection timeout*. No reconnect attempt will occur in this case.

If an application fails to send a message due to connection errors, the following behavior must be adopted by the application, depending on the message type sent:

- *Notification* and *Error*: Periodically retry to send the message until it has been sent successfully.
- *Request*: Periodically retry to send the message until it has been sent successfully or a *request timeout*, see below, has been reached. In the latter case, the application decides all further communication behavior, e.g., if the *Request* message will be dropped or resend.
- *Reply*: Drop the message.

The rationale behind the deviating behavior for *Reply* messages is, that reply destinations may be temporary destinations which may have ceased to exist and thus will never reappear, see, e.g., [13]. Figure 7 visualizes the reconnect behavior in case of *Notification* or *Error* messages on the left side. After the *connection timeout* has been exceeded, the application automatically disconnects from the communication system and tries to reconnect itself to it. Once the connection has been reestablished, the message is send. In contrast, a *Reply* message is dropped once sending the message fails, as shown on the right side of Figure 7.

For *Request* messages, the behavior depends on the relationship between the request timeout, and the time necessary to reestablish the connection, as shown in Figure 8. The *Request* will only be send once the connection has been reestablished if the *request timeout* has not been exceeded.

B. Communication problems

Functional errors resulting from message computation should always be handled by the application itself. All other kinds of exceptions are based on either configuration or programming errors and usually cannot be corrected by the application during runtime. Therefore, this kind of errors must be signaled to the application management.

C. Communication supervision

In order to inform the application management promptly, the protocol layer uses a mechanism based on the *Control Bus* pattern [9]. If necessary, *Error* messages are sent to separate destinations, supervised by the application management.

1) Error message construction: Like other message types, *Error* messages consist of a header and a body section. The header contains a set of header fields as described in Table I, while the body content depends on the cause of the *Error* message. If the latter has been triggered by another message, the body of the *Error* message should contain a copy of the content of the original triggering message. Otherwise, the body of the *Error* message will be empty.

2) Error destinations: All messages that are either

- syntactically incorrect, i.e., those failing the syntactical validation, or
- out-of-sequence, i.e., those received unexpected from the defined message flow of the interface, or
- have an insufficiently defined set of message header fields, or
- have an incompatible version number

must result in an *Error* message sent to a destination called, e.g., *invalid message channel*.

In case of synchronous communication being used between applications, crack propagation due to blocked threads or slow responses must be prevented using a timeout pattern [5]. Whenever a sender sends a *Request* to another application, the sender will wait for a configurable amount of time, denoted *request timeout*, to receive the *Reply*. If the *request timeout* exceeds, the sender converts the *Request* message into an *Error*

TABLE I. Header fields and their valid content for an *Error* message

field name	field content
message ID	the message ID
message type	*Error*
message name	*invalid message error* or *timeout message error* or *fatal error*, where the message name used depends on the cause of the error.
message sender	name of the application sending this message
error text	a short and meaningful text describing the error
version	the version number of the interface

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

message and sends it to a destination called, e.g., *timeout message channel*.

For severe errors that cannot be handled by the application itself, e.g., connection problems or broken database connections, an *Error* message should be sent to a destination called, e.g., *fatal error channel*. Obviously, this can be done in case of *connection problems* only, if the connection to the application management still exists. Note that these kind of failures typically threaten the running state of applications.

D. Communication monitoring

All errors reported to one of the error destinations, i.e., *invalid message channel*, *timeout message channel* or *fatal error channel*, have to be analyzed by the application management to supervise the application landscape. In other words, the error destinations act as part of the *Control Bus* pattern [9].

The required activities of the application management in conjunction with the error destinations is described in the following subsections. For a better overview, a short tabular summary of the most important facts is given at the end of each subsection, consisting of

- the priority necessary to react to an *Error* message,
- the strategy necessary to deal with an *Error* message,
- the environment, i.e., development, test or production stages [32], within most *Error* messages will be produced and
- the information when the *Error* message should be deleted from the error destination.

1) Invalid message channel: Messages sent to this destination mainly occur due to syntactical or dynamic problems, so each message within this destination is important. None of the errors will directly threaten the running state of an application but business processes may degrade depending on their design.

All errors reported to the *invalid message channel* are based on programming or process design errors and cannot be solved during runtime. Instead, they must be delegated to the second level support. Exceptions are *Error* messages due to incompatible interface version numbers. In this case, the application management should first check whether the interface configurations of the involved applications are correct, second, try, if convenient, to up- or downgrade one of the participating applications to a new version that supports the required interface version and third, inform the second level service if none of the before mentioned activities leads to the desired behavior. Once an *Error* message has been analyzed, it should be directly deleted from the *invalid message channel*. It is expected that the main load for the *invalid message channel* occurs within the test stage. Normally, no *Error* messages should be produced in the production stage, otherwise this indicates misbehavior due to unsufficient test cases.

TABLE II. Monitoring of the invalid message channel

summary	
production priority	medium, may effect execution of business processes
evaluation strategy	based on single error messages
main error occurances	test stage
error message deletion	immediately after analysis by application management

2) Timeout message channel: An individual *Error* message within the *timeout message channel* has no further meaning and can be ignored, but the history of the message amount within the *timeout message channel* is very important. The rationale behind this property lies in the unique source of *Error* messages within the *timeout message channel*: in case of synchronous communication the *Request* has not been replied within the *request timeout* by the targeted application. For a single *Error* message this behavior is not problematic, but large amounts of errors indicate either severe load problems within the target application of the *Request* or problems with the communication infrastructure used for sending the *Request* and *Reply*. The running state of the requesting applications is not threatened, but the problem may influence the business processes.

All messages within the *timeout message channel* should be analyzed in real time per targeted application. It is recommended that this analysis should result in a graphical illustration projecting the number of *Error* messages in the *timeout message channel* over time to indicate behavior trends of the targeted applications. If certain thresholds are exceeded, the application management should have a closer look to the targeted application and the corresponding communication infrastructure.

Messages exceeding a configurable amount of time, e.g., one hour, within a *timeout message channel* should be automatically deleted. Failing this will result in resource allocation problems within the corresponding communication infrastructure. It is expected that the main load for the *timeout message channel* occurs in the production stage since these errors typically occur during unexpected load scenarios.

TABLE III. Monitoring of the timeout message channel

summary	
production priority	low, serves as a general health status indicator of applications
evaluation strategy	based on multiple error messages
main error occurances	production stage
error message deletion	after a given time frame of, e.g., one hour

3) Fatal error channel: Errors within this destination can be categorized as either severe programming errors, which must be handled by the second level support or resource errors, e.g., missing table space or wrong port numbers, that can be handled by the application management directly. Once an *Error* message has been analyzed, it should be directly deleted from the *fatal error channel*. It is expected that the load for the *fatal error channel* occurs equally in the test and production stages.

TABLE IV. Monitoring of the fatal error channel

summary	
production priority	high, immediate action necessary
evaluation strategy	based on single error messages
main error occurances	test and production stages
error message deletion	immediately after analysis by application management

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

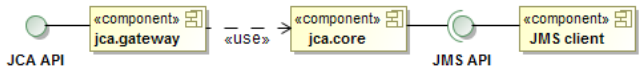

Figure 9. The JCA framework can be used in any Java application. A provider specific JMS client has to be included in the application classpath.

VII. Example: Implementation of an Integration Framework

Based on the recommendation for the communication technology given in Section IV, a Java based integration framework has been developed at the Hamburger Hafen und Logistik AG (HHLA) supporting a standardized integration. The framework, which is based on JMS [13] satisfies the transmission protocol and communication error handling requirements described in Sections V and VI, respectively. This section gives an overview about the resulting implementation, which currently guarantees a robust communication within the HHLA production environment since three years.

A. Communication adapter framework

The Java Communication Adapter (JCA) framework is organized by two modules, core and gateway, to support the required communication services, see Figure 9 for an overview. It is packaged as Java Archive (JAR) to be easily deployed and reused while integrating applications into the HHLA application landscape. In case of products, which cannot be modified directly, or applications using different programming languages, appropriate adapter have been introduced to technically integrate applications by reusing this framework.

1) Core module: This module encapsulates the transport layer, see Figure 1, i.e., it is responsible for establishing a robust connection to the underlying communication technology. Therefore, the standard transmission protocol JMS [13] has been used, implemented by a JMS provider. The JCA supports all JMS providers that are compatible to the JMS Specification, version 1.1 [13]. It was developed and tested with two different JMS providers, ActiveMQ [38] and SwiftMQ [39], the former one representing an open source implementation and the latter being a commercial product.

As main functionality, the core module of the JCA establishes the connection to the JMS server and extends the standard JMS communication behavior to handle connection and communication problems as described in Section VI.

When *connection* or *communication problems* occur, they are directly reported to the corresponding error message channel, which are supervised by the application management. Broken connections are automatically detached by the core module once the configurable *connection timeout* has been reached and reinitialized immediately, until the connection has been successfully reestablished. Additionally, a thread and timeout handling has been realized to support synchronous and asynchronous communication in parallel via standard JMS. Furthermore, the core module constructs the JMS message itself using the `Session.createTextMessage()` method of the JMS standard application programming interface and sets the corresponding JMS message header fields *JMSMessageID*, *JMSType*, *JMSReplyTo*, *JMSCorrelationID*, *MessageName*, *MessageSender*, *MessageError*, *MessageVersion* and *TraceID*, according to the message type being send. Note that all header fields starting with *JMS* represent standard JMS header fields that always exist, while all other header fields have to be added using the appropriate `message.set<Type>Property()` method, where `<Type>` denotes one of, e.g., `String`, `Long`, etc.

When receiving JMS messages, the header fields are read and provided to the gateway module to handle extra communication services like business process logging. All transmitted messages are logged with the message header field *TraceID*, which can be used for error analysation in correlation with the related business process.

2) Gateway module: As shown in Figure 10, the *CommunicationService* is the central interface containing all relevant operations to integrate applications. It offers the following principal communication services:

- Connection handling: *start*, *stop* and *isStarted*
- Create messages: *getMessageFactory*
- Report errors: *reportError*
- Send messages: *sendNotification* and *sendRequest*
- Receive messages: *registerReceiver* and *unregisterReceiver*

The gateway module, encapsulating the protocol layer of Figure 1, extends the functionality of the core module by mapping Java objects onto XML representations and vice versa while sending and receiving messages, respectively. This marshalling and unmarshalling is realized by the Java Architecture for XML Binding (JAXB) framework, which is part of every Java Runtime Environment (JRE) [40]. Additionally, the required message header fields and their content are determined and given to the core module.

The robust communication of the core module is extended within the gateway module by a validation of message header fields and message content. Furthermore, a configurable support of different interface versions has been realized here. Validation errors, i.e., syntactical errors, are directly reported to the supervised *invalid message channel* via the core module. All converted Java objects are logged with the message header field *TraceID*, which, again, can be used for error analyzation in correlation with a business process.

VIII. Interface Design Styles

The main problem to be solved in interface design concerns the intended functional semantic on the interface, i.e., the construction of the functional layer shown in Figure 1. It directly influences the kind of service offered by the *server* and therefore the necessary number and style of all messages.

Looking at existing interfaces, they can be categorized in our experience by their semantic design styles: CRUD based interfaces, use case based interfaces and business process based interfaces, each of them described in detail in the following sections.

A. CRUD based design

The Create, Read, Update, Delete (CRUD) based design directly uses the business objects described within the requirements and ignores any given business context. This results in interfaces consisting of a minimal set of messages, representing

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

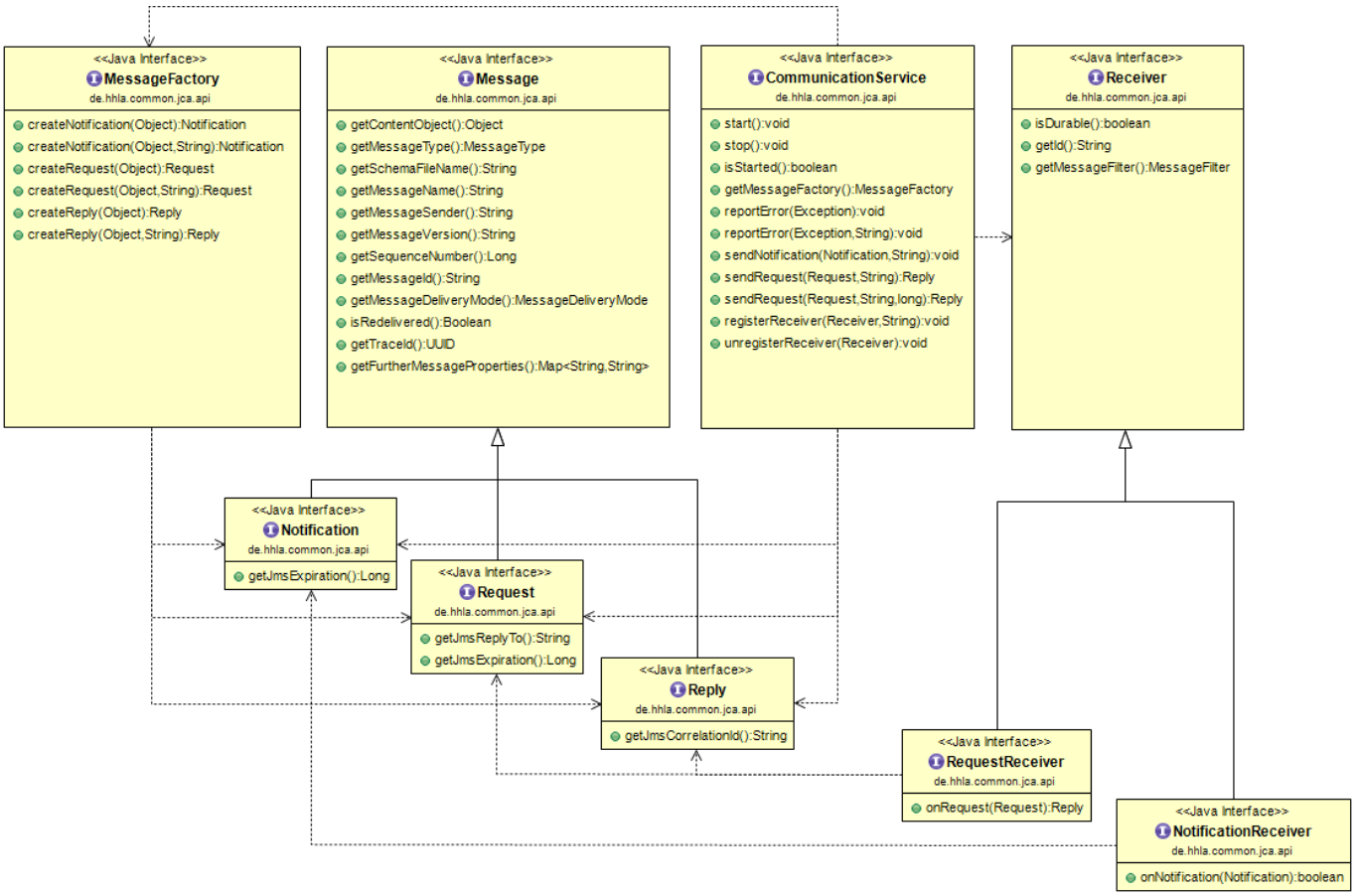

Figure 10. UML class diagramm depicting the main interface structure and methods of the JCA application programming interface (API).

a set of CRUD messages for every business object the *server* is functionally responsible for. Besides the advantages of requiring very little design efforts and being very stable, this interface design style has some important disadvantages.

First, the interface bears absolutely no business context, leading to severe difficulties in understanding the underlying business processes [2]. Second, the read operation demands synchronous communication, which represents an explicit control flow leading to a tight coupling of applications [5], [26] and third, missing business context either leads to a distribution of business functions over the *clients* or to business objects incorporating the results of applied business functions.

B. Use case based design

An interface design based on use cases rests upon requirements formulated from the perspective of the primary actors for individual systems only [41], i.e., the underlying business process is not directly present. Due to the characteristics of use cases, describing non-interrupted interactions with the system [42] that represent the view of the primary actor [41], these requirements are limited to the context of single activities, which are typically independent with respect to each other. A representation of the underlying business process triggering the desired activities is missing and therefore difficult to reconstruct.

These preconditions usually lead to rather fine granular and use case oriented interfaces comprising of a large number of messages, carrying specific use case based information only. Some important consequences arise from this design style. First, all business functions that are identical from the business process point of view, are hard to identify based on a use case analysis only. The absence of a business context leads to an interface design supporting individual use cases, which bear no evident business process semantics. Consequently, these interfaces usually offer a broad range of identical functionalities named differently. Second, the missing business context significantly increases the difficulty to understand the functional behavior of the interface over time [2], leading to serious problems in its usage. As a consequence, further unnecessary messages are often introduced in order to provide some use case specific information. Third, the lower level of abstraction of a use case - compared to the business process - leads to a rather fine granular interface structure. Performance issues may arise with this interface style due to the enforced frequent interface access [9]. And fourth, synchronous communication often arises in order to collect all necessary information to execute the use case, so a control flow arises [26] leading, again, to a tight coupling of applications [5], [31].

Note that using a use case based design must not lead compulsorily to a bad interface design. But given the size of current applications with their numerous use cases and the typical usage of distributed programming teams within industrial projects, the necessary refactoring to introduce an appropriate abstraction on the interface is usually omitted in

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

our experience.

C. Business process based design

This design uses business process descriptions and further requirements formulated with respect to those descriptions, to align between individual business process activities and applications. Using the Business Process Model and Notation (BPMN) [43] and representing applications via pools, interfaces can be directly derived from the exchanged information between individual business activities within the pools.

The resulting interfaces focus on business semantics and directly support objects, events and functions of the business processes, thus leading to a domain model bound to interfaces [7] with the following consequences. Business processes support a high level of abstraction, thus leading to rather coarse granular interfaces with respect to the number of messages. The communication is driven by business events, so asynchronous communication is naturally supported, leading to data flows [26]. Finally, the functionality provided by the *server* within the business processes becomes rather clear, i.e., the business context is represented on the interface.

D. Design example

To explain and clarify the differences between these design styles, the simplified process of loading a truck at a container terminal will serve as an example throughout this section. This process consists of the following steps, executed in the given order:

- *order clearance*: the customer gives an order to the container terminal to load a container on a truck.
- *load clearance*: in order to deliver the container, several clearances must be given, e.g., by customs and the container owner.
- *transport planning*: the container terminal plans the necessary equipment to execute the order.
- *load container*: the container is loaded on the truck using the planned equipment.

Two applications shall be constructed in order to implement the process: the *Administration*, dealing with the administrative parts of the process, and *Operating*, handling the physical transport of the container. An interface between both applications will be designed according to the design style considered, thus showing the differences between the design approaches.

1) CRUD based design: All relevant business objects of the truck loading process are represented in a data model that provides methods to create, read, update and delete the objects. These methods represent the interface of the owning application, i.e., the *server*, and are called by the *clients*, in order to execute the business process. For example, after creating an order using `createOrder()`, the *Administration* calls `createInstruction()` to start the loading of the container on a truck. Subsequently, *Operating* calls `readCustomsClearance()` and `readReleaseOrder()` to check if the container is released to be loaded on a truck. The corresponding return objects must be interpreted within *Operating* to make this decision. If the container has been loaded, *Operating* finally calls `deleteOrder()`, `deleteCustomsClearing()` and `deleteReleaseOrder()` to clear the *Administration*.

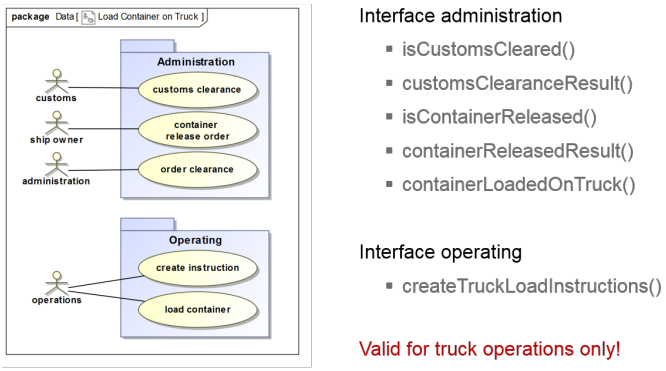

Figure 11. Constructed interface (right) resulting from applying the use case based design approach.

It becomes clear that both applications, i.e., *Administration* and *Operating* must implement some part of the underlying business logic to deal with these type of interfaces. Since the interface style bears no business semantics, the underlying business process cannot be reconstructed easily. Note that the size of the interface directly depends on the number of business objects the server is responsible for.

2) Use case based design: Based on the requirements of the truck loading process, corresponding use cases like order clearance or create instruction can be derived, as shown in Figure 11. Each of these use cases handles a specific functional aspect with respect to its primary actor. The underlying business process is executed through a set of use cases interacting with each other.

For example, if an order has been given, *Administration* calls `createTruckLoadInstructions()` to initiate the container transport. Prior to loading, *Operating* checks the container release status, using `isCustomsCleared()` and `isContainerReleased()`. If the container has been released, it is loaded on truck and *Operating* informs *Administration* via `containerLoadedOnTruck()` that the order has been executed. *Administration* may then clean up its internal data structures.

As depicted on the right side of Figure 11, the resulting interface contains a lot of methods for specific actions, i.e., the level of abstraction is rather low. Consequently the interface is valid for truck operations only and would require a couple of additional methods to incorporate, e.g., vessel and train operations.

Furthermore a use case based interface introduces synchronous communication, as indicated by, e.g., the method pairs `isCustomsCleared()` and `customsClearanceResult()`, leading to a blocking of *Operating* while accessing the information.

3) Business process based design: In this case, the business process itself serves as basis for interface design. Using BPMN, the process of truck loading can be mapped onto the applications as shown on the left side of Figure 12. Due to the given high level of abstraction within the business process, it is valid for all types of carriers, i.e., no further messages are necessary to include vessel and train operations.

Once an order has been given, *Administration* informs *Operating* via `orderPlaced()` that a new order has been

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

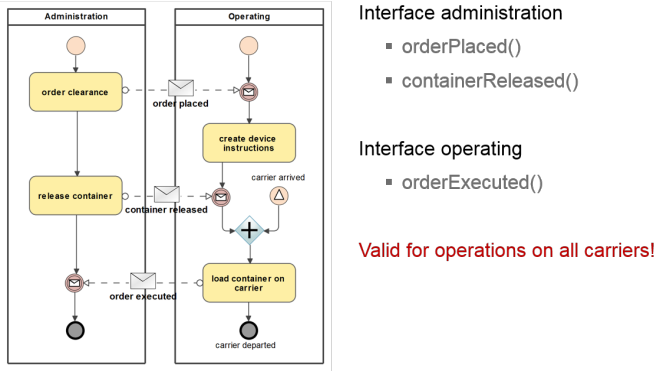

Figure 12. Constructed interface (right) resulting from applying the business process based design approach.

accepted. Within *Operating*, all necessary instructions for container loading will be created. Once the truck has arrived and *Administration* has published via `containerReleased()` that the container is released to be loaded on a truck, the physical moves are executed. Afterwards, `orderExecuted()` informs *Administration*, to clean up its internal data structures.

The dynamic behavior of the interface can be derived directly from the BPMN description, as shown on the left side of Figure 12. The resulting interface is quite small, meaningful and abstract, so it can be easily extended to other carriers. Additionally, the communication between both applications is asynchronous. Note that both applications, *Administration* and *Operating*, do not technically depend on each other, instead they simply publish their information without knowing the receiver, resulting in a data flow [26].

E. Comparison

To give a recommendation for a specific interface design style all design approaches described above have been compared to each other using typical interface design goals like robustness, performance and understandability [2].

1) Robustness: Interfaces are crucial with respect to the stability of the overall application landscape. Poorly designed interfaces may propagate internal application errors during runtime, thus causing damage within other applications [2], [5]. Robustness is achieved by avoiding functional distribution, distributed transactions [6] and semantical ambiguity.

In case of a CRUD interface, the information provided by the interface must be functionally interpreted by the *client* since the *server* informs about changes on business objects only without any functional context. This leads to multiple and distributed implementations of business functions according to the usage of the interface. In contrast, the use case and business process based design styles can both concentrate the business functions within the *server*, so no functional distribution will arise.

In general, distributed technical transactions can be avoided in all three design approaches. But modeling a control flow instead of a data flow bears a higher risk of introducing distributed transactions within the application landscape, due to the usage of synchronous communication. None of the design approaches specifically supports the construction of an efficient message field structure nor prohibits the introduction of content based constraints.

2) Performance: Obviously, interfaces must satisfy the required performance, i.e., they must be able to deal with the given quantity description. Otherwise, the business process will not work correctly since required business functions may not be executed in time. Performance is supported by designing minimal interfaces with respect to the number of messages and avoiding synchronous communication [4], [9].

The more abstract the interface is, the less messages are needed due to the restriction of transmitting core concepts only. With a CRUD based design, the most abstract design is chosen while a use case based design includes relatively less functional abstraction. Asynchronous communication is usually directly supported in the business process based design, while the other two approaches support a rather synchronous communication style. This holds especially for the CRUD based design, where the `read()` operation always enforces synchronous communication.

3) Understandability: Well designed interfaces must have a strong and documented relation to the underlying business context [2], thus ensuring a good usability of the interface. This will enhance the cost efficiency of the interface over time since a much better acceptance of the interface within the development teams will arise because the interface will be easier to learn, remember and use correctly [2].

Understandability is given by a strong functional binding between the domain model and the implementation [7], the usage of business objects and business events as message content [4], [9] and a meaningful message naming schema.

Naturally, a business process based design leads to a direct mapping between interface and business process description thus enriching the interface with a comprehensive business context. On the contrary, a CRUD based design bears no business context at all due to its high level of abstraction.

Although all three approaches directly support the exchange of business objects, differences occur considering the publication of business events. A CRUD based design supports none of them per se, i.e., this approach forces a mapping of business events onto business objects. This will lead to serious problems in understanding the dynamic behavior of the application landscape. Using a business process based design instead, the published business events can be directly derived from the underlying business process. In contrast, a use case based design does not primarily focus on business events but on individual user operations thus obscuring the business context. While the business process and the use case based designs both support message naming schemas providing a rich functional context, a CRUD based design uses only the given names for create, read, update and delete messages.

4) Recommendation: Considering the above mentioned design goals and the important advantage of supporting a direct link between domain model and interface design, the business process based design is the recommended design style for interfaces, leading to the best design compromise.

IX. Example: Integrating an Application Using the Business Process Based Design

The business process based design approach has been used at the HHLA to integrate a new application into an

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

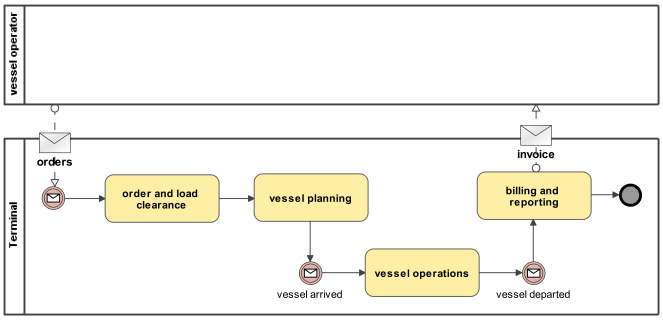

Figure 13. Business process for vessel handling.

TABLE V. Responsibility of applications for business objects.

administrative system	
business object	**description**
carrier order	order to handle a vessel
work instruction	planned instruction to move a container
work queue	planned list of work instructions for a specific device type
vessel geometry	vessel type geometry
...	...

terminal control system	
business object	**description**
container	physical data of a container
carrier	physical data of a carrier, e.g., a vessel
devices	equipment to move a container, e.g., straddle carrier
device instruction	instruction for a specific device to move a container
...	...

existing complex application landscape. This new application will be responsible for the required order management and planning tasks involved with vessel handling processes at a deep sea container terminal. A critical aspect during this integration was the required design of a new interface to an existing terminal control system, which controls all devices and logistical aspects on a container terminal.

Starting with a business process definition at a level showing a rather general process context, i.e. the summary level [41] as depicted in Figure 13, the business process has been further detailed. Using a domain model for HHLA container terminals, this refining leads to all relevant business objects required within this process.

Thereafter, these business objects have been assigned to the administrative system and terminal control system as given in Table V. Being assigned to an application implies that this application acts as a *server* for that specific business object, i.e., only this application is responsible for the business objects entire lifecycle. Consequently, each of the applications has to provide an interface to enable an access to its business objects and events.

The assignment of business objects to applications is usually guarded by the existing functional scope of the applications. In case of ambiguities, i.e., cases where business objects are required by multiple applications, the domain model of the HHLA has been used to determine the *server*. Note that other

TABLE VI. Mapping of business objects onto existing application objects. To decouple them, an integration model has been introduced. A — denotes a missing object in that application.

business object	**integration model**	**administrative system**	**terminal control system**
carrier order	carrier order	carrier visit	—
work instruction	work instruction	work instruction	transport order
work queue	transport directive	work queue	transport order
vessel geometry	carrier geometry	vessel master data	vessel geometry
container	container	unit	container
carrier	carrier	vessel	vessel
device	device	—	device
device instruction	device instruction	—	order

clients may still use caches for these business objects, which have to be updated properly using an appropriate interface of the *server*.

Once this has been done, an IT alignment took place mapping each business object onto existing application objects. Since applications use different internal objects to represent these business objects, an integration model will be derived, decoupling applications and the domain model from each other. Table VI shows the resulting mapping for the vessel handling process.

Given the integration model, the business process based design approach described in the previous section can be applied. Using BPMN, all applications have been represented as pools with the functional activities being assigned accordingly, see Figure 14 for details. Note that all functional activities of the vessel handling process as well as their sequence have been adapted to the existing functionalities and process sequences within the corresponding applications. Furthermore, one of the given pools directly represents the vessel itself instead of an application. Incorporating the vessel into the BPMN model is helpful in this case since it represents an important physical event source for the whole process.

Looking at Figure 14, nearly all messages functionally required to exchange the necessary information between the administrative system and the terminal control system can be grasped directly from the BPMN description. Table VII summarizes the resulting interfaces of both applications. Note that some messages required for resilience are still missing, e.g., messages that allow a complete download of business objects from the *server* to reset caches located at *clients*.

Finally, individual XML structures using appropriate elements and/or attributes have to be designed for each message. Since most messages contain business objects only, the required information within a message is usually determined by the attributes of the business objects. In case of business events, the information to be carried is usually the event itself enriched with the business object identifier to whom the event applied.

Using the business process based design approach offers a structured approach for designing a functionally complete interface while supporting the nonfunctional aspects given in Section III-E.

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

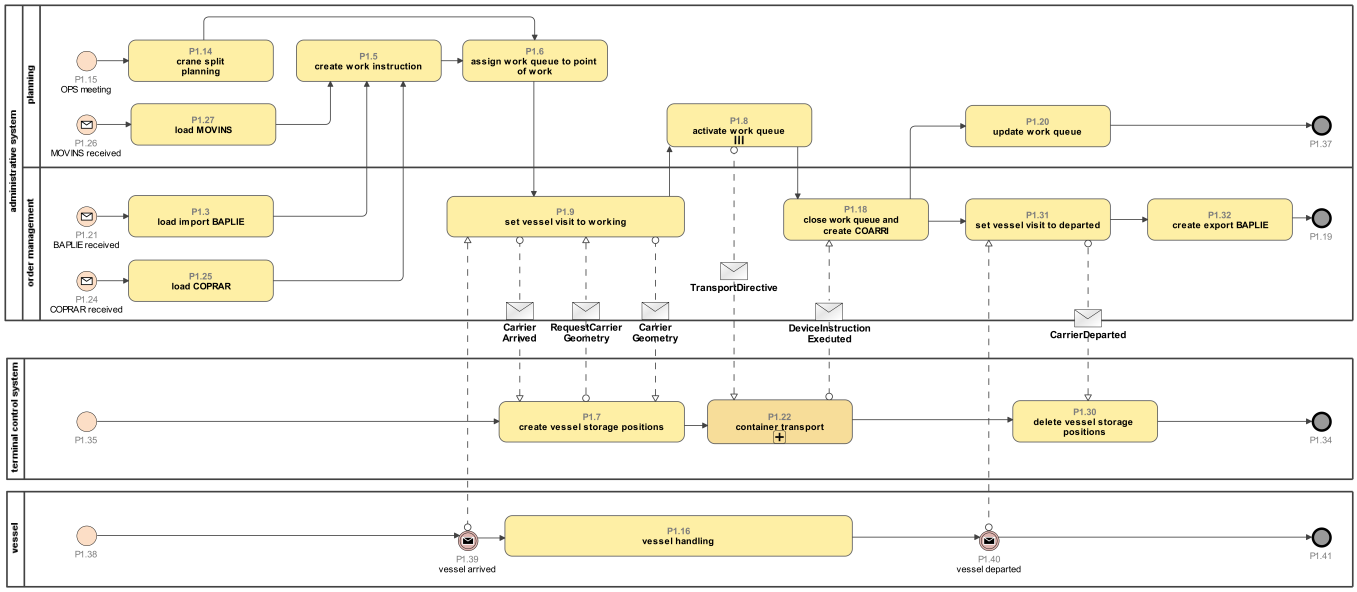

Figure 14. Resulting business process description using the business process based approach. All messages received are UN/EDIFACT messages carrying required order and storage information. Note that the events *vessel arrived* and *vessel departed* will be recorded manually in the administrative system.

TABLE VII. Resulting interfaces using the business process based design approach. All messages have been assigned to an interface according to the application responsibilities for business objects. For each message, its message type and its semantical content is given.

interface of the administrative system

message name	message type	message content
RequestCarrierGeometry	Request	requesting vessel geometry
CarrierGeometry	Reply	vessel geometry
TransportDirective	Notification	work queue assigned to a quay crane

interface of the terminal control system

message name	message type	message content
CarrierArrived	Notification	business event
DeviceInstructionExecuted	Notification	transport acknowledgement
CarrierDeparted	Notification	business event

X. INTERFACE OPERATIONS

Large application landscapes are usually operated in a 24/7 hour mode, requiring appropriate control, maintenance and support by an application management. The application management has to solve upcoming communication failures and supports application updates due to business changes and life cycle management requirements. Updating applications and communication infrastructure components requires deployment into an already running application landscape, resulting in business acceptable downtimes only.

To achieve these goals, interfaces must be versioned and the implementing applications have to be deployed during runtime, using migration patterns as described below.

A. Interface versioning

Every interface specification evolves over time due to syntactic, semantic or dynamic changes on the interface. These changes lead to different versions of the interface specification that are not compatible to each other. Therefore, the implementing applications must implement the correct version of the interface specification. In a complex application landscape, this is a common situation [8].

In order to guarantee a unique identification of a specific interface occurrence over time, each individual interface occurrence must have a version number [1], [8]. Note that any change on an interface leads to a new interface version [8]. This includes syntactical changes in any message, changes within the message sequence flow, i.e., all changes of the dynamic behavior, and changes of the semantic behavior. Even the obviously simple cases of adding either a field to an existing message or introducing a new message to an interface represents a semantical change of the interface. This requires compatibility of the receiving application with the new interface specification version. Otherwise, severe problems may arise, if, e.g., a *client* executes syntactical message checks based on a specific interface version.

B. Migration patterns

Rolling out a new interface version in an application landscape operating 24/7 hours, requires the usage of an appropriate migration pattern.

1) Big bang migration pattern: The simplest approach of an interface migration is *big bang*, where all applications are shutdown, redeployed and restarted at the same time, resulting in

$$1 + c \tag{1}$$

migration steps, where c denotes the number of participating *clients*. In case of a fall-back, the *server* and all *clients* must be redeployed again.

2) Client first migration pattern: Within this pattern, the migration path is dominated by the *clients*. Each *client* will be successively migrated onto a new version that can handle both interface versions in parallel, as shown in Figure 15. In

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

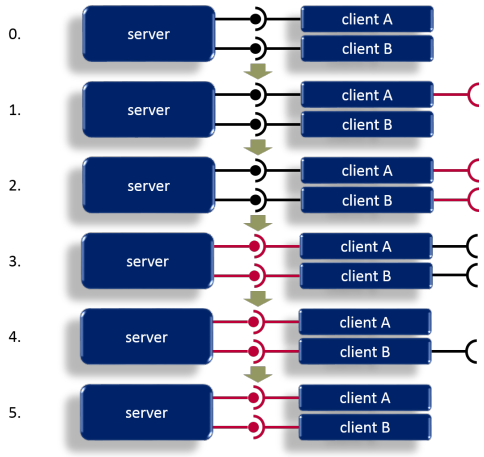

Figure 15. Steps of the *client first migration* pattern. The new interface version is denoted red.

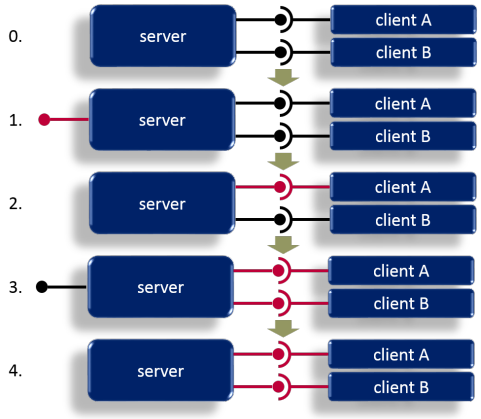

Figure 16. Steps of the *server first migration* pattern.

steps 1 and 2, the *clients* are changed to support additionally the new interface specification version. In step 3, the *server* is merged to the new interface implementation. Steps 4 and 5 are necessary to remove the support of the previous interface specification version from the *clients*. After finishing all *client* migrations, the *server* will be upgraded to support the new interface version. Afterwards, all *clients* will be updated a second time in order to remove the support of the old interface version. During steps one to four of this migration path, the *server* will receive messages with a wrong interface version that must be ignored by the *server*.

The *client first migration* pattern will result in

$$1 + 2 * c \tag{2}$$

deployments, where c denotes the number of *clients* connected to the *server*. An advantage of this migration path is that *clients* can be upgraded independently from each other, i.e., no temporal coupling of the individual *client* migrations exist. The price for this migration behavior is the necessary number of deployments: each *client* must be deployed two times, while the *server* is deployed only once. Furthermore, in case of a failure, the operational safe position of step 2 must be reached again. This is done by falling back with the *server* supporting the old interface version only and all *clients* whose support of the old interface version has been removed so far, requiring

$$1 + c_{+} \tag{3}$$

steps, where c_{+} denotes the number of *clients* migrated after the *server* migration.

3) Server first migration pattern: In contrast to the *client first migration* approach, the migration path can be reversed resulting in a *server* migration first followed by *client* migrations, see Figure 16. At step 1, the *server* provides support for two interface specification versions. In steps 2 and 3, both *clients* are merged successively. Finally, support of the previous interface specification version is removed from the *server* implementation, resulting in

$$2 + c \tag{4}$$

deployments. Again, c denotes the number of participating *clients*. The advantage of this pattern is, that the number of deployments is

$$(1 + 2 * c) - (2 + c) = c - 1 \tag{5}$$

less than with the *client first migration* pattern. Note that during steps one to three of the migration path both *clients* A and B will receive invalid messages, which must be ignored, due to the concurrent interface version support of the *server*. If a failure on the interface occurs within the migration path, all *clients* upgraded so far must fall back onto the previous interface version using c_{+} roll-out steps, where, again, c_{+} denotes the number of *clients* migrated after the *server* migration. Thus, the operational safe position of step 1 is reached again.

4) Mixed migration pattern: If the migration of an interface starts as a *client first migration* it could be changed to a *server first migration* at any time. The resulting *mixed migration* pattern is depicted in Figure 17. A *mixed migration* always starts with some *client* migrations followed by the *server* migration. Thereafter the remaining *clients* are migrated, resulting in

$$2 + 2 * c_{-} + c_{+} \tag{6}$$

deployments, where c_{-} denotes the number of *clients* migrated prior and c_{+} the number of *clients* migrated after the *server* migration. Important within this migration path is step 2, where the *server* and all *clients* migrated so far, i.e., c_{-} *clients*, must be deployed as a big bang. Otherwise, the *server* and these *clients* would communicate using two different interface versions in parallel usually leading to race conditions and doubled messages.

5) Comparison: The main differences between the migration patterns are the number of roll-out and fall-back steps as shown in Table VIII and the required support of multiple interface versions within the applications. Beside the advantages of a lacking necessity to support multiple interface versions and a minimal number of roll-out steps, the *big bang* pattern bears a high risk during fall-back situations where multiple applications must fall-back in parallel. Therefore, this pattern is only recommended if the number of *clients* is very small and a simultaneous fall-back is organizational manageable. Considering the other strategies, the *mixed migration* pattern suffers similar problems due to the required big bang at

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

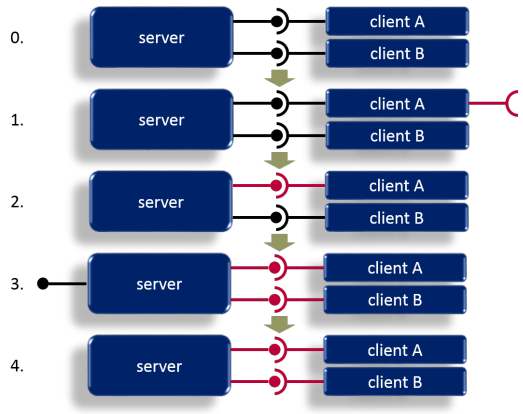

Figure 17. Steps of the *mixed migration* pattern.

migration step 2. Being a more complex pattern than *big bang*, its usage is not recommended. In contrast, *client first migration* and *server first migration* patterns reduce the risk involved with a possible fall-back compared to *big bang* at the cost of some additional roll-out steps. Since the *server first migration* pattern requires less roll-out and fall-back steps than the *client first migration* pattern, it is the recommended roll-out strategy.

XI. Conclusion

Due to the growing distribution of business functionalities, interfaces have become most important while operating applications within an application landscape. Badly designed interfaces have a critical impact on their functional and operational behavior [4]. To overcome these issues, this paper presents a structured and holistic approach to construct interface specifications considering all aspects and requirements of the business domain as well as IT operations.

Depending on the nonfunctional communication requirements to be satisfied to successfully integrate an application, adequate communication technologies have to be selected. Using messaging as integration style typically results in a good integration behavior for applications. In addition, further communication services encapsulated in an appropriate transmission protocol should be offered to deal with integration and error handling issues during operations. As a result, all applications implementing this transmission protocol can communicate in a robust and consistent way. This is a fundamental property for professional application management to support 24/7 hour operations in a large application landscape.

Interface specifications serve as contracts between applications. Thus, it is inevitable to define the artifacts *message description*, *dynamic description*, *semantic description*, *infrastructure description* and *quantity description* to properly describe an interface with respect to these different aspects. In order to construct an interface, different design approaches have been presented and compared to each other. It turns out that the business process based design approach is most likely leading to the best result with respect to robustness, performance and understandability.

Finally, different migration patterns have been presented introducing a new interface version into production stage. Due to the minimal number of required fallback steps in case of a severe error and one additional roll-out step compared to the *big bang* pattern the *server first migration* pattern is recommended, at least for larger application landscapes.

TABLE VIII. Comparison of the migration patterns with respect to the number of migration steps necessary and the number of applications that must be redeployed in case of a fallback.

pattern	rollout steps	application fallbacks
big bang migration	$1 + c$	$1 + c$
client first migration	$1 + 2 * c$	$1 + c_+$
server first	$2 + c$	c_+
mixed migration	$2 + 2 * c_- + c_+$	c_+

XII. Acknowledgment

The authors would like to thank Andreas Henning for valuable comments.

References

[1] A. Hagemann and G. Krepinsky, "(Inter)facing the Business," in FASSI 2016 - The Second International Conference on Fundamentals and Advances in Software Systems Integration. IARIA, Jul. 2016, pp. 1–7, ISBN: 978-1-61208-497-8.

[2] M. Henning, "API Design Matters," ACM Queue Magazine, vol. 5, 2007.

[3] J. Bloch, "How to Design a Good API and Why it Matters," 2006, URL: http://landawn.com/HowtoDesignaGoodAPIandWhyitMatters.pdf [accessed: 2016-06-08].

[4] R. J. Wieringa, Design Methods for Reactive Systems. Morgan Kaufmann Publishers, 2003, ISBN: 1-55860-755-2.

[5] M. Nygard, Release It!: Design and Deploy Production-Ready Software. O'Reilly, Apr. 2007, ISBN: 978-0978739218.

[6] U. Friedrichsen, "Patterns of Resilience," 2016, URL: http://de.slideshare.net/ufried/patterns-of-resilience [accessed: 2016-06-06].

[7] E. Evans, Domain Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2004, ISBN: 0-321-12521-5.

[8] B. Bonati, F. Furrer, and S. Murer, Managed Evolution. Springer Verlag, 2011, ISBN: 978-3-642-01632-5.

[9] G. Hohpe and B. Woolf, Enterprise Integration Patterns. Addison-Wesley, 2012, ISBN: 978-0-133-06510-7.

[10] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," The Internet Society, Specification, 1998.

[11] R. Fielding and J. Reschke, "Hypertext transfer protocol (http1.1): Message syntax and routing," 2014, URL: https://tools.ietf.org/html/rfc7230\#section-2.6 [accessed: 2017-01-09].

[12] "File Transfer Protocol," 1985, URL: https://tools.ietf.org/html/rfc959 [accessed: 2017-02-13].

[13] M. Happner, R. Burridge, R. Sharma, J. Fialli, and K. Stout, "Java Message Service," Sun microsystems, Specification, 2002.

[14] "Java Remote Method Invocation API," 2016, URL: http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/ [accessed: 2017-18-01].

[15] "Advanced Message Queuing Protocol (AMQP) specification," 2014, URL: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=64955 [accessed: 2017-25-01].

[16] "Blink Protocol," 2012, URL: http://blinkprotocol.org/ [accessed: 2016-06-06].

[17] "Financial Information eXchange," 2016, URL: https://en.wikipedia.org/wiki/Financial_Information_eXchange [accessed: 2016-06-06].

[18] "FAST protocol," 2016, URL: https://en.wikipedia.org/wiki/FAST_protocol [accessed: 2016-06-06].

[19] "UN/CEFACT Domains," 2017, URL: https://www2.unece.org/cefact/pages/viewpage.action?pageId=9603195 [accessed: 2017-18-01].

[20] "Service-oriented architecture," 2016, URL: https://en.wikipedia.org/wiki/Service-oriented_architecture [accessed: 2016-06-08].

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[21] R. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," dissertation, University of California, Irvine, 2000.

[22] M. Iridon, "Enterprise Integration Modeling," International Journal on Advances in Software, vol. 9, no 1 & 2, 2016, pp. 116–127, ISSN: 1942-2628.

[23] H. Kerner, Rechnernetze nach ISO-OSI, CCITT. H. Kerner, 1989, ISBN: 3-900934-10-X.

[24] "OSI model," 1984, URL: https://en.wikipedia.org/wiki/OSI_model [accessed: 2017-01-26].

[25] "Client-server model," 2016, URL: https://en.wikipedia.org/wiki/Client-server_model [accessed: 2016-03-03].

[26] R. Westphal, "Radikale Objektorientierung - Teil 1: Messaging als Programmiermodell," OBJEKTspektrum, vol. 1/2015, 2015, pp. 63–69.

[27] "OMG Unified Modeling Language," 2017, URL: http://www.omg.org/spec/UML/2.5/PDF/ [accessed: 2017-24-01].

[28] J. Hopcroft, R. Motwani, and J. Ullman, Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 2001, ISBN: 0-201-44124-1.

[29] U. Friedrichsen, "The promises and perils of microservices," 2017, URL: https://www.slideshare.net/ufried/the-promises-and-perils-of-microservices [accessed: 2017-02-24].

[30] P. Bernstein and E. Newcomer, Principles of Transaction Processing. Elsevier Inc., 2009, ISBN: 978-1-55860-623-4.

[31] U. Friedrichsen, "Watch your communication," 2016, URL: https://www.slideshare.net/ufried/watch-your-communication [accessed: 2017-02-24].

[32] J. Humble and D. Farley, Continous Delivery. Addison-Wesley, 2010, ISBN: 978-0-321-60191-9.

[33] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerland, Security Patterns, ser. Software Design Patterns. John Wiley & Sons, Ltd, 2005.

[34] "Extensible Markup Language," 2008, URL: https://www.w3.org/TR/2008/REC-xml-20081126/ [accessed: 2017-02-13].

[35] "XML Schema Definition Language," 2012, URL: https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/ [accessed: 2017-02-13].

[36] "ISO 8601," 2016, URL: https://en.wikipedia.org/wiki/ISO_8601 [accessed: 2016-11-10].

[37] "Base64," 2016, URL: https://en.wikipedia.org/wiki/Base64 [accessed: 2016-11-10].

[38] "ActiveMQ," 2017, URL: http://activemq.apache.org [accessed: 2017-05-25].

[39] "SwiftMQ," 2017, URL: http://www.swiftmq.com [accessed: 2017-05-25].

[40] "Java Language and Virtual Machine Specifications," 2017, URL: http://docs.oracle.com/javase/specs/ [accessed: 2017-05-25].

[41] A. Cockburn, Writing Effective Use Cases. Addison-Wesley, 2001, ISBN: 978-0-201-70225-5.

[42] B. Oestereich, Objektorientierte Softwareentwicklung: Analyse und Design mit der UML 2.0. Oldenbourg, 2004, ISBN: 978-3486272666.

[43] "Information technology – Object Management Group Business Process Model and Notation," 2013, URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=62652 [accessed: 2017-02-14].

2017, © Copyright by authors, Published under agreement with IARIA - www.iaria.org