On the Application of the Service-Oriented Architectural Style to Heterogeneous Application Landscapes

Dissertation
zur Erlangung des Grades
des Doktors der Naturwissenschaften (Dr. rer. nat.)
der Fakultät
Wirtschaftsinformatik und Angewandte Informatik
der Otto-Friedrich-Universität Bamberg

vorgelegt von
Helge Hofmeister

Bamberg,
Dezember 2008

1. Gutachter: Prof. Dr. Guido Wirtz, Otto-Friedrich Universität Bamberg
2. Gutachter: Prof. Dr. Mathias Weske, Hasso-Plattner-Institut Potsdam

Tag der Disputation: 3. April 2009

Abstract

This thesis investigates the application of the service-oriented architectural style in the context of industrial enterprises. This style provides a commonly perceived paradigm to organize distributed software systems. However, beyond the attention this style attracts, little description exists on what differentiates service orientation from already more mature styles, such as component orientation.

This dissertation argues that the service-oriented style is an approach that centralizes control over distributed functionality that is provided by the application systems of an organization. This means that service orientation is a paradigm for application integration. Moreover, we argue that it is not possible to fully formalize the inherent principles of service orientation as part of an architectural style description. In fact, *soft* design principles are a differentiator of this style. These principles, however, are rarely implied in the context of industrial enterprises. This is why we analyze how such design principles could be objectively described and what attention should be paid to them. This analysis is performed by categorizing the potential benefits of this style and assigning the respective principles to the identified benefits. Subsequently, a reference architecture is defined on the basis of these findings. This architecture focuses on structuring service-oriented applications — so-called composite applications.

In order to apply the defined reference architecture in the context of actual projects, we describe a design methodology for composite applications. This design methodology focuses on predominantly using business processes to design various types of services and their interconnections as they are described by the reference architecture. To achieve this, a service design algorithm is included that derives services for business processes by incorporating the results of a statistical analysis of service design principles.

The deliverable of this methodology is the platform-independent design of a composite application that incorporates the restrictions of the application landscape in which a composite will be deployed. To allow the realization of composite applications, we informally map the platform-independent reference architecture to a platform that is widely used in industrial enterprises. This mapping is, together with the design methodology and the reference architecture, applied to an industry-scale use case. This way the applicability of the concepts is demonstrated.

Zusammenfassung

Diese Dissertation untersucht, wie der service-orientierte Architekturstil von großen Konzernen angewandt werden kann. Dieser Architekturstil strukturiert verteilte Systeme. Neben allgemeiner Aufmerksamkeit sind jedoch nur wenige Beschreibungen verfügbar, die zeigen, wie sich dieser Stil von anderen – so z.B. von dem der Komponentenorientierung – absetzt.
Die vorliegende Arbeit beschreibt, dass der service-orientierte Stil ein Ansatz zur Kontrollzentralisierung ist. Dabei zentralisiert er die Kontrolle über der Funktionalität, die von den Applikationssystemen einer Organisation bereitgestellt wird. Daher kann er auch als Integrationsansatz verstanden werden. Zusätzlich ist es nicht möglich, alle Prinzipien des service-orientierten Stils als Architekturstil zu beschreiben. Tatsächlich stellen *weiche* Designprinzipien das Alleinstellungsmerkmal dieses Stils dar. Diese Prinzipien werden allerdings von großen Organisationen kaum verinnerlicht. Diese Dissertation untersucht deshalb den Stellenwert der einzelnen Prinzipien und beschreibt, wie sie objektiviert werden können. Diese Untersuchung wird dadurch betrieben, dass die möglichen Vorteile dieses Architekturstils zunächst kategorisiert werden. Danach werden diesen potentiellen Vorteilen die Prinzipien zugewiesen, die zu ihrer Erlangung beitragen. Auf Basis dieser Analyse wird danach eine Referenzarchitektur erstellt. Diese Architektur beschreibt eine Struktur für service-orientierte Applikationen, so genannte composite applications, die es erlaubt *weiche* Designprinzipien zu berücksichtigen.
Um diese Referenzarchitektur auf reelle Problemstellungen anwenden zu können, wird außerdem eine Methodik zum Design von composite applications beschrieben. Im Fokus dieser Methodik steht es, Geschäftsprozesse als Ausgangspunkt für das Design von Services zu benutzen und deren Verbindungen zu beschreiben. Dabei orientiert sie sich an den möglichen Verbindungen, die von der Referenzarchitektur definiert werden. Um dies zu erreichen ist ausserdem ein Algorithmus für das Design von Services enthalten. Dieser Algorithmus leitet aus Geschäftsprozessen Services ab und berücksichtigt dabei die Ergebnisse einer statistischen Auswertung, die die Wiederverwendbarkeit von Services als Untersuchungsziel hat.
Das Resultat dieser Methodik ist das plattformunabhängige Design einer composite application. Dieses Design ist an einem Geschäftsprozess orientiert, berücksichtigt aber auch die Einschränkungen der Systemlandschaft, in der die Applikation betrieben werden soll. Um auf der Basis eines solchen Designs eine composite application erstellen zu können, ist weiterhin eine informelle Abbildung der plattformunabhängigen Architektur auf eine häufig eingesetzte Plattform in der Arbeit beschrieben.
Zusammen mit der Methodik und der Referenzarchitektur wird diese Abbildung dann auf ein industrielles Fallbeispiel angewandt. Auf diese Weise wird die Anwendbarkeit der beschriebenen Konzepte demonstriert.

Contents

1 Introduction

Classical application integration is a substantial part of a large organizations' information technology (IT) landscape in terms of resources, time and budget. As a rule of thumb, practitioners estimate that 20% of project costs are integration costs. In addition to project costs, integration projects increase complexity within an IT landscape. This causes an overall increase in the total costs of IT ownership. However, even more strategically important than cost is the flexibility of an IT landscape in terms of how it can respond to changes in business models and processes.

Integrating different application systems to support certain business requirements is a fairly inflexible mechanism. This is due to the fact that integration systems do not (and should not) implement business logic. Furthermore, they are adapted to multiple application systems that do implement this business logic. Evolutions in business that lead to changes on an IT level consequently require modifications of (back-end) application systems. These changes usually affect interfaces to other application systems as well. Thus, integration systems need to be changed fairly frequently. Additionally, new business models or mergers and acquisitions draw special attention to the integration systems as they allow for connectivity among different intra- or inter-organizational entities.
As a result, changes in business models usually affect several application and integration systems. Making changes to these systems is a costly and, more importantly, time consuming business.

In order to reduce the complexity of integrated application systems, the concept of consolidation was proposed (cf. [1]). Consolidating both application and integration systems aimed at reducing complexity by first eliminating the need for interfaces among application systems and second by standardizing interface development and maintenance.
To avoid consolidation leading to highly complex application systems, organizations were more or less forced to use standard software (global players especially chose that approach). By doing so, particular competitive process-related advantages of an organization were not supported anymore by the IT systems. This is why large organizations kept a variety of commercial-off-the-shelf software (COTS) and home-grown applications in their application landscape. These systems were integrated using integration systems.

1.1 Problem Statement

Service-oriented architecture (SOA) or service orientation (SO) is an architectural style that allows the construction of applications that reuse distributed functionality of heterogeneous application landscapes. Applications that reuse functionality and expose their functionality as web-based applications are so-called composite applications (cf. e.g. [2]). Together, SO and composite applications promise to protect investments in legacy landscapes by reusing the existent functionality while allowing for the incorporation of recent business changes.

Garlan identified uncertainty about the control model as a major issue when building systems that reuse existing parts (cf. [3]). This issue persists when building service-oriented composite applications.
One way to address this issue is the concept of Business Process Integration Oriented

Application Integration (BPIOAI) introduced by Linthicum (cf. [4]). This concept centralizes the control model outside the participating application systems and uses business processes as the central control instance over distributed functionality. This functionality can be exposed by the means of services that have a formally described interface (cf. [4]).

Papazouglou stated that an SOA allows business process-centered control over distributed services by introducing process-centered service aggregation, or so-called service orchestration. The latter is introduced as a part of a service-oriented architecture. It serves as a mechanism for aggregating basic services to more specialized services (cf. [5]).
From the proposed aggregation of services, another possible benefit for the industry can be identified: required changes for functional enhancement can be realized as additional services that are aggregated together with services that expose standard functionality of COTS. Such aggregators could thus provide the required functionality which is typically offered by separate systems. This way SO could also contribute to keeping COTS unmodified – which is a major aspect of today's IT governance (cf. [6, pp. 69f.]).

SO obviously comes along with some promise that could be beneficial both for industry companies and for software/IT services companies. However, these groups' perception of SO is today mainly influenced by promises of vendors (examples are SAP [7], IBM [8], Oracle [9] and IONA [10]). Also, science has begun to approach the topic (cf. e.g. [11], [5], [4], [12], [13] to name a few). In the meantime, even an addition to the service-oriented architectural style is intensively researched: the area of semantic service provisioning and consumption is an interesting topic that shows first promising results (eg. [14]).

However, the gap between fundamental research, vendor promises and customer still exists and seems to be growing. One reason for this is that companies have not yet determined how to approach the topic and how to make the beneficial ideas of SO tangible. On the other side, there are still no concepts available that point out what could make SO beneficial for the industry and IT suppliers, how they could leverage these potential benefits and how this new way of application development could be approached (even if software vendors would pretend the opposite). Today, there is little information available about how to structure problems in such a way that service orientation can be beneficial. Further, there are few ways to structure composite applications such that service-oriented principles are incorporated while actual requirements are realized in a heterogeneous application landscape.
This analysis is shared by Schelp and Winter. In a recent analysis they state that "existing literature does not sufficiently address the integration layer and its importance for decoupling business related structures on one hand, and IT related structures on the other. This decoupling however is a necessary precondition for buffering changes and supporting alignment, hence for contributing to agility on a sustainable level" [15, p. 68].

The objective of this thesis is to analyze existing work in the domain of service orientation, identify gaps between the current state of research and real-life requirements and to fill these gaps with concepts that make service orientation applicable.

1.2 Approach

The findings of this thesis were partially developed during a BASF IT Services project that addressed service orientation. BASF IT Services is an IT services provider that generates large portions of its revenue through industry customers. Thus, evaluating the service-oriented architectural style seemed promising to the company because of the integration issues described above.
In order to analyze the impact SO could and should have on application integration, the application landscape, business process support, application governance and the realization of web-based applications, BASF IT Services decided to start with a first analysis of this architectural style. As there was little experience in terms of requirements engineering, design and programming, a project was launched in 2004 that would provide experience in the area of SO and process-based application development. This experience took both a conceptual approach as well as an implementation approach.

This thesis is related to this project. The requirements of the project were used as exemplary industry requirements to develop and verify the concepts presented in this thesis. By incorporating these concepts back into the project allowed to analyze their value for the company and to prepare the introduction of SO into companies such as BASF IT Services and its customers.
Essential to achieving this objective is the structuring of application development and application integration from an architectural point of view. This will be done by constraining and detailing the service-oriented architectural style as it is commonly described today. By providing more structure, constraints and guidance, SO will become more applicable in real life. Further, by using this approach, it will also be possible to determine what aspects of an SOA are related to the architectural style and what aspects are actually introduced by soft design principles that can not be formalized as elements of a style.
In order to apply SO with all its principles, a reference architecture will be defined that structures composite applications. Additionally, a design and development methodology is required. Only methodological support will help organizations internally propagate these concepts and comply with the principles this style comes along with. Finally, the developed concepts need to be evaluated with regards to their applicability.

First, an appropriate definition of the service-oriented architectural style will be given. By analyzing existing work, a formal model of an SOA will be established and described using an architectural description language (ADL). Additionally, how vendor platforms can support the application of this style will be described.
Based on this description, it will become clear that the assumed benefits of SO can not be fully captured by these descriptive means as they are too generic.
Accounting for this descriptive deficiency, a framework will be established that supports the analysis of how SO can provide additional benefits to software engineering. This will be done by identifying key metrics that describe how certain principles of service orientation can be measured. The discussion of these metrics will also be used to describe service-oriented principles in greater depth.
Reusability is one of the commonly accepted principles of service orientation. In order to determine whether the design of services can influence their reusability, a quantitative analysis will be performed, that will investigate the significance of reusability to this architectural style.
Having highlighted what service-oriented principles mean and what they can and can not

improve, a detailed, constrained and therefore tangible blueprint for composite applications will then be presented. This blueprint aims to support projects in their application of the service-oriented architectural style and to leverage the identified concepts. To make it possible to apply the blueprint to actual problems, a design methodology will also be presented. This design methodology can be used in real-life settings to drill-down on business processes and to derive a design for composite applications. In conjunction with the reference architecture, this is a major contribution of this thesis.
In order to evaluate the previously identified concepts, a case study will be presented. This case study was conducted during 15 months at BASF IT Services and involved around 20 project members. In order to realize the use case, a generic mapping of the architectural blueprint to a strategic SOA platform of BASF IT Services will be given.
Based on these findings, the design of the use case will be described. By analyzing this use case it will be possible to assess how the service-oriented style could contribute to the success of realizing cross-system applications. Additionally, it will help to verify whether the presented concepts provide an additional value compared to using the more generic definition of an SOA.

1.3 Thesis Structure

This thesis reports on the background and main steps sketched above.

First, the meaning and definition of an SOA will be analyzed in chapter 2. Based on the identified common understanding, SO will be defined as an architectural style. Additionally, a methodology will be introduced that supports the categorization of composite applications and SOA platforms.

As these definitions demonstrate, only a generic part of the definition of service orientation can be formalized as an architectural style. Not all service-oriented principles can be captured by such definitions.
In chapter 3, potential benefits and trade-offs of SO will be introduced and discussed. This will be achieved by detailing potential benefits and analyzing how they can be leveraged. Based on the identified potential, basic measures and more complex metrics for the assessment of service-oriented design will be introduced. To support future projects, an approach will be presented that can be used to evaluate the suitability of the service-oriented style for given requirements.

Chapter 4 will provide a quantitative analysis of a web service-based component architecture. The analysis will determine whether service design can influence service reuse and how much attention should be given to the design of the actual services. The objective is to identify design principles that are significant discriminators for the re-usability of services. Based on these findings, recommendations for design methodologies will be made that can be incorporated into actual design methodologies.

In chapter 5, the identified benefits and requirements for an applicable service-oriented style will be used in order to define a reference architecture for composite applications. This reference architecture will allow business process-centered reuse of legacy functionality and define a structure on top of the basic service-oriented architectural style. It will allow the structuring of composite applications and the integration of all principles of service orientation into actual projects. The reference architecture will be described without

any reference to actual target platforms and service standards. This is why it allows for designing composite applications in a platform-independent way. While the description of the reference architecture is independent from any platform, the reference architecture emphasizes the notion of heterogeneous application landscapes. This is achieved by integrating several mechanisms that allow for realizing composite applications in heterogeneous application landscapes.

A step-wise design methodology for composite applications will be presented in chapter 6. This methodology incorporates the findings of chapters 3 and 4. It allows for designing composite applications and services in alignment with the reference architecture of chapter 5. The steps of the methodology combine a top-down approach that uses business requirements for the design of a composite application with the bottom-up integration of constraints that arise from the actual application landscape. By incorporating these constraints, the platform-independent design is adjusted for a specific target platform. By integrating key metrics of chapter 3 into the methodology, reaching compliance with service-oriented principles will be optimized.
This approach will also introduce a problem-oriented meta-model for services. This model will, in conjunction with the findings of chapter 4, facilitate the integration of legacy systems into service orchestrations that underlie composite applications and support reusability of certain types of services.

The composite application reference architecture will be mapped to an actual platform in chapter 7. By using the findings of chapters 5 and 6, this platform-specific reference allows for the realization of service-oriented solutions for actual business requirements.
The platform that is relevant for BASF IT Services will be chosen as the target platform of this mapping.

In chapter 8, the described concepts will be applied to a case study. The suitability of the service-oriented style for the given requirements will then be discussed. Subsequently, the design methodology of chapter 6 will be applied to these requirements. The result will be the design of an actual composite application for an actual business problem as described in chapter 5.
The metrics that are introduced in chapter 3 will be applied to the design to provide a quantitative evaluation of the design and to assess the applicability of metrics in that context. Finally, a composite application that is based on the elaborated design will be described in order to demonstrate the applicability of the concepts.

The findings and contribution of this thesis in contrast to related work will be discussed in chapter 9.

Chapter 10, the conclusion, will analyze and synthesize the concepts elaborated upon throughout the thesis.

The main achievements of this work have been published in a series of international conferences over the last years (cf. [16], [17], [18], [19], [20], [21], [22]).

2 Service-Oriented Architecture

Service-oriented architecture (SOA) is a kind of software architecture that is mainly visible within the industry because software vendors often push this concept into the market (cf. e.g. SAP [7], IBM [8], Oracle [9], IONA [10]).
Even if the diffusion of this concept is purely driven by marketing and thus could be ignored, organizations deploying products from vendors that switched to SOA-like platforms need to be prepared. This is because future updates of their standard software will involve service-oriented principles. Hence, only because of the market power of some software vendors, companies need to deal with this paradigm.
An architectural style is a "collection of conventions that are used to interpret a class of architectural descriptions" [23, p. 320]. Such an architectural description consists of components, connectors, compositions of components and connections (aka configurations), properties and constraints (cf. [24]).
In an SOA, the components are services. Connectors are the calling relations among those services that all rely on a common protocol (cf. [4, p. 218]). However, *component* is a rather overloaded term. Szyperski gave a widely referenced definition of what a component is: "A software component is a [...] unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" [25, p. 50].
Following the definition of *services* as the components of SOA, services are individual units that exist autonomously and expose their functionality via self-describing interfaces (cf. [11], [5]). By this definition it becomes obvious that component orientation and service orientation become almost identical approaches.
Cerventes et al. identified in [12] the difference between components and services in that services are abstract descriptions of components that become concrete during runtime through dynamic lookups. Fowler notes that components are actually services intended to be used locally while services are used remotely (cf. [26]).
Erl states that *how* the elements of the service-oriented architectural style are designed distinguishes service orientation from component orientation (cf. [11, p. 36]).
Despite these definitions, the fact that an SOA can be realized (and, more important, *is* realized) using open and easily understandable standards like the XML-based SOAP [27] also contributes to the diffusion of this style. Services that are accessible via the SOAP protocol are referred to as *web services* (cf. [5]).

Birman and Vogels discussed whether or not web services are distributed objects (cf. [28] and [29]). The discussion is based on topics like life-cycles of objects and web services, reliability of services and service referenciation. The common understanding is that web services are *not yet* distributed object-based systems because they still lack certain aspects that industry-proof software has to fulfill. This, however, is not important from a methodological point of view and time has brought additional so-called *WS-** standards (e.g. [30]) that address these issues.
Does that mean that SOA will solely force organizations to SOAP-enable their software components to implement this new architectural style? This actually depends on *the way* that the existing components are designed and used. Beside registries and remoting, service orientation is an architectural style that emphasizes the building of applications by (re-) using distributed functionality which were not realized as components initially. SOA concerns functionality as it can be found in heterogeneous application landscapes. Service-oriented applications that are built on top of legacy functionality are called com-

posite applications (cf. [2]). Landscapes that are (re-) used by composite applications consist of both a variety of commercial-off-the-shelf software (COTS) and home-grown applications.
In a non-service-oriented landscape, these applications are of course usually already integrated with each other in order to achieve certain functionalities. The paradigm used for integration does, however, vary.

2.1 Application Integration

Today, different paradigms are deployed to support business processes that span multiple systems. Linthicum identified four approaches to application integration (cf. [4, pp. 6-19]) that describe these paradigms. This section gives an overview of these approaches and highlights typical advantages and disadvantages.

Portal-Oriented Application Integration Portal-Oriented Application Integration does not focus on automating the connections between application systems. It aims to allow a user to interact with different application systems without much effort. For that sake, it does not aim at automating the conjuncted support of different application systems for an abstract task. Furthermore, different application systems need to be unified into one single user interface.
While allowing for quick results, this integration approach can not be considered to support the realization of cross-system processes. The reason is that it does not aim at automation. Furthermore, it facilitates manual work. Providing centralized portals with access to different back-end systems is a commonly appreciated approach as it frees workers from the requirement of knowing details about the actual system landscape, though.

Information-Oriented Application Integration Information-Oriented Application Integration (IOAI) approaches intend to provide simple mechanisms to automatically exchange information between different application systems.
This can be achieved using different categories of IOAI: Data Integration and Interface Processing.

- **Data Integration**. Linthicum does not group data replication and data federation into one major group. The approaches described below are very similar, though. Both aim at allowing different application systems which are using databases to operate on the same sets of data. Integration is thereby done by one application system writing a data entity to a database and another application system reading the data entity.
  Technically this can be done in the following ways:

  Data Replication. Linthicum defines data replication as "simply moving data between two or more systems" [4, p. 7]. Because different application systems use different databases, the application systems could operate on the same set of data even if the data schemes differ whenever the databases are replicated. Replication is realized using an additional layer above the database functionality. This layer communicates changes from one database to another replicated database through

messages.
Data replication is widely applied today. This is because it allows for failure tolerant set-ups that do not require all systems to be available. The described layer above the databases is usually an Enterprise Application Integration (EAI) integration server in addition to database mechanisms such as triggers. The EAI platform delivers data as scheduled. The trigger or a similar mechanism transfers the data from a transfer space within the database to the productive data area of the database.
This approach has two drawbacks. On one hand, the integration is not performed in real-time but on a time-scheduled base. The replication is usually performed once per day. In case of changes that become necessary (for instance due to business events), clerks that have to perform these changes need to know whether the data was already replicated when they perform such a change.
The described scheduling does usually de-couple data transfer from functionality of the application system that operates on the database. This is how the complexity of the EAI integration platform can be controlled. On the other hand, the control of integration is distributed. At least the EAI platform and the database internal transfer mechanism are both controlling the integration. This renders such solutions into hardly modifiable constructs.

Data Federation. This approach describes "the integration of multiple databases and database models into a single, unified view of the database" [4, p. 7]. Here, an additional layer is used to make different applications virtually operate on a local database which, in reality, is a set of distributed databases.
A benefit is that this approach, "allows access to any connected database in the enterprise through a single, well-defined interface" [4, p. 9].
This integration principle is rarely used within commercial organizations. There are two reasons: differing data formats of application systems need either to be unified or converted on the fly. The first approach is hardly achievable due to the immense efforts that would be necessary. The second approach leads to a runtime overhead that has negative effects on the performance of the applications. This is because any data accesses is not only performed remotely but also converted. As database operations are usually very fine granular, the performance is affected negatively.
A further obstacle that comes with this solution is that the integration is achieved implicitly by multiple applications operating on the same set of data. This tightly couples the applications and prohibits the definition of a clear point of control. Especially when integrating COTS, this approach is hardly applicable.

There are non-technical constraints related to the autonomy of the participating organizations. Such constraints include that organizations will rarely grant external partners access to their databases. If further common data schemes are required, data integration is not feasible. This is why data integration approaches are only useful for application integration within a single organization.

An integration approach that is more applicable to inter-organizational integration is interface processing. This approach is, of course, also applicable for the integration of application systems within organizations.

- **Interface Processing**. This approach does not only focus on data. Interface processing uses well-defined application interfaces to access data and *functions* of application systems as well. Interface processing is a mechanism that uses functions of application systems to extract information and communicate it to other applica-

tion systems (cf. [4, p. 10]).
If those functions are described by structural interfaces, the integration approach is described as **Service-Oriented Application Integration** (SOAI) (cf. [4, p. 10]).
The distinction of the interface processing and the SOAI is obviously blurred. In their genuine form, interface processing is a message-based approach that is used for the transmission of discrete sets of data among application systems. SOAI describes discrete function calls instead of messaging. But of course, processing data can also involve functionality that operates on the exchanged data. This is why the two approaches are not easy to distinguish.
SOAI has the advantage of combining integration and application logic while interface processing usually distinguishes between integration and business functionality (cf. [4, p. 10]). In this basic form, both approaches share one major downside: the connected application systems all share the control of overall integration. Thus, changing the underlying business process necessarily involves modifying application systems.

2.2 SOA as a Paradigm for Control Centralization

In [3] Garlan identified a lack of certainty about the control model as a major issue when building systems by re-using existing parts. As the described integration paradigms implicitly combine data and functionality of application systems with each other, the control model can not be obvious.
One way to resolve the issue of uncertainty about the control model is the concept of Business Process Integration Oriented Application Integration (BPIOAI) that is also introduced by Linthicum in [4]. This concept centralizes the control model outside the participating application systems and uses business processes as the central control instance over distributed functionality. This functionality can be exposed by means of services that have a syntactically described static interface. Hence, it can be considered as an extended concept in addition to SOAI (cf. [4, pp. 55ff.]).
Papazouglou identified the business process-centered re-usability of business functionality as a major benefit of SO (cf. [5]). Service orientation allows for such business process based *configurations*[1] by introducing the concept of *service orchestration*. Orchestration is a specialization of service aggregation that assembles basic services in a process-oriented way to higher-level services (cf. [5], [31]).

We consider BPIOAI that relies on (possibly aggregated) services as a way to realize the service-oriented architectural style, since it describes a control model for the integration of application systems using services. This means that we see service orientation as an architectural style that allows for developing applications by (re-)using existing software. Hence, it is a process-centered application integration principle.
Other inherent features of an SOA are shown in figure 1. These include service discovery, monitoring and management. The most important implication of this description is, however, the layered specialization of basic services to managed composite services. The specialization of services to more problem-oriented services is achieved by the concept of aggregation.

[1] In terms of an architectural description language.

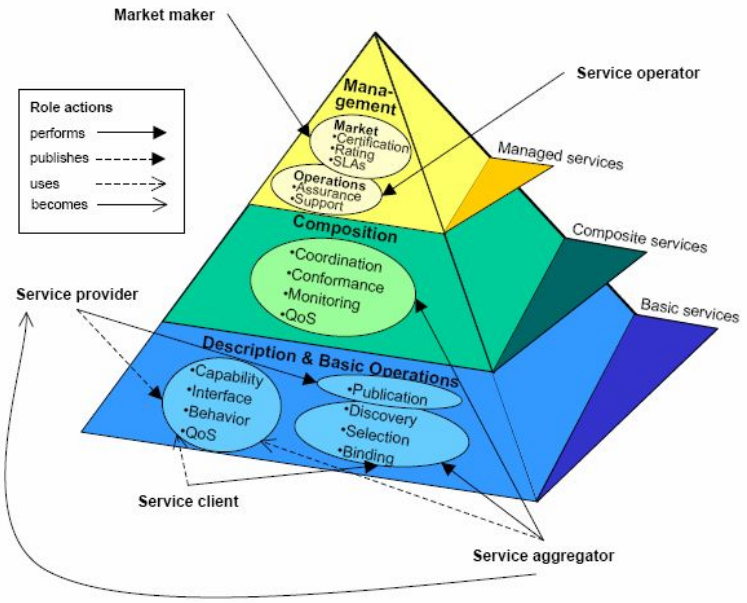


Figure 1: Service-Oriented Architecture according to [5]

Another aspect of SO is that aggregation of services can be used to realize changes that are required for functional enhancement of legacy COTS-based applications. By aggregating services that expose standard functionality with custom-made services that add the required functionality, SO could contribute to keep COTS unmodified. This is a major objective of today's IT governance.

2.3 SOA Defined

To our knowledge, the service-oriented architectural style has not yet been defined formally. This section introduces a definition of *service orientation* with the minimum set of required components, connections and constraints. This definition underpins the further elaborations of this thesis. However, it is not used to decide whether a given architecture is compliant with this architectural style.
Based on this definition, it is possible to describe which concepts are incorporated into the service-oriented style while being optional for other distributed computing concepts. It will also discuss the fact that not all aspects of service orientation can be captured through a style description. Based on this identified delta, metrics are defined in chapter 3 that measure both compliance to adherent service-oriented principles of a given design as well its SOA-specific qualities. Since a major portion of service orientation is the designing of services and thus can not be formalized by an architectural style, service design is discussed in more depth in chapter 4.

There exist different languages for the purpose of describing architectural styles (cf. [24]). We chose *ACME* [32] as the architectural description language for three reasons. First, ACME is considered a universal architectural description interchange language (cf. [24], [32]). Thus, the language unifies different approaches of architectural description and is therefore a universal mechanism for such descriptions. Second, ACME-models can be analyzed using first-order relational logic (cf. [33]). This will allow, if required, for a more

fundamental access to the analysis of the design of composite applications. For instance it could be verified that a given design is created using the service-oriented style. And third, (recent) tool-support for ACME exists (cf. [34]). This allows the definition done here to be usable in actual projects and shows that the ACME project is still active and supported.

Yanchuk et al. state that the most basic occurrence of an SOA is a client-server architecture (cf. [35]). We use the basic definition of the client-server style in order to approach the definition of the service-oriented style.
As a foundation for further definitions, the client-server architectural style is described in listing 1 using ACME language on the basis of the definition given in [32].

Listing 1: The Client-Server Architecture Style according to [32]

```
STYLE clientServer = {
  COMPONENT Type Client = {
    PORT send-request : RPCPort ;
  }
  COMPONENT Type Server = {
    PORT receive-request : RPCPort ;
    PROPERTY hasState : BOOLEAN;
    PROPERTY state : SET {};
    PROPERTY functionality : SET {};
  }
  CONNECTOR Type Rpc = {
    ROLE caller : ClientRole;
    ROLE callee : ServiceRole;
    PROPERTY asynchronous : boolean;
    PROPERTY protocol : CommunicationProtocol;
  }
  ROLE Type ClientRole;
  ROLE Type ServerRole;
  PROPERTY Type CommunicationProtocol = SEQUENCE <>;
  PORT Type RPCPort = {
     RULE rule1 = INVARIANT FORALL c : Client in Self.Components |
                            EXISTS r : Rpc in Self.Connectors |
                            attached(c, r);
     RULE rule2 = INVARIANT FORALL s : Server in Self.Components |
                            EXISTS r : Rpc in Self.Connectors |
                            attached(s, r);
  }
}
```

Listing 1 shows that systems shall, in order to comply with the client-server style, consist of two types of components: *Clients* and *Servers*. Clients send requests to servers using *Remote Procedure Calls* (RPC) in a synchronous or asynchronous fashion. It is implicitly included in this definition that clients and servers can be distributed among different machines in a network.

Listing 2: The Service-Oriented Architecture Style (Part I)

```
STYLE serviceOriented EXTENDS clientServer WITH {
  COMPONENT Type Service = {
     PORT deployable : DeploymentPort;
     PROPERTY realization : Language;
     PROPERTY protocol : CommunicationProtocol;
  }
  COMPONENT Type Agent = {
     PORT deployment : DeploymentPort;
     PROPERTY canExecute : Language;
     PROPERTY canCommunicate : CommunicationProtocol;
     PROPERTY physicalAddress : Address;
     // Every service that is deployed on a platform ...
     RULE canExecute = INVARIANT FORALL s : Service in Self.Components,
                                        a : Agent in Self.Components,
                                        d : Deployment in Self.Connectors |
                          attached(s, d) AND  attached(a, d)->
                          // ... needs to be executable there ...
                          (s.realization == a.canExecute) AND
                          // ... and its protocol must be supported
                          (s.protocol == a.canCommunicate);
  }
  CONNECTOR Type Deployment = {
     ROLE deployee : deployeeRole;
     ROLE deployTarget : deployTargetRole;
  }
  ROLE Type deployeeRole;
  ROLE Type deployeTargetRole;
  PORT Type DeploymentPort = {
      // Every service needs to be deployed to an agent:
     RULE deployAll = INVARIANT FORALL s : Service in Self.Components,
                                       a : Agent in Self.Components |
                        EXISTS d : Deployment in Self.Connectors |
                        attached(s, d) AND attached(a, d);
  }
                              [...]
```

Listing 2 introduces the service-oriented style as a specialization of the client-server style. A distinction is made between the *services* and the *agents* that execute the services. This distinction allows for a more transparent distribution of services within a landscape of agents. In turn, this implies that services represent an abstract set of functionality unless they are deployed on an agent. We refer to this abstract set of functionality as *service type* (cf. [36]).
A constraint of that style is that services can (and must be) deployed on appropriate agents. *Appropriate* thereby refers to two characteristics: first, the realization of a service must be compatible with the runtime environment of the agent it is deployed on. Second, the communication protocol a service relies on must be supported by the agent as well. The configuration of service and agent forms a resource that provides or consumes functionality that exists in a given set of systems.

The notion of providing and/or consuming functionality is expressed in listing 3 which is a continuation of listing 2. Two components extend the client and server components: *serviceConsumers* that use functionality that is exposed by a *serviceProviders*. Service providers are stateless. This is an important aspect of services in comparison with components. It is one of the *service-oriented principles* stated in [11]. However, as Erl states,

services should be stateless to a *certain extent* (cf. [11, p. 308]) as statelessness is not always possible. This is especially true if services are used to change data in back-end systems. The service instance that performs an action should, however, not be important. Hence, the services themselves should be stateless in the sense that they do not preserve a conversational state. Their operations do not have to (but should) be idempotent, though.
Service consumers and service providers are connected by *serviceCalls*. Such a connection implies a *type* a service needs to have. There is, however, no constraint from the architectural style point-of-view that a service consumer needs to be capable of dealing with certain types of providers.
Additionally, call-back facilities are introduced. These facilities may also exist in client-server architectures but are crucial for SOAs. Hence, as a constraint of a *serviceCall*, asynchronous calls require a call-back connection from the provider to the appropriate consumer.
An additional constraint on connections between service providers and consumers is that all connections need to rely on the same (arbitrary) protocol. This principle facilitates exchangeability of services and eases recomposability of service aggregations.

On top of this basic structure of the service-oriented style, there are other important components. One of these components is the *ServiceRegistry*. A service registry allows for dynamic lookup of service references during runtime of a system. Service consumers may use this feature in order to determine the actual agent upon which a service provider is deployed. This lookup is usually based on the syntactic description of the service provider's interface. However, more recent approaches extend the idea towards semantic service lookups (eg. [14]). Here, enriched lookups are used in order to lookup *matching* services. From an architectural style point of view such approaches do not have an impact.
ServiceAggregation is a component that acts both as a service consumer and as a service provider. It uses other service providers to, in turn, expose the aggregated functionality as a service provider. Service aggregators, in contrast to basic service providers, can have a state and do not need to by stateless. From the design point of view, service aggregations do offer more problem-oriented functionality to other service consumers. In parallel, aggregators act as mediators. The *mediator* pattern was introduced by the Gang of Four in [37] as a pattern for object-oriented systems. The essence of this pattern is also true for service orientation. This is the notion that the interaction among a set of objects or services are encapsulated. This is why the set of services that are encapsulated by an aggregator/mediator can change their interaction independent of other services. As a consequence, aggregators promote loose coupling.
A specialization of service aggregation is service orchestration. A service orchestration aggregates basic service providers following a certain process or workflow. That process is a set of steps (and branches) that describes the way service providers are aggregated. Every step of such a process must correspond to a service provider that is aggregated to a more specialized service provider (cf. [11, pp. 200-207]).

Listing 3: Structure and Constraints of the Service-Oriented Style (Part II)

```
[...]
COMPONENT Type ServiceProvider EXTENDS Service, clientServer.Server WITH {
    PORT receiveCall : ServiceCallIn;
    PORT replyToCall : ServiceCallOut; // for asynchronous call-back
    PORT register : ServiceCallOut;      // for registering with a registry
    PROPERTY callBack : ServiceReference;
    PROPERTY operations : SET;
    RULE stateless = INVARIANT hasState == false;   // statelessness
    // functionality is exposed by operations:
```

```
    RULE exposeAll = INVARIANT FORALL op IN operations |
                                EXISTS f IN functionality |
                                    OP == f;
  }
  COMPONENT Type ServiceConsumer EXTENDS Service, clientServer.Client WITH {
    PORT serviceCall : ServiceCallOut;
    PORT callBackPort: ServiceCallIn; // for asynchronous call-back
    PORT serviceLookup : ServiceCallOut; // for registering with a registry
  }
  COMPONENT Type ServiceRegistry EXTENDS ServiceProvider WITH {
    PORT receiveRegistration : ServiceCallIn; // No call-back as considered sync.
    PORT lookup : ServiceCallIn;   // No call-back as considered sync.
  }
  COMPONENT Type ServiceAggregation EXTENDS ServiceProvider, ServiceConsumer WITH {
    PROPERTY changableState : SET;
    // Abbv.: interfaces that are aggreagted:
    PROPERTY aggregatedServices : SET {type:TypedInterface};
    RULE aggregateExistent = INVARIANT FORALL call IN aggregatedServices |
            EXISTS p : ServiceProvider in Self.Components,
                   con : ServiceCall in Self.Connectors |
            attached(p, con) AND attached (call, con);
    RULE canHaveState = INVARIANT hasState == true OR hasState == false;
  }
  COMPONENT Type ServiceOrchestration EXTENDS ServiceAggregation WITH {
    PROPERTY process : Process;
    RULE hasService = INVARIANT FORALL step IN process |
                                EXISTS service IN aggregatedServices |
                      step.type == service.receiveCall.typedInterface;
  }
  CONNECTOR Type ServiceCall EXTENDS clientServer.Rpc WITH {
     ROLE caller : ServiceConsumerRole;
     ROLE callee : ServiceProviderRole;
     // for every asnc. call there must be a call-back port
     RULE callBack = INVARIANT FORALL con:ServiceCall in Self.Connectors |
              con.asynchronous == true ->
              EXISTS callback : ServiceCall in Self.Connectors |
              callback.caller == con.callee;
     // all protocols of a system must be the same
     RULE commonProtocol = INVARIANT FORALL c, d in Self.Connectors |
              c != d ->
              c.protocol == d.protocol;
  }
  ROLE Type ServiceConsumerRole;
  ROLE Type ServiceProviderRole;
  PORT Type ServiceCallIn EXTENDS clientServer.RPCPort WITH  {
       PROPERTY typedInterface : InterfaceType;
  }
  PORT Type ServiceCallOut EXTENDS clientServer.RPCPort WITH  {}
  PORT Type Registration EXTENDS ServiceCallOut WITH {
       PROPERTY serviceReference : ServiceReference;
  }
  PROPERTY Type InterfaceType = SEQUENCE <dataTypes . interfaceName>;

  PROPERTY Type ServiceReference EXTENDS Address WITH {
       interface : InterfaceType;
  }
  PROPERTY Type Process = SET {step};
}
```

As described by Garlan et al. in [38], styles expressed using ACME can also be described using *OMG Unified Modeling Language* (UML) (cf. [39]) models. This will be the way architectural sketches will be described throughout this thesis. In order to sketch out certain architectural concepts, the UML-components shown in figure 2 will be used. The components shown are a simplification of the ACME language constructs and are solely meant for illustration purposes.

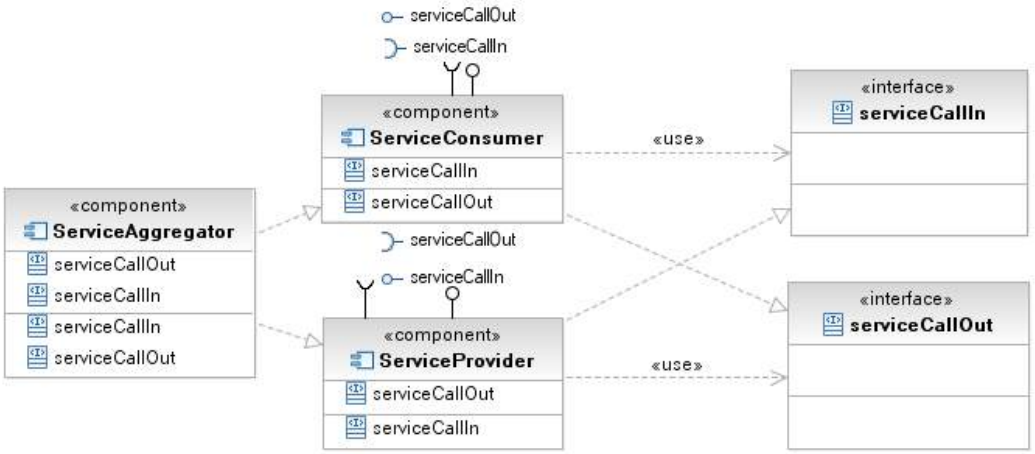


Figure 2: Some Elements of the ACME Definition Represented as UML Components

The definition of the service-oriented style can only be a first approach to capture what an SOA is, what it means, how to apply it and how to use it. This is due to the fact that the definition given above could be true for component-oriented systems as well. The difference between SO and component orientation is the way *how* services are built and composed — not the fact that they can be built and composed.
Analyzing the eight service-oriented principles given by Erl (cf. [11, p. 37]), only four can be fully captured by the definition of the architectural style. These are *composability*, *statelessness*, *service contract* and *discoverability*.
The other four principles are *loose coupling*, *autonomy*, *abstraction* and *reusability*. These principles, that are more *soft* design principles compared to the style elements formalized by the architectural style, are discussed in more detail next.
"Loose coupling is a condition wherein a service [consumer] acquires knowledge of another service [provider] while still remaining independent of that service [provider]. Loose coupling is achieved through the use of service contracts that allow services to interact within predefined parameters" [11, p. 297]. This definition refers to the fact that a service consumer solely relies on functionality that is provided elsewhere by a defined service provider rather than on a specific service provider. This principle is important to make services composable since the assumptions about the services are minimized. In turn, composing services by the means of aggregator promotes loose coupling, too. This is because aggregators reduce the dependency of a service consumer to a dependency on the respective service aggregator(s) rather than on all aggregated service providers.
When referring to loose coupling, it is shown that service consumers depend on contracts rather than on service providers. The existence of service contracts is part of the style definition. This is achieved by introducing typed interfaces and references to service providers that are deployed on agents. The way these contracts are "design[ed] is extremely important" [11, p. 295] — the *how* is more important than the fact that it exists.
"Autonomy requires that the range of logic exposed by a service exists within an explicit

boundary. This allows the service to execute self-governance of all its processing. It also eliminates dependencies on other services, which frees a service from ties that could inhibit its deployment and evolution [...] Service autonomy is a primary consideration when deciding how application logic should be divided up into services and which operations should be grouped together within a service context. Deferring the location of business rules is one way to strengthen autonomy and keep services more generic. Processes generally assume this role [...]" [11, pp. 303f.]. From the point of view of style, only the required preconditions can be established by introducing service providers and service orchestrations. How functionality is assigned to (basic) service providers and service orchestrations is a design decision that is not covered by the style definition, though. It is part of a design approach for a composite application.

"There is no limit to the amount of logic a service can represent [...] Operation granularity is therefore a primary design consideration that is directly related to the range and nature of functionality being exposed by the service [...] Operations [...] collectively abstract the underlying logic. Services simply act as containers for these operations" [11, pp. 298f.]. Erl's explanation of abstraction by service design indicates that the common principle of abstraction in software engineering is also a crucial part of an SOA. This style supports abstraction as interface-level abstraction by hiding underlying processing details of service providers' realizations from service consumers (cf. [11, p. 299]). In addition, services are considered as business relevant tasks that support the achievement of business goals (cf. [5]). This way of abstracting processing details to business-related tasks is also beyond the definition of style.

"Service orientation encourages reuse in all services [...] By applying design standards that make each service potentially reusable, the chances of being able to accommodate future requirements with less development effort are increased" [11, p. 292]. This principle is neither new nor specific to SO. However, that services can be "reused everywhere and by anybody" [5, p. 2] is a very well noted principle. As stated by Erl, the reuse only comes with the way services are designed and are again not part of the style definition.

These latter four service-oriented principles often gain most of the attention that is given to this architectural style. This might be caused by disenchantment among users regarding the level of support COTS and home-developed software reach with regards to actual business needs. It is because of this that the biggest part of concerns about an SOA relate to business issue-motivated service design.

There is, however, one major trade-off to be dealt with when approaching the way SO is described and observed. This is that SOA obviously affects the perception of how service designers abstract from technical issues and focus on business issues. The design of services is what is perceived as a major benefit. On the other hand service orientation is, as discussed above, a principle of integrating legacy systems and reusing their functionality. Designing how legacy systems should be built and (re-)used is obviously not influenced ex-post. This is why this basic definition of the service-oriented architectural style is not sufficient in the context of heterogeneous application landscapes. In order to realize all principles of an SOA here, a more detailed and applicable description of how heterogeneity and consistency can be addressed is required. This will be done by the reference architecture that is introduced in chapter 5. It describes a blueprint for composite applications and emphasizes the integration of heterogeneous landscapes and business process-centered control centralization while considering the identified benefits and trade-offs of chapter 3. In order to apply these mechanisms and to support designers to stick with the described design principles, a design methodology for service-oriented systems is introduced in chapter 6.

2.4 Platform Requirements for the Service-Oriented Architectural Style

It is not possible to describe one single platform on which composite applications can be executed. As outlined above, distribution is an inherent characteristic of an SOA. Therefore, there is little limitation to the platform that can be used. Every agent in a service-oriented landscape could be realized using different technologies.
There is, however, the need to use a central platform for service orchestration if the control of a system should be centralized. This agent hosting the service orchestration (or aggregation) component uses distributed services that are often provisioned by another central element – the so-called Enterprise Service Bus (ESB) (cf. e.g. [5], [40]). An ESB is often considered a mediator between service consumers and service providers (cf. [41], [42]). The benefit of introducing such a mediator into the architecture is that the complexity of a distributed system becomes better manageable (cf. [43, p. 38]). An ESB is, however, not part of the basic service-oriented architecture style, while an orchestration platform is.

Composite applications are applications or services that aggregate service providers to more specific functionality (cf. [44] or [5]). In this thesis, composite applications are understood as applications that aggregate basic service providers and expose their functionality with the notion of a user interface. Whether a composite *service* underlies such an application is not defined. In order to expose such an user interface we also consider a web-based portal as a part of an SOA platform.[2]

A portal and an orchestration platform would allow for the realization of composite applications that are built following the service-oriented style.
This description of platform requirements is less restrictive than the description of Tsai et al. (cf. [40]). The architecture classification given in [40] is, being a sub-set, compatible with the point of view taken above, though. [40] classifies SOA platforms by four dimensions: *structure*, *runtime re-composition capability*, *fault-tolerance* and *system engineering support*.
The structure of a composite application running on a platform can either be static (S) or dynamic (D). 'Static' refers to the fact that all components of a composite application are fixed and known before runtime. A dynamic structure of a composite application indicates that parts (esp. services) of the respective application can by dynamically recombined.
A platform can offer means for re-composing composite applications. This feature might either be supported (R) or it may be not supported (N). If the feature is supported, existing services can be replaced at runtime with other services. This replacement is not achieved by changing aggregations. Furthermore, this feature refers to the dynamic lookup facilities coming along with service registries.
The fault-tolerance capability of a platform can either be FN (no fault-tolerance), FB (fault tolerant communication backbone; ESB) or FC (fault-tolerant control service). Unfortunately, the authors do not specify the exact semantics of FB and FC. FC seems to refer to a composite application that can deal in the central control instance with faults of any kind. FB seems to refer to an ESB that ensures reliable communication and buffering.
System engineering support refers to the degree an SOA platform supports the development of composite applications. SY thereby refers to an SOA platform that supports system engineering. SN to the opposite case.

[2] The user interface that is realized with a portal is considered as a service provider.

In order to categorize different platforms, single dimensions can have arbitrary characteristics. This is indicated by “XX”.

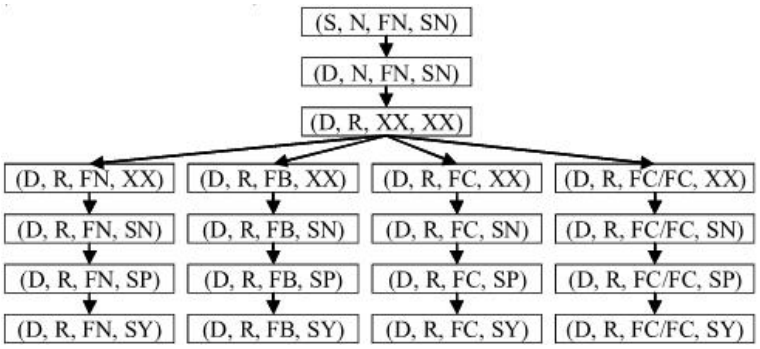


Figure 3: SO Architecture Evaluation Roadmap [40]

Figure 3 shows a tree from the most basic service-oriented architectural style (S, N, FN, SN) to more sophisticated styles as they are all described in [40]. This evaluation roadmap can be considered as the continuation of the style definition. This is because it adds features like fault-tolerance and provides an ordered set of constraints that are applied to the service-oriented style in addition to the given definition.

In order to realize a composite application according to the definition of the service-oriented architectural style as it was provided in this chapter, using a platform (D, R, XX, XX) is the minimal requirement. The reason thereof is that the given style definition does focus on design principles. It does not incorporate principles for fault tolerance and the support of building composite applications.
This means that the given definition of the service-oriented style would not allow for failure tolerant applications. As reliability is crucial and its absence can prohibit the application of service-oriented principles in the context of commercial organizations, the definition of a refined service-oriented style as it is provided by the reference architecture of chapter 5 is required to bridge this gap.

3 Assessing the Application and Applicability of SOA

Everything that can be counted does not necessarily count; everything that counts can not necessarily be counted

A. Einstein [45]

When discussing service-oriented architectures in the context of a company like BASF IT Services, it is crucial to discuss the advantages and disadvantages this architectural style can provide for the company. This does not necessarily involve the discussion of economic impacts such as saving potentials or additional turnover. If a company that is both an IT service provider and an expert in the needs of industry customers, discussions about an architectural style can also focus on architectural benefits.

In order to identify these architectural benefits, we will analyze what was identified in chapter 2 to distinguish service orientation from component orientation or the client-server style. Grouping these aspects within a structure of quality characteristics will help to identify what overall quality aspects the service-oriented style can contribute to. For each of the identified categories it is then possible to define (basic) measures and more complex metrics (which might be defined using the basic measures) that indicate the extent to which a design supports the respective quality aspect[3].
When defining metrics, emphasis is placed on eased applicability of the respective metrics. This is important so as to ensure that the metrics are applied in actual projects. The drawback to this approach is, of course, that the metrics can at times oversimplify given circumstances or produce incorrect conclusions due to underlying assumptions not being true for a given use case. This is why several metrics are defined in order to generate overall conclusions that are based on several indicators. Only through a frequent application of such metrics can experience in interpreting them be gained.

Defining the overall quality aspects of service orientation will also help outline a framework that can support the assessment of given requirements in terms of an advantageous application of the service-oriented style *prior* to its application. Based on this qualitative framework companies are able to then assess whether the application of the service-oriented style might be advantageous for a given requirement.

3.1 Potential Benefits and Trade-Offs of SOA

Quality is a nonfunctional aspect of software engineering. Therefore, any given software can reach "a certain quality". A *quality attribute* is a non-functional characteristic that represents the degree to which a given software system possesses a desired combination of attributes (cf. [48]). In order to best analyze quality attributes, it is necessary to group these attributes into categories of qualities. The ISO standard 9126 provides six so-called *characteristics* with according sub-categories that refer to quality aspects of software (cf. [49]).
The following list describes, on a qualitative level, how service-oriented principles could improve the degree to which software fulfills the actual quality characteristic.

[3]The difference between a 'measure' and a 'metric' is that a measure confers to an extent a system possesses a certain attribute (e.g., measured as a simple count) whereas a metric describes a quantitative measure of the degree to which a system possesses an attribute (cf. [46] and [47]).

- **Maintainability**. "Maintainability is the capability of the software product to be modified. Modifications may include corrections, improvements or adoptions of the software to changes in environment, and in requirements and functional specification" [49]. Lee and Lee have shown in a quantitative evaluation that – according to stakeholders of software development projects – maintainability is the most important quality characteristic when it comes to component-based software development (cf. [50]).
  In [51] Bengtsson et al. narrow the definition of maintainability down to ***modifiability*** by excluding bug-fixing and related activities. They define modifiability of a software system as "[...] the ease with which it can be modified to changes in the environment, requirements or functional specification" [51, p. 2].
  As stated before, SO aims at centralizing control over loosely coupled, stateless and autonomous services. The place where required changes need to be adopted is in such an architecture easier to locate than in a monolithic application. In particular, the explicit control model combined with rarely interacting basic components eases the application of changes to a software system. This is because changes will potentially affect the few components of a system that are in an obvious (aggregation-) relation with each other.
  This is why modifiability is not only an important aspect for stakeholders; it is also a characteristic to which service-oriented principles could contribute.

- **Functionality**. A software has to deliver certain functionalities to its users. Service orientation, as a "distinct approach to separate concerns" [11, p. 32], can support software creators as it allows them to focus on functional aspects while realizing requirements. That said, as software is considered to deliver the required functionality (and here we focus on non-functional attributes), service orientation can solely support *how fast* the functional requirements are met. This is, in turn, a non-functional aspect of software engineering. We consider this aspect to be clearly related to the characteristic of modifiability, as it describes how easy requirements can be realized.

- **Reliability**. Two main characteristics of reliability are the fault tolerance of a system and how it can recover from failures. The latter is referred to as recoverability. Considering fault tolerance, it is important to define "failure". Christian gave a classification of failures in [52]. Failures that we consider as specific sources of issues are failures that do not concern the correctness of computations. Further, we consider *omission failure* and the related *crash failure* as well as the *timing failure* as crucial. This is because the inherent distribution of a service-oriented architecture introduces many *depends on* relations among the components (cf. [52]). Failures to single services might render complete applications unavailable. Additionally, physical distribution of components might lead to timing failures. Hence, when talking about failure-tolerance the service-oriented principle might lead to a decreased quality in terms of failure tolerance and special attention needs to be paid.
  The same is true for recoverability. For a distributed system, recovering from failures might impose more effort than it is true for monolithic systems. In contrast to other distributed systems, service-oriented applications rely on stateless services. Hence, recoverability might be easier to achieve.
  In sum, reliability becomes more endangered than improved by applying service orientation. In conducting a trade-off analysis, this should be balanced with the potential benefits.

- **Usability**. ISO 9126 defines usability as "a set of attributes that bear on the effort needed for use and on the individual assessment of such use, by a stated or implied set of users [...and...] the capability of the software product to be understood, learned and liked by the user, when used under specified conditions" [49][4]. Considering processes rather than products as assets to be utilized, service-oriented applications can improve the usability of *processes* as described by the standard. This is because composite applications put a single web-based user interface on top of a user interface-independent implementation of business processes. Hence, a user does not necessarily need to know which systems are involved in an actual business process and how they are integrated with each other. Looking at today's reality, clerks often need to know under which conditions (especially time) and among which systems information is exchanged and how business events should be handled accordingly. Eliminating burdens on clerks could decrease required learning efforts and result in higher user acceptance. Hence, SO can – like other architectural styles – increase usability.[5]

- **Efficiency**. The term *efficiency* refers to the ratio between the performance a software system can reach and the resources it uses for reaching that performance level (cf. [49]). Due to services being discoverable, service-oriented applications can use scheduling mechanisms for resource allocation without including this concern in the composite application. Here, however, this principle is considered a specialization of service orientation. Architectures like the Open Grid Services Architecture (OGSA) [54] aim to define means for job management and resource management for web services. This is beyond the service-oriented architectural style and therefore considered beyond the scope of this paper. Hence, it is assumed that efficiency is a quality characteristic that is not concerned by (genuine) service orientation.

- **Portability**. Portability refers to the quality of a software to be easily transferred between environments (cf. [49]). As SO applications are not monolithic applications, portability does not play a major role when analyzing them. Portability of the single components is not affected by the application of service orientation. Portability is therefore not considered a quality characteristic that is affected by the service-oriented architectural style.

To summarize, it can be assumed that the service-oriented architectural style influences the quality of software systems in terms of modifiability, reliability and usability. The major impact is assumed to be on the modifiability of systems. This is why the following analysis of quality characteristics emphasizes modifiability before examining reliability and usability.

3.2 Assessing Design Quality

This section proposes and describes metrics and measures that might be used to assess the quality of a system's design. Measures are simple mechanisms that provide values that can be computed by more complex metrics. The values that can be computed using the

[4] cited from [53, p. 5].

[5] In contrast to other architectural styles that can also address the multi-interface issue, service orientation could address this issue by additionally improving e.g. modifiability.

presented means are considered to provide an estimation of the extent to which a software system might fulfill the according quality characteristic. In presenting these means, we focus on characteristics specific to the service-oriented architectural style. This section does not aim to fully describe the assessment of software design.
Having defined what the term *quality* means in the context of software systems, the definition of the term *design* is still needed. Brathall and Runeson defined in [55] a taxonomy of properties for software architectures (TOPSA). [55] differentiates three orthogonal dimensions of software architecture: abstraction level, dynamism and aggregation level. These three dimensions form the so-called TOPSA space (cf. [55, p. 2]). The abstraction level dimension can have from two up to an arbitrary amount of abstraction levels. The actual code is referred to as the "realizational" level. Each more abstract level is referred to as "conceptional". [55] suggests the abstraction levels "{*FunctionalUnit*, *ConcurrentStateMachine*,*SoftwareProcess*}" [55, p. 4]. For the aggregation levels the following fragmentation is proposed: "{*System*, *Subsystem*,*Component*, *Class*}" [55, p. 4]. The fragmentation sets for both dimensions are on the ordinal scale. Using these definitions, we can define that a design that is assessed at an arbitrary point in time[6] τ by the means defined below has to be (at least) $\overline{\texttt{TOPSA}(\tau)} \leq [\texttt{FunctionalUnit}, \texttt{Static}, \texttt{Component}]$[7] or "closer" to the actual executable implementation. A software architecture of this quality we refer to as the *design* of the system that is represented at the lowest level in the TOPSA space. To summarize these definitions: assessing service-oriented quality of a design means applying the means given below to a design of a software system.

In order to consider the fact that systems that are built using a service-oriented style are built by (partially) re-using COTS-based services, all metrics that are defined for the assessment of service-oriented designs need to deal with a certain lack of knowledge about the single services. Only by treating the single services as black-boxes and analyzing them from an outside point of view, will the metrics be applicable in the context of COTS integration. An additional consideration is the effort that needs to be put into calculating the metrics: if it relies on source code (that is often not accessible) or is just too complex, the metrics would just not be used in the context of a commercial organization.

All measures and metrics that are presented below are first introduced by describing their respective purpose. After presenting the applicable formula, the mechanism of the formula and the according value range are introduced. If applicable, the actual metric is also analyzed in terms of satisfying some basic "desiderata" – properties a measure or metric of a certain class should have.
Beyond indicating the value range of a metric and providing rules for the interpretation of these values, it is desirable to give fixed thresholds. As described in [56], Boolean discriminant functions are supportive when assessing design in terms of fulfilling quality aspects. At this point in time the empirical base for analyzing existent systems is nonexistent. This is because building (real) composite applications is still a long-term project that is quite expensive – and today there are hardly any existing applications that could be analyzed. Additionally, fixing thresholds for metrics requires the categorization of actual software systems. These categories indicate whether the respective system fulfills a certain quality characteristic. For some quality characteristics, such as modifiability

[6]This includes the important notion that assessing a system is always an action at a certain point in time. Hence, each presented metric (implicitly) includes as a variable the time when they are measured. This becomes especially important as soon as re-usability and re-use are considered.

[7]While using definition from [55], a *component* is in the service-oriented style the actual (outside) description of *service*.

or usability, this can only be a subjective and qualitative fragmentation. Deducing fixed values based on a purely subjective categorization is also questionable. For this reason, this chapter only describes rules for the interpretation of values that are derived from overall design metrics. The metrics will then, in turn, be applied to the case study that is presented in chapter 8. In order to get an understanding of the reasonable thresholds for the presented metrics, this chapter should also be considered.

3.2.1 Assessing Modifiability

Analyzing prior work, measuring and assessing the modifiability of software systems is basically measuring the complexity of the system (cf. [55], [57]). Usually this solely involves measuring complexity as a directly composable attribute of the single components.
In order to measure the complexity of a service-oriented system we propose the following measures and metrics that are complementary to other complexity measures.
First, we introduce four very simple measures for measuring the *size* of a system. In [58] Briand et al. have described six properties size measures and metrics for systems should satisfy. These "desiderata" are introduced while discussing the first relevant size measure. After introducing four size measures, the concept of object coupling is transferred to services. Based on these basic measures, a new complexity metric SSC is introduced and evaluated.[8]
Afterwards, another aspect of maintainability is discussed: how well a system deals with its complexity.

A modular (finite) system is described in the following as a tuple $\Omega =< E, R, M >$.
E denotes the elements of the system Ω. R denotes the relations among those elements. M is a collection of modules of Ω in a way that (cf. [58, p. 70]):

$$\forall e \in E(\exists m \in M(m \ =< E_m, R_m > \ \wedge e \in E_m)) \wedge$$
$$\forall m_i, m_j \in M(m_i \ =< E_{mi}, R_{mi} > \ \wedge m_j \ =< E_{mj}, R_{mj} > \ \wedge E_{mi} \cap E_{mj} \ = \ \emptyset)$$

$OuterR(m)$ defines the inter-module relations a given module m is involved in.

For a service-oriented system this can be interpreted as follows: methods of services are considered to be the elements of a system (E). Relations among these methods, such as calling relations (irrespective of any communication semantics), are the relations of a system (R). Services are considered as modules (M) that group methods together.
$\Omega.\Psi$ is then defined as the set of all service types in a given system Ω: $\Omega.\Psi \subseteq E \times E$.
$\Omega.R$ denotes the calling relations among the services of Ω: $\Omega.R \subseteq \Omega.\Psi \times \Omega.\Psi$.

The following size measures are described using these conventions.

- **Number of Services (NS)** $NS(\Omega) = |\Omega.\Psi|$.

  Mechanism NS is a simple count of all services in a system. It is a *size* measure that only considers the pure count of services in a system.

[8] According to [59], the coupling of a system is an indicator for its complexity. This becomes also visible by analyzing the metric properties in [58]. There, coupling and complexity metrics only differ in terms of the *symmetry* property that is defined for complexity but not for coupling metrics. The concept of symmetry addresses the conventions that are used to describe a system. As such conventions are not included here, coupling is considered a valid measure for complexity.

Value range NS is limited to the range of $[0, +\infty[$

Discussion NS is a simple first measure of a system's complexity: the more services are meant to be used in a system, the more complex (and less modifiable) the system might be. In its simplicity, NS is a basic measure that can be used within other measures and metrics – for complexity measures and metrics for other characteristics. It is also suitable to weight values of more complex metrics as it provides the context in which these metrics should be analyzed.

The following properties are desirable for size measures and metrics. They are introduced by evaluating NS and are subsequently used to analyze the other size measures as well.

`Size.I` (cf. [58]) a size measure or metric should satisfy is *non-negativity*. It demands that a size measure $\chi(\Omega)$ is not negative: $\chi(\Omega) \geq 0$ (cf. [58, p. 71]). NS satisfies this property as it is a count of elements.

`Size.II` Is the null value property: $\Omega = \emptyset \Rightarrow \chi(\Omega) = 0$ (cf. [58, p. 71]). As $NS(\Omega) = |\Omega.\Psi|$, $NS(\emptyset) = 0$ holds true and NS satisfies `Size.II`.

`Size.III` Is the module additivity property: $s_1 \subseteq \Omega$ and $s_2 \subseteq \Omega$ and $\Omega.\Psi = \Psi_{s1} \cup \Psi_{s2}$ and $\Psi_{s1} \cap \Psi_{s2} = \emptyset \Rightarrow \chi(\Omega) = \chi(s_1) + \chi(s_2)$ (cf. [58, p. 71]). As $NS(\Omega) = |\Omega.\Psi|$, the compound can be calculated as $NS(s_1 \cup s_2) = |\Psi_{s1}| + |\Psi_{s2}|$. Then, $(\Psi_{s1} \cap \Psi_{s2} = \emptyset) \Rightarrow (|\Psi_{s1}| + |\Psi_{s2}| = |\Psi_{s1} \cup \Psi_{s2}|)$. Hence, NS satisfies `Size.III`.

`Size.IV` demands that the size of a system Ω can be determined by the knowledge of the size of its disjoint parts $s_e =< s, R_e >$[9]: $\chi(\Omega) = \sum_{s \in \Omega.\Psi} \chi(s_e)$ (cf. [58, p. 71]). `Size.IV` is a consequence of `Size.III` (cf. [58, p. 71]). Hence, NS satisfies `Size.IV`.

`Size.V` is the monotonicity property: $\Omega' =< E', \Omega'.R' >$ and $\Omega'' =< E'', \Omega''.R'' >$ and $\Omega' \subseteq \Omega'' \Rightarrow \chi(\Omega') \leq \chi(\Omega'')$. Monotonicity follows from the properties `Size.I` - `Size.III` (cf. [58, p. 71]). Hence, NS satisfies `Size.V`.

`Size.VI` "From the above properties, `Size.I` - `Size.III`, it follows that the size of a system $[\Omega =< E, \Omega.R >]$ is not greater than the sum of the size of any pair of its modules. [...]" [58, p. 71]. Hence, NS also satisfies `Size.VI`.

Using these properties, more size measures for service oriented systems can be described and analyzed:

- **Service Consumers (SC)** $\Omega.C$ is the set of all service types that consume (operations of) service providers in a system Ω. $SC(\Omega) = |\Omega.C|$; $\Omega.C \subseteq \Omega.\Psi$.

  Mechanism SC is a simple count of all service consumers in a system.

  Value range SC is limited to the range of $[0, +\infty[$

  Discussion As NS, SC is a basic measure that is used in more complex metrics.

  $\Omega.C$ is defined as $\Omega.C \subseteq \Omega.\Psi$. Therefore it exists a set $\Omega.S'$ consisting of non-service consumers in a form $\Omega.\Psi = \Omega.C \cup \Omega.S'$ and $\Omega.C \cap \Omega.S' = \emptyset$. Denoting a system $S' =< \Omega.\Psi \setminus \Omega.S', \Omega.R \setminus \Omega.R' >$ that equals a system Ω in all elements, besides

[9] [58] does not use the notion of a modular system. Despite this, the notion of a system is equivalent to the above definition.

the elements $\Omega.S'$ and their relations among themselves and other elements $\Omega.R'$ as $\Omega \setminus \Omega.S'$ we can state that $SC(\Omega) = NS(\Omega \setminus \Omega.S')$. As $|\Omega.\Psi \setminus \Omega.S'| \geq 0$ holds true and deducting a positive integer from NS is a linear transformation, it can be concluded that the properties `Size.I` - `Size.VI` are satisfied by SC.

- **Service Providers (SP)** $\Omega.P$ is the set of all service types in a system Ω that expose operations that are consumed by service consumers. $SP(\Omega) = |\Omega.P|;\ \Omega.P \subset \Omega.\Psi$

  Mechanism SP is a simple count of all service providers in a system.

  Value range SP is limited to the range of $[0, +\infty[$

  Discussion As NS, SP is a basic measure that is used in more complex metrics. $\Omega.P$ is defined as $\Omega.P \subseteq \Omega.\Psi$. Therefore it exists a set $\Omega.S'$ with non-service providers in a form $\Omega.\Psi = \Omega.P \cup \Omega.S'$ and $\Omega.P \cap \Omega.S' = \emptyset$. Hence, $SP(\Omega) = NS(\Omega \setminus \Omega.S')$. As $|\Omega.\Psi \setminus \Omega.S'| \geq 0$ holds true and deducting a positive integer from NS is a linear transformation, it can be concluded that the properties `Size.I` - `Size.VI` are satisfied by SP.

- **Service Aggregators (SA)** $\Omega.A$ is the set of all service types in a system Ω that both act as service provider and service consumer. $SA(\Omega) = |\Omega.A|;\ \Omega.A \subseteq \Omega.C, \Omega.A \subseteq \Omega.P, \Omega.A = \Omega.C \cap \Omega.P$.

  Mechanism SA is a simple count of all service aggregators in a system. Service aggregators are – as defined by the service-oriented architectural style – sub-types of both service providers and service consumers.

  Value range SA is limited to the range of $[0, +\infty[$

  Discussion As NS, SA is a basic measure that is used in more complex metrics. The fact that the sets of consumers and providers overlap in the set of all aggregators is an interesting mechanism that is used in some more complex metrics. The SA value is the most interesting value of the basic measures as it slightly indicates how a systems complexity is made up and addressed.

  As the set of a system's aggregators is defined as $\Omega.A = \Omega.C \cap \Omega.P$ and $\Omega.\Psi = \Omega.C \cup \Omega.P$, SA can be described as $SA(\Omega) = SP(\Omega) + SC(\Omega) - NS(\Omega.P \setminus \Omega.A) - NS(\Omega.C \setminus \Omega.A)$. As $SP(\Omega) \geq NS(\Omega.P \setminus \Omega.A)$ and $SC(\Omega) \geq NS(\Omega.C \setminus \Omega.A)$, NS is linear transformation of NS that is always positive. Hence, SA satisfies the properties `Size.I` - `Size.VI`.

Next to these basic measures for the *size* of a system, the following coupling measures and metrics are also applicable for measuring a system's complexity and hence indicating its modifiability.

- **Coupling of Service (cos)** Two components are "coupled if and only if at least one of them acts upon the other" [60, p. 4]. As a service-oriented principle, services should be "loosely coupled" (cf. [11]). As stated in listing 3, services expose solely operations for interaction with other services. Hence, the *Coupling Between Object Classes* metric (CBO) as defined in [60] is applicable for services as well. This metric is defined as "a count of the number of other classes to which it is coupled" [60, p. 11]. Transferred to services that means that cos is defined as *the count of services a given service calls operations on.* cos is a function of a given service. We denote the cos of a service $s \in \Psi$ as:

$$\cos(s) = |\{\Omega.s\} \times \Omega.\Psi| \tag{1}$$

Mechanism Being a simple count, $\cos(s)$ is a basic measure for the complexity of a single service. As services encapsulate their variables, in order to calculate $\cos(s)$ it is only checked whether a service uses methods of another service. If so, the value is increased by one. The number of methods a service is actually using is not considered.

Value range $[0, +\infty[$

Discussion Still treating a service as a black-box, calculating $\cos(s)$ requires some sort of insight into the mechanisms of a given service. This will be possible to analyze as this basic need to be documented exists for COTS-based services, too. An absolute high value will indicate that the given service depends on many other services. The impact on modifiability depends on the actual class of service that is analyzed. High values for (sole) service consumers might indicate a low modifiability while high values for aggregators might indicate the opposite. Note that $cos(p) = 0$ holds true for a (sole) service provider p.
As all measures and metrics that are presented here aim to assess the overall design of a system and not of single services, this measure is sufficient as it is used in other metrics. Of course, knowledge about how many methods of a service provider a consumer is depending on is important for the design of a service. This is discussed in more detail in chapter 4.
According to the ACME UML meta-model in figure 2, table 1 shows some topologies and the corresponding cos-values.

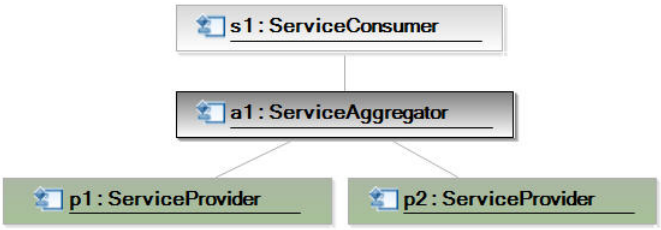
	
`cos(s1)=1;cos(p1)=0`	`cos(s1)=1;cos(a1)=2;cos(p1)=0;cos(p2)=0`

Table 1: Examples of `cos` Values

In [61] nine properties a measure or metric that measures the complexity of a software artifact should satisfy are discussed[10]. In [60] the analysis is conducted that the CBO metric satisfies five out of the six relevant properties {non-coarseness, non-uniqueness, design details, monotonicity, nonequivalence of interaction, interaction increases complexity}. Only the property *interaction increases complexity* is not satisfied (cf. [60]). In order to facilitate the application of this measure in the context of service-oriented systems, the cos-value of a service s is limited the number of possible, unidirectional connections among all services of a system Ω: $\cos(s) \leq 2 \times \binom{NS(\Omega)}{2} \mid s \in \Omega.\Psi$. Hence, the coupling between two services is limited to 2.
On the other hand this leads to the effect that merging two services s' and s'' can decrease the number of couples of the new service $s = s' \cap s''$ by more than

[10] [61] actually introduced the properties [58] applied to (modular) systems.

$|\{\Omega.s'\} \times \{\Omega.s''\}|$. Hence, cos does not satisfy the monotonicity property a coupling measure should satisfy.
Not satisfying this property can, however, be desirable under certain circumstances. This is because the count of channels among services might sometimes not be as important as the count of services another service is coupled with.
A metric that satisfies the monotonicity property is λ (see (3) below).

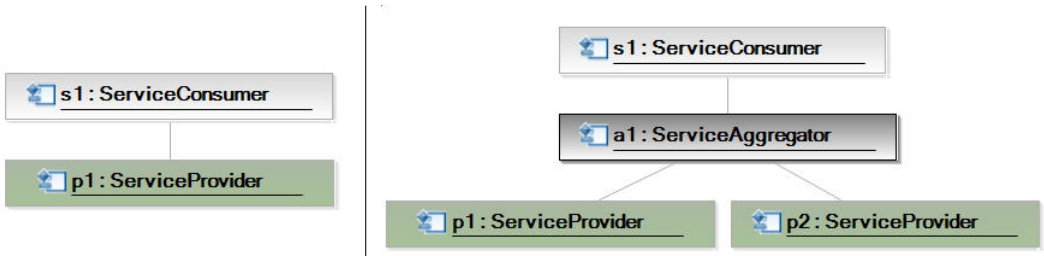
- **Coupling to Service** (cts) Being the opposite measure of cos, cts indicates how many services are actually coupled to a certain service. Obviously, cts can only be measured for service providers. This means that cts is defined as *the count of services that call operations on a given service*. cts is a function of a given service. We denote the cts of a service $s \in \Omega.\Psi$ as:

$$cts(s) = |\Omega.\Psi \times \{\Omega.s\}| \tag{2}$$

Mechanism Being a simple count, $cts(s)$ is also a basic indicator for the complexity of a single service. As services encapsulate their variables, in order to calculate $cts(s)$ it is only checked whether a service uses methods of the given service s. If so, the value is increased by one. How many methods a service actually uses is not considered.

Value range $[0, +\infty[$

Discussion An absolute high value will indicate that the given service is a crucial service that many other services depend on. The impact on modifiability is that such a service is harder to modify as changes will have a higher impact on the remaining part of the system as it would be the case for services with a lower cts-score.
Table 2 shows some topologies and the corresponding cts-values.



`cts(s1)=0;cts(p1)=1` | `cts(s1)=0;cts(a1)=1;cts(p1)=1;cts(p2)=1`

Table 2: Examples of `cts` Values

cts is the conversely value of cos. As such, the maximum cts value is limited to the same boundary as cos:
$cts(s) \leq 2 \times \binom{NS(\Omega)}{2} \mid s \in \Omega.\Psi$. Being the reciprocal of cts satisfies the same properties. These are {non-coarseness, non-uniqueness, design details, nonequivalence of interaction}.

- **Inter-Service Coupling** (λ) Let $p.\Pi$ be the set of all *receiveCall*-ports of a service provider p. The function π shall be the count of *receiveCall*-ports of a service provider: $\pi(p) = |p.\Pi|$.[11]
Let $c.\Gamma$ be the set of all *serviceCall*-ports of a service consumer. The function γ

[11] As a size measure π satisfies the properties `Size.I-Size.VI`

shall be the count of *serviceCall*-ports of a service consumer c: $\gamma(c) = |c.\Gamma|$.[12]
Let $\Omega.\Lambda$ be the set of all channels between *receiveCall*-ports and *serviceCall*-ports in a system Ω: $\Omega.\Lambda \subseteq c.\Gamma \times p.\Pi$. The function λ is then defined as the cardinality of $\Omega.\Lambda$:

$$\lambda(c,p) = |c.\Gamma \times p.\Pi| \tag{3}$$

Mechanism $\lambda(c,p)$ is the count of channels between the two services c and p. $c \in \Omega.C$ and $p \in \Omega.P$.

Value range The value range of λ is $[0, +\infty[$

Discussion λ is equivalent to the CBO metric as defined in [60]. Therefore it satisfies the same properties – including monotonicity.

- **System's Service Coupling (SSC)** SSC measures the degree of coupling in a given system Ω with regards to its modifiability. It is defined as:

$$SSC(\Omega) = \frac{\sum_{c \in \Omega.C} \cos(c)}{SC(\Omega) \times SP(\Omega)} \mid SC(\Omega), SP(\Omega) \geq 1 \tag{4}$$

Mechanism In order to indicate the overall coupling of a given system, the sum of all single cos-values of a system's service consumers is set in relation with the maximum couplings that could occur in a system if no aggregators were used at all. If service aggregators occur in a system Ω, they increase both $SP(\Omega)$ and $SC(\Omega)$. This mechanism increases the denominator and therefore decreases the value of SSC whenever service aggregators are deployed in a system. This is because aggregators are considered to help decrease the overall coupling of services in a system (cf. the discussion of the mediator pattern and aggregators in chapter 2.3).

Value range SSC is limited to the range of $[0, 1]$. As stated in the definition of the service-oriented style, a service consumer needs to be coupled with at least one service provider, 0 will never be seen for service-oriented systems, though.

Discussion The SSC metric indicates to which extent services of a system are cross-linked. The fact that – by definition of the service-oriented architectural style – services need to call each other, a SSC value of 0 is not reasonable. If a system scores a relatively high SSC value this is an indication that a lot of interaction without mediation between services takes place. As aggregators automatically decrease the SSC value, a value close to 1 will indicate that a system is hard to modify as it is very complex and not mediated. If a medium value is reached, other indicators should be considered in order to assess the modifiability. This is because systems with a certain level of functionality will need a certain level of coupling, too. Low SSC values indicate a loosely coupled system. Such systems are considered to be better modifiable than more coupled systems.
Examples for SSC-values are given in table 3.

Briand and Basili have described in [58] five properties to which a coupling metric for a modular system should comply. In order to analyze the behavior of the SSC metric more in detail, the metric will be checked against these five properties. For this sake, the above definition of a system as a triple $< E, R, M >$ and the relation $OuterR(m)$ are used.

[12] As a size measure γ satisfies the properties `Size.I-Size.VI`

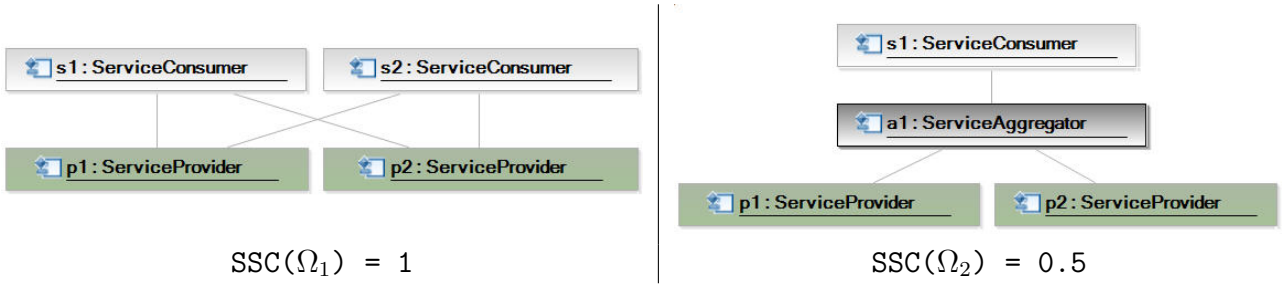


Table 3: Examples of `SSC` Values

In order to analyze the presented coupling metric SSC, we consider methods of services as the element of a system, relations among these methods as calling relations (irrespective of any communication semantics) and services as modules that group together sets of methods. This definition fulfills the given definition of element, relation, module and system.
Taking a coupling metric $\alpha(\Omega)$ (or $\alpha(m)$), the first property `Coupling.I` is non-negativity (cf. [58, p. 78]):

$$[\alpha(m) \ \geq 0 \mid \alpha(\Omega) \geq 0]$$

As neither $\cos(m)$ nor $SC(\Omega)$ nor $SP(\Omega)$ can be negative, SSC obviously satisfies `Coupling.I`.

`Coupling.II` is defined the following way (cf. [58, p. 78]):

$$\forall m \in \Omega(OuterR(m) = \emptyset) \Rightarrow \alpha(\Omega) = 0$$

`Coupling.II` defines that the coupling for a system without connections among its modules shall be zero. As $(\forall c \in \Omega \mid cos(c) = 0) \Rightarrow (SSC(\Omega) = 0)$ holds true, SSC satisfies `Coupling.II`.

`Coupling.III` is the monotonicity property. It describes that a new relation between modules does not decrease the coupling (cf. [58, p. 78]): "Let $MS' =< E, R', M' >$ and $MS'' =< E, R'', M'' >$ be two modular systems [...] such that there exist two modules $m' \in M'$, $m'' \in M''$ such that $R' - OuterR(m') = R'' - OuterR(m'')$, and $OuterR(m') \subseteq OuterR(m'')$. Then," [58, p. 78]

$$[\alpha(m') \leq \alpha(m'') \mid \alpha(MS') \leq \alpha(MS'')]$$

Measuring coupling as described by the SSC metric, $R - OuterR(m) = \emptyset$ holds true. This is because internal calling relations of a single service are not considered. As cos simply counts method calls[13], the demand that $OuterR(MS') \subseteq OuterR(MS'')$ has the effect that $\sum_{m' \in MS'} \cos(m') \leq \sum_{m'' \in MS''} \cos(m'')$.[14]
Another case to consider is the fact that an additional relation can turn a service consumer (or provider) into a service aggregator. The new relation would always increase the cos value for one service, hence increase $\sum_{m \in MS} \cos(m)$ by 1. On the other hand the new service aggregator would increase $SC(\Omega) \times SP(\Omega)$ by $SC(\Omega)$ (or $SC(\Omega)$, depending on what service is changed). As $SC(\Omega) \geq 1$ and $SP(\Omega) \geq 1$,

[13] Additionally, it simplifies it in such a way that all methods a service s calls at another service p can only contribute to the cos-value by 1.
[14] The values might be equal if another relation between two already coupled services is added.

SSC could be decreased if a channel turns a provider (or consumer) into an aggregator:

$$a = \sum_{c \in \Omega.C} \cos(c),\ b = SC(\Omega) \times SP(\Omega)$$

$$\frac{a}{b} \geq \frac{a+1}{b+SC(\Omega)}$$

$$a \times (b + SC(\Omega)) \geq b \times (a+1)$$

$$ab + a \times SC(\Omega) \geq ab + b$$

$$a \times SC(\Omega) \geq b$$

$$SC(\Omega) \geq \frac{b}{a}$$

This means, as soon as more than $\frac{SC(\Omega) \times SP(\Omega)}{\sum_c \cos(c)}$ service consumers are deployed in a system, turning a provider into an aggregator does not increase the complexity of a system. Hence, there are circumstances under which SSC does not satisfy `Coupling.III`. As the aim of the metric is to introduce the idea of service aggregators as coupling-reduction mechanism, monotonicity can not be satisfied by SSC.

`Coupling.V` is a property that describes the merging of unrelated modules in a system. It describes a (modular) system obtained by merging two non-interacting modules as being as complex as the initial system. (cf. [58, p. 79]).
"Let $MS' =< E, R, M' >$ and $MS'' =< E, R, M'' >$ be two modular systems [...] such that $M'' = M' - \{m'_1, m'_2\} \cup \{m''\}$, with $m'_1 \in M', m'_2 \in M'$, and $m'' \notin M'$, and $m'' = m'_1 \cup m'_2$. (The two modules m'_1 and m'_2 are replaced by the module m'', union of m'_1 and m'_2). If no relationships exist between the elements belonging to m'_1 and m'_2 [...], then" [58, p. 79][15]

$$[\alpha(m'_1) + \alpha(m'_2) = \alpha(m'') \mid \alpha(MS') \geq \alpha(MS'')]$$

For the context of service-oriented systems that would mean that two services with disjoint methods are merged together into one bigger service.
SSC does not satisfy `Coupling.V`. This is because the merger of two (unrelated) services could even increase the SSC value for a system (what is also a violation of `Coupling.IV` that demands that a merger of arbitrary modules should decrease or not affect the coupling value (cf. [58, p. 78f.]):
By not superseding channels with the merge of services it is possible that $\sum_{m' \in MS'} \cos(m') = \sum_{m'' \in MS''} \cos(m'')$ while $SC(\Omega) \times SP(\Omega)$ is decreased. In these cases the SSC-value is increased (as $\sum_{m' \in MS'} \cos(m') \leq SC(MS') \times SP(MS')$).
Thus, merging services only improves (decreases) the SSC value of a system, whenever (bi-directional) relations between services can be spared, too. There also exists a trade-off, in that transforming aggregators into sole providers is considered negative since it results in higher SSC values.
If SSC would be the only rule by which a system is designed, the design would lead to a decentralized and distributed system with numerous services that do not interact heavily and are mediated by aggregators.
SSC violates (under certain conditions) `Coupling.III` - `Coupling.V`. As SSC incorporates the concept of aggregators as a mechanism to decrease complexity and

[15] Literal error in source was corrected.

increase modifiability, the aim of SSC is to reward the use of aggregators. Thus, SSC supports loosely coupled services that are mediated, and are therefore easier to modify than bigger services – even if it does not satisfy all the desiderata for object-oriented coupling metrics.

In order to assess the coupling regardless of service aggregators, the metric that is presented next – SCF – should also be considered. Both metrics can indicate how heavily services of a system interact and whether mediators are used or not. Especially if $SA(\Omega)$ scores a relatively high value, other metrics (e.g., SCF and other complexity *handling* metrics that are introduced later) should also be considered.

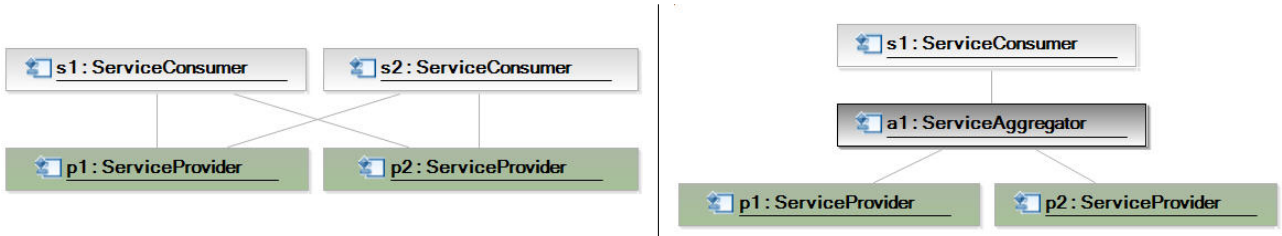
- **Service Coupling Factor (SCF)** In [59], Washizaki et al. have defined a complexity metric called Component Coupling Factor ($CCOF$). As discussed in the introduction, component orientation and service orientation are similar architectural styles. Only differences exist between some "soft" design principles (cf. chapter 2). Thus, the $CCOF$ complexity metric can also indicate the complexity of a service-oriented system. Using the notation used in this chapter, we define the Service Coupling Factor (SCF) in complete analogy with CCOF as defined in [59]:

$$SCF(\Omega) = \frac{\sum_{c \in \Omega.C} \cos(c)}{NS(\Omega)^2 - NS(\Omega)} \mid NS(\Omega) \geq 2 \tag{5}$$

  Mechanism In order to indicate the overall coupling of a given system, the sum over all single cos-values of a system's service consumers is set in relation with the maximum couplings that could occur in a system.
  In contrast to SSC it does not consider the fact that aggregators decrease the coupling of a system.

  Value range SCF is limited to the range of $[0, 1]$. As by the definition of the service-oriented style, a service consumer needs to be coupled with at least one service provider, 0 will never be seen for service-oriented systems.

  Discussion SCF is a metric designed for component-oriented systems that we apply to service-oriented systems, too. Consequently, this metric can not incorporate service-oriented principles.
  As discussed in [59], $CCOF$ (and therefore SCF, too) satisfies the properties `coupling.I - coupling.IV`. Hence, in contrast to SSC it satisfies the "merging of modules" and "monotonicity". This is because it does not include the notion of service aggregators.
  Relatively low SCF values imply a loosely coupled system while high values indicate a dense coupling in a system. Especially when used in combination with SSC, SCF can indicate how much the complexity of a system influences its modifiability. Whenever a system possesses a high coupling in terms of the SCF value while the SSC value indicates a low coupling, the system's designer obviously tries to address the high coupling by service-oriented principles. However, whether or not this is beneficial is not indicated by these two metrics. In such cases other metrics that assess the quality of aggregation should also be considered.
  Examples of SCF values are shown in table 4.



$\texttt{SCF}(\Omega_1) = 0.\overline{3}$ $\texttt{SCF}(\Omega_2) = 0.25$

Table 4: Examples of SCF Values

These presented metrics, especially SSC and SCF give an indication of a system's complexity and therefore modifiability. SSC additionally incorporates a mechanism that reflects the fact that complexity can be addressed in order to reach higher levels of modifiability. The extent to which complexity is addressed can also be measured by non-complexity metrics.

The following metrics are introduced in order to assess how well a service-oriented system addresses its complexity. As the types of metrics that are discussed below have not been described yet, there are no objective "desiderata" these metrics could be checked against.

- **System's CentraliZation (SCZ)** SCZ describes to what extent a system is centralized. Control centralization is seen as a task for service aggregators.

$$\kappa(\Omega) = 0.9 \times (SC(\Omega) - SA(\Omega)) - (SA(\Omega) - 1)^2$$

$$SCZ(\Omega) = 1 - \frac{\sum_{c \in \Omega.C} (\cos(c)) \; - \sum_{a \in \Omega.A} (\cos(a)) \; - \; \kappa(\Omega)}{\sum_{c \in \Omega.C} (\cos(c)) \; + \; (SA(\Omega) - 1)^2} \tag{6}$$

  Mechanism The basic mechanism of SCZ is to set the extent of a system's consumers' coupling less the coupling of the aggregators in relation with the overall consumers' coupling. The more the aggregators are coupled, the smaller the value gets. Hence, SCZ converges to a value of 1. In order to capture the fact that a service consumer is always coupled with one service provider, the amount of (sole) service consumers is deducted from the denominator. As the use of multiple service consumers is a slight indicator for a de-centralized system, the number of consumers is not deducted completely from the count of service consumers. As the excessive use of (i.e., more than one) service aggregators is contrary to the idea of centralization, a "punishment" for the excessive use of aggregators is also included. This is reflected by the use of the supporting function κ.

  Value range The value range of SCZ is $]0, 1[$.

  Discussion SCZ addresses the need for control centralization in a system that (re-) uses existing parts. Both, the BPIOAI approach (cf. [4]) as well as the discussion in [3] demand to centralize the control on-top of existing functionality.
  A high value of SCZ indicates that a system uses centralized components. With regards to modifiability, a higher SCZ-value is better than a lower one. Important to note is that a high value might be caused by central aggregators but can also be

caused by single, central (sole) service providers. This is why the SCZ value for a system can also be high if a system is centralized without control centralization. This can be the case if multiple service consumers use one single service provider. Even if the control is completely de-centralized, this case leads to a high SCZ value. This is acceptable since such a hub-and-spoke architecture is also easy to modify (and of course, the system has a central component).
Using multiple centralization components decreases the SCZ value. This is why service aggregators should not be deployed exhaustively. This is especially true for multi-purpose services that act both as consumer and provider without explicit control purposes. This is why the metric includes a "punishment" for the excessive use of aggregators.
A SCZ value close to 1 indicates a high degree of centralization. With regards to an optimal modifiability, such a very high centralization might not be the optimal design for any system, though. This is because a highly complex system that is controlled from one instance might lead to an over-complex, and therefore unmodifiable, central control instance. In these cases, a less centralized system might be advantageous. Hence, there is a trade-off between complexity and centralization. This is why the SCZ value of a system should be looked at with the knowledge of SSC and SCF values as well as the knowledge of the absolute counts of the different service types.
Also important to remember is that the metric is based on the assumption that an aggregator centralizes the control of a system. This might not always be the case. Whenever an application uses aggregators in order to adapt to external services, aggregators might be used extensively while the control is centralized in one component. In such cases, the SCZ value can be misleading. Examples for SCZ-values are given in table 5.

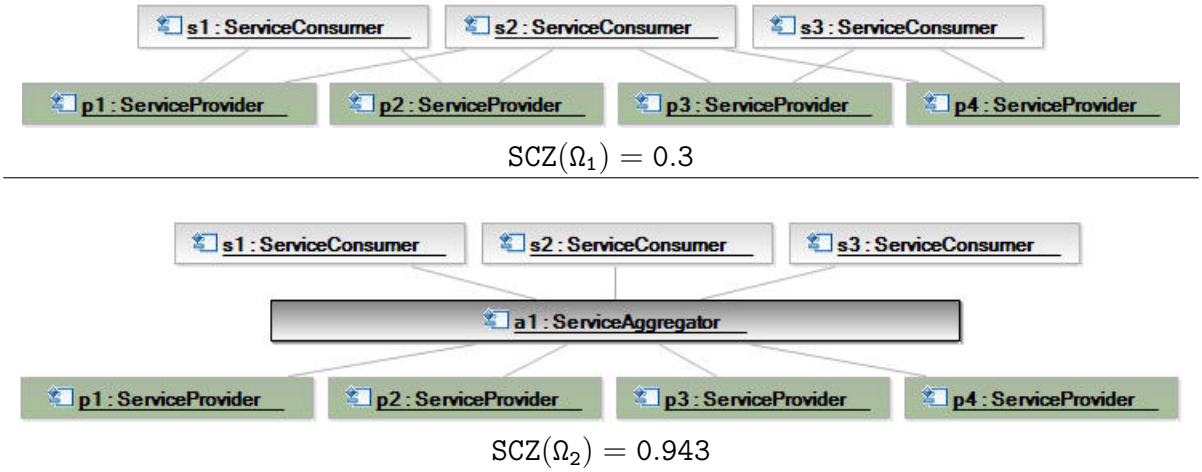


Table 5: Examples of SCZ Values

Another indicator of the degree to which a system addresses its complexity is the level of *aggregation*. Aggregation "... refers to if and to what extent a structure is made from other structures" [55, p. 2]. Measuring the degree of aggregation within a system has to analyze a system's aggregators more in detail. However, in order to allow the metric to be applicable for black-box COTS it does not analyze the internals of the aggregators.

- **Extent of Aggregation (EOA)** There are two metrics that indicate the degree

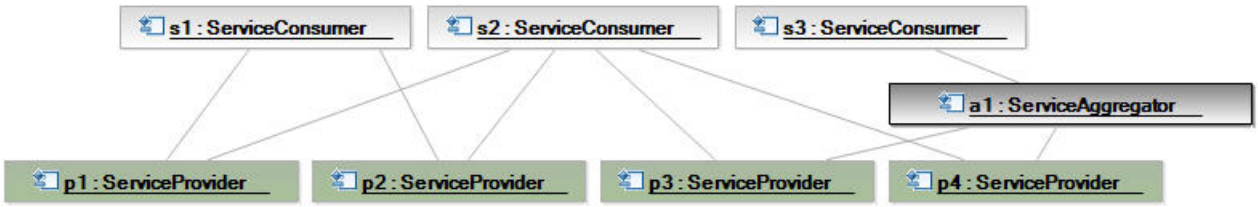
of a system's aggregation. The first metric is the *Extent of Aggregation* (EOA):

$$EOA(\Omega) = \sum_{c \in \{\Omega.C \setminus \Omega.A\}} \frac{\sum_{a \in \Omega.A} \lambda(c,a)}{\sum_{p \in \Omega.P} \lambda(c,p)} \tag{7}$$

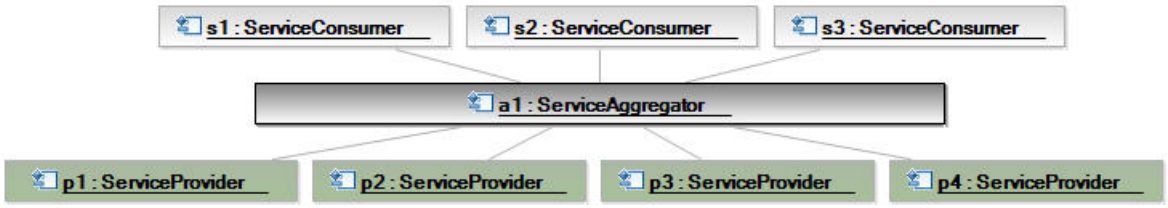
Mechanism *EOA* relates the count of channels between non-aggregative consumers and aggregators with the overall count of channels from non-aggregative consumers to arbitrary service providers.

Value range The value range of *EOA* is $[0, 1]$.

Discussion *EOA* describes the extent of hierarchy in the system: the ratio between the count of channels that are mediated by aggregators and the total count of total channels from sole service consumers to all service providers. Low values indicate arbitrary channels among a system's components while relatively high values indicate a high degree of aggregation. Usually, a higher degree of aggregation is better as it indicates that complexity is addressed. The *EOA* value of a system should be looked at with the knowledge of *SSC* values as well as the knowledge of the absolute counts of the different service types as these values indicate how much complexity a system needs to deal with. Important to note is that *EOA* relies on a system heavily using consumers. If an application is solely triggered by a service consumer, the result might not be representative for the complete system.
Examples for *EOA*-values are given in table 6.



$\texttt{EOA}(\Omega_1) = 0.143$



$\texttt{EOA}(\Omega_2) = 1$

Table 6: Examples of `EOA` Values

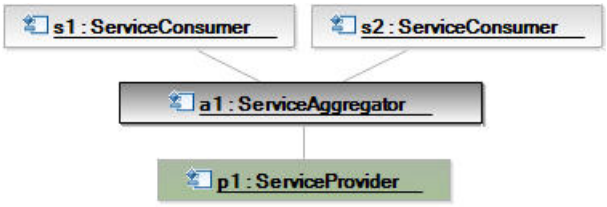
- **Density of Aggregation (DOA)** *DOA* indicates *to which extent* the aggregation in a system combines more basic services to more complex services:

$$DOA(\Omega) = \sum_{a \in \Omega.A} \ln\left(\frac{\gamma(a)}{\pi(a) + \gamma(a)} \times 2\right) \tag{8}$$

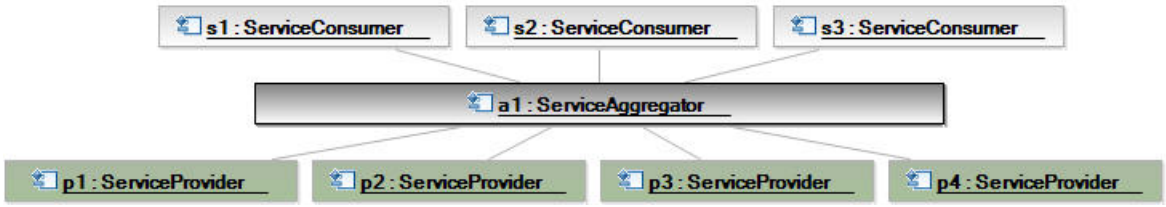
Mechanism *DOA* relates for each service aggregator the count of *serviceCall*-ports to the overall count of the service's ports. The value range for this ratio is $]0, 1]$. A value of ≥ 0.5 for an aggregator indicates that it consumes more ports than it provides. By multiplying it by 2 and calculating the logarithmic value for this result, such aggregators get a low positive score. For aggregators that offer more ports than they consume, a relatively high negative value is the result.

Value range The value range of *DOA* is $]-\infty,\ +\infty[$.

Discussion By "punishing" the non-aggregative use of aggregators with relatively high negative values while "rewarding" only low positive scores to "real" aggregators, an overall positive value indicates proper use of aggregators. The absolute value of this metric is irrelevant. Especially in combination with the *SCZ* the *DOA* value can indicate whether centralization of a system goes along with a good aggregation. Hence, for high *SCZ* values, the *DOA* value should be positive. If a negative *DOA* meets a high *SCZ* value, a system might use improper centralization mechanisms that decrease the level of modifiability.
Examples for *DOA*-values are given in table 7.



$\texttt{DOA}(\Omega_1) = -0.405$



$\texttt{DOA}(\Omega_2) = +0.134$

Table 7: Examples of `DOA` Values

- **Aggregator CentraliZation (ACZ)** ACZ indicates the degree of centralization in a system by considering the use of mediating services. Mediators are identified by the supporting measure AD.

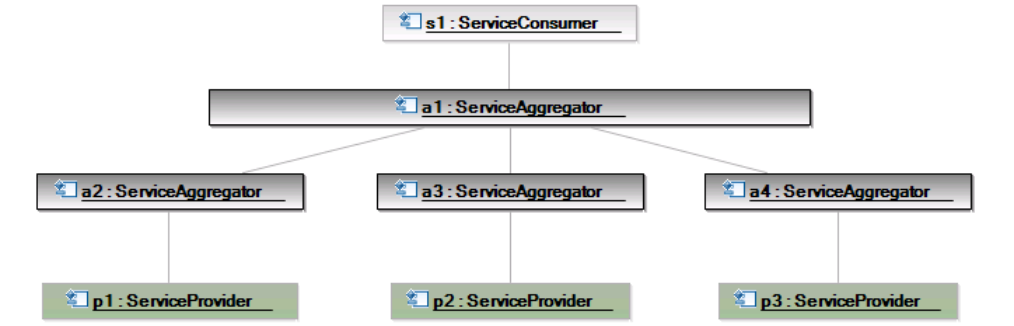
$$AD(\Omega, a) = \begin{cases} 0 & \text{if } 0.5 \leq \dfrac{\gamma(a)}{\pi(a) + \gamma(a)} \leq 0.\overline{6} \\ 1 & \text{otherwise} \end{cases} \mid a \in \Omega.A$$

$$ACZ(\Omega) = \begin{cases} 1 & \text{if } SA(\Omega) = 1 \\ 1 - \dfrac{\sum\limits_{a \in \Omega.A} AD(\Omega, a)}{SA(\Omega)} & \text{otherwise} \end{cases} \tag{9}$$

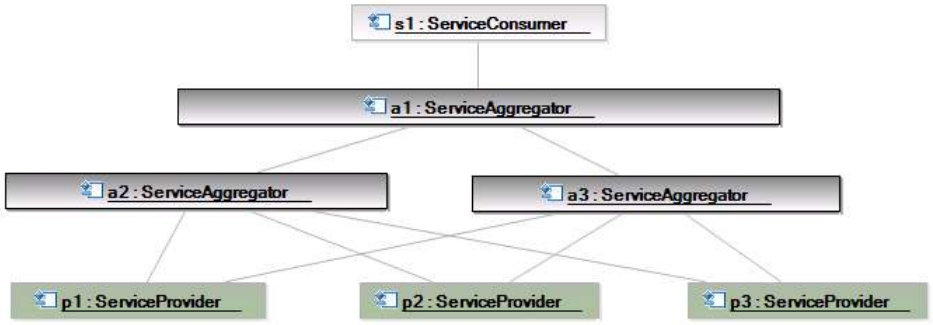
Mechanism ACZ combines the idea of control centralization via aggregators and considers the actual density of an aggregation. An aggregator that might not compose several services is indicated by an *Aggregator Density* (AD) of 0. AD incorporates the idea that little to no control is executed if the density of an aggregation is low. Such an aggregator is considered to be a *mediator*. If services that consume one service and also act as service providers should be seen as service mediators, the AD measure needs to be adjusted accordingly (e.g., $\frac{\gamma(a)}{\pi(a)+\gamma(a)} = 0.5$ or $\gamma(a) = 1$).
By relating the count of non-mediators with the overall count of service aggregators in a system the ratio of such aggregators is calculated. By deducting this ratio from 1 the degree of centralization into non-mediators is indicated.

Value range The value range of ACZ is $[0, 1]$.

Discussion ACZ can be used as a metric to interpret SCZ values. SCZ describes to which degree a system mediates service calls and interprets the use of few aggregators as a centralization. Without considering the internals of an actual aggregator, this can be a misleading interpretation. As the analysis of component internals is considered to be hardly applicable in real-life settings, the interpretation of the internals is supported by the ACZ metric. The ACZ metric incorporates the assumption that control can only be exercised by a service aggregator whose count of *receiveCall*-ports is disparate from the count of *serviceCall*-ports.
Values close to 1 indicate a high degree of centralization while values close to 0 indicate a low degree of centralization in a system. A complete centralization in terms of a ACZ-value of 1 can only occur if only one aggregator is used or only mediators are used. In real-life settings this is unlikely to occur.
ACZ is valuable for the interpretation of the results of the SCZ metric for a given system: if a system's design scores a relatively low SCZ-value, a high ACZ-value indicates that the mediators are used in the design of the system and that the design incorporates the idea of control centralization. However, if both the ACZ and the SCZ values are low, the system does not follow a centralized control model. Of course, this metric has also to be carefully applied. This is because it also solely puts an interpretation of an externally visible structure over the actual internal structure of a component that determines the visible part. However, in conjunction with the SCZ, DOA and EOA values it is considered valuable.
Examples for ACZ-values are given in table 8.



$\texttt{ACZ}(\Omega_1) = 0.75$



$\texttt{ACZ}(\Omega_2) = 0.25$

Table 8: Examples of ACZ Values

The set of metrics described above can indicate to what extent a system is modifiable. Another important influencing factor of modifiability is re-use. In discussing re-use it is necessary to distinguish between to what extent a system is re-usable and to what extent it re-uses functionality of other systems.
These two aspects of re-usability are different perspectives on the same phenomenon. The degree of being re-usable can only be known ex-post while assessing to which extent a system is (or will be) built using existent parts can be achieved ex-ante. If a system is designed to re-use a large portion of its functionality, it can be assumed that the initial implementation of the system will be faster than the alternative of starting from scratch. In the introduction of this chapter it was defined that the quality characteristic of functionality concerns basically how fast functionality can be realized. As re-use has a high impact on this quality characteristic, it is discussed as a parameter of modifiability. Besides the fact that re-usability and re-use can have positive effects on modifiability, re-use can reduce cost (cf. [62]). This is why it is important to assess both re-use and re-usability. The assessment of re-use is discussed in this chapter. Re-usability of a system's services is discussed in chapter 4. In both cases, considering re-use of services is the most fine granular aggregation level. How services are designed internally and whether services are realized by re-using entities that are not accessible as services is not discussed. Additionally, verbatim re-use (cf. [63]) is the only way of re-use that is discussed here.

To assess to what extent a system Ω is built using existing parts, partitioning the system is required. As defined above a (modular) service-oriented system can be defined as $\Omega =< E, R, M >$ or $\Omega =< E, R, \Psi >$, respectively. In order to determine ex-ante the degree of reuse in a system, the sets Ψ_r and Ψ_n as well as R_r and R_a are important.
Ψ_r is the set of services that are shared among a system Ω and at least one other sys-

tem Ω': $\Omega.\Psi_r \subseteq \Omega'.\Psi \mid \forall\ r \in \Omega.\Psi_r\ .\ \exists\ \Omega' \neq \Omega,\ r \in \Omega'.\Psi$. Additionally, it is obvious that the set of both new and re-used services compose the set of all services in a system: $\Omega.\Psi = \Omega.\Psi_r \cup \Omega.\Psi_n$ and $\Omega.\Psi_r \cap \Omega.\Psi_n = \emptyset$.
This definition introduces the problem of delineating systems. We introduced the notion of a system as a tuple of sets. However, in real world applications this definition is ambiguous, as the borders of such distributed systems are blurred. Using services of "other systems" could be considered a merge of two disjoint systems. This is why an applicable mechanism for delineating systems is required for some discussions of re-use: a system is a triple of methods, relations and services that is used in order to realize a business process. If multiple implementations of business processes share elements, modules or relations they are still considered disjoint systems[16].
Still, re-use across systems can be ambiguous because the notion requires an initial possessor. The decision of re-use can then only be made if the ownership of a service is known. This is a governance problem and considered out of scope of this thesis. This is why we incorporate the notion of time into re-use: if a service s that is used in a system Ω' at the time $(\tau - 1)$ is also used in a system Ω at the time τ, s is considered to be re-used in Ω. This is why re-use can only be measured for a point in time.
In order to assess how services are re-used, two sets of relations need to be distinguished, too. These relations are $R_r \subseteq \Omega.\Psi_r \times \Omega.\Psi_r$ as well as $R_a \subseteq \Omega.\Psi_n \times \Omega.\Psi_r$. Of course, these sets are disjoint: $R_a \cap R_r = \emptyset$ while $R_a \cup R_r \subseteq R$.
R_r are the relations that exist between services that are used in multiple systems. The relations might be either already part of another system $\Omega.R_r \subseteq \Omega'.R$ or might be introduced (partially or completely) by the new system.
R_a is the set of all relations that are added between the new services of a system and the services that are shared with other systems. All relations $a \in \Omega.R_a$ are introduced together with the system Ω. R_a can also be considered as the set of *inter-system relations*.

The following measures are introduced in order to discuss the concepts of re-use and to propose means for assessing the extent of re-use in a system.

- **Re-used Services (RS)** RS is the count of services a system uses that are also part of other systems at the time τ: $RS(\Omega, \tau) = |\Omega.\Psi_r|$.

  Mechanism RS is a simple count of elements.

  Value Range The value range of RS is $[0, +\infty[$

  Discussion If a system is built by re-using existing parts, the required functionality can be achieved faster. This is why an absolute high RS value indicates a better modifiability of a system. However, if multiple systems heavily re-use certain services, the modifiability of all systems might decrease under certain circumstances. This is because changes to these services affect all other systems, too. This is why RS is only a basic measure and needs to be evaluated with other metrics like the MRR metric that is also introduced later in this chapter.

- **Multi-used Services (MS)** MS is the count of services in a system that are used more than once at the time τ: $MS(\Omega, \tau) = |\Omega.X| \ |(cts(x) > 1)\ \Rightarrow (x \in \Omega.X)\ ; x \in \Omega.\Psi$

  Mechanism MS is a simple count of elements.

  Value Range The value range of MS is $[0, +\infty[$

[16] Which, of course, fits into the formal definition of a modular system, too.

Discussion MS is the amount of services that are used more than once. The fact that a service is used more than once does not lead to the notion that this service is re-used. Re-use is defined here as the use across system borders or the existence prior to the creation of a system. This definition indicates $MS \leq RS$.
However, this measure possesses some significance. A high MS-value is a good indicator that a system consists of many re-usable services, as high cts-values indicate that a service might be re-usable.[17] Absolute high MS-values indicate that changes to single services might effect large portions of a system. This effect can be beneficial in terms of modifiability. Nevertheless, mediating these services by using aggregators should also be considered. This is because mediation could reduce the impact of changes to providers. If aggregators are used, the density of aggregation (DOA) should be taken into account.

- **Number of Usages (nou)** nou is the count of systems a service is used in: $nou(s, \tau) = |S| \mid \forall \Omega \in S \ ; \ s \in \Omega.\Psi$

  Mechanism For each system Ω for that a relation to a given service s is part of its relations, nou is increased.

  Value Range The value range of nou is $[0, +\infty[$

  Discussion If a service s is used at τ in nou systems, changes that are made to s will not only effect the system the changes are required for but nou systems. This is why a high value for this measure can indicate a decreased modifiability. This is because changes can only be implemented by considering side-effects. On the other hand, necessary changes (e.g., due to regulatory requirements) can be implemented across systems more quickly and easily.
  In order to address this trade-off, the concept of mediation shall be used. The re-use through mediators is measured by the MRR value. Absolute high nou-values should only occur for aggregators.

- **Reused Connections (RECON)** $RECON$ is the count of relations between a system and services that are also used in other systems. $RECON(\Omega, \tau) = |\Omega.R_a|$

  Mechanism $RECON$ is the amount of connections (or channels) from a system to services that are used also in other systems.

  Value Range The value range of $RECON$ is $[0, +\infty[$

  Discussion $RECON$ is different from $OuterR(m)$ as it does not assume fixed system (module) boarders. It is defined as the cardinality of the set of all connections from services that are introduced by one system and services that existed before. It is a basic measure for more complex metrics.

- **Mediated Re-Use (MRU)** MRU indicates how many exclusively used aggregators of a system Ω are connected with services that are shared across systems: $MRU(\Omega, \tau) = |\Omega.R_{aa}| \mid \Omega.R_{aa} \subseteq \Omega.A_n \times \Omega.\Psi_{pr} \mid \Omega.A_n \subseteq \Omega.SA \ ; \ \Omega.\Psi_{pr} \subseteq \Omega.SP \ ; \ \Omega.R_{aa} \subseteq \Omega.R_a$

  Mechanism MRU is the number of relations between exclusively-used mediators of a system Ω and shared services. The aggregators that are used exclusively in a system are a subset of the systems aggregators.

  Value Range The value range of MRU is $[0, +\infty[$

[17] Re-usability assessment is discussed more in depth in chapter 4.

Discussion As discussed, aggregators promote loose coupling and make mediated services more independent from other parts of the system. This is why the negative effects extensive re-use can have on modifiability, can be addressed by using mediators. High MRU-values indicate that mediators are introduced together with a system. The alternative would be to also re-use aggregators. The preferable method of mediation is dependent on the actual use case. MRU is one of the basic measures that are used in the indicator for mediated re-use MRR.

- **Re-Used Mediation (RUM)** RUM indicates how many relations among exclusively-used consumers to shared mediators exist in a system: $RUM(\Omega, \tau) = |\Omega.R_{ar}| \mid \Omega.R_{ar} \subseteq \Omega.C_n \times \Omega.\Psi_{ar} \mid \Omega.C_n \subseteq \Omega.SC \ ; \ \Omega.\Psi_{ar} \subseteq \Omega.SA \ ; \ \Omega.R_{ar} \subseteq \Omega.R_a$

  Mechanism RUM is the count of relations among exclusively-used consumers (aggregators and pure consumers) of a system Ω to shared aggregators.

  Value Range The value range of RUM is $[0, +\infty[$

  Discussion RUM indicates the re-use of already mediated services. This design option is more beneficial if quick first results are required when building a system. For a long-term modifiability self-developed aggregators might be preferable, though. The preferable option depends on the actual use case.

- **Aggregator to Aggregator Re-Use (AAR)** AAR indicates how many relations among exclusively-used aggregators to shared aggregators exist in a system: $AAR(\Omega, \tau) = |\Omega.R_{aar}| \mid \Omega.R_{aar} \subseteq \Omega.A_n \times \Omega.A_r \mid \Omega.A_n \subseteq \Omega.A \ ; \ \Omega.A_r \subseteq \Omega.A \ ; \ \Omega.R_{aar} \subseteq \Omega.R_a$

  Mechanism AAR is the count of relations among exclusively-used aggregators of a system Ω to shared aggregators.

  Value Range The value range of AAR is $[0, +\infty[$

  Discussion AAR indicates "double" mediation. High AAR-values arise whenever mediators are used in systems to re-use services that are already mediated by aggregators. This approach might only be advantageous for certain use cases. Generally, AAR should indicate values close to 0.
  $AAR(\Omega, \tau)$ is always less or equal $max(MRU(\Omega, \tau), RUM(\Omega, \tau))$.

- **Re-Use Ratio (RUR)** RUR indicates whether a system is built using existing parts or not:

$$RUR(\Omega, \tau) = \frac{RS(\Omega, \tau)}{NS(\Omega)} \tag{10}$$

  Mechanism RUR is the ratio of re-used services to the overall count of services in a system NS.

  Value Range The value range of RUR is $[0, 1]$

  Discussion RUR-values close to 1 indicate that systems are (to be) built using large proportions of services. On one-hand side this will lead to rapid, cost efficient realizations. However, future modifiability might be affected in a negative way by this design option, as indicated by high RUR-values. This is why RUR should be considered together with MRR as well as DOA in order to estimate the effects the extensive re-use of services might have on the modifiability of the system.
  An interesting notion is again the dependence on time: the RUR for a system can (as with all re-use measures and metrics) change without modifying the system. Such

effects should be addressed by organizations by introducing governance structures and processes.

- **Mediated Re-Use Ratio (MRR)** MRR is the ratio of mediated re-use to the overall re-use:

$$MRR(\Omega, \tau) = \frac{MRU(\Omega, \tau) + RUM(\Omega, \tau) - AAR(\Omega, \tau)}{RECON(\Omega, \tau)} \tag{11}$$

Mechanism As mediated re-use and re-used mediation are non-disjoint aspects of a system, $MRU(\Omega, \tau) + RUM(\Omega, \tau)$ might be greater than $RECON(\Omega, \tau)$. This is why the ratio between mediated re-use and the overall re-use is adjusted by deducting $AAR(\Omega, \tau)$ from the numerator.

Value Range The value range of MRR is $[0, 1]$

Discussion Aggregators promote loose coupling and make mediated services more independent from other parts of the system. This is why an extensive re-use of services that are not used directly but rather, are mediated by aggregators, indicates an overall loose coupling in a system. Therefore MRR-values close to 1 indicate a "good" sort of re-use and, in turn, a higher modifiability of the respective system.
Not included in the MRR-value is the point of mediation. But this is another important aspect. Since this design decision is dependent on actual requirements, it can only be given the notion that RUM and MRU values should be looked at independently, too.

Having discussed and defined metrics that assess the degree of re-usability in a system, it should also be considered to what extent it is re-usable. This is mainly a concern of the design of the single services and not of the overall system. This aspect is discussed in chapter 4.

All metrics that were described in the previous section indicate the modifiability of a system – some as indicators for high modifiability, some as indicators of low modifiability. An optimal mechanism would be to identify a boolean discriminant function (BDF) with according thresholds for the single metrics in order to determine the modifiability of a system that realizes the analyzed design (as described similar in [56]). This approach has two problems. Realizing a significant application that goes beyond scientific prototyping is cost intensive. This is why the first issue is to get relevant data for running regression tests. A second issue is the required categorization of systems into "easy to modify" and "hardly modifiable". We do not consider such a fragmentation feasible, as it will heavily reflect subjective parameters. This is why an aggregated discrimination function that forecasts the modifiability of a system is considered unrealistic.
We believe that the introduced metrics can be better used to highlight certain aspects for service-oriented design principles and their impact on modifiability. This is why a qualitative description of inter-relations among the metrics is considered better applicable than a quantitative discriminant function. As a rule it can be stated that if the metrics give a "bad" picture of a design, a redesign is appropriate. If the picture is "good", the design can still be suboptimal. They are a necessary indicator but not sufficient indicators of design quality. For the sake of demonstrating their applicability, the following case study shows the metrics in the context of real-life requirements.

Figure 4 shows the presented metrics, their relations and trade-offs. It represents a graphical summary of the discussions given for the single metrics in this section. In addition,

an approach to the interpretation of the metrics in the context of a realistic application will be discussed in section 8.3.

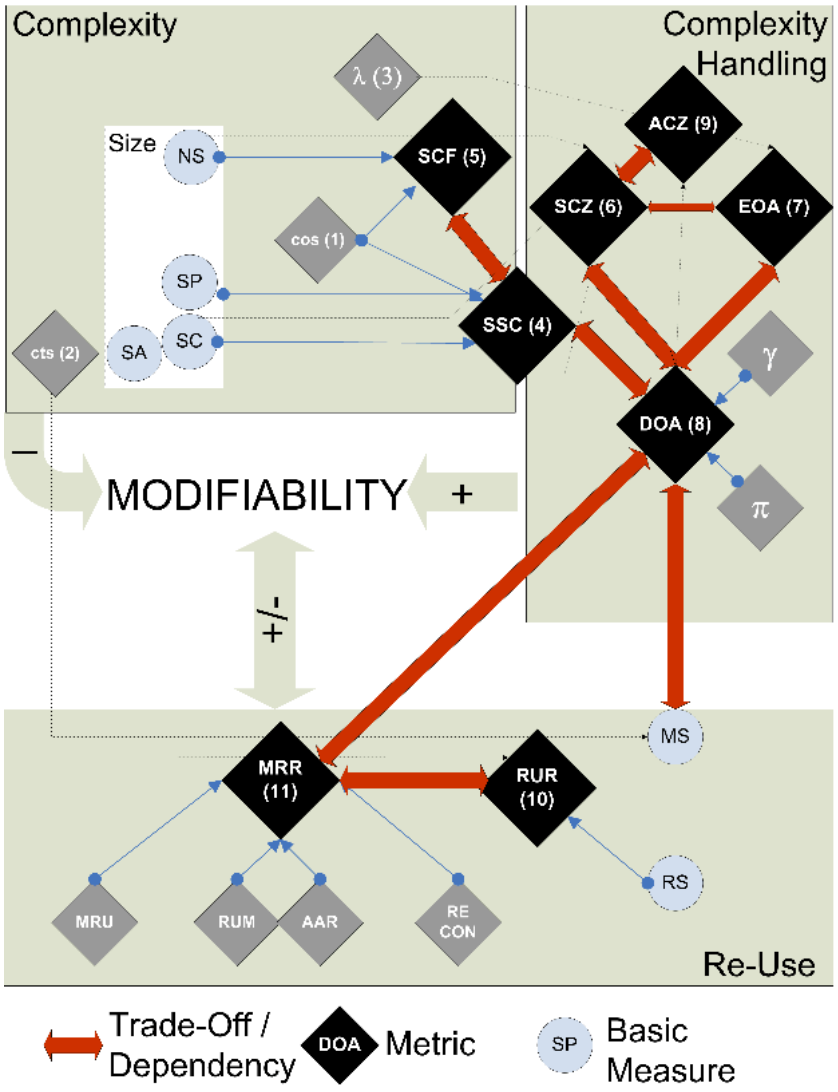


Figure 4: Overview of Modifiability Metrics and their Dependencies

3.2.2 Assessing Reliability

As stated in the beginning of this chapter, applying service-oriented principles can have a negative effect on the reliability of a system. In particular, omission failures, crash failures, and timing failures of single services or their channels can effect multiple systems. How much a system is effected by such failures is described as *fault tolerance* of a system.

Another important parameter that indicates a system's reliability is the *availability* of the overall system. In the following, a mechanism is described that allows for measuring the availability of a service-oriented application. Afterwards, it is discussed how a service-oriented system could deal with unavailabilities.

Empirical models that aim to estimate and improve software availability, such as the Schneidewind software reliability model (cf. [64]), were proposed. Such models aim to

analyze failures that are caused by errors in a software system and that can be observed by testing these systems. Observed failures are used in order to predict future failures of the system and to describe the error correction process. Such models are complementary to the (simple) metrics that are presented in this section. The aim of the metrics here is not to improve the error rate of services. Rather, the impact a given design has on the overall availability of a system should be assessable. The availability of single services should of course be increased independently. This distinction is especially important since a certain proportion of services will be provided by COTS.

Tsai et al. developed in [65] a software availability model for web services that is the basis for the presented availability metrics. It describes both, atomic services and overall design availability measures. The availability for a service[18] S is dependent on the behavior the service shows in the scenarios he is used in:

$$Avl_s = \frac{\sum(w_i \times Avl_{scenario_i})}{\sum(w_i)}$$

w_i is the execution rate of the corresponding scenario i. A scenario is triggered by events and can trigger events by itself (cf. [65, p. 148]). The availability for each scenario can be derived by testing the service under the respective scenarios. [66] proposes a method that uses the data input profile of a scenario in order to derive the availability parameter for a component in the respective scenario.
However, for customers of commercial applications the empirical estimation of software components is not applicable overall. This is because failures that occur in systems, modules or services lead to operative issues of the respective organization. This is why all modules are operated on the basis of so-called service-level agreements (SLA). SLAs are composed by the SLA for a platform a certain software system is deployed on and the software itself. The SLA for the software is responded by the software manufacturer. As we discuss the application of SO as an integration paradigm, the outlined approach for deriving a service's availability should be used by the manufacturer in order to define the SLAs he is offering to his customers. The notion of availability is nevertheless important. From the black-box viewpoint we take towards services, it is not relevant whether the availability of a given service is measured or agreed on. This is why we assume that each (sole) service provider that provides functionality for a service-oriented system has an availability Avl_s. The availability of different services is assumed to be mutually independent.

Singh et al. have presented in [67] an approach for availability prediction and assessment of component-based systems. This approach treats components as black-boxes (cf. [68]) and is suitable for assessing design in the early development stage (cf. [69]). Additionally, this model was integrated with UML (cf. [70]). This is why we extend this model with dependencies among components in order to apply it to service-oriented systems.
The probability of a failure of a service can be denoted as: $\theta_{sj} = 1 - (1 - \theta_s)^{bp_{sj}}$. This simple description assumes regularity (cf. [69, p. 179]). This means that the service $s \in \Omega.\Psi$ has a constant failure rate θ_s $(1 - Avl_s)$. This assumption fits with the notion of SLAs. It needs to be considered, though, that the failure rates that are assumed by [67]

[18] Discussing availability involves both a service type and an agent. Thus, referring to services means referring to usable resources. This implies that the availability of a service includes the availability of hardware components and network availability.

are expressed in failures per execution of a scenario. In contrast, SLAs are sometimes usage-independent. When using the notion of availability Avl_s and SLAs, it is assumed here that SLAs are agreed on a per-usage basis.[19]
In the above description, bp_{sj} describes the *busy period* of a service s in a scenario j. A busy period simply indicates the number of calls a component receives and issues throughout the actual scenario.

Considering the aggregation of sole service providers to a higher-level service as a scenario, the availability of a service aggregator $a \in \Omega.SA$ can be then noted as:

$$Avl_{atom}(a) = \left(\prod_{s \in P}(1-\theta_s)^{\lambda(a,s)}\right) \times (1-\theta_a)^{\sum_s \lambda(a,s)} \quad | \; a \in \Omega.SA \tag{12}$$

$$P = \{\Omega.SP \setminus \Omega.SA\}$$

The availability of an aggregator is the product of the failure rates of the aggregated service providers with regards to their *busy time* – which is derived from the coupling between the aggregator and the provider – with consideration of the failure rate θ_a of the service aggregator itself during its execution.[20] In order to keep the assessment applicable for COTS-based aggregators, each aggregated service is considered to be included in every execution $\lambda(a, s)$ times. This leads to a conservative assessment of the availability of service aggregators as the probability of defect services not being used is ignored.
A special case is when service aggregators use service aggregators themselves. The availability of such an higher-level aggregator Δ can then be noted as

$$Avl(\Delta) = \left(\prod_{s \in P}(1-\theta_s)^{\lambda(\Delta,s)}\right) \times \prod_{a \in Y} Avl(a) \times (1-\theta_\Delta)^{\sum_s \lambda(\Delta,s)} \tag{13}$$

$$\Delta \in \Omega.SA$$

$$\forall a \in Y \; . \; \{\Delta\} \times \{a\} \in \Omega.R \mid a \in \Omega.SA$$

$$P = \{\Omega.SP \setminus \Omega.SA\}$$

Avl provides a mechanism that can be used in order to assess how the availability of a service aggregator is affected by the availability of the services it aggregates. Low availability of the aggregator can either be compensated by changing the design of the aggregator (while considering the modifiability metrics) or by changing the SLAs of the used service providers — without the need to test the overall system.

[19] By using slightly different semantics than the one that was derived here from the ISO norm [49], [71] differentiates availability and reliability in order to solve this issue (cf. [71, p. 362]). We assume that the indicated availability describes the failures during a time interval (= the scenario). In order to avoid confusion, we do not use the phrasing of reliability that is used in [71] for describing this quality property.

[20] This failure rate is assumed to be the SLA of the platform the aggregator is deployed on. If necessary, the availability of the aggregator can be tested. In order to keep the application simple, this is not incorporated into the presented approach.

The overall failure probability of a system is defined in [67] similar as:

$$\theta_{\Omega} = 1 - \sum_{j=1}^{K} p_j \left(\prod_{i=1}^{N} (1 - \theta_i)^{bp_{ij}} \right) \tag{14}$$

K is the set of all possible scenarios of a system. p_j denotes the probability of the j^{th} scenario to be executed.
As we transferred the notion of scenarios to the scenario of service aggregation, the availability of service aggregators can be included in this overall picture.
If needed, the overall availability of a system should be derived using (14), though. This is because, the availability metric for aggregators assumes that all aggregated services are always used. This simplification is acceptable for aggregators. But for the overall system availability this could be too pessimistic. In order to calculate the p_j-values, the execution path for every service consumer needs to be analyzed with a method as described in [70].

As stated in the introduction, availability of a system can be endangered by the application of service orientation as a paradigm for distributed systems. The formulas (12) and (13) are applicable to calculate – with respect to service-oriented principles – the availability of service aggregators. This is an important mechanism because aggregators are single-point-of-failures in a composite application while being helpful in terms of modifiability. In order to allow a system designer to assess how the use of aggregators that improve modifiability effects the availability, these formulas should be used in order to simulate how the availability of the single service providers effect the availability of the aggregators. A statistical approach that allows for such simulations is described in [67].

However, the availability of a system can only be one aspect used to assess a system's reliability. As important as availability is a system's tolerance to failures as well as how it does recover from failures.
It is not the aim to describe how failure tolerant distributed systems must be built. Again, the special notion of service orientation should be applied to prior work in this area in order to derive indicators that help assessing the reliability of a service-oriented system.
When analyzing work in the area of fault-tolerant distributed systems (esp. [71, pp. 361-413]), two things are important: that failures do not prohibit the atomicity of service interactions and that a system can recover from failures. Discussing these issues, it is assumed that a composite application that is analyzed will be executed using a common protocol that incorporates failure tolerance on the communication layer. Hence, atomicity and recoverability are analyzed from an application-layer point of view. Gray and Reuther discuss failure tolerance and recoverability in this way: "atomicity is more than a mere definition; it requires precise specifications for two fundamental aspects of each operation. One is the question of when and how the results of the operation are made accessible. [...] The other aspect is [...] how the (partial) results of an operation that do not complete successfully be rolled back such that no side effects (or only well-controlled side effects) occur " [72, p. 160]. This definition is also applicable for the service-oriented architectural style. This is that services expose these operations that should complete successfully (atomicity of service interactions) or be rolled back (recovering from failures). These are the two remaining aspects that complete the picture of reliability.
In order to answer the question "when and how the results of the operation are made accessible" [72, p. 160] in the context of aggregated services, two cases need to be distinguished. According to Grefen et al., two orthogonal transactional layers are required: one

for relatively short-living and one for long-living transactions (cf. [73]). The transactional layer for short-living transactions can consist both of single services as well as of multiple, distributed services. In both cases, the ACID (*A*tomicity, *C*onsistency, *I*solation and *D*urability) (cf. [72, pp. 166f]) properties need to be satisfied in order to achieve failure tolerance. As soon as distributed services and their underlying application systems need to ensure such a level of reliability, this needs to be addressed by the overall design. This is why the design of a system needs to incorporate a protocol such as the two-phase commit protocol (2PC) (cf. [72]). In order to allow for a distributed 2PC in a system, a *coordinator* needs to be included in a system (cf. e.g. [71, p. 394]). Following the definition of the service-oriented architectural style, such a coordinator might be an aggregator.
Thus, the simple assessment of a design is whether it incorporates a service aggregator that is able to perform a distributed 2PC among service providers (including the underlying application systems). If such an element is present in the design, this is an indicator for a failure-tolerant design. If not, the software system might have limitations in terms of reliability.

The remaining aspect of reliability corresponds to the transactional layer for long-running transactions: recoverability. If a part of the system is unavailable or fails, the system has to recover to a correct state (cf. [71, p. 401]). This can be achieved using two forms of recovery: backward recovery or forward recovery. For backward recovery it is important to mark certain states in a computation as checkpoints. In the event of failures, backward recovery restores the state that was saved in a checkpoint (cf. [71, p. 401]). In order to realize such checkpoints during run-time, a distributed snapshot of the system might be used (e.g., by using the method described in [74]). For such snapshots, an arbitrary process can notify all other entities in a distributed system to record their current state. This way, an overall consistent state is recorded.
Service orientation has, however, the notion that services should be stateless. If that was fully achieved in a system, no need for snapshots or other forms of coordination would be required. This is unrealistic because stateless services can of course change the state of their underlying application system. The conversational state should be kept by the aggregators of a system. If the state of such aggregators would be mutually exclusively accessible by all participating services, ensuring a consistent state among all services would be easy to achieve (cf. [75, p. 643]). However, changed states of connected application systems can not be included in such a globally accessible state. This problem can be addressed by forward recovery.
Forward recovery is a mechanism that can be used to reach a consistent state that is different than the previous consistent state. How this is achieved is purely application dependent as compensating transactions try to "semantically reverse what the original sub-transaction has done" [72, p. 203]. Such compensation operations are important for service-oriented applications as they are the sole mechanism that can reverse data that was changed in back-end systems after a (probably distributed) short-living (ACID) transaction has been committed.
Long-running transactions that are compensated by forward recovery have relaxed ACID properties. Meeting the long-term characteristics of global transaction, Grefen et al. relax the properties of isolation and atomicity. Relaxed isolation is achieved by publishing intermediated results to the global state of a system context (which is again easy if a common store is available) (cf. [73]). Atomicity is relaxed by introducing compensating transactions. Both context publication and compensating transactions are short-term transactions. Thus, the fundamental support for global transactions is also formed by

ACID (short-term) transactions (cf. [73, p. 319]).
In order to assess the recoverability of a system it is therefore necessary to include compensating service operations into the design. Together with a transaction coordinator for failure tolerance, the existence of compensating services and a globally accessible data store (that manages the state) can be an indicator of a system's ability to recover from failures.

3.2.3 Assessing Usability

Applying the service-oriented architectural style may also have impact on usability. However, it may not necessarily impact the usability of *a system* that is built. The impact is on the usability of the supported business process. Such processes are often conducted by using multiple back-end systems. The portal-based integration approach (see section 2.1) addresses this issue. Following this approach by integrating "all participating systems through the browser, although it does not integrate the applications [...]" [4, p. 99], the usability of the underlying process is increased. This is achieved by providing process users with the ability to work with one application (the browser) in order to perform the process's tasks. However, this approach has notable weaknesses. The most crucial one is that "information does not flow in real time and so requires human interaction. As a result, systems do not automatically react to business events [...]" [4, p. 103].
The BPIOAI-approach addresses the issue of automated and explicitly controlled computation of business events. Using portals for the human-interaction with such processes would combine the advantages while avoiding the (obvious) drawbacks of purely portal-based integration. To assess the usability of a composite application, it is possible by simply counting the user interfaces a user interacts with in order to perform tasks in a business process. A specialization of this count, is to count the logins a user needs in order to perform the tasks his role(s) in the process requires. The simple rule here is: the fewer logins, the greater the usability of the system.
In order to focus on service-oriented principles, other usability characteristics such as the ability of an application to be learned or liked is considered out of scope.

3.3 Assessing the Suitability of SOA

The above discussion highlighted potential benefits and trade-offs that the service-oriented principle presents. When receiving requirements for new systems or change requests, it is necessary for suppliers to assess whether the service-oriented architectural style is suitable for quickly realizing the given requirements. Therefore an inexpensive mechanism is required that allows for assessing the suitability as well as for explaining customers the potential benefits of service orientation in the actual context without spending major efforts.
Due to organizational as well as timing reasons, the use case underlying the case study that is used in chapter 8 for applying, testing and verifying the concepts and findings that are compiled in this thesis, was not chosen by using the approach presented here. Furthermore, the methodology that is proposed here incorporates the experience made with that first approach from a practitioner's point of view.

In order to assess middleware architectures, [76] proposes a scenario-based evaluation

method that can be used for the evaluation of a middleware architecture (MEMS). It is a methodology that was derived from the Architecture Tradeoff Analysis Method (ATAM) [77]. MEMS is a method that consists of seven steps and involves the implementation of prototypes that are used to compare different designs. We consider this approach as too laborious in the context of fast requirement assessment. Important to notice, however, is that the methodology is "founded on key scenarios that describe the behavior of a [...] architecture with respect to particular quality attributes and in particular contexts" [76, p. 14]. In order to achieve this, key quality attributes that are determined in step 1 of MEMS are applied to newly-generated key scenarios of the actual requirements. Based on this application, prototypes for different alternatives are defined and realized. We believe that the first two steps can already support the assessment of given requirements with regards of service orientation.
Based on the quality attributes that are described in the sections 3.1 and 3.2, standard scenarios have been derived. This fits with the first two steps of the MEMS methodology. If a new change request is made or a new project is requested, the supplier can simply assess whether these standard scenarios fit with the requirements. Based on this fit, a first idea of the suitability of the service-oriented style can be generated and communicated to the customer. The single scenarios are described according to the quality characteristics that they profit from.

- **Modifiability** In order to assess whether the benefits of eased modifiability can improve the realization of actual requirements, the following scenarios can be compared to the requirements:
  - **Frequent Changes in Processes** If the business process that needs to be realized is subject to frequent changes, this is an indicator for the suitability of service orientation.
  - **Frequent Changes of Process Tasks** If the underlying business process has several variants of its tasks, service orientation could be suitable. This is because if an orchestration is used to realize the process, the orchestrated services can easily be replaced – by the use of a registry, this replacement can be performed transparently during runtime (cf. e.g. [14]).
  - **Outsourcing of Process Tasks** If the execution of tasks could be (partially or completely) outsourced, a centralized control can improve the flexibility of the system in a way that allows for flexible outsourcing. As the process is not executed by monolithic back-end applications, services that represent these tasks can be called at the outsourcing partner. It does not matter whether the tasks are fully automated or require human interaction.
  - **Use of a Shared Service Center** If a company centralizes business process execution into so-called shared service centers, service orientation can also be supportive. Shared service centers execute supporting business processes (such as human resources, billing etc.) on an enterprise-wide basis. If the process is standardized while the single tasks vary for organizational entities (e.g., due to local legal regulations), service orientation could be beneficial. In a solution for such scenarios, the shared service center would provide the process orchestration for the de-centralized services that are exposed by the application systems of the single organizational entities.
  - **Different Client Applications** If the requirements articulate that different client applications are necessary, service orientation can help. One example

could be a process that involves the purchasing of goods. Some suppliers might offer a system-to-system interface, others might require a login into the supplier portal of the ordering company. Thus, one can see that different applications are required for fulfilling the same task. From an implementation point of view, composite applications are agnostic to this aspect as services are solely described from an interface point of view and accessible via a common protocol. What type of application is therefore used for implementing the respective service is not important.

- **Reliability** As previously described, reliability is a characteristic that needs to be taken into consideration when designing composite applications. That is why composite applications are not considered less reliable than other distributed applications. However, if requirements fit into the following usage scenario, service orientation might not be an appropriate principle.
  - **Production Control** If an application needs to control critical processes such as shop-floor processes or logistic execution, alternatives to service orientation should be evaluated as well. Because minimal unavailability of a control application can stop the complete operative business of a company, highly replicated monolithic applications might be more appropriate. However, if the control involves multiple applications that need to be integrated, service orientation might be a suitable integration paradigm.
- **Usability** Service orientation is not a principle that can be used to improve user interfaces. As described above it can contribute to the usability of processes by exposing a single user interface for users of that process.
  - **Process Requires Human Interaction in Multiple Back-end Applications** Simply stated, if a process spans multiple application systems and users need to interact with these application in order to fulfill their tasks, service orientation can be suitable for improving the usability.

Clearly, these scenarios are rough descriptions of possible requirements. Further, the mapping of requirements to these scenarios is a subjective task. But this is necessary for keeping the concept applicable for real-life requirements. Only after a first evaluation has indicated the suitability of the service-oriented style, a more detailed design can be created and assessed using the means that were presented in section 3.2.

4 Is There Reuse by Design? A Quantitative Approach

Service-orientation is a paradigm for reusing functionality that is exposed by (legacy) application systems in business processes that possibly involve multiple application systems. The functionality that is being reused is exposed through services. Discussions that involve the design of services often include the notion of reusable services. The assumption is that services can be designed in a way that increases the reusability of the designed services.
Several design principles have been identified that support the design of reusable services. Papazoglou et al. describe e.g. in [78] that a high service cohesion leverages the reusability of a service (cf. [78, p. 417]). In addition he states that "low [service] coupling [...] indicates a well-partitioned system that avoids problems of service redundancy and duplication" [78, p. 416]. Erradi et al. identify an appropriate service granularity as a design principle that leverages reusability (cf. [79, p. 156]).
In the context of object-oriented design, Etzkorn et. al described in [80] that the reusability of a class is achieved by a lack of coupling, a cohesive design and the appropriate size of the class's interface (cf. [80, p. 300]). Hence, the design principles that make up a reusable software entity are very similar for both approaches.
Several approaches aim to design (reusable) services (cf. e.g. [11], [78], [81]). All of these approaches respect the design principles of service cohesion, coupling and granularity to a certain extent. What lacks, however, is an objective definition of these design principles. Additionally, none of the known approaches have been quantified whether and to what extent the single design principles really contribute to the reusability of services.

In order to analyze which design principles contribute to the reusability of services, a set of metric candidates is presented. These candidate metrics are, in turn, applied to a non-academic use case of a distributed application and analyzed using quantitative means to derive the discriminative power of the single candidate metrics. This analysis aims to frame the discussion of service reusability on an objective level that is more reliable than the qualitative descriptions that underlie this discussion today. Further, this analysis is not based on a survey that represents subjective opinions of "experts". It is based on objective data about reused services.
The objective of this analysis is to derive valid principles of service design from existent service design approaches. The identified principles should be applicable to the definition of methodologies that focus on applicability while considering the dimension of reusability.

4.1 Candidate Metrics for Reusable Service Design

All candidate metrics that are described here can be categorized by four dimensions. The first dimension is the object a metric is applied to. Possible values are {`field, operation, service`}. The second dimension distinguishes whether the object a metric is applied to is analyzed as a black box or under consideration of its source-code: {`internal, external`}. The third dimension describes the `scope` of the metric. In the domain of service orientation, five different scopes are possible: {`class-internal, service-internal, aggregation-internal, system- internal, cross-system`}. These dimensions indicate the reference for the actual metric.

The fourth dimension describes four variants that describe the quality of objects that are computed. For every object in {`field, operation, service`} it can be distinguished which property of the object is computed: name and type (a), the respective semantics of the variable (b), their semantic type (c) and their type (d).
Summarizing, a metric `m` is categorized by the vector $\overline{\texttt{MV(m)}} \leq$ `[ {field, operation, service}, {internal, external}, {class-internal, service-internal, aggregation-internal, system-internal, cross-system}, {name and type, semantics of variable, semantic type, type} ]`.
The following example shall outline this fourth dimension: considering a service with two methods, a metric y should be computed. The first three dimensions are classified by $\overline{\texttt{MV(m)}}$ = [`operation, external, service-internal, X`]. Hence, the operations of a given service should be analyzed from an outside point of view. The service that is to be analyzed shall have the following two operations:

```
x:  long getTimeDifferenceInMillis(String startDate, Date endDate)
y:  Date getTimeDifference(long startDateInMillies, String endDate)
```

In the fourth dimension the following can be observed: considering variant (a), six different variables are used. None corresponds with another one. Using (b), three different elements can be distinguished: a time difference, a start date and an end date. Subsequently, (c) would analyze two different variables: a point in time and a period of time. Considering (d) there would be three variables: `long`, `String` and `Date`.
The alternative, (d), is the easiest to measure as it is un-ambiguous. However, the unique use of types will degenerate any metric — especially if elementary data types are used. (c) can only be measured if semantic types are introduced in a system. Dimensions (a) and (b) rely on the use of a naming convention that names the same variable for the same purpose always the same way. If such a naming convention is in place and the design is made accordingly, same variables will have the same type and names if the same semantic object is concerned. Applying variant (a) or (b) will not likely result in different values for a given metric.

Services group sets of methods. The methods expose the underlying functionality to arbitrary systems. Methods themselves provide a relation to their input values $\{Inp_m\}$ and their output values $\{Outp_m\}$. A basic measure that can be used in several candidate metrics describes the intersection of these two sets among different operations.
(15) describes for a given parameter `p` its occurrences in the set of all input parameters of all methods {M}. The set of all methods is determined by the actual scope of the candidate metric.

$$v^i(p, \{M\}) = \sum^{m \in \{M\}} \sum^{in \in \{Inp_m\}} 1 \mid in = p \tag{15}$$

(16) describes for a given parameter `p` its occurrences in the set of all output parameters of all methods {M}.

$$v^o(p, \{M\}) = \sum^{m \in \{M\}} \sum^{out \in \{Outp_m\}} 1 \mid out = p \tag{16}$$

(17) describes for a given parameter `p` its occurrences in the set of all parameters of all operations {M}.

$$v^t(p, \{M\}) = \sum^{m \in \{M\}} \sum^{tot \in \{Outp_m\} \cup \{Inp_m\}} 1 \mid tot = p \tag{17}$$

Based on the introduced principles, three classes of candidate metrics are analyzed in terms of their influence on reusability. These three classes are the ones often referred to in approaches to service design: cohesion, coupling and granularity.

Papazoglou et al. describe that "highly related functionality supports increased reuse potential as a highly cohesive service module can be used for very specific purposes." [78, p. 417].

Class-internal cohesion describes the functional relatedness of methods in their implementing class. Whenever services are realized in an object-oriented way, class-internal cohesion could be measured for any of the classes used for implementing the service. Metrics that analyze class-internal cohesion are described in the following.

- **Lack of Cohesion in Methods (LCOM)** Chidamber et al. defined in [82] the metric *lack of cohesion in methods* ($LCOM$) as "the number of disjoint sets [of instance variables used by a method] formed by the intersection of the n sets." [82, p. 204]. According to [83, p. 76], with $\{I_i\}$ being the set of attributes used by a method $\{M_i\}$, $P = \{(I_i, I_j) I_i \cap I_j = \emptyset\}$ and $Q = \{(I_i, I_j) | I_i \cap I_j \neq \emptyset\}$. $LCOM$ is defined as described by (18).

  $$LCOM(C) = \begin{cases} |P| - |Q| & \text{if } |P| > |Q| \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

  Mechanism "$LCOM$ is the number of pairs of methods in a class having no common attribute references, $|P|$, minus the number of pairs of similar methods, |Q|. However, if |P| < |Q|, LCOM is set to zero" [83, p. 76]. High $LCOM$ values indicate a highly cohesive class while low values indicate a low cohesion for a class.

  Value range $LCOM$ is limited to the range of $[0, +\infty[$

  Discussion Considering $LCOM$, it becomes obvious that cohesion and uniqueness are related properties. If the attributes of classes are clustered into disjoint sets so that all sets are exclusively used by the respective methods, the cohesion is high. Hence, unique methods from a parameter point of view increase the cohesion.
  $LCOM$ is applicable for several points within the $\overline{\texttt{MV}}$-space: $\overline{\texttt{MV(LCOM)}} \leq$ `[operation, internal, {class-internal, service-internal},{name and type, semantics of variable, semantic type, type}]`. Of course, it always measures class-internal relations. However, if services are realized using one class, $LCOM$ is applicable for services.

- **Lack of Cohesion in Methods by Henderson-Sellers ($LCOM^*$)** Henderson-Sellers defined in [84] the metric $LCOM^*$ as follows: $M = M_1 \cup M_2 \cup ... \cup M_n$ being all methods of a class C, $F = F_1 \cup F_2 \cup ... \cup F_n$ being all attributes of class C, $r(f)$ is defined as the number of methods that access a field f:

  $$r(f) = |M^*| \mid f \in I_n \hat{} f \in F; \forall m_n \in M^* \Rightarrow I_n \neq \emptyset$$

  $< r >$ is then the mean r for all fields of a class C:

  $$< r >= \frac{\sum^{f \in F} r(f)}{|F|}$$

$LCOM^*$ is defined (according to [85]) as described by (19).

$$LCOM^*(C) = \begin{cases} 0 & \text{if } |M| = 1 \\ \frac{<r> - |M|}{1 - |M|} & \text{otherwise} \end{cases} \tag{19}$$

Mechanism $LCOM^*$ is the relation between the amount of fields the methods of a class use in average on-top of one field and the overall count of methods.

Value range $LCOM^*$ is limited to the range of $[0, 2]$ (cf. [83])

Discussion $LCOM^*$ is applicable for several points within the $\overline{\texttt{MV}}$-space: $\overline{\texttt{MV(LCOM*)}} \leq$ `[operation, internal, {class-internal, service-internal}, {name and type, semantics of variable, semantic type, type}]`. Of course, it always measures class-internal relations. If services are realized using one class, $LCOM^*$ is applicable for services, though.
Of course, $LCOM$ and $LCOM^*$ might generate different values for the same class.

"Cohesion is the degree of the strength of functional relatedness of operations" [78, p. 416]. Understanding "relatedness" as intersecting sets of parameters, methods or services, measuring cohesion can also mean measuring "uniqueness" (cf. [60, p. 479]). If objects are highly cohesive, their respective combination of parameters is also – to a certain extent – unique within a scope.
In order to define candidate metrics for uniqueness that can be applied if services are solely available from an outside view, three basic functions are required:

(20) describes for a certain parameter `p` and a set of operations `{M}` that is determined by the scope of the metric, whether `p` is unique in that set. Important to note is that this assumes that `p` itself is member of `{M}`. A value of 1 indicates a unique method.

$$isUnique^i(p, \{M\}) = \begin{cases} 0 & \text{if } \upsilon^i(p, \{M\}) > 1 \\ 1 & \text{otherwise} \end{cases} \tag{20}$$

(21) is similar to (20). The sole difference is that the parameter is analyzed with regards to all output parameters.

$$isUnique^o(p, \{M\}) = \begin{cases} 0 & \text{if } \upsilon^o(p, \{M\}) > 1 \\ 1 & \text{otherwise} \end{cases} \tag{21}$$

(22) finally describes whether a parameter `p` occurs more than once in the set of all parameters of all operations.

$$isUnique^t(p, \{M\}) = \begin{cases} 0 & \text{if } \upsilon^t(p, \{M\}) > 1 \\ 1 & \text{otherwise} \end{cases} \tag{22}$$

In conjunction with (15) - (17) allow (20) - (22) to analyze cohesion for service-oriented systems without considering the actual source code of the services. The motivation for that approach is given by Reijers. In [86] he proposes a cohesion metric for workflows that is based on sharing information elements of workflow activities. These information elements are visible as parameters of these activities. This metric can not, however, be

applied in all contexts. This is because it focuses on grouping methods into activities. If an actual design does, however, not allow for analyzing the grouping of methods to services, the metric is not applicable. It solely allows the analysis of the reuse of methods themselves. This is why Reijers' concept has to be transferred in order to be applicable only to methods.
As outlined earlier, it is crucial to analyze services just by their interfaces. This is because services are often black-boxes that can not be analyzed more in-depth than on the level of their interfaces. The drawback to this is that relations among services are not obvious.

Briand and Basili have described in [58] four properties to which a cohesion metric for a modular system should comply. In order to analyze the behavior of the introduced metrics more in detail, the metrics should be checked against these four properties.
The basis for these properties of cohesion is, again, the definition of a modular system that was introduced in chapter 3.
According to the procedure presented in section 3.2.1, methods of services are considered to be the elements of a system. Relations among these methods, such as calling relations (irrespective of any communication semantics), are the relations of a system. Services are considered as modules that group together sets of methods.
The single properties are introduced while analyzing the first new cohesion metric UM

- **Uniqueness of Methods (UM)** Based on the presented basic metrics (20) - (22), the Uniqueness of Methods (UM) for a method `op` can be proposed as a cohesion metric as described in (23).

$$UM(op, \{M\}) = \sum_{p \in \{Outp_{op}\} \cup \{Inp_{op}\}} isUnique^t(p, \{M\}) \tag{23}$$

  Mechanism UM is a count of unique parameters of a given method within the actual scope.

  Value range The value range of UM is $[0, +\infty[$.

  Discussion Stipulating that the uniqueness of a method is an indicator for its cohesiveness, UM is the most basic measure for that principle. High values indicate highly unique methods. This is, in turn, an indicator for a cohesive method.
  Considering UM as a cohesion metric, it should be evaluated against the four properties a cohesion metric should satisfy.
  Taking a cohesion metric $\alpha(m)$, the first property `Cohesion.I` is non-negativity and normalization (cf. [58, p. 77]): $\alpha(m) \in [0, Max]$. UM obviously fulfills non-negativity. However, the maximum value of UM is not normalized. This is achieved by the Relative Uniqueness of a Method in a Service (NUM).
  `Cohesion.II` is defined the following way (cf. [58, p. 77]): $R_m = \emptyset \Rightarrow \alpha(m) = 0$. `Cohesion.II` is called `Null Value`. It describes that a module without relations to other modules shall have a cohesion value of 0. `Cohesion.II` is not fulfilled by UM. This is because connections among services can not be captured by considering the methods of services. This is why a method with a unique signature might exist that is not connected to any other service. Thus, in such a case the UM value might be high. Of course, if services do not call each other directly but instead use an aggregator for coordination, a relation among services is indicated if the output of one method (partially or totally) corresponds with the input signature of another method.

`Cohesion.III` is the monotonicity property. It describes that a new relation between modules does not decrease the cohesion: "Let $MS' =< E, R', M' >$ and $MS'' =< E, R'', M'' >$ be two modular systems [...] such that there exist two modules $m' =< E_m, R_m >$ and $m'' =< E_{m'}, R_{m''} >$ [...] such that $R' - R_{m'} = R'' - R_{m''}$ and $R_{m'} \subseteq R_{m''}$ [...]. Then, $\alpha(m') \leq \alpha(m'')$" [58, p. 77]

As additional relations among services do not imply that methods of the services are changed, additional relations within a system do not increase UM. Hence, UM satisfies `Cohesion.III`.
`Cohesion.IV` is called `cohesive modules`. It states that putting together unrelated modules should not increase the cohesion of the new module compared with the old modules: "Let $MS' =< E, R', M' >$ and $MS'' =< E, R'', M'' >$ be two modular systems [...] such that there exist two modules $m' =< E_m, R_m >$ and $m'' =< E_{m'}, R_{m''} >$ [...] such that $M'' = M' - \{m'_1, m'_2\} \cup \{m''\}$ with $m'_1 \in M', m'_2 \in M', m'' \notin M'$ and $m'' = m'_1 \cup m'_2$ [...]. If no relationships exist between the elements belonging to m'_1 and m'_2, i.e. $InputR(m'_1) \cap OutputR(m'_2) = \emptyset$ and $InputR(m'_2) \cap OutputR(m'_1) = \emptyset$, then $max\{\alpha(m'_1), \alpha(m'_2)\} \geq \alpha(m'')$" [58, p. 77]. By definition, services are modules that group methods. If unrelated methods are regrouped this does not effect the UM-value at all. Hence, UM fulfills `Cohesion.IV`.
UM is applicable for several points within the $\overline{\texttt{MV}}$-space: $\overline{\texttt{MV(UM)}} \leq$ `[operation, external, {class-internal, service- internal, aggregation-internal, system-internal, cross-system}, {name and type, semantics of variable, semantic type, type}]`.

- **Normalized Uniqueness of Methods (NUM)** Based on UM, the Normalized Uniqueness of Methods (NUM) can be defined as described in (24).

$$NUM(op, \{M\}) = \begin{cases} 0 & \text{if} |\{Outp_{op}\}| + |\{Inp_{op}\}| = 0 \\ \dfrac{\sum\limits_{p \in \{Outp_{op}\} \cup \{Inp_{op}\}} isUnique^t(p, \{M\})}{|\{Outp_{op}\}| + |\{Inp_{op}\}|} & \text{otherwise} \end{cases} \tag{24}$$

Mechanism The count of unique parameters is set into a relation with the overall count of parameters a given operation uses. This normalizes the value range of the metric.
High values indicate a unique method and therefore a high cohesion.

Value range The value range of UM is $[0,\ 1]$.

Discussion Being a normalized version of UM, NUM also satisfies the properties `Cohesion.III` and `Cohesion.IV`. Additionally, `Cohesion.I` is satisfied as the value range is normalized. As is the case for UM, NUM does not satisfy `Cohesion.II`.
NUM is applicable for several points within the $\overline{\texttt{MV}}$-space: $\overline{\texttt{MV(NUM)}} \leq$ `[operation, external, {class-internal, service-internal, aggregation-internal, system-internal, cross-system}, {name and type, semantics of variable, semantic type, type}]`.

- **Normalized Relative Uniqueness of Methods ($NRUM$)** The Normalized Rel-

ative Uniqueness of Methods ($NRUM$) is defined by (25).

$$NRUM(op, \{M\}) = 1 - \frac{\sum_{m \in \{M\}} \sum_{t \in \{Outp_m\} \cup \{Inp_m\}}^{p \in \{Outp_{op}\} \cup \{Inp_{op}\}} (v^t(p, \{M\}) - 1)}{\sum_{m \in \{M\}} \sum_{t \in \{Outp_m\} \cup \{Inp_m\}} 1} \tag{25}$$

Mechanism In contrast to NUM and UM, $NRUM$ does not consider whether the binary property of parameters are unique or not. Furthermore $NRUM$ relates the count of (re-)usages of a parameter in all methods within the scope to the overall count of all parameters of all methods within the scope.
For the sake of readability the case of a denominator-value of zero is not included in the function. In such cases, the value shall be zero.

Value range The value range of $NRUM$ is $[0,\ 1]$

Discussion In contrast to UM and NUM, $NRUM$ "reacts" to parameters that are only hardly reused. However, with a large scope, the actual usage count of parameters becomes irrelevant and $NRUM$ approaches 1. This is why the absolute value N should also be considered.
The property `Cohesion.I` is satisfied by $NRUM$, as its value range is normalized. `Cohesion.II` is satisfied by $NRUM$ if relations among services are exclusively realized using aggregators. This is because in such cases input and output parameters of the connected methods have to correspond (at least partially). A method that is not connected at all with other services (resp. their methods), results in $NRUM$ to be zero. Important to note is that a zero value does not imply that relations with other services do not exist. However, if no aggregators are used, $NRUM$ does not satisfy `Cohesion.II`.
With UM and NUM, additional relationships or regrouping do not effect the signatures of methods. Hence, $NRUM$ satisfies both `Cohesion.III` and `Cohesion.IV`.
$NRUM$ is applicable for several points within the $\overline{\text{MV}}$-space: $\overline{\texttt{MV(NRUM)}} \leq$ `[operation, external, {class-internal, service-internal, aggregation-internal, system-internal, cross-system}, {name and type, semantics of variable, semantic type, type}]`.

- **Relative Uniqueness of Methods (RUM)** The Relative Uniqueness of Methods (RUM) is defined as described by (26).

$$RUM(op, \{M\}) = \sum_{m \in \{M\}} \sum_{t \in \{Outp_m\} \cup \{Inp_m\}}^{p \in \{Outp_{op}\} \cup \{Inp_{op}\}} (v^t(p, \{M\}) - 1) \tag{26}$$

Mechanism RUM is the count of (re-)usages of a parameter in all methods within the given scope.

Value range The value range of RUM is $[0,\ +\infty[$.

Discussion Underlying the same mechanism as $NRUM$, RUM also indicates the use of the parameters of a given method within the given scope. The value range is, however, not limited in order to avoid large scopes to marginalize the usage count of the single parameters. Low values indicate highly cohesive methods.
The property `Cohesion.I` is not satisfied by RUM since its value range is not normalized. `Cohesion.II` is also not satisfied by RUM. In contrast to $NRUM$, the

lack of relations (in aggregated scenarios) results in a RUM value of one instead of zero.
As for UM, NUM and $NRUM$ additional relationships or regrouping do not effect the signatures of methods. Hence, RUM satisfies both `Cohesion.III` and `Cohesion.IV`.
RUM is applicable for several points within the $\overline{\texttt{MV}}$-space: $\overline{\texttt{MV(RUM)}} \leq$ `[operation, external, {class-internal, service- internal, aggregation-internal, system-internal, cross-system}, {name and type, semantics of variable, semantic type, type}]`.

Besides cohesion, coupling might influence the reusability of services. In contrast to the coupling metrics presented in section 3.2.1 which measure the overall coupling of a system, more granular metrics are required in the context of reuse. Additionally, the metrics need to be applicable from an outside-view of the services.

Generally any action of one method on another method or field constitutes coupling (cf. [60, p. 479]). As the control flow is not obvious when solely considering structural interfaces, only the notion of the use of identical parameters can be used for analyzing coupling.

While focusing on single services rather than a whole system, the following metrics should also satisfy the notion of the properties `Coupling.I` - `Coupling.V`. They will be checked according to the definition given in section 3.2.1.

- **Input-Output Coupling of Methods ($IOCM$)** The Input-Output Coupling of Methods ($IOCM$) is defined as described by (27).

$$IOCM(op, \{M\}) = \sum^{i \in \{Inp_{op}\}} isunique^{o}(i, \{M \setminus \{op\}\}) \tag{27}$$

Mechanism $IOCM$ is the count of input parameters of a given method that are unique within the set of output parameters of a given scope. The intersection of input and output parameters of the actual method is not included in the count.

Value range The value range of $IOCM$ is $[0,\ +\infty[$

Discussion If the output of a method `y` overlaps with the set of input parameters of a method `op`, the two methods are considered to be coupled. Of course, this is only an indirect implication, as two non-coupled methods could also have overlapping sets of input and output parameters. However, as this is a way of estimating coupling without having an insight into the services (and not even into the aggregators) it is proposed as a candidate metric for the assessment of coupling. Low values indicate high coupling while high $IOCM$-values indicate a low coupling of the actual method within the given scope.
As the count of unique parameters can not be zero, $IOCM$ obviously satisfies the `Nonnegativity` property `Coupling.I`. As low $IOCM$-values indicate high coupling, the `Null Value` property `Coupling.II` is not satisfied by $IOCM$. $IOCM$ does, however, satisfy the `Monotonicity` property `Coupling.III`. This is because additional relations among services are not reflected by the signature of the services.

The `Merging of Modules` property `Coupling.IV` is also satisfied. This is because the merging of methods from different services into another service does not affect the signatures of the respective methods. Property `Coupling.V`, `Disjoint Module Additivity`, is satisfied for the same reason.
$IOCM$ is applicable for several points within the $\overline{\texttt{MV}}$-space: $\overline{\texttt{MV(IOCM)}} \leq$ `[operation, external, {class-internal, service-internal, aggregation-internal, system-internal, cross-system}, {name and type, semantics of variable, semantic type, type}]`.

- **Relative Input-Output Coupling of Methods ($RIOCM$)** The Relative Input-Output Coupling of Methods ($RIOCM$) is defined as described by (28).

$$RIOCM(op, \{M\}) = \sum^{i \in \{Inp_{op}\}} v^o(i, \{M \setminus \{op\}\}) \tag{28}$$

Mechanism $RIOCM$ also utilizes the principle of overlapping input and output parameter sets. In contrast with $IOCM$, it is not a count of the binary property `uniqueness` of the input parameters of a given method in a given scope. It is actually a count of occurrences of the single input parameters within the set of output parameters. The intersection of input and output parameters of the actual method is excluded from that count.

Value range The value range of $RIOCM$ is $[0, +\infty[$

Discussion Low $RIOCM$-values indicate a low coupling while high values indicate high coupling of the actual method. The indicated coupling is also indirect. This is for the same reasons that apply to $IOCM$.
As $RIOCM$-values can only be zero or positive, $RIOCM$ satisfies `Coupling.I`. Also `Coupling.II` is satisfied if all relations among services are mediated by aggregators. This is because in such cases relations are reflected by the signature of the services' methods. If relations are not mediated, `Coupling.II` is not satisfied. In addition $RIOCM$ satisfies `Coupling.III - Coupling.V` for the same reasons that apply for $IOCM$.
$RIOCM$ is applicable for several points within the $\overline{\texttt{MV}}$-space: $\overline{\texttt{MV(RIOCM)}} \leq$ `[operation, external, {class-internal, service-internal, aggregation-internal, system-internal, cross-system}, {name and type, semantics of variable, semantic type, type}]`.

- **Normalized Input-Output Coupling of Methods ($NIOCM$)** The Normalized Input-Output Coupling of Methods ($NIOCM$) is defined as described by (29).

$$NIOCM(op, \{M\}) = \begin{cases} 0 & \text{if} |\{Inp_{op}\}| = 0 \\ 1 - \dfrac{\sum^{i \in \{Inp_{op}\}} isunique^o(i, \{M \setminus \{op\}\})}{|\{Inp_{op}\}|} & \text{otherwise} \end{cases} \tag{29}$$

Mechanism $NIOCM$ sets the count of unique input parameters (that are equivalent to $IOCM$) into a relation with the count of input parameters of the respective

method. By subtracting the normalized value from one, a method that solely uses unique parameters as input gets a $NIOCM$-value of zero.

Value range The value range of $NIOCM$ is [0, 1]

Discussion High $NIOCM$ values indicate a high coupling while low values indicate a low coupling.
By reversing the values (high values for high coupling and vice versa) the idea of satisfying `Coupling.II` is addressed. In mediated scenarios $NIOCM$ satisfies that property. According to $IOCM$, $NIOCM$ also satisfies the properties `Coupling.I` and `Coupling.III` - `Coupling.V`.
$NIOCM$ is applicable for several points within the $\overline{\texttt{MV}}$-space: $\overline{\texttt{MV(NIOCM)}} \leq$ `[operation, external, {class-internal, service-internal, aggregation-internal, system-internal, cross-system}, {name and type, semantics of variable, semantic type, type}]`.

- **Normalized Relative Input-Output Coupling of Methods ($NRIOCM$)** The Normalized Input-Output Coupling of Methods ($NRIOCM$) is defined as described by (30).

$$NRIOCM(op, \{M\}) = \frac{\sum_{i \in \{Inp_{op}\}} \sum_{m \in \{M \setminus \{op\}\}} \sum_{o \in \{Outp_m\}} (\upsilon^o(i, \{M \setminus \{op\}\})}{\sum_{m \in \{M \setminus \{op\}\}} \sum_{o \in \{Outp_m\}} 1} \tag{30}$$

Mechanism $NRIOCM$ sets the count of (re-)usages of all input parameters of a given method within the set of output parameters of all methods in relation with all output parameters of all other methods within the scope. The intersection of input and output parameters of `op` is not included into the count.
Of course, the denominator could be zero for an empty system. For the sake of readability $NRIOCM$ is not noted as a split function. Whenever the denominator is zero, the value of $NRIOCM$ will also be zero.

Value range The value range of $NRIOCM$ is [0, 1]

Discussion Analogous to $RICOM$, high $NRIOCM$ values indicate a highly coupled method while low values indicate low coupling. In order to normalize this metric, $RIOCM$ is set into relation with the count of all output parameters of all methods within the scope excluding the actual method itself. The drawback of such normalization is that a large scope blurs the (re-)use count of the parameters.
According to the discussion outlined for $NIOCM$, $NRIOCM$ satisfies the properties `Coupling.I` and `Coupling.III` - `Coupling.V`. `Coupling.II` is also satisfied for mediated scenarios.
$NRIOCM$ is applicable for several points within the $\overline{\texttt{MV}}$-space: $\overline{\texttt{MV(NRIOCM)}} \leq$ `[operation, external, {class-internal, service-internal, aggregation-internal, system-internal, cross-system}, {name and type, semantics of variable, semantic type, type}]`.

- **Output-Input Coupling of Methods ($OICM$)** The Input-Output Coupling of Methods ($OICM$) is defined as described by (31).

$$OICM(op, \{M\}) = \sum_{o \in \{Outp_{op}\}} isunique^i(o, \{M \setminus \{op\}\}) \tag{31}$$

Mechanism The mechanism of $OICM$ is the same as for $IOCM$ but uses the opposite sets of parameters: it counts unique output parameters with regards to the set of input parameters of the methods within the scope.

Value range The value range of $OICM$ is $[0, +\infty[$

Discussion While $IOCM$ indicates whether a method is dependent on the output of other methods, $OICM$ indicates the same but in the opposite direction. It indicates whether there are methods that use the output parameters of `op` as their input parameters.
Low $OICM$-values indicate high coupling while high $OICM$ values indicate a low coupling of the actual method within the given scope.
Utilizing the same mechanisms as $IOCM$, $OICM$ satisfies the same properties `Coupling.I` and `Coupling.III - Coupling.V`. `Coupling.II` is also not satisfied.
$OICM$ is applicable for several points within the $\overline{\texttt{MV}}$-space: $\overline{\texttt{MV(OICM)}} \leq$ `[operation, external, {class-internal, service-internal, aggregation-internal, system-internal, cross-system}, {name and type, semantics of variable, semantic type, type}]`.

- **Relative Output-Input Coupling of Methods ($ROICM$)** The Relative Output-Input Coupling of Methods ($ROICM$) is defined as described by (32).

$$ROICM(op, \{M\}) = \sum^{o \in \{Outp_{op}\}} \upsilon^i(o, \{M \setminus \{op\}\}) \tag{32}$$

Mechanism As $RIOCM$, $ROICM$ utilizes the principle of overlapping input and output parameter sets and uses the count of occurrences of the single output parameters within the set of input parameters. The intersection of output and input parameters of the actual method is excluded from that count.

Value range The value range of $ROICM$ is $[0, +\infty[$

Discussion Low $ROICM$-values indicate a low coupling while high values indicate high coupling of the actual method. The indicated coupling is also indirect.
As $RIOCM$, $ROICM$ satisfies `Coupling.I`. Also `Coupling.II` is satisfied if all relations among services are mediated by aggregators. In addition $ROICM$ satisfies `Coupling.III - Coupling.V`.
$ROICM$ is applicable for several points within the $\overline{\texttt{MV}}$-space: $\overline{\texttt{MV(ROICM)}} \leq$ `[operation, external, {class-internal, service-internal, aggregation-internal, system-internal, cross-system}, {name and type, semantics of variable, semantic type, type}]`.

- **Normalized Output-Input Coupling of Methods ($NOICM$)** The Input-Output Coupling of Methods ($NOICM$) is defined as described by (33).

$$NOICM(op, \{M\}) = \begin{cases} 0 & \text{if} |\{Outp_{op}\}| = 0 \\ 1 - \dfrac{\sum^{o \in \{Outp_{op}\}} isunique^i(o, \{M \setminus \{op\}\})}{|\{Outp_{op}\}|} & \text{otherwise} \end{cases} \tag{33}$$

Mechanism $NOICM$ is the normalization of $OICM$. In order to indicate low coupling with low values the fraction is deducted from 1.

Value range The value range of $NOICM$ is $[0,\ 1]$

Discussion The values of $NOICM$ are to be interpreted according to $NIOCM$. Low values indicate that hardly any methods within the scope use the same parameters as their respective input parameters as the method `op` uses as output parameters.
Utilizing the same mechanisms as $NIOCM$, $NOICM$ satisfies the same properties `Coupling.I` and `Coupling.III - Coupling.V`. `Coupling.II` is also only satisfied in mediated systems. $OICM$ is applicable for several points within the $\overline{\texttt{MV}}$-space: $\overline{\texttt{MV(NOICM)}} \leq$ `[operation, external, {class-internal, service-internal, aggregation-internal, system-internal, cross-system}, {name and type, semantics of variable, semantic type, type}]`.

- **Normalized Relative Output-Input Coupling of Methods ($NROICM$)** The Normalized Output-Input Coupling of Methods ($NROICM$) is defined as described by (34).

$$NROICM(op, \{M\}) = \frac{\sum\limits_{o \in \{Outp_{op}\}} (v^i(o, \{M \setminus \{op\}\})}{\sum\limits_{m \in \{M \setminus \{op\}\}} \sum\limits_{i \in \{Inp_m\}} 1} \tag{34}$$

Mechanism $NROICM$ sets the count of (re-)usages of all output parameters of a method within the set of input parameters of all methods in relation to all input parameters of all other methods within the scope.

Value range The value range of $NROICM$ is $[0,\ 1]$

Discussion High $NROICM$ values indicate a highly coupled method while low values indicate low coupling. The value is normalized to the value range of $[0,\ 1]$. The extent to which the input parameters of the methods depend on the output parameters of `op` is indicated as well.
According to the discussion outlined for $NRIOCM$, $NROICM$ satisfies the properties `Coupling.I` and `Coupling.III - Coupling.V`. `Coupling.II` is also satisfied for mediated scenarios.
$NROICM$ is applicable for several points within the $\overline{\texttt{MV}}$-space: $\overline{\texttt{MV(NROICM)}} \leq$ `[operation, external, {class-internal, service-internal, aggregation-internal, system-internal, cross-system}, {name and type, semantics of variable, semantic type, type}]`.

- **Normalized Total Coupling of Methods ($NTCM$)** The Normalized Total Coupling of Methods ($NTCM$) is defined as described by (35).

$$NTCM(op, \{M\}) = \begin{cases} 0 & \text{if} |\{Inp_{op}\}| + |\{Outp_{op}\}| = 0 \\ 1 - \frac{OICM(op,\{M\}) + IOCM(op,\{M\})}{|\{Outp_{op}\}| + |\{Inp_{op}\}|} & \text{otherwise} \end{cases} \tag{35}$$

Mechanism $NTCM$ indicates the ratio of both unique input and output parameters of the given method.

Value range The value range of $NTCM$ is [0, 1]

Discussion Instead of analyzing only one part of a methods parameters, $NTCM$ includes all parameters of all methods within the scope into the analysis. High values indicate a high coupling (and few unique parameters) while low values indicate low coupling.
Being a compound of $NIOCM$ and $NOICM$, $NTCM$ satisfies the properties `Coupling.I` and `Coupling.III - Coupling.V`. `Coupling.II` is also satisfied for mediated scenarios.
$NTCM$ is applicable for several points within the $\overline{\texttt{MV}}$-space: $\overline{\texttt{MV(NTCM)}} \leq$ `[operation, external, {class-internal, service-internal, aggregation- internal, system-internal, cross-system}, {name and type, semantics of variable, semantic type, type}]`.

- **Normalized Relative Total Coupling of Methods ($NRTCM$)** The Relative Total Coupling of Methods ($NRTCM$) is defined as described by (36).

$$NRTCM(op, \{M\}) = \begin{cases} 0 & \text{if} |\{Inp_{op}\}| + |\{Outp_{op}\}| = 0 \\ \frac{NRIOCM(op,\{M\}) \times |\{Inp_{op}\}|}{|\{Inp_{op}\}| + |\{Outp_{op}\}|} + \frac{NROICM(op,\{M\}) \times |\{Outp_{op}\}|}{|\{Inp_{op}\}| + |\{Outp_{op}\}|} & \text{otherwise} \end{cases} \tag{36}$$

Mechanism $NRTCM$ is the weighted sum of $NRIOCM$ and $NROICM$ of a given method `op` within the specified scope $\{M\}$.

Value range The value range of $NRTCM$ is [0, 1]

Discussion $NRTCM$ is the summarizing candidate metric for coupling. As both $NIOCM$ and $NOICM$ are normalized, it is possible to weight these single metrics to indicate the total coupling of a method. Low values indicate low coupling while high values indicate high coupling.
According to $NRIOCM$ and $NROICM$, $NRTCM$ satisfies the properties `Coupling.I` and `Coupling.III - Coupling.V`. `Coupling.II` is also satisfied for mediated scenarios.
$NRTCM$ is applicable for several points within the $\overline{\texttt{MV}}$-space: $\overline{\texttt{MV(NRTCM)}} \leq$ `[operation, external, {class-internal, service-internal, aggregation-internal, system-internal, cross-system}, {name and type, semantics of variable, semantic type, type}]`.

Legner et al. describe that the granularity of a service should be "oriented towards business suitability" [87, p. 6]. This is of course a subjective design principle. Furthermore it does not give any support to the definition of granularity. Erradi et al. approach the quantitative analysis of service granularity. They state that granularity can be quantified using "the number of components that are invoked through a given operation on a service interface [and] the number of resources' state changes like the number of database tables updated" [79, p. 157].
Measuring the number of state changes a service causes from an outside point of view is not possible. This is why the presented granularity metrics – similarly to the presented cohesion and coupling candidate metrics – focuses on elements of the public structural interface of services.

- **Signature Size of Method (SSM)** The Signature Size of Methods (SSM) is defined as described by (37).

$$SSM(op) = |\{Inp_{op}\}| + |\{Outp_{op}\}| \tag{37}$$

Mechanism SSM is a count of both the input parameters and output parameters of a method.

Value range The value range of SSM is $[0,\ +\infty[$

Discussion SSM simply indicates the count of parameters of a method. Obviously low SSM-values indicate a low granularity while high values indicate a high granularity. Important is the specification of the $\overline{\texttt{MV}}$-space in which the metric is applied. This is because the SSM-value can vary if, for example, not types but semantics of parameters are measured. This is why SSM should be used very carefully.
Being a size metric, SSM needs to be evaluated against the six properties a size metric should – according to [58] – satisfy. A more detailed description of these properties is given in section 3.2.1.
The property `Size.I` a size metric should satisfy is *non-negativity*. It is obviously satisfied by SSM.
`Size.II` is the `Null Value` property that demands a metric value of 0 for empty systems. This property is also satisfied as SSM is a count of elements of a system.
`Size.III` is the module additivity property. In order to prove additivity for SSM it is necessary to consider a method as a system and its properties as its elements. The joint SSM value for two methods op_1 and op_2 can then be calculated as $|\{Inp_{op_1}\} \cup \{Inp_{op_2}\}| + |\{Outp_{op_1}\} \cup \{Outp_{op_2}\}|$. Since by definition the methods are disjoint, the above calculation can be resolved to $|\{Inp_{op_1}\}| + |\{Inp_{op_2}\}| + |\{Outp_{op_1}\}| + |\{Outp_{op_2}\}|$. Hence, SSM satisfies `Size.III`.
As the properties `Size.IV - Size.VI` follow from the satisfaction of properties `Size.I- Size.III` SSM also satisfies them.
SSM is applicable for several points within the $\overline{\texttt{MV}}$-space: $\overline{\texttt{MV(SSM)}} \leq$ `[operation, external, {class-internal, service-internal, aggregation-internal, system-internal, cross-system}, {name and type, semantics of variable, semantic type, type}]`.

- **Signature Size Deviation of Method ($SSDM$)** The Signature Size Deviation of Methods ($SSDM$) is defined as described by (38).

$$SSDM(op, \{M\}) = \begin{cases} 0 & \text{if } SSM(op) = 0 \\ \dfrac{(SSM(op) - \sum_{}^{m \in \{M\}} \frac{SSM(m)}{|\{M\}|})^2}{SSM(op)} & \text{otherwise} \end{cases} \tag{38}$$

Mechanism According to the variance of variates, $SSDM$ indicates the distance between an actual method's granularity (expressed by SSM) and the mean granularity within the scope. This value is set into a relation with the method's granularity in order to achieve comparability for various levels of granularity.

Value range The value range of $SSDM$ is $[0,\ +\infty[$

Discussion As stated by [79] services may be "offered at different levels with different granularity. High-level business process functionality is externalized as coarse-grained services for cross-LOB [line of business], cross-channel or external access.

These are realized by composing fine-grained services harvested from existing custom/legacy systems or packages" [79, p. 156]. Besides the implication of a service meta-model, this quotation outlines that there is an appropriate granularity of services (and their methods) according to the scope in which the granularity is measured.
If design principles are used in order to fix the general design of single services and methods, *SSDM* can be used as an indicator that describes whether the actual method fits into the overall granularity that is used for methods in the scope. However, the definition of that scope is crucial. It should be aligned with a service meta-model in order to include services with equal scopes into the calculation.
As *SSDM* does not measure the actual granularity of a method it is not considered to be a size metric. Thus, no properties are checked for this candidate metric.
SSDM is applicable for several points within the $\overline{\texttt{MV}}$-space: $\overline{\texttt{MV(SSDM)}} \leq$ `[operation, external, {class-internal, service-internal, aggregation-internal, system-internal, cross-system}, {name and type, semantics of variable, semantic type, type}]`.

As the above discussion revealed, not all metrics satisfy all desiderata for their respective class of metric. This is due to the fact that all metrics were designed with focus on eased applicability. As a consequence, they can be applied in the following to a productive application. However, as the metrics do not all satisfy all desiderata, applying them can only be an approach towards the assessment of whether the design principles they represent have an impact on reusability of services.

4.2 Introduction to the Case Study

In order to investigate whether the presented candidate metrics – and the concepts they stand for – are applicable for designing reusable services, they are applied to a real-life project that was conducted independently of this research or any other SOA activities within BASF IT Services.
The project is a multi-million dollar project that aims to unify the processes of organizational changes. For the sake of nondisclosure it is unfortunately not possible to describe the processes in complete detail. However, the general software architecture can be outlined and the services can be roughly described.

The project consists of three layers: a user interface, a web service layer and various back-end systems. The user interface acts as mediator and service orchestrator. All process steps are backed up by one method of a web service.
The web services are exposed by the web service layer. No integration server is used. Instead, J2EE components are exposed as web services. The exposure takes place by using dedicated web service classes. These classes transfer the data via data transfer objects to the session-bean layer of the J2EE part. This part in call interacts with the back-end systems. In most cases this is achieved by accessing the databases of the respective back-end systems. This architecture is outlined in figure 5.

The single web services are groupings of methods that are required by each process. Hence, there is no dedicated grouping for reuse nor is there any actual reuse. The web service layer is dedicated for each process. However, there is common functionality. Some methods

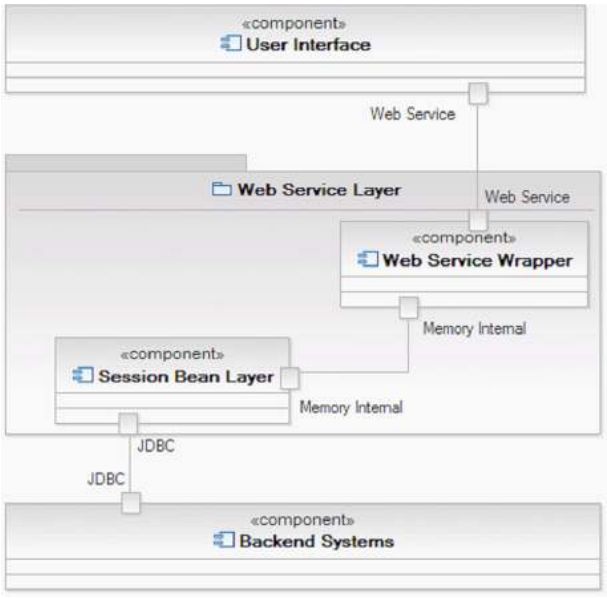


Figure 5: Architectural Sketch of the Application System

are used by multiple processes. These methods have the same name and signature as the corresponding methods of other services. Their implementation on the level of session beans is also reused.

The web services are named based on the respective process they support. Their methods correspond to the process steps in the user interface. Unfortunately, it was determined that every method has its own input and output parameters. However, same methods have the same parameters and re-used methods can be identified.
Such a method looks like the following:

```
approveNewEmployeeReturn approveNewEmployee(ApproveNewEmployeeIn)
```

By breaking down the complex types of all methods there are only elements of the type `String`. Fortunately, the elements of all types are formed using the same naming convention. Hence, by considering the name of an element, the respective semantics of that element is unambiguous. Therefore, in order to analyze the given implementation, all metrics needed to be applicable in the $\overline{\mathtt{MV}}$-space at a point that satisfies these requirements: `[operation, {internal, external}, {class-internal, service- internal, aggregation-internal, system-internal, cross-system}, semantics of variable]`.

In total, 87 processes are supported by the application system. Each process consists of three to 20 steps. In order to apply the presented candidate metrics both, the source code as well as the WSDL-files of the web services were considered. Randomly, 81 methods were chosen out of 12 processes. Whenever two methods had the same name they were considered reused. For every method all candidate metrics were calculated. The metrics $LCOM$ and $LCOM^*$ were calculated both for the session beans as well as for the data access objects (DAO) using the tool eclipse metrics [85]. All other candidate metrics were manually calculated using a spreadsheet. Based on this data, the discriminative power of the candidate metrics was tested.

As the application does not incorporate service-oriented principles such as service aggregation, the coupling metrics that were applied have to be interpreted accordingly.

4.3 On the Candidate Metrics' Discriminative Power

		Statistics			
				Valid Values	
REUSE		Mean	Standard Deviation	Not Weighted	Weighted
0	UM	,966	,680	29	29,000
	NUM	,188	,187	29	29,000
	RUM	100,069	52,533	29	29,000
	NRUM	,213	,107	29	29,000
	IOCM	,828	1,365	29	29,000
	NIOCM	,760	,334	29	29,000
	RIOCM	20,172	10,684	29	29,000
	NRIOCM	4,827E-02	3,547E-02	29	29,000
	OICM	2,655	2,663	29	29,000
	NOICM	,160	,252	29	29,000
	ROICM	8,483	16,232	29	29,000
	NROICM	1,193E-02	2,181E-02	29	29,000
	NTCM	,499	,242	29	29,000
	NRTCM	3,803E-02	2,509E-02	29	29,000
	SSM	6,724	4,503	29	29,000
	SSDM	5,728	6,026	29	29,000
	pubBean_LCOM	,000	,000	29	29,000
	pubBean_LCOM*	,172	,344	29	29,000
	DAO_LCOM	,000	,000	29	29,000
	DAO_LCOM*	,000	,000	29	29,000
1	UM	2,688	4,827	16	16,000
	NUM	,227	,254	16	16,000
	RUM	143,313	78,223	16	16,000
	NRUM	,200	,108	16	16,000
	IOCM	1,500	2,309	16	16,000
	NIOCM	,809	,247	16	16,000
	RIOCM	27,563	26,934	16	16,000
	NRIOCM	3,943E-02	2,757E-02	16	16,000
	OICM	4,750	5,825	16	16,000
	NOICM	,195	,245	16	16,000
	ROICM	19,938	35,252	16	16,000
	NROICM	1,540E-02	2,607E-02	16	16,000
	NTCM	,545	,214	16	16,000
	NRTCM	3,872E-02	1,890E-02	16	16,000
	SSM	12,875	9,858	16	16,000
	SSDM	6,968	6,155	16	16,000
	pubBean_LCOM	,000	,000	16	16,000
	pubBean_LCOM*	,000	,000	16	16,000
	DAO_LCOM	,000	,000	16	16,000
	DAO_LCOM*	,000	,000	16	16,000

Table 9: Group Statistics

Out of the 81 methods that were randomly chosen, 16 were reused[21] and 29 were not reused. This means that the average use of a reused method was 3.25. It was verified that the 29 methods that were not reused were also not reused outside of the drawn sample. Based on these values, a discriminant analysis was performed.[22] Two groups were defined. The first group (0) contained all methods that were not reused (used only once). The second group (1) contained all methods that were used twice or more often.
For the methods of both groups, all the previously-described candidate metrics were calculated. Methods represent the population of the analysis while the single candidate represents the dependent attributes. For each group the mean and standard deviation was calculated. These values are shown in table 9.

[21] Reused methods were counted as often as they were used.
[22] SPSS version 10.0.5

Test of Function	Wilks-Lambda	Chi-Square	df	Significance
1	,462	27,002	16	,041

Table 10: Test of Discriminant Function based on Wilks' Lambda

In order to analyze whether there was a significant difference between the two groups of methods, a discriminant analysis was performed (Wilks' Lambda). The analysis showed that there was a significant discriminator that varied between the two groups ($p = 0.041$). The qualitative properties of the discriminant function are shown in table 10.

In order to determine the significant attributes, an analysis of the means of the two groups was performed. The results of this comparison of the two groups are shown in table 11. This indicates that the two candidate metrics that are mainly responsible for the difference between the two groups are SSM ($p = 0.006$) and RUM ($p = 0.032$).

	Wilks-Lambda	F	df1	df2	Significance
UM	,922	3,628	1	43	,064
NUM	,992	,365	1	43	,549
RUM	,898	4,904	1	43	,032
NRUM	,997	,149	1	43	,701
IOCM	,966	1,517	1	43	,225
NIOCM	,994	,258	1	43	,614
RIOCM	,962	1,720	1	43	,197
NRIOCM	,983	,744	1	43	,393
OICM	,940	2,750	1	43	,105
NOICM	,995	,202	1	43	,655
ROICM	,951	2,236	1	43	,142
NROICM	,995	,228	1	43	,635
NTCM	,991	,401	1	43	,530
NRTCM	1,000	,009	1	43	,925
SSM	,839	8,281	1	43	,006
SSDM	,990	,430	1	43	,515
pubBean_LCOM	,[a]				
pubBean_LCOM*	,915	3,984	1	43	,052
DAO_LCOM	,[a]				
DAO_LCOM*	,[a]				

Table 11: Test of Equality – Group Mean Values

A step-wise procedure was performed next to determine the most significant discriminant function coefficient. The single steps select the most significant candidate metrics according to their discriminant power. This was measured by calculating the Wilks' Lambda value for each candidate metric in every step. The single step for the selection of candidate metrics were:

1. Choose candidate metrics with the lowest Wilks' Lambda value that satisfies the acceptance criterion (F-value of at least 3.84).
2. Choose candidate metrics with the second lowest Wilks' Lambda value that satisfies the acceptance criterion. If the combination of the first and the second candidate metric have significant discriminant power, both candidate metrics are accepted. If not, the first candidate metric is rejected.
3. The procedure stops as soon as no other candidate metrics satisfies the acceptance criterion.

Step		Tolerance	F-value for exclusion
1	SSM	1,000	8,281

Table 12: Variables Included in the Analysis of Step 1

A summary of the (only) step for the presented candidate metrics is shown in table 12 and table 13. Table 12 indicates that SSM is considered in the first step for inclusion. Table 13 shows the candidate metrics that were excluded from this step.
According to this step-wise analysis the most significant candidate metric that discriminates both groups of methods is SSM. Including additional candidate metrics into the analysis does not improve the significance of the discriminate function.

The raw data for the analysis can be found in appendix B.

4.4 Conclusion

According to the step-wise procedure and the comparison of group mean values, two candidate metrics have a certain discriminant power in terms of reusability: RUM and SSM. This means that none of the coupling metrics introduced showed significance on the reuse of a service method.

Granularity According to the statistics outlined above, the size of a method's signature influences the reusability significantly. For the group of methods that were not reused at all, the mean SSM-value is 6.7 while the group of reused methods shows a mean of 12.9. This means that methods of services should be cut in a way that allows them to compute several parameters. As there was no significance for the $SSDM$ metric, it can not be stated whether there is an optimal distribution of input and output parameters of a method.
Interesting to note is that more coarse-grained methods and services are likely to decrease the overall amount of services. Even if not proven statistically, this is an important yet trivial principle to consider when designing services: the less services exist within an organization, the more easily the landscape of services can be managed.

Cohesion Cohesion also showed significance for re-usability. RUM, in particular, possesses the discriminant power to differentiate between reused methods and methods that are not reused. The mean RUM-value for reused methods is 143.3 while the mean value for methods that are not reused is 100. Hence, reused methods are less unique. Following the argumentation from [60], reused methods would be less cohesive than methods that are used only once.
This analysis contradicts the assumption described about cohesion. It can be stipulated that a service method is more likely to be re-used if it uses common parameters, though. This is an indicator for the benefit of using a global type system rather than designing service methods by the principle of cohesion.

In sum, it can be said that the analysis of the accessible data did support the assumptions about the impact of cohesion and coupling that are widely described.
Only the accepted fact that service methods should be coarse grained proved to have significant influence on reusability. This leads to the conclusion that "over engineering"

Step		Tolerance	Minimal Tolerance	F-value for inclusion	Wilks-Lambda
0	UM	1,000	1,000	3,628	,922
	NUM	1,000	1,000	,365	,992
	RUM	1,000	1,000	4,904	,898
	NRUM	1,000	1,000	,149	,997
	IOCM	1,000	1,000	1,517	,966
	NIOCM	1,000	1,000	,258	,994
	RIOCM	1,000	1,000	1,720	,962
	NRIOCM	1,000	1,000	,744	,983
	OICM	1,000	1,000	2,750	,940
	NOICM	1,000	1,000	,202	,995
	ROICM	1,000	1,000	2,236	,951
	NROICM	1,000	1,000	,228	,995
	NTCM	1,000	1,000	,401	,991
	NRTCM	1,000	1,000	,009	1,000
	SSM	1,000	1,000	8,281	,839
	SSDM	1,000	1,000	,430	,990
	pubBean_LCOM	,000	,000	.	.
	pubBean_LCOM*	1,000	1,000	3,984	,915
	DAO_LCOM	,000	,000	.	.
	DAO_LCOM*	,000	,000	.	.
1	UM	,898	,898	,889	,821
	NUM	,961	,961	1,175	,816
	RUM	,449	,449	,011	,838
	NRUM	,660	,660	2,071	,799
	IOCM	,895	,895	,082	,837
	NIOCM	,994	,994	,436	,830
	RIOCM	,848	,848	,035	,838
	NRIOCM	,749	,749	,367	,831
	OICM	,472	,472	,324	,832
	NOICM	,962	,962	,011	,838
	ROICM	,752	,752	,004	,838
	NROICM	,954	,954	,016	,838
	NTCM	,980	,980	,901	,821
	NRTCM	,937	,937	,583	,827
	SSDM	,994	,994	,631	,826
	pubBean_LCOM	,000	,000	.	.
	pubBean_LCOM*	,948	,948	1,551	,809
	DAO_LCOM	,000	,000	.	.
	DAO_LCOM*	,000	,000	.	.

Table 13: Variables NOT Included in The Analysis of Step 1

services with regards to their possible reusability will provide little benefit. Hence, a service design methodology has to provide fast solutions in a problem-oriented manner. While supporting fast design of services, this methodology should also respect the following principles:

- Design services for a given use case. Do not design them for reuse.
- Analyze existing services before designing new ones. If reuse is possible, reuse services.
- Keep services (especially their methods) coarse-grained.
- Include different levels of services. Each level should have its own level of granularity.
- Use a common type system. All services should use the same types, names and semantics for their parameters.

The design methodology for composite applications that is discussed in chapter 6 includes an approach for deriving suitable services. This service design methodology incorporates the principles that have been identified by the analysis of this chapter.

5 Standardizing the Application of the Service-Oriented Style: A Reference Architecture for Business Process-Oriented Composite Applications in Heterogeneous Application Landscapes

The discussion in chapter 2 has shown that the difference between the service-oriented and the component-oriented architectural style is minimal. The advancement made by the introduction of service orientation is, however, crucial for the applicability of the concept. On one hand, the eased understandability of protocols and platforms (such as web services [36]) can not be underestimated for the success of an architectural style. More interesting is the explicit platform neutral definition of concepts that are important in the context of industrial and heterogeneous software landscapes, though.
Concluding the analysis of the service-oriented architecture in chapters 2 and 3, this architectural style is beneficial if the functionality of back-end systems can be reused in flexible, business process-oriented applications. Such composite applications are built using services that are exposed by back-end systems.

Several approaches exist that aim to provide means for structuring composite applications. They all describe a meta-model for a *software architecture* (cf. [88]) that incorporates service-oriented principles to a certain extent. [11], for instance, proposes the use of different types and levels of services to build composite applications. The idea of using a business process-like service orchestration is also included in this reference architecture. A service interface layer, an orchestration layer, a business service layer and an application service layer are described. The service interface layer is in between the business process layer and the application layer. It describes a service mediation layer. There, emphasis is put on abstraction. Application services describe functionality exposed by the application layer while business services represent functionality that is used to reach business goals by orchestrating in alignment with a business process.
Such a business process-oriented orchestration on top of services employs the idea of control-centralization that is a major advantage of the service-oriented style. In order to relate the orchestrated services with back-end services, multiple layers of service composition are proposed. [89] and [81] also define a reference architecture with multiple levels of service compositions. So-called macro- and micro-workflows are used to reuse functionality of back-end systems through services. Emphasis is put on the possible realization of the workflow engines that are used to orchestrate the single services.
These models are conceptional frameworks that are meant to provide an understanding of how composite applications should be designed. As outlined in [15], they lack, however, a description of appropriate integration mechanisms that are a fundamental element of service-oriented systems (cf. [15, p. 68]).
Since this is not their purpose, they also lack a more detailed description of the single layers. These approaches neither handle the issues of data, context nor transaction handling as they were described as necessary in section 3.2.2.
This is because these approaches include best practices for designing composite applications. They do not describe how a composite application should be implemented.
An approach that includes implementation aspects is the composite application framework WS-CAF [30]. Here, transactional security and separation of long-running and short-running processes are incorporated. This approach describes a set of protocols that

a runtime environment might use. It does not incorporate design principles of the service-oriented paradigm at all. Furthermore, it is only platform independent on a very general level: all platforms implementing the protocols are compliant.

In order to provide a structure that allows industry companies to leverage the advantages of the service-oriented architectural style in the context of heterogeneous application landscapes, a reference architecture that helps both, to design and implement composite applications is presented in this chapter. The underlying design concepts are similar to the ones of [11], [89], and [90] — they use multiple levels of abstraction in order to separate the concerns of composite applications. In the approach presented here, special attention is put on the ability of composite applications to utilize distributed and heterogeneous application systems that are not necessarily accessible by the means of services.
The presented concepts are defined independently from any technology that may be used to realize composite applications. This way, a development methodology becomes possible that can be used to design composite applications without being too limited by the constraints of a platform for central components. Further, the reference architecture is additionally designed in a way that a top-down methodology of composite application development is feasible, service-oriented principles are supported and the resulting artifacts can be used for realizing a composite application. This methodology is presented in chapter 6.
In terms of the Model Driven Architecture, the reference architecture that is described below is twofold. It describes first a meta structure for platform-independent models of a software architecture. Second, it contains a functional description as well as requirements for a *virtual machine* that is used to execute a platform-specific model (cf. [91, p. 2-6]).

5.1 Outline of the Architecture

The *soft* design principles of the service-oriented architectural style that were identified in chapter 2 are *loose coupling*, *autonomy*, *abstraction* and *reusability*. As described above, reference architectures for service-oriented applications usually incorporate abstraction by the notion of several *layers*. The reference architecture introduced here also incorporates this principle. It uses a central service orchestration layer that aggregates services exposed by less abstract layers. The reference architecture allows for a business process-based definition of this orchestration. For this sake it abstracts from "technical" details. It includes a data repository for context handling that also participates in the management of distributed transactions. This, of course, increases the extent of coupling between the central elements of the composite application (cf. [6, pp. 185ff]). This is necessary to create efficient composites that can be deployed on arbitrary platforms (cf. [6, pp. 191ff]). The notion of an eventing system is used for consistency management within the context and for de-coupling the different processes and tasks that are possibly supported by one composite application.
By connecting these components, the reference architecture defines a second layer of service aggregation — the *service coordination layer*. On this layer, if required, orchestrated entities from the top layer can be described as aggregations of functionality that is exposed by systems of the application landscape. It provides a mechanism for the *mediation* of the services. This coordination includes the "details" of transactional management and connects application systems either directly to the composite using the common service

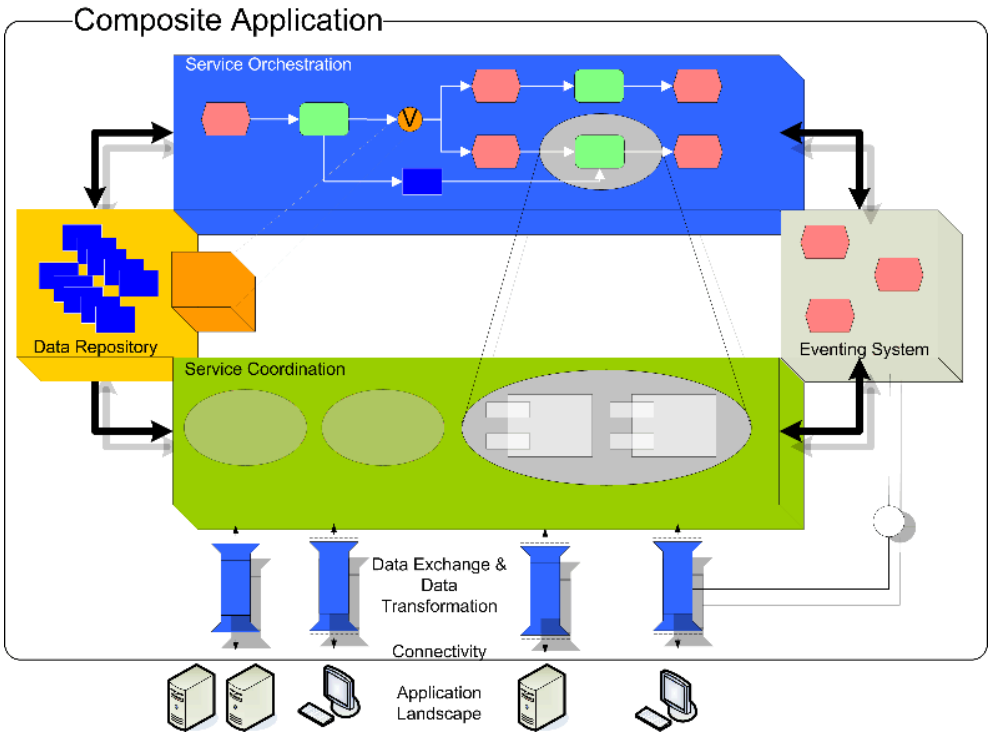


Figure 6: Reference Architecture for Composite Applications

protocol or by using the *data exchange and data transformation layer*. This layer ensures, together with the layer of *connectivity*, various ways of interactions with back-end systems while abstracting from communication semantics.

This reference architecture allows for defining service boundaries in a business-driven way. Together with the design methodology of chapter 6, a business process and its functions can be used to define the services for the single layers. The boundaries of these services are determined by the business descriptions. This way, the principles of *autonomy* (cf. chapter 2) and *intersection points* (cf. [15]) are incorporated.
As described in chapter 2, the use of aggregators promotes loose coupling. This is because the interaction necessary for the provisioning of a service is encapsulated by them. By including the layer of service coordination, this idea is incorporated into the reference architecture. It encapsulates application system-specific coordination from the process orchestration and includes loose coupling this way. The dedicated eventing system used to manage processes also decouples the components of a composite that is aligned with the reference architecture. By the means of the data exchange and data transformation layer, loose coupling is promoted by the means of validating the actual content of service interactions (cf. [6, p. 192]).
By providing the technical circumstances for dividing business-focused service consumers (the orchestration), the architecture also allows a distinguished service meta-model that, together with the findings of chapter 4, allows for the design of reusable services.
Finally, the data exchange and data transformation layer allows for another principle of service orientation: reusing the functionality of back-end systems. The aspect of *reusability* is then addressed by the service design methodology of chapter 6 that is based on the reference architecture.

5.2 Events

Business processes can be seen as event-driven computations of data (cf. [92]). In the context of a business process-controlled service-oriented architecture, this simply means that both, the data as well its computation are dependent on events. In order to outline how the reference architecture allows composite applications that are based on it to operate correctly on consistent data, the understanding of events as they underlie the architecture is first presented.

Starting from scratch, an event can be defined as anything that happens. In a composite application, an event is anything that happens and that is noteworthy for the composite application. For instance, the change of the state in an application system that provides functionality for a composite application is an event. Further, an event is always a result of preliminary events that caused the change of the state. A relevant event as it is defined here is if a certain state within a connected application system is reached and that this situation has to be treated by a business process that is supported by a composite application. Consequently, the understanding of an event is that it is an aggregation of things that *happened* within an integrated system. Hence, an event is an aggregation of *pre-events*.

Ferstl et al. describe integration as the task of integrating tasks (cf. [93, p. 220]). In [93] a task is described from an outside view by events that cause the need for performing a task, goals to be achieved by the task and a set of data called *task object* that is concerned with the task's performance. A task is performed by executing a *solution procedure* that operates on the task object (cf. [93, pp. 89ff.]). A task is determined by a set of pre-events. Transitively the solution procedure and the task object are also determined (cf. [93, p. 91]). As business processes describe the central control flow of composites, the solution procedure is considered a certain business process. Thus, an event identifies a business process.

In order to make use of this interpretation, an event needs to be described in greater detail. Since the above argumentation involves states and changes of states, an understanding of a system as a state space system is implied. A system Ω can be seen as a set of possible state transitions within the Cartesian product of all states Z. Thus, a system Ω is defined as $\Omega \subseteq Z \times Z$ (cf. [93, p. 15]). Since a system is a set of transitions, an event E — that is identified to be a state transition — is an element of a relation on Z: $E \in Z \times Z$.
This understanding of an event is not operational. In order to use this definition of an event, it would need to identify two states of an application system — which means two sets of certain values occurring in an application system's memory — as an event that determines a solution procedure and therefore a business process within a composite application.
By using abstraction, this issue can be overcome. Thus, events should be described by types of events. A type of event is simply a set of different events indicating *similar* state transitions. Essentially, this makes a type of an event named T being also a relation on Z: $T \subseteq Z \times Z$.
Consequently, an event type describes the actions which are to be performed and, more importantly, what set of data is possibly affected.

Taking these ideas into account, some relations between event types can be used to describe relations between sets of actions and data that are modified by those actions. Such

relations allow for the definition of rules that can support the design of composite applications.

5.2.1 Event Relations

Our model assumes four types of relations between event types: *Concurrent* ($\|$), *blocks* ($\perp$), *updates* ($\vdash$) and *follows* ($\triangleright$). *Concurrent* means that actions of instances of processes computing events of those types compute on disjoint task objects. By defining $a \parallel b$, it is expressed that a process A computing an event of type a will not modify data that is needed by process B and vice versa. This is the default relation between event types.
The relations $\perp$ and $\vdash$ are only reasonable between event types that lead to actions, which are changing the data used by actions that are computing the other type of event. $a \perp b$ states that a process B handling an event of type b can not be started while a process A that handles an event $\vec{a}$[23] is processed. Hence, $a \perp b$ implies that $\vec{b}$ can not be handled while $\vec{a}$ is active. This is because the changes that A is going to perform would make further processing of $\vec{b}$ impossible.[24] This relation has to be used whenever the computation of events *could* affect each other in an unacceptable way.
The relation $a \vdash b$ implies that actions that are performed in order to compute $\vec{b}$ have to make use of the state of A, indicating that the state is unidirectionally shared between A and B. This relation can also occur bidirectional, meaning that A and B share their states. The benefit of having such a relation is that loosely coupled services that have disjoint memories – similar to different process instances – can easily share data without sophisticated communication. The definition of this relation among different event types changes the communication paradigm for the set of affected data from *messaging* to *spaces* (cf. [94]).

Relation	reflexive	irreflexive	symmetric	antisymmetric	transitive	total
$\parallel \subseteq \Theta \times \Theta$	$\emptyset$	$\surd$	$\surd$	$\emptyset$	$\surd$	$\emptyset$
$\perp \subseteq \Theta \times \Theta$	$\surd$	$\emptyset$	$\emptyset$	$\emptyset$	$\surd$	$\emptyset$
$\vdash \subseteq \Theta \times \Theta$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\surd$	$\emptyset$
$\triangleright \subseteq \Theta \times \Theta$	$\emptyset$	$\surd$	$\emptyset$	$\emptyset$	$\surd$	$\emptyset$

Table 14: Properties of Relations

The *updates* relation is an *optimistic* relation. This is because the actions that compute events whose types are set into an *update* relation, can share the data read but should not write the same data. This is because writing to the same data could lead to race conditions and data inconsistency.
The *follows* relation indicates that an event of type $\tilde{a}$ is generated because of the processing of an event of type a: $a \triangleright \tilde{a}$. This causal relation is needed to allow a fine-granular distinction of event types. An example of an event of type $\tilde{a}$ could be the generation of an invoice by an ERP-system because of the arrival of an order (event of type a).
Considering again Z as all possible states, an event type T as $T \subseteq Z \times Z$ and the set of all event types Θ as $\Theta \subseteq \mathrm{Z} \times \mathrm{Z}$ ($T \subseteq \Theta$), table 14 shows the properties of the described

[23] $\vec{a}$ is an event of type a.
[24] Therefore an already running process instance B will have to fail if a process A becomes active.

relations.[25]

One event type determines one (sub-) process. Hence, a relation called *triggers* ($\gg$) between an event type and a process type is introduced. $\gg$ is a surjection from the set of all event types Θ into the set of all business processes types Φ so that $\gg \subseteq \Theta \times \Phi$. Hence, $a \gg X$ implies that an event of type a triggers a process of type X. Alternatively, one can say that whenever an event of type a occurs, a process $\vec{X}$ of type X has to be executed. This relation can be used together with the relations between event types in order to define simple rules that support the definition of guidelines for designing business processes. These rules are described in table 15.

Nr.	Rule name	Rule
I.	*Process Unambiguity*	$(a \triangleright c) \Rightarrow ((a \gg X) \wedge (c \gg X))$
II.	*Single Event*	$((a \gg X) \wedge (c \gg X)) \Rightarrow$ $((a \triangleright c) \vee (c \triangleright a) \vee (a \perp c) \vee (c \perp a) \vee ((a \vdash c) \wedge (c \vdash a)))$
III.	*Same Process Replication*	$((a \vdash c) \wedge (d \vdash c)) \Rightarrow ((a \gg X) \wedge (c \gg X) \wedge (d \gg X))$

Table 15: Process Rules

Rule number one, *Process Unambiguity*, states that events of two causally related event types have to be processed by the same type of process. Rule number two, *Single Event*, states that two events that are to be computed by the same process are either causally related (and therefore executed one after another), blocking each other or updating their states bidirectionally. In consequence, one process can only process one event at a time or the memory of the processes must be shared. That achieves strict or sequential consistency for the process' private memory (cf. [71, pp. 299ff.]). This is because rule III defines that memories can only be shared between services of the same process type.

The introduced definitions and relations also allow for the definition of rules between event types only. Most of the rules that could be defined are dependent on the semantics of the business processes. Hence, the procedure of implementing a composite application should also include the rules that describe relations only between event types. Either way, some rules can be defined independently of actual scenarios.
However, some rules are independent of any actual scenario. These rules, which can be predefined, include rules that involve the set of data that is modified by processes. Although they are general rules, most of them target data consistency.
The scenario-independent rules are introduced in table 16.

Rule IV defines relations $\|, \perp$ *and* $\vdash$ as being mutually exclusive. Rule V states that each two event types have to be in a relation to each other. The default relation is $\|$.

By using rule VI, *Unambiguity* an unambiguous replication between the data repository instances of different composite applications can be assured if required. Only instances are addressed because rule III prescribes that all events that lead to a replication of memories need to be handled by the same type of process.
The rules *Serialization* (VII) and *Replication* (VIII) are needed to guarantee data consistency within a single composite application that realizes one process.

[25]Note that one event type can be defined to be in an *updates* relation with itself. This is, however, not the default.

Nr.	Rule name	Rule
IV.	*Exclusiveness*	$\forall_{\otimes\in\{\parallel,\perp,\vdash\}}.\forall_{\odot\in\{\parallel,\perp,\vdash\}}.(a \otimes c) \Rightarrow \neg(a \odot c);\ (\otimes \neq \odot)$
V.	*Totality*	$\neg((a \otimes b) \vee (b \otimes a)) \Rightarrow (a \parallel b);\ (\otimes \in \{\perp, \vdash\}, (a \neq b))$
VI.	*Unambiguity*	$((a \vdash c) \wedge (x \vdash c)) \Rightarrow ((a \vdash x) \wedge (x \vdash a))$
VII.	*Serialization*	$(a \perp c) \Leftrightarrow (c \perp a)$
VIII.	*Replication*	$(a \vdash c) \Leftrightarrow (c \vdash a)$

Table 16: Event Type Rules

Events are important for the control of data consistency. In order to allow for a flexible realization of a composite application on any platform, the context of a process is explicitly kept as a shared memory for all elements of a composite application. This shared memory – the so-called data repository that is described in chapter 5.6 – ensures data consistency and potentially reduces communication overhead. In order to effectively realize a data repository, an identifier of the relevant data is required. This identifier is introduced to the reference architecture by the means of `Event`-objects. They can be seen as *tickets* or message *slips* (cf. [95, pp. 301ff.]).
An `Event` is a data structure that represents business events within the composite application. Events are categorized using event types. In order to capture this relation an `Event` is associated with an `EventType`. `Event`-objects as well as `EventType`-objects are identified by the use of unique numbers. These numbers are part of an event's state. These identifiers are generated respectively and put into the state the moment the `Event` is generated. This is done using a special component of the reference architecture — the eventing system that is introduced in chapter 5.5.

5.2.2 Realizing Data Visibility using Event Types and Relations

The representation and use of data in workflows is described in [96] through the notion of workflow data patterns. Those patterns are categorized into four clusters. One of them is the *data visibility patterns* (cf. [96, p. 358]). Seven different scopes of visibility have been identified there. Six of those scopes can be realized by an appropriate definition of event type boundaries and event relations. Since a composite application that is formed by this reference architecture constitutes a workflow system, these data patterns might be required for the realization of an actual composite application. The notions of event types and a data repository are suitable for creating some of those patterns. How the single data visibility patterns can be realized is described in the following.

- **Task Data** "Data elements can be defined by tasks which are accessible only within the context of individual execution instances of that task" [96, p. 359]. Considering a service orchestration as the workflow and every orchestrated service (enterprise service; cf. section 6.1) as one task, by defining one event type per enterprise service, the respective data is only visible within the services that are aggregated to an enterprise service (coordination services and application services; cf. section 6.1). If the visibility should be further restricted to single services that are aggregated, additional event types need to be defined.[26]

[26] This weakens the principle of centralizing control into a service orchestration. This is why it is not recommended to restrict the data visibility further than to enterprise services.

- **Block Data** "Block tasks [...] are able to define data elements which are accessible by each of the components of the corresponding sub-workflow" [96, p. 360]. If the event type boundaries are solely managed by the service orchestration, this is the default behavior. As described for the *Task Data* pattern, if one event type is defined per enterprise service the *Block Data* pattern is realized.

- **Scope Data** "Data elements can be defined which are accessible by a subset of the tasks in a case" [96, p. 362]. By simply defining less event types than enterprise services, enterprise services that compute the same event type are able to access the same data.

- **Multiple Instance Data** "Tasks which are able to execute multiple times within a single workflow case can define data elements which are specific to an individual execution instance" [96, p. 363]. In order to have multiple instances accessing the same data, an *updates*-relation ($\vdash$) needs to be defined on the event type that determines the respective task. Assuming that an event e triggers a relevant type of task, the relation $e \vdash e$ needs to be defined in order to realize this pattern.

- **Case Data** "Data elements are supported which are specific to a process instance or case of a workflow. They can be accessed by all components of the workflow during the execution of the case" [96, p. 365]. This visibility pattern can be realized by defining an event type per business process. As an `Event` is generated for each process instance, data that is stored in the data repository by using this event is visible throughout the respective process.

- **Workflow Data** "Data elements are supported which are accessible to all components in each and every case of the workflow and are within the control of the workflow system" [96, p. 366]. The definition of having a single event type per process is equivalent to the definition for the *Case Data* pattern. In order to realize the *Workflow Data* pattern, an *updates*-relation ($\vdash$) needs to be defined on this event type. Assuming that an event e triggers a process E, the relation $e \vdash e$ needs to be defined if the *Workflow Data* pattern should be realized.

- **Environment Data** "Data elements which exist in the external operating environment are able to be accessed by components of the workflow during execution" [96, p. 367]. This visibility pattern is not supported by the notion of event types. Data of external resources (application systems) can only be accessed by the means of explicit service invocations.

5.3 Heterogeneous Application Systems

The providers of functionality for composite applications are the various application systems that exist within the landscape of an organization. These application systems could just be a set of services that run on application servers. In such a scenario an organization is free to develop services as they are needed. However, this is rare in reality.
More likely application systems are COTS. Over time, organizations usually have bought and consolidated application systems for various purposes. Most likely these application systems are also integrated using Information-Oriented Application Integration. Of course, these landscapes support the processes of an organization well. They are not flexible, though (cf. the discussion of section 2.2). A BPIOAI approach would be preferable.

However, it is simply because of budget that flexibility issues can not be addressed by reorganizing application systems into service providers.
This is why the application systems need to be "integratable" by exposing their functionality as services — services referred to in this thesis as "application services". Some proposals exist that aim to support the definition of application services in a way that required functionality and possibilities of application systems are considered as a trade-off (cf. [97], [11]). Despite the argument of budget, the idea of deploying COTS in an environment is to use standard software that is supported by the respective vendor. This is why it is usually not an option to deploy services on application systems at will (e.g., by the means of wrappers as proposed by [97]).
If service orientation could be applied to a heterogeneous landscape, this is only possible if application systems remain unchanged (this means not reprogrammed but possibly reconfigured). The heterogeneity needs to be addressed elsewhere.
This is why there are no constraints that can be put onto application systems. Furthermore, a flexible and independent mechanism is required that enables heterogeneous application systems to participate in a composite application. This mechanism is provided by the presented reference architecture.
Application systems are also considered as point of human-interactions with composite applications. This is because it is not feasible to remove all user interactions from application systems in order to establish a composite application. However, one advantage of the service-oriented architectural style is the unification of user-interaction (cf. section 3.2.3). This is why a composite application usually consists of some sort of user portal (cf. chapter 2). The presented reference architecture does, however, not prescribe the usage of a portal. The service-oriented architectural style is considered an architectural style that establishes a central control instance in a heterogeneous application landscape to thus increase the maintainability of the overall process. Achieving this is also possible without using a portal. Hence, from a software architecture point of view, portals that form the presentation logic of a composite application are considered back-end application systems. As a consequence, there are no constraints or requirements imposed by this reference architecture towards portals.
This is a specific approach of this reference architecture. It has two additional advantages: first, it completely de-couples the control logic from user interactions or "wizards" that guide a user through a graphical user interface. Second, it allows for the transparent replacement of user interactions. This way the degree of automation can be increased without changing the overall process. Also outsourcing of certain tasks becomes easier.

5.4 Connectivity to Application Systems

Purpose and Functionality As a prerequisite, composite applications rely on a common protocol that is shared by service consumers and providers (cf. [4, p. 218]). The connectivity layer addresses the integration of application systems that do not provide functionality by using the common protocol of a specific composite application. It connects application systems and the composite application by homogenizing the protocol that is used to access functionality. The connectivity is realized by adapters. In general, an adapter "convert[s] the interface of a class into another interface [...]. An adapter lets classes work together that couldn't otherwise because of incompatible interfaces" [37, p. 139]. This description signifies that an adapter basically handles differences between two components by creating an intermediary abstraction between them. In the context of

composite applications the definition of an adapter must be narrowed. Linthicum defines adapters from an application integration point of view as constructs that "[...] remove us from the need to deal with the interface details that communicate with a variety of different source and target systems. What's more, adapters provide more consistency from interface to interface because they are, by design, reusable from problem domain to problem domain. [...] They merely deal with the connectivity to the source or target systems" [4, pp. 23f.]. Hence, adapters are domain-independent intermediary components that connect in a system-specific way to application systems and to make the application systems accessible by composite applications. Adapters represent "[...] layers between the [composite application] and the source or target application" [4, p. 218].
Adapters address heterogeneous protocols and provide a translation that is specific for a certain (class) of application system. In addition to heterogeneous protocols, data formats are likely to differ in application systems. According to [95], data representation, data structures and data types can differ. Heterogeneous data is addressed by the data exchange and data transformation (DET) layer's heterogeneity service (cf. section 5.7.3). In some scenarios, though, the platform of the DET might rely on a common data representation. In order to realize such scenarios, the adapters for the single application systems need to address differences in the data representation and transform the data prior to forwarding them to the DET. Even if it is conceptually possible, it is unlikely to deploy adapters and not a DET to connect to application systems. This is because application systems that do not natively support a service interoperability protocol usually do not store data using the data structures and data types used by a composite application.

Realization Requirements Adapters need to support any kind of interaction that is required between a composite application and application systems. Possible ways of (service-based) interaction are described in [98]. The different ways of interacting can be realized by using the DET (cf. section 5.7.8). As the DET relies on the functionality of the connectivity layer, it needs to provide the following functionality:

- **Send - Service Interaction Pattern 1** of [98].
  Application systems need to be capable of intrinsically sending requests to a composite application. In order to realize this functionality, the connectivity layer needs to seamlessly adapt to the standard mechanisms of the application system. Usually this is realized by configuring application systems in a way to use the connectivity layer as endpoint of their outbound communication. The connectivity layer in turn transforms the protocol to the common protocol of the composite application. This way, extrinsic event generation becomes possible. This approach allows for *interface processing*-based integration (cf. [4]).
  An alternative is to have the connectivity layer monitoring state transitions within the application systems and sending requests to the composite application based on such transitions. This approach is a mixture of *interface processing*-based integration and *data replication*-based integration (cf. [4]).
  In both cases, the connectivity layer should allow issuing of acknowledgments to the application system if data was successfully received by the composite application.

- **Receive - Service Interaction Pattern 2** of [98].
  In order to invoke service operations that are provided by an application system, the composite application system needs to send requests to the connectivity layer so that the request can be received by an application system. The connectivity layer

must enable application systems to receive such requests. Additionally, it should issue acknowledgments to the composite application if data was successfully received by an application system.

- **Send/Receive - Receive/Send - Service Interaction Pattern 3** of [98].
  The connectivity layer needs to enable the composite application to send requests to an application system and to receive replies. This interaction needs to be possible with both, synchronous and asynchronous communication semantics. If asynchronous communication is used, the connectivity layer does not offer functionality for realizing correlation. This functionality is provided by the DET (cf. section 5.7.1). However, if required, the connectivity layer must provide means for addressing dynamic callback-endpoints.
  Since a composite must send requests to application systems and receive replies using the connectivity layer, the connectivity layer also needs to enable application systems to send and receive requests.

- **Transactional Support** If atomic transactions are initiated by the service coordination layer of a composite application (cf. section 5.8), they also usually involve operations of application systems. In order to allow for distributed transactions, both, the common protocol of the composite application and the communication from the coordination layer to the application systems need to support a transactional protocol. Established transactions can possibly be active in both, application systems and the composite application.
  A possible approach to realize this is to register adapters of the connectivity layers as resources in a distributed transaction initiated by the service coordination layer. The connectivity layer may act in such scenarios as a transaction manager for the application-internal transactions and populate the results of these transactions to the coordination layer.

- **Conversion of Data Representation** One aspect of heterogeneity of data is the differences between data representations within application systems and composite applications. The functionality of data transformation and homogenization is addressed by the heterogeneity service of the DET (cf. 5.7.3). There might exist, however, technical constraints on the DET platform and/or the heterogeneity service that require a transformation of the data representation at the layer of connectivity.

- **Dynamic Addressing**: **Relayed request - Service Interaction Pattern 12**
  To relay a request means that a receiver of a requests redirects the request to another service provider. In order to support this type of interaction, the connectivity layer needs to provide an addressing protocol that supports request relaying. This involves the ability of receiving a pro-active relay-action of an application system, and transparently redirecting such a request to another application system (or more general service provider). Important to note is that the protocol of the application to which the request is relayed might differ from the protocol that is used by the relaying application. From a composite application point of view such a relay should be handled transparently.

5.5 Eventing System

Purpose and Functionality The eventing system is the entry point that is used to trigger composite applications. Possible ways of triggering composite applications are intrinsic and extrinsic generations of events. In case of intrinsic event generation, the composite application itself starts a business process independent of the connected application systems or user interactions. Intrinsic events could, for example, indicate that a scheduled computation of potentially changed data is needed. Intrinsic generation of events is similar to well-known batch processing (cf. e.g. [99]). Extrinsic event generation describes when application systems or users trigger composite applications from the outside.
In order to trigger composite applications from outside the application, the eventing system is exposed as a service. This service is called `EventService`. An `EventService` creates `Event`-objects that are used as keys to determine the appropriate service orchestration that implements the applicable business process.

According to the two possible scenarios of event generation, there exist two types of event services. The publicly accessible version is the "push-version". In this version of an `ExtrinsicEventService`, the event service behaves similar to an *Observer* in the Observer Pattern (cf. [37]) and filters out all relevant events. Such an event service puts the filter logic into the composite application in order to avoid changing a connected application system. There is no need to customize an application system to use an `EventService` as it is described by the reference architecture. As outlined below, application systems can be connected via the standard means of connectivity while the data exchange and data transformation layer (that is discussed in section 5.7) transparently handles the communication with the eventing system.
The intrinsic version of an event service generates events by itself only dependent on time. Based on its configuration, an eventing system triggers appropriate processes. This is the highest degree of decoupling that is achievable. This is because there is no application system involved in the generation of an `Event`-object at all.

The first (and publicly accessible) part of the eventing system is the generation of `Event`-objects by the means of an `EventService`. The other part is the determination of appropriate process orchestrations that handle the actual events. As described in section 5.2, an event determines the process that computes it. This is why the eventing system needs to maintain the relation between event types and processes. Technically, the eventing system needs to maintain a reference between an event type and an endpoint reference of a process orchestration. This is why the eventing system realizes a service registry.
In order to maintain these relations, an eventing system exposes a management interface that is used for administering the relations and keeps an internal data structure for the relations of event types and endpoints. An eventing system additionally ensures proper configuration by respecting the event rules that were described in section 5.2.

Realization Requirements and Syntactical Definition An eventing system is exposed to external application systems through an `EventService`. Such a service exposes an `update`-operation that can be invoked using an `EventType`-object as parameter. If called, an `EventService` then generates an `Event`-object that is dispatched to the service orchestration layer (cf. section 5.9.1). `Event`-objects contain unique numbers to allow the

use of these objects as keys in other parts of a composite application. An implementation of the eventing system has to ensure the uniqueness of these event identifiers for all `Event`-objects it generates.
As soon as an event is computed, the orchestration layer terminates the computation of an event by invoking the `finish`-operation of the `EventService`. Potentially suspended events are then dispatched by the `EventRegistry` subsequently. In most real-life applications of this reference architecture, an application system will very likely not invoke an event service directly. Furthermore it is likely to populate an arbitrary message or service call (transparently) via the connectivity layer to the exchange and transformation layer. The data exchange and data transformation layer transforms such arbitrary messages into a service call to the `EventService`'s `update`-operation that generates the `Event`.

An eventing system has to allow for various communication semantics and interaction patterns for invoking the `EventService`. The following interaction patterns need to be supported by the actual implementation of the eventing system:

- *Receive* pattern (cf. [98], pattern 2). By supporting this pattern, the composite application can be triggered asynchronously ("fire-and-forget").
- *Receive/Send* pattern (cf. [98], pattern 3) with synchronous response. An `update`-operation must not block until the event is processed further. This way it is transparent to the application system whether an event is processed successfully or not. This allows another decoupling between the application system and the composite application.
  As the return-value of an `update`-operation is the `Event`-object that was created, an application system can be informed that the triggering of the composite application was successful, though. The actual `Event` might be transformed into another structure using a heterogeneity service (cf. section 5.7.3) if an application system can not directly compute `Event`-objects.
  If an application should also be informed about the final success of a process, a dedicated application service should be designed and then later invoked.
- *Receive/Send* pattern (cf. [98], pattern 3) with asynchronous response. This type of interaction needs to be supported by the implementation of the exchange and transformation layer in order to allow asynchronous acknowledgments of the initiation of composite applications. Besides different communication semantics, this scenario is the same as *Receive/Send* with synchronous response.

For the first interaction scenario, events have to be communicated reliably to the event service since acknowledgments are not possible. For all three scenarios – as the `update`-operation is non-blocking – the composite application needs to ensure that the event is processed further. This is why a communication bus is required that supports *Messaging* (cf. [95, p. 53]) together with *Guaranteed Delivery* (cf. [95, p. 122]).
Other communication patterns do not need to be supported by the eventing system. More complex interactions are realized using the exchange and transformation layer that encapsulates the event service.

The interface the `EventService` exposes to the application systems and the data exchange and data transformation layer is shown in figure 7.

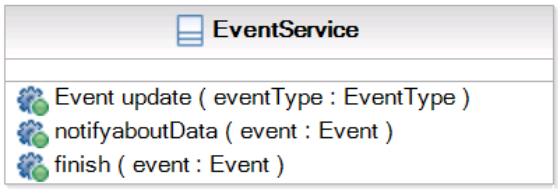


Figure 7: Public Interface of the `EventService`

The implementation of the eventing system consists of two additional components: an `EventIdGenerator` as well as an `EventRegistry`. The interfaces of these two components are depicted in figure 8.

Figure 8: Interfaces of the `EventRegistry` and `EventIdGenerator` Components

The collaboration of the single components of the eventing system is described in the sequence diagram shown in figure 9.

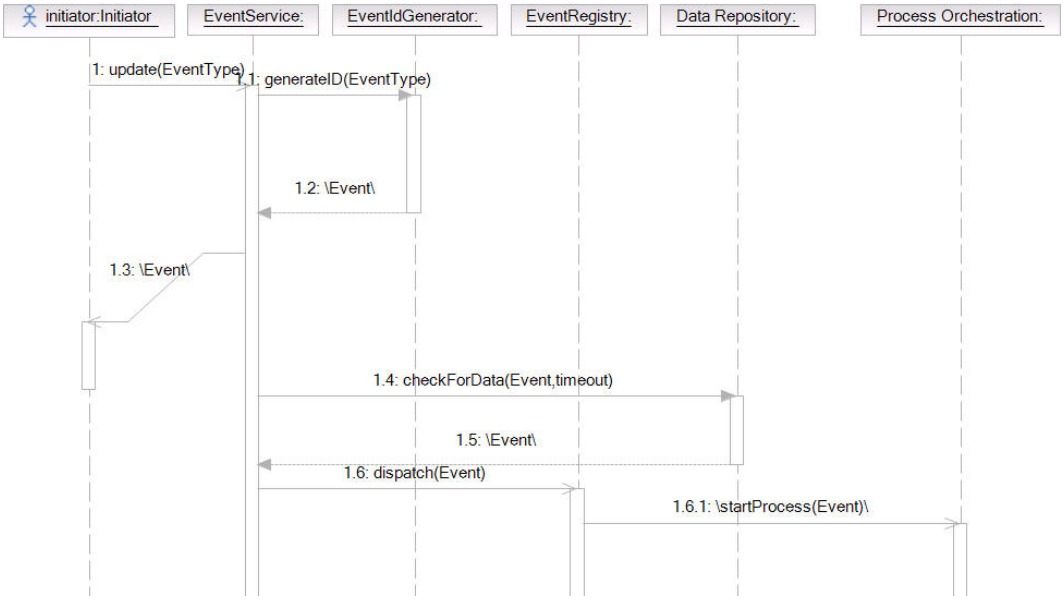


Figure 9: Overview of the Event Creation and Process Initiation

An arbitrary initiator (this could be an application system, the data exchange and data transformation layer or an intrinsic `TimerEventService`), calls the `update`-operation of the `EventService` by passing it an `EventType`-object (1). In turn, the `EventService` calls the `EventIdGenerator` (1.1) to get an `Event`-object that corresponds with the `EventType` (1.2). Next, the `EventService` invokes the `checkForData`-operation of the `DataRepository` (1.4). This operation is described in detail in section 5.6. Basically, it checks whether the data that is required for the execution of a process is registered in the data repository. A call-back is initiated to the `notifyaboutData`-operation of the `EventService` as soon

as the required data is registered. As an alternative, synchronous calls can be used that block until the data is registered.
As soon as the `DataRepository` confirms that the required data is registered (1.5), the `EventService` calls the `EventRegistry` (1.6) that looks up the appropriate process endpoint and invokes the process using the `Event` as an argument (1.6.1).

The interface that can be used to configure the eventing system is depicted in figure 10. It exposes 10 operations. The operation `setProcessEndpoint` defines at which endpoint an appropriate service orchestration for an event type can be started.
The four `scheduleEvent` operations are used for configuring intrinsic event generation. All operations are similar in that they all register an `EventType`. The difference between the operations is when exactly the `EventService` should be called in order to generate an appropriate `Event`. Using the first method, a single event generation can be scheduled to be generated during a `duration` of time that begins with the call of the operation. The second method will be useful if the event should be raised after a certain time has expired.

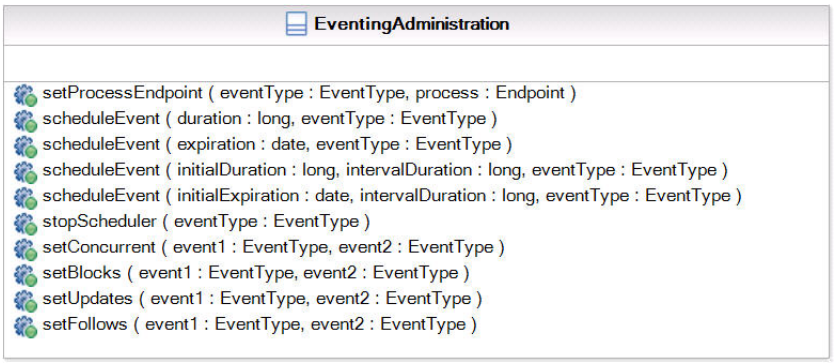


Figure 10: Public Interface of the `EventingAdministration` Service

The latter two schedule-operations schedule the event generation just like the first two operations but are different in that the events are generated repetitively with a time `intervalDuration` of milliseconds between the generations.
The `stopScheduler`-operation can be invoked at any time in order to stop further raising on an `EventType`.
The last four operations are used to set the relations between event types as they were defined in chapter 5.2.

A platform that is used to realize an eventing system needs to support the following functionality:

- **Message Filter** (cf. [95, pp. 237ff.]) If triggered from the outside, an eventing system must be capable of filtering out events.
- **Receive** pattern (cf. [98], pattern 2) that allows to have the event service invoked.
- **Receive/Send** pattern (cf. [98], pattern 3) with synchronous response. The support of this pattern is required in order to invoke the event service synchronously.
- **Receive/Send** pattern (cf. [98], pattern 3) with asynchronous response. The support of this pattern is required in order to invoke the event service asynchronously.

- **Guaranteed Delivery** is required in order to allow for failure tolerant dispatching of `Event`-objects.
- **Timer** Intrinsic generation of `Events` is required for some use cases. In order to realize this functionality, the platform has to offer some sort of a timer mechanism that initiates the event generation.
- **Persistent Storage** The configuration of the eventing system must be kept in a persistent storage.

5.6 Data Repository

Purpose and Functionality In order to realize composite applications that utilize heterogeneous application systems, several aspects must be incorporated. This reference architecture introduces several layers that address these aspects. Consequently, if these layers span multiple platforms, a process executed by a composite application is distributed over all these platforms.
As the design methodology that will be proposed in chapter 6 should be applicable to any actual target platform, this reference architecture, needs to be realizable with any arbitrary environment that follows the IT-strategy of the respective organization.
Multiple platforms are potentially required for realizing composite applications. Such platforms include application servers, integration servers and service orchestration tools. In order to apply the reference architecture to arbitrary combinations of such platforms, it needs to be ensured that all required platforms are able to collaboratively support business processes. As most of the architecture's elements operate on the processes' data, it is necessary that all elements have access to that data while ensuring acceptable performance and data consistency (cf. [6, pp. 185f.]).
In order to allow this, the reference architecture introduces a global process context that is called the data repository. The data repository is the non-persistent[27] memory of composite applications. It keeps data referenced by events and makes it accessible to arbitrary components of a composite. This follows the idea of Kossmann that proposes to establish "a globally interconnected set of objects known as the ObjectUniverse, positioned in a huge address space referred to as the ObjectCosmos" [100, p. 1] in order to allow interoperability in a distributed system. In the context of the different workflows used to realize a composite application, the data repository can be seen as a *blackboard* (cf. [101]) that is used in a workflow system (cf. [102]). It partially replaces the messaging paradigm with a spaces approach (cf. [94]).

The structure of the data stored in the data repository of a composite application is called *canonical data model* (CDM) (cf. [95, pp. 355-360]). A CDM provides an additional level of indirection between the single heterogeneous applications' individual data formats. If a new application is connected to a composite application, only transformations between CDM and the application's data format have to be realized, regardless of the number of applications that already participate (cf. [95, p. 356]). The CDM is a composite's own data format. If required, the transformation between the application's data formats and the CDM is realized by the data exchange and data transformation layer (cf. section 5.7). It is reasonable to use an already modeled and established data schema as the CDM. A

[27] In order to address failure tolerance, the memory is persistent. However, the data store that is used is different from the owning application system that is used for storing the data

good candidate for a CDM is the data model that is used in the organisation's main Enterprise Resource Planning (ERP) system. Alternatively, global standards like UN/CEFACT (cf. [103]) could be applied. A CDM is the data model for one composite application that usually supports one business process. Hence, if required, realizing multiple CDMs is possible. The usage of a CDM in a composite application does not introduce a company-wide data model.

The data repository acts similar to a tuple space (cf. [104]). Elements of the composite application get/read and store data to and from the data repository. These operations need to potentially be protected with transactions. Additionally, the data repository uses events to allow for smooth long-running transactions by already blocking requests that could interfere with already running processes.
It was introduced in section 5.2 that events determine both, the appropriate processes to trigger and the set of data that is possibly affected by a certain event. Based on this idea relations between event types were introduced. These relations are used to describe consistency models for the data repository.

Four relations between event types were defined: *concurrent* ($\|$), *blocks* ($\perp$), *updates* ($\vdash$) and *follows* ($\triangleright$). The *follows* relation can not be used to describe consistency because it provides no information about affected data.
For the *concurrent* relation, data is not considered to be shared between the affected processes at all. *Strict consistency* is achieved for the data that is kept for two *concurrent* events in the data repository (cf. [71, p. 298]).
The *blocks* relation implies that two process instances possibly share all their data. Therefore, it is considered that their execution needs to be serialized. This achieves also *strict consistency* (and needs to be realized by the eventing system).
The *updates* relation can be used to share data among different process instances.
The relation $a \vdash b$[28] is used to describe that an instance of a composite application (a process instance) $\vec{B}$ has to load the according data object from the data repository of a process instance $\vec{A}$ if $\vec{a}$ is being computed at the time of the access through $\vec{B}$. If no data can be found at the data repository instance of $\vec{A}$, $\vec{B}$ will create an instance of the required data. If at a later time $\vec{A}$ would be active, the data from $\vec{B}$'s data repository would be queried.
Whenever process instances need to mutually update their instances of their data repositories ($(a \vdash b) \wedge (b \vdash a)$), concurrent data access is possible. Here write accesses of one process update the data of the other process, too. Strict consistency is not achievable in such cases as the initiation of the data changes might be triggered by distributed services. As this prohibits the identification of "most recent" operations, only *sequential consistency* is achieved (cf. [71, pp. 30ff.]).

To ensure that the relations are appropriately defined in terms of data consistency, some of the rules defined in table 15 in section 5.2 are helpful. These are *Serialization* and *Replication* ($(a \perp c) \Leftrightarrow (c \perp a)$ and $(a \vdash c) \Leftrightarrow (c \vdash a)$).
They describe that *blocking* and *updating* can only be defined symmetrically. By prescribing that two event types that indicate processes, which usually change the memory of each other, need to be serialized if the relation was identified for one direction. Serializing the execution leads to strict consistency and is preferable. As a drawback, the reactivity

[28] Note that this would also be applicable if $a = b$ was true.

of a composite application is lowered.
If, however, events are identified for several steps within a process, processing these events is quicker as if one event type per process was identified. Hence, blocking an event's computation for the duration of another event is not very critical in terms of loosing system reactivity.
Replication indicates that two processes A and B for event types a and b can also be set in relation $((a \vdash b) \wedge (b \vdash a))$. This guarantees that both instances of the implied processes access the same set of data. In this case, writes would still have to be serialized between those instances to guarantee a strictly consistent data set. Due to the lack of a notion of absolute time, sequential consistency is achieved, though.
The rule *Unambiguity* $(((a \vdash c) \wedge (x \vdash c)) \Rightarrow ((a \vdash x) \wedge (x \vdash a)))$ assures an unambiguous replication between data objects of different processes' data repository instances. As a consequence, the data repository instances of the concerned processes act as if they were one. Without this rule it would not be possible to properly share data between more than two process instances.

Realization Requirements and Syntactical Definition When creating a data repository, it is important to consider that the purpose of composite applications is to integrate various heterogeneous application systems. As a prerequisite, the architecture of a composite application can rely on common protocols and a common data format. These elements are ensured by the connectivity layer as well as by the data exchange and data transformation layer. However, whenever services that fit in terms of data and protocol as well as in terms of appropriate service design, a composite should be able to utilize them directly. It is not an option to introduce protocol requirements in terms of including the data repository into the reference architecture. This would finally forbid the seamless use of any external service.

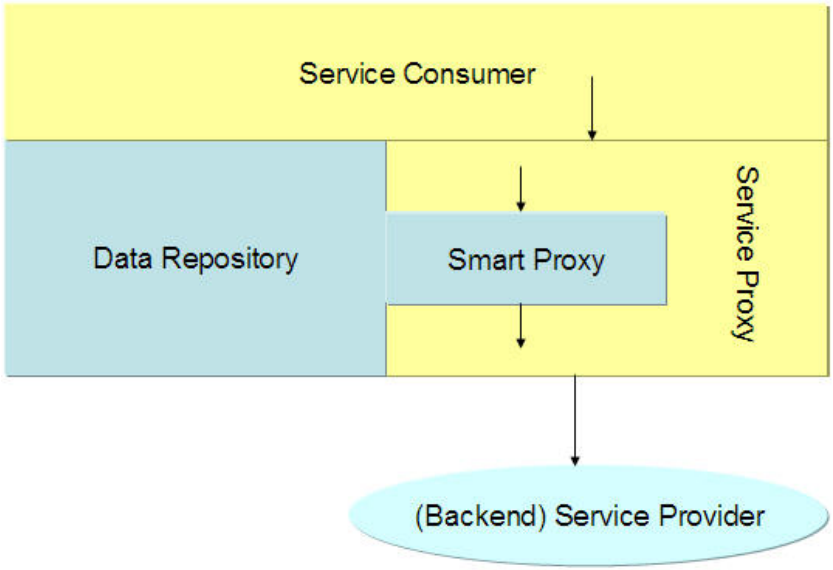


Figure 11: Smart Proxy Concept for Data Repository Integration

This is why the reference architecture introduces the notion of "smart proxies". On any target platform that is used to implement a composite application, external (back-end) services are likely to be used by the means of proxies that locally represent the remote service (cf. [37]). The reference architecture solely specifies the interface of such proxies

with the data repository in order to allow a transparent use of both, the remote services as well as of the data repository.
The concept of a smart proxy is illustrated in figure 11.

A smart proxy is the interface between an arbitrary service proxy for external services and a data repository. An external service can be both, a service consumer and service provider. Whenever a composite application exposes itself as a service consumer to back-end systems, smart proxies can be used for storing the data of the request into the data repository. To keep track of data changes and ensure data consistency, an `Event`-object must be used. As agnostic external services are unlikely to accept and produce `Event` objects, consuming such services is only possible if the data exchange and data transformation layer aggregates the creation of an event and storing of the corresponding request data to the data repository. The smart proxy interface can be applied both, for service consumers and service providers. Its interface is specified in figure 12.

Figure 12: Public Interface of the `SmartProxy`

The smart proxy interface is defined in order to allow the basic operations of a tuple space. It is inspired by the `JavaSpace` interface of [105].
The `read`-operation reads data that is associated with the passed `Event` from the data repository. The data repository returns the data only if no changes are currently being performed on the data. If the read-only access is not permitted, the operation signals an exception (in a platform dependent way). The `take`-operation reads data from the data repository and locks them for modification if no event is active that might require reading the data. Also this operation signals an exception in case of a locking issue. The `write`-operation writes data to the data repository. A prerequisite is that the concerned data is either locked for the passed `Event` or is not yet existent in the data repository. The same is true for new events. If non-locked data is written for the first time and is associated with a new `Event`, this `Event` is registered with the data repository. These operations signal exceptions concerning locked data. The data is passed between the actual proxy or stub and the data repository by the means of serialized data.
These operations are also offered with a `Transaction`-object as a parameter. This can be used if the respective service consumer and/or provider are capable of handling transactions and transaction management is not performed by the platform. If the notion of transactions is used, the existence of the data objects in the data repository, respectively their locking for *taken* objects, is bound to the state of the actual transaction that is represented by the `Transaction`-object. By the additional notions of transactions, a more fine-granular (short-running) locking mechanism is realized on top of the event-based locking for long-running transactions.
Either way, the implementation of a data repository needs to support persisting the con-

text data. This is necessary for re-establishing the process context after a system failure and to increase a composite's reliability.

An actual `SmartProxy` may not directly implement the described interface. This is because the operations expect serialized data. By serializing the data, the data repository may potentially loose information about dependencies between different data objects. If a fine-granular locking mechanism is required, an `EventType` for each data object would be required. Other reasons for not implementing the data might be platforms that do not allow for the serialization of data or persisting serialized data. This is why the actual implementation of the smart proxy and the data repository might not be agnostic towards the managed data. It is proposed that a data repository implements `read`, `take` and `write` operations for any object that is potentially stored in a data repository. An actual example for a non-generic smart proxy is included in the implementation of the reference architecture that is described in section 7.2.2.

Various approaches exist to realize smart proxies. They might be explicitly included in the code of actual service proxies. Alternatively, they could be realized using method-call interception (cf. [106]). Smart proxies might be realized as part of the runtime environment of the service coordination layer (cf. section 5.8). Alternatively, they could be realized as stand-alone service providers that are referenced by application systems and composite applications. In such cases, attention has to be drawn to the fact that a transaction monitor might be needed in order to use transactions with the data repository.

A data repository also exposes a management service. This service is accessible via the eventing system. It provides functionality that might be used by application systems if special data consistency is required. Generally, it is used at design-time in order to configure the data repository.
The `checkForData`-operation of the `DataService` is used by the eventing system in order to check whether a service orchestration can be triggered. Normally, no data should be required before a process starts. This is because a process should retrieve the actual data by itself. However, some processes might be initiated (implicitly) by application systems in order to compute a certain data object. In such cases, the object has to be registered with the data repository before the actual process orchestration can be started. This is done using the normal `write`-operation of a data repository.
The `checkForData`-operation either blocks until the required data is registered or the indicated `timeout` is reached. The operation returns `true` if the required data is registered and otherwise returns `false`. If no timeout is indicated, asynchronous replies are used. In such a case, a data repository invokes the `notifyAboutData`-operation of the `EventService` as soon as the necessary data is registered.
The `setRequiredData`-operation is used to configure a data repository. It sets a dependency between an object type and the data it is checked for when the `checkForData`-operation is invoked. The actual notion of describing data types is dependent upon the actual platform with which the data repository is realized.
The `setConcernedData`-operation is also used to configure the data repository at design time. It is used to describe which data is possibly concerned by the execution of a certain type of event. "Concerned" data is important if events of types that are set into an *update* relation occur simultaneously. If data objects are marked as "concerned", it is possible that events of different types mutually access each other's data.
All related event type information in the data repository can be reset using the `resetDataConfig`-operation.

An actual platform for the realization of a data repository could potentially also not support the generic interface as it is described here. In such cases, the implementation of the data repository must ensure that the identified relation between event types and data object types is respected.
The `setUpdates`-operation, analogous to the operation of the `EventingAdministration` service, informs a data repository that data of processes belonging to the indicated event types should be shared.
The `invalidate`-operation can be used by application systems in order to inform a composite application that all data that concerns a certain event type has become invalid. Of course, this should be avoided. But whenever both composite applications and especially (back-end) application systems operate on data and that data is changed in the back-end system, the data that is possibly cached in the process context of the composite application can be marked invalid. In such cases, the data repository has to abort all active transactions that operate on the concerned data. If a `read`, `take` or `write` operation is executed, the respective operation has to inform the caller that the data has been invalidated. Usually, the caller of the smart proxy then has to rollback the respective operations of the long-running transaction.[29].
The `eventIsFinished`-operation is finally required to clean up a data repository. As soon as the eventing system is notified about the end of an event's computation, the data repository also needs to be notified. By invoking this operation, all cached data for the respective event is deleted. The operation throws an exception if, at the moment of the invocation, data is still checked out using this event. In such cases manual support procedures will usually apply. The interface of the data repository is shown in figure 13.

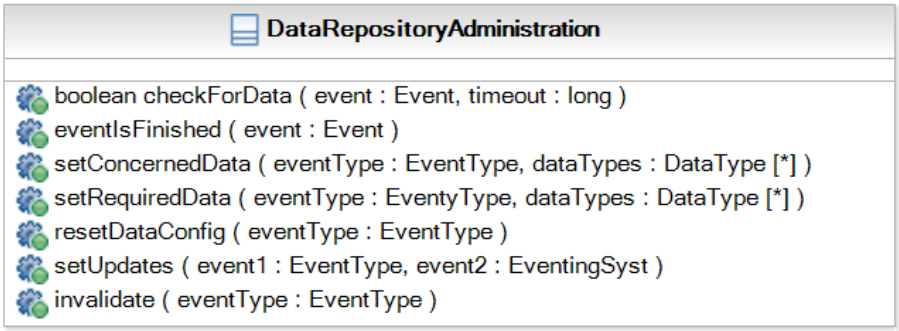


Figure 13: Public Interface of the `DatRepositoryAdministration` Service

The following platform requirements apply for the realization of a data repository:

- **Transaction Management** If transactional security should be established for complex operations on the data repository (multiple take/write operations), both, the data repository's platform as well as the platform the smart proxy is deployed on needs to support a (multi-resource) transactional protocol (e.g., two-phase commit, cf. [72]). As a smart proxy does not necessarily need to be realized using the common protocol of the composite application, the applicability of transactional concepts can be augmented to platforms that do not support transactional security of service interactions (e.g., "vanilla" web services).

[29]Note that the invalidation of data has to be a separate type of event and might be populated through the integration layer to a data repository.

- **Messaging with Guaranteed Delivery** In order to allow for reliable registration and de-registration, messaging with guaranteed delivery is required if the eventing system and the data repository are not deployed to the same platform. In the latter case, the event registration can be protected by using local transactions.
- **Persistent Storage** The state of a data repository must be persistent in order to allow for a failure tolerant behavior of a composite application.

5.7 Data Exchange and Data Transformation Layer

One objective of this reference architecture is to standardize the way composite applications are built in such a way that both, the implementation of composite applications is eased and the conventional application integration is facilitated. This is achieved by the concept of integration flows. These integration flows describe a behavioral element on top of data-centric integration concepts and facilitate the realization of composite applications in the context of heterogeneous application landscapes.

This second layer of the reference architecture is an optional layer that must be used whenever application systems do not stick to a globally defined data model, the appropriate data semantics or the required communication semantics. It unifies the data format of the connected application systems to a canonical data format (cf. [95, pp. 355-360] or [107]). It provides additional functionality for validity checking of data in terms of syntax and semantics as well as error handling procedures that need to be invoked whenever errors occur on this layer.[30] This way, integration aspects and business logic can be separated.

The data exchange and data transformation layer (DET) provides the functionality of an enterprise service (ESB) bus to a composite application. It mediates the business logic and the back-end application systems that are used to realize the business logic (cf. [41, p. 68]).
In [95], several so-called integration patterns are described and set into a relation. This relation is a basic pipe-and-filter architecture that describes a sequence of single patterns. According to [89], basic services can be orchestrated by so-called micro-workflows in order to be orchestrated themselves by macro-workflows. This idea is combined with the integration patterns for the definition of the DET. In order to facilitate cross-system process orchestration, the DET describes a sub-set of the integration patterns that are orchestrated to so-called integration services. These services are, in turn, orchestrated by so-called integration flows.

In order to standardize and simplify the DET, all its functionality is encapsulated in so-called integration services (IS). These services are orchestrated by two different integration processes. One provides data to the upper layers of a composite — the integration in-flow (IIF). The other one publishes data from upper layers to the connected legacy systems. This process is called integration out-flow (IOF). Both, the IIF and the IOF, act as service providers: they expose the functionality of application systems to upper layers of the composite application as services.
As mentioned, the integration flows are a replacement of the pipe-and-filter architecture

[30] The error handling at this layer basically covers support procedures that need to be initiated whenever errors occur (human errors are mostly the cause of these errors).

that is used by [95] to categorize integration patterns for message based application integration. Additionally, they limit the applicable set of patterns to an extent that is required to realize composite applications in a heterogeneous application landscape. This way, business processes can be used to centralize the control of the integration of application systems into composite application and prohibit the use of business logic inside the integration layer. The integration flows only handle the details of distributed and heterogeneous interactions.

This chapter introduces the single integration services as well as the requirements for realizing them. Subsequently, the integration flows are described. In section 5.7.8, how the integration flows can be combined in order to support higher-level interaction protocols is described. This discussion also demonstrates how the design of the service coordination layer supports the design decisions that have to be taken for the design of the DET layer.

5.7.1 Data Service

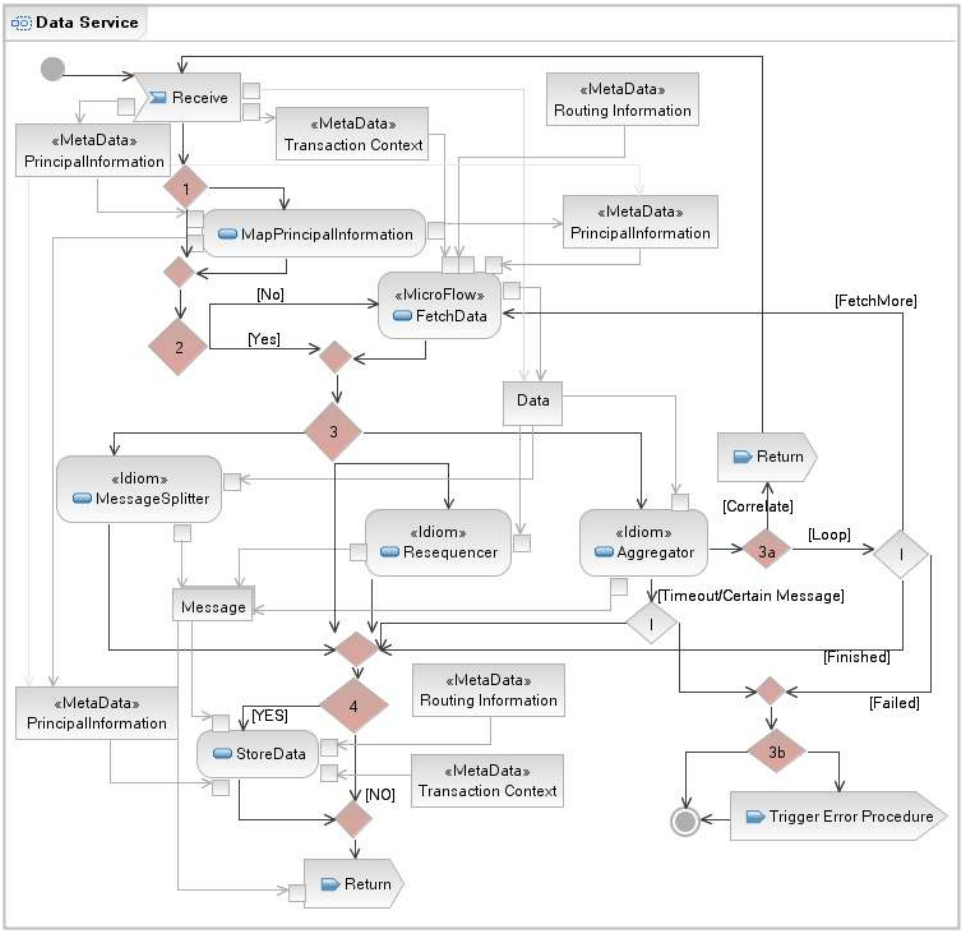


Figure 14: Micro-Flow of the `Data Service`

Purpose and Functionality The integration service that is most crucial for the DET is the `Data Service`. Its task is to load data from back-end systems and to store data to back-end systems. A `Data Service` can be used both as part of the IIF and in the IOF. For the sake of connectivity it might use adaptors that establish a connection to the

back-end systems.
A `Data Service` is described as a micro-flow. The micro-flow that describes the orchestration of the building blocks of a `Data Service` is depicted in figure 14.

The depicted activity diagram describes a superset of possible designs for a `Data Service`. The design decisions that are necessary are marked as red decisions with Arabic figures within the activity diagram. If not implemented at design time but realized using configurations, they correspond with the workflow pattern 16 *Deferred Choice* of [108].
The first design decision is whether a `Data Service` should map authentication data for the actual application system. Alternatively, the mapping could also simply propagate principal information of a composite application to a back-end system.
Then it either fetches data from a back-end system (indicted by *No* at the second design decision) or uses the data that was passed to the `Data Service` as parameters of the request.
The fetching of data can be described as a sub-process of the `Data Service`. This process is depicted in figure 15.

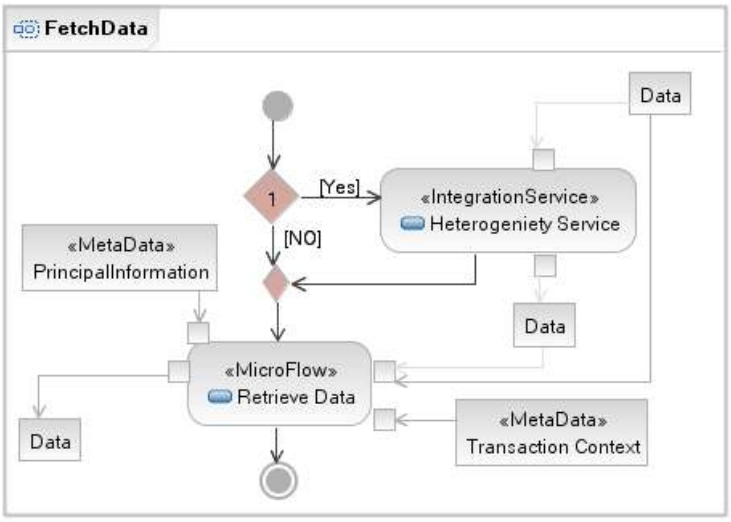


Figure 15: Micro-Flow of the `Fetch Data` Activity

Fetching data might require transforming the data that is passed to a `Data Service` in order to perform a lookup based on this data. If this is required, a `Heterogeneity Service` (cf. section 5.7.3) might be used for transforming the data of the request in a way that the `Retrieve Data` activity can lookup data from a back-end system. The translation should only be used to translate data into a format that is specific to the actual technical connectivity. An example would be to transform a message containing business data into a Structured Query Language statement (cf. [109]).
Fetching data might involve the notion of an atomic transaction that is realized via a 2-phase commit protocol (cf. [72]). A `Data Service` might have access to a transaction context that is then further propagated to the back-end applications.
The actual realization of the `Retrieve Data` activity is the second design decision that has to be made for the `Fetch Data` flow. The decision is determined by the design choices of the *request/response* interaction pattern of [98]. These choices are

- "The outgoing and incoming messages must be correlated. In other words, there is a common item of information in the request and the response that allows these two

messages to be unequivocally related to one another.

- The sender may block a thread of execution to wait for the response or fault, or it may provide a single continuation for both, the response and the fault or two separate continuations." [98, p. 309]

These two design decisions indicate different approaches to a realization. They allow for both, synchronous and asynchronous interaction with the back-end system.
Realizing a `Retrieve Data` activity always requires information for performing a lookup. This includes information about the actual payload as well as meta-data about the principal under whose identity the lookup has to be performed. This information might be included in the data passed to this activity or contained in its configuration. Whenever no request to the back-end system is required, the `Data Service` is called directly and the use of the `Fetch Data` flow is not required.
The interaction mode with the application system has to be indicated for each `Retrieve Data` activity (design decision 1 in figure 16). Possible options are synchronous and asynchronous communication, while asynchronous communication might be realized using correlation or call-back endpoints. In all options, a transactional context might be passed to the application system. In case of asynchronous communication, a timeout can be specified in order to avoid indefinite waiting periods of the `Data Service`. For situations in which a timeout occurs, an appropriate behavior needs to be defined (design decision 1a).

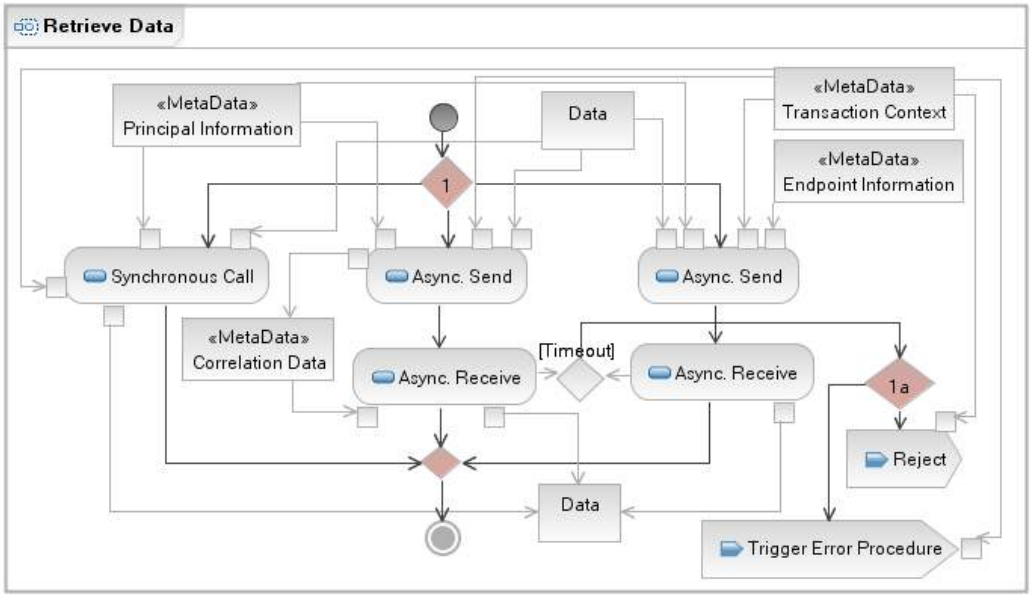


Figure 16: Micro-Flow of the `Retrieve Data` Activity

Returning to the description of the overall `Data Service`, the next design decision is regarding the treatment of the received data. Four mutually exclusive options are available:

- `MessageSplitter`. Whenever the data that is retrieved from a back-end system or directly received contains information about multiple data objects that need to be computed separately, the data needs to be split into several messages. This functionality is equivalent to the description of the enterprise integration pattern in [95, p. 259].

- `Resequencer` According to [95, pp. 283ff.], applying the *Resequencer* pattern is appropriate if multiple messages require in-sequence processing (e.g., if referential integrity needs to be maintained in subsequent processing steps). If an IIF or a composite application in general require a certain sequence of messages that is neither created by the `Retrieve Data` activity nor received by an IIF, this pattern might be applied as a part of the `Data Service`.

- `Aggregator` In contrast, if data needs to be processed as a whole, the aggregator pattern is required (cf. [95, pp. 268ff.]). Here, several messages are combined into one single message in order to have all contained data computed at once.
  The aggregator can either be implemented as a loop that retrieves all required data (the `Loop`-branch of design decision 3a in figure 14) or as an asynchronous collection of required data via a correlation. In the latter case, the aggregator appends the actual data to a list and suspends the execution of the `Data Service`. A `Data Service` is suspended in order to allow subsequent requests to be correlated with previous ones. This way, the aggregator enables a `Data Service` to collect related data and transfer them as a whole to a composite application.
  An aggregator may also wait for incoming messages for a certain time, until a certain message is received or until a certain amount of messages was received. Whenever a certain message is received or a timeout is reached, the aggregator can either be successfully finished or terminate with an error (decision I). If designed accordingly, an error procedure might be triggered (design decision 3b).

- Whenever the incoming or fetched data is appropriate, no additional action is required at this point.

Despite the design decision described above, one or several messages can be processed by a `Data Service`. If multiple messages are existent, each of it is processed by a dedicated instance of the surrounding IIF. Hence, if multiple messages exist, subsequent activities are executed in different, unsynchronized control flows as described by the workflow pattern 12 *Multiple Instances Without Synchronization* of [108].[31]

If a `Data Service` is designed accordingly, data can be transmitted to the respective application(s) by using a `StoreData` activity that uses (possibly mapped) meta-data in order to authenticate against the back-end system. A transactional context might also here be propagated to the back-end system.

Design Decisions and Realization Requirements The following list provides an operational overview of the design decisions that have to be taken if a `Data Service` is designed and upon which factors these decisions depend.

- *What is the actual data format?* Since the realization of many integration services depends on the actual format of the data that is sent and/or received by the `Data Service`, the format needs to be described.

- *Is Principal Propagation Required?* Whenever application systems require authentication it is necessary to provide a `Data Service` with principal information. This

[31] If the IIF is used in the context of an IOF, the IIF must not open multiple control flows (cf. section 5.7.7).

information can either be derived by mapping the credentials that were used to initiate a `Data Service` or by using a fixed configuration.

- *Fetch data from back-end or data in request to data service?* Depending on the actual connected back-end system, a `Data Service` either receives data from a back-end system or polls, based on what it has received, for additional data. Fetching data is required only if an actual `Data Service` is solely used for retrieving data and not for writing data. If fetching data is required, the following design decisions have to be made.
  - *Does data need to be transformed in order to perform the data retrieval?* Depending on the application system, a `Data Service` receives data but needs to transform the data in order to poll for data out of the application system.
  - *Communication semantics of the retrieval activity.* Depending on the actual application system, data retrieval might be synchronous or asynchronous. In both cases, a transactional context might be propagated to an application system. If asynchronous communication is chosen, the following design decision has to be made.
    - *Correlation or fixed call-back endpoint?* Depending on the application system, the connected application might be able to respond to an endpoint that identifies the actual instance of the retrieval activity. Whenever the application is not capable of using different endpoints, the platform of the DET has to correlate the incoming data with the request to the application system. This correlation is dependent on the actual data format of the request and the response. This influences the design of the `Retrieve Data` activity. Additionally, a timeout needs to be specified for the receive-steps that realize the asynchronous communication.
      Usually, these options are only used whenever a `Data Service` is used by an IIF.
- *Treatment of received/retrieved data?* Depending on the received data, the data must either be split into several messages, put into a certain sequence of messages, aggregated into one message or not treated at all. The splitter, resequencer and aggregator need to be configured according to the actual data format. If an aggregator is used, additional design decisions have to be made. Those are:
  - *Loop or Correlate?* Depending on the application system, the aggregation can either be asynchronous or synchronous. In an asynchronous case the aggregator ends the computation of a `Data Service` and relies on the platform to correlate subsequent requests to the suspended instance. In a synchronous scenario, a `Data Service` loops over the retrieval step in order to fetch all data that need to be aggregated.
  - *Completeness Condition?* Depending on the scenario (expressed by the semantics of the computed data), the condition for the aggregation to be complete must be defined. This is usually a configuration option.
  - *What if the Completeness Condition Failed?* Depending on the scenario (expressed by the semantics of the computed data), the failure of the condition (e.g., a timeout) might trigger an error handling procedure that needs to be described.

- *Store Data?* Depending on the use of a `Data Service` the transmitted data might need to be stored into an application system. The actual connectivity is dependent on the application system and realized using adapters. The stored data format is also application-specific. A `Data Service` relies on a `Heterogeneity Service` to transform the data into the application-specific format. If data has to be stored, the credentials that should be used for storing the data are also required.

As outlined above, the implementation of a `Data Service` is dependent on the connected application system and on the actual data format. Thanks to the more fine granular design using idioms, not every component of a `Data Service` is dependent on both factors.
A `Data Service` might be used as a service within the IIF and the IOF of the DET. In order to realize a `Data Service`, this platform needs to support the following patterns:

- **Request/Response Service Interaction Pattern 3** of [98]
- **Aggregator Integration Pattern 268** of [95]
- **Resequencer Integration Pattern 283** of [95]
- **Message Splitter Integration Pattern 259** of [95]
- **Workflow Pattern 12 – Multiple Instances Without Synchronization** of [108]
- **Workflow Pattern 16 – Deferred Choice** of [108]
- **Principal Propagation**. In order to authenticate against a back-end system using the credentials of an actual user of a composite, the DET needs to support principal propagation.
- **Correlation or Call-Back**. The platform needs to either support correlation or a referencing mechanism that allows both, asynchronous interaction with back-end systems and message aggregation.
- **Routing Meta-Data** The platform needs to offer an interface to provide meta-data about an ongoing request. This meta-data has to include information about the determined receiver(s) of a payload. The determination is performed by the `Routing Service`
- Additionally, the platform needs to support **Exactly-Once in Order Messaging** in order to allow for re-sequencing messages so that subsequent tasks receive the message in a defined order.

5.7.2 Validity Service

Purpose and Functionality A `Data Service` can be used by integration flows for interacting with application systems. These application systems might be internal applications or applications that are operated by organization-external partners. Especially in the latter case, data submitted by these applications might be erroneous or even malicious. In order to avoid such data going through a DET and reaching a composite application,

the data needs to be filtered. This is why a `Validity Service` is introduced. It acts as a `Message Filter` that filters out erroneous or malicious messages (cf. [95, 237]).
The detection of such messages is dependent on two factors: the data format and the actual scenario. Based on these factors it needs to be defined what "valid" means. This will be, on one hand, the well-formedness of the overall structure and validity in terms of the actual data format. Additionally, it might be necessary to add additional constraints such as partner-specific value ranges, etc. A `Validity Service` performs these checks by applying the defined rules to the received message and by providing the validity as a return value. Based on this value the orchestrating integration flow can react on valid or invalid messages.[32] A `Validity Service` might also filter out messages with inappropriate user credentials.
A `Validity Service` is usually designed specifically for one data format. However, a generic `Validity Service` might be realized that uses configuration information. In such cases, the orchestrating integration flow has to provide the information that is required for looking up the actual configuration for a certain message type.

Design Decisions and Realization Requirements The design decisions for the `Validity Service` concern the data format of the respective application system. Based on this format as well as on the actual requirements, the constraints for well-formedness and validity need to be defined.
The following list provides an operational overview of the design decisions that have to be taken if a `Validity Service` is designed and upon which factors they depend.

- *How is 'valid' and 'well-formed' defined for the actual data format?* Depending on the connected back-end system, a `Validity Service` might be needed in order to perform several levels of validity checks. Possible levels include the data representation ("well-formedness") and the actual values of data entities. The latter is usually specified by a data format definition.
- *What are the possible return values?* In order to allow the integration flow to react on valid and invalid data, the possible return values for a given data format need to be defined for a `Validity Service`.

There are no special realization requirements for a `Validity Service`.

5.7.3 Heterogeneity Service

Purpose and Functionality Different application systems expose different operations that can provide functionality to a composite application. Because of heterogeneity, a `Data Service` needs to be implemented specifically for both, the targeted application system and its data format(s). In order to realize composite applications without dependencies on the various application systems, it utilizes a Canonical Data Format (cf. [95, pp. 355-360]) or Canonical Data Model (CDM).
A `Heterogeneity Service` is the integration service that handles data translations from

[32] This is in contrast to the `Message Filter` pattern since the `Validity Service` does not filter the message directly. It solely provides the information based on which an integration flow can filter out messages.

application-specific data formats to the canonical data format used internally by the respective composite application. It also realizes the *Message Translator* pattern of [95]. It involves several levels of transformation. According to [95], these are data representation, data structure and data type (cf. [95, p. 87]). The data representation also needs to be homogeneous within a composite application. Whenever the connectivity layer can not address heterogeneous data representations, a `Heterogeneity Service` is also required to unify the data representation. Required transformations might be chained by defining a sequence of different `Heterogeneity Services`. Also several `Heterogeneity Services` that address the same level of transformation could be chained to allow for better reusable translators.
A `Heterogeneity Service` might also use a `Data Service` to lookup data for data enrichment during transformations. This is especially important whenever the content of a message needs to be replaced.

Design Decisions and Realization Requirements The design decisions of a `Heterogeneity Service` are dependent on the data format of the respective application system(s) and the canonical data format of the composite. Additionally, it should be determined whether lookups to external systems are required.
The following list provides an operational overview of the design decisions that have to be taken if a `Heterogeneity Service` is designed and upon which factors they depend.

- *Required Levels of Transformation?* Depending on the actual connected back-end system, a `Heterogeneity Service` might be needed in order to realize several levels of transformation. Possible levels are data structures and data types. Also the level of data representation might be addressed by a `Heterogeneity Service`.
- *Required Steps of Transformation?* Depending on the actual data format, an actual message translation has to be defined for every level of transformation. By defining a translation, the translation can be split into multiple steps in order to ease the reuse of these steps.
- *Is a Lookup Required?* Depending on the data format and the data management of the connected application systems, a lookup of data using a `Data Service` might be required. If so, the necessary authentication information must be propagated to the `Data Service`.

A `Heterogeneity Service` might be used as a service within the IIF and the IOF of the DET. In order to realize a `Heterogeneity Service`, the platform for the DET needs to support the following patterns:

- **Message Translator Integration Pattern 78** of [95] including chaining transformations.
- **Request/Response Service Interaction Pattern 3** of [98] in order to utilize a `Data Service` for lookups.

5.7.4 Trigger Service

Purpose and Functionality A `Trigger Service` can be used by an IIF as a translator between the integration-focused DET and a composite application. It transforms arbitrary incoming data that is either received or fetched from an application system, based on the respective sending application, into an `EventType` and triggers the composite application using an `Event`-object. The micro-flow of the `Trigger Service` that describes its interactions with the eventing system and the data repository is depicted in figure 17.

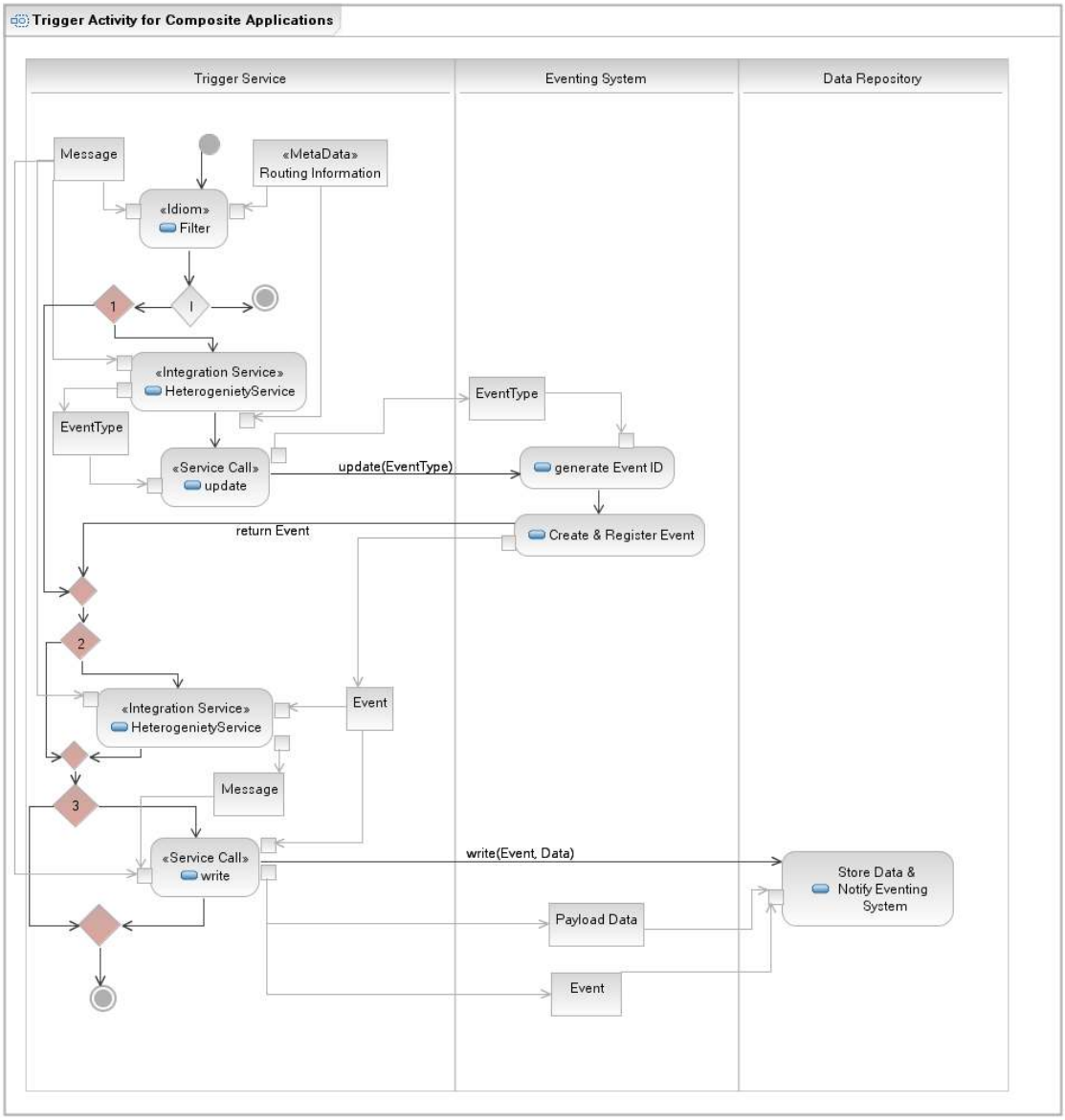


Figure 17: Micro-Flow of the `Trigger Service`

A `Trigger Service` first filters incoming messages based on its content and on routing information that is provided by the platform of the DET. As the integration pattern *Message Filter* (cf. [95, 237]) describes, the filter activity either allows continuation of processing the actual data or terminates its computation. The actual filter logic is dependent on the scenario as well as the data.

Based on the routing information and potentially on the actual data format, a `Heterogeneity Service` might be defined that creates an `EventType`-object by transforming the data from the application system.[33] Such an `EventType`-object is transmitted to the eventing system by invoking the `update`-operation of the according `Event Service`. This way, a composite application is started.
If the actual type of event can only be computed whenever certain data is available to the composite application, a `Trigger Service` stores the received data in the data repository by using the `write`-operation of the smart proxy and the `Event`-object that was returned by the `Event Service`. Potentially, the data that is to be stored needs to be transformed prior to its transmission.

Whenever an IIF that a `Trigger Service` is part of is used in a send/receive interaction (cf. section 5.7.8), the branch that *writes* data to the data repository is not used.

Design Decisions and Realization Requirements The design decisions for the `Trigger Service` are influenced by the canonical data format, the connected application systems and the business logic of the anticipated supported business process.
The following list describes the design decisions that are necessary in order to realize a `Trigger Service`.

- *What Data can be Processed?* This decision determines how a `Trigger Service` should filter-out incoming messages. The way of filtering depends on the actual business logic of the business process and on the connected application system(s). Also technical reasons might exist for application systems to propagate more events than required for the anticipated business logic of a composite application. In such cases, the filter has to be designed accordingly.
  A filter might be required to be stateful to implement all scenarios.
- *What Type of Event is Appropriate?* Based on the payload of the received message and based on routing information, the translator activity needs to be parameterized in a way that an accurate `EventType`-object can be created. The information this decision is based on can be derived from the business process model.
- *What is the Mandatory Data for the Determined Type of Event?* If the type of event that was determined by the transformation step requires some data in order to be computable, that data needs to be transmitted to the data repository. In such cases, a `Trigger Service` should be designed in a way that it stores this data in the data repository. This design decision depends on the actual business logic.
  - *Does the Received Data need to be Transformed Prior to its Transmission to the Data Repository?* If a data prerequisite exists, an additional transformation step might be required prior to transmitting the payload data to the data repository. This might include structural conversion as well as filtering certain data items out of a more broad data set.

A `Trigger Service` might be used as a service within an IIF of the DET. In order to realize a `Trigger Service`, the platform for the DET needs to support the following patterns and functionalities:

[33] In such cases, a `Heterogeneity Service` must not use lookups as no principal information is available.

- **Message Filter Integration Pattern 237** of [95] including stateful filters.
- **Routing Meta-Data** The platform needs to offer an interface to provide meta-data about an ongoing request. This meta-data has to include information about the actual sender of a certain payload.

5.7.5 Routing Service

Purpose and Functionality The actual service endpoints that indicate the agent that hosts a service provider is maintained within a service registry. Such a registry is described in section 5.10. Whenever application systems are not directly acting as service providers and are therefore not registered with a service registry, an IOF is used to connect to such a system. In order to determine the physical location of an application system, a `Routing Service` can be used as part of the IOF.
A `Routing Service` can be realized using different routing patterns. The micro-flow of the `Routing Service` is depicted in figure 18.

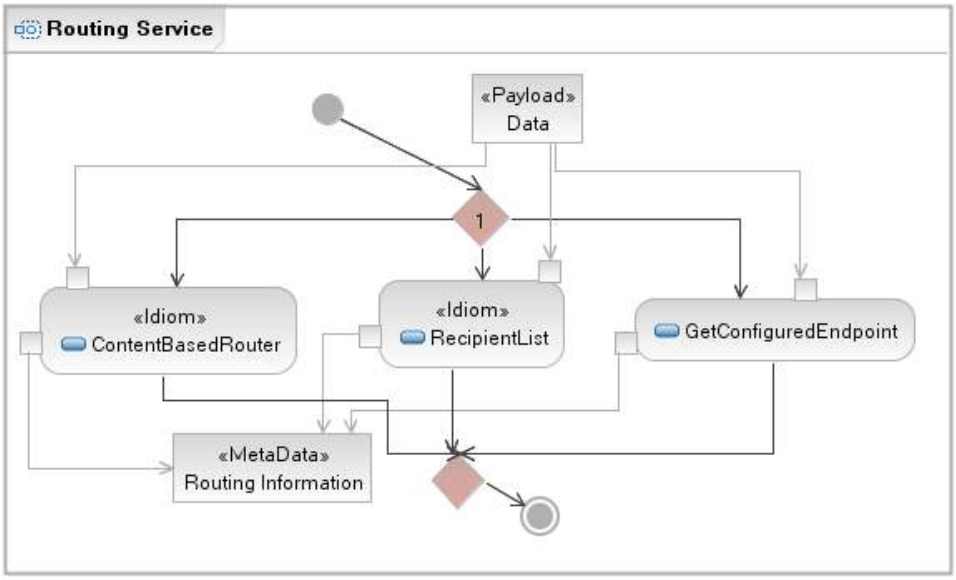


Figure 18: Micro-Flow of the `Routing Service`

A `Routing Service` can use three types of routers. The first option is that a *Content-Based Router* (cf. [95, 230]) is used. In such a case, the `Routing Service` determines the recipient of payload based on the content of the payload.
If a *Recipient List* (cf. [95, 249]) is used, there will be multiple recipients based on the payload. It is an extension of the *Content-Based Router* (cf. [95, p. 249]). If multiple receivers are determined, the meta-data containing the routing information consists of multiple entries. For each entry a new control flow of an IOF is created. Hence, subsequent activities are executed in different control flows. Usually, they do not need to be synchronized as described by the workflow pattern 12 *Multiple Instances Without Synchronization* of [108]. However, sometimes the final acknowledgment must involve all instances. In such cases, support for the workflow pattern 14 *Multiple Instances With a Priori Runtime Knowledge* of [108] is required.

The third option is to have a fixed receiver for a certain payload. The configuration is either static for a specific IOF or can use a service registry for looking up a service reference. Such a lookup is then based on the static, structural interface of the required service.

Design Decisions and Realization Requirements The design decisions of a `Routing Service` depend on the actual business process logic (the resource perspective of the process) and might be dependent on the canonical data format.
The following list provides an operational overview of the design decisions that have to be taken if a `Routing Service` is designed and upon which factors they depend.

- *Is the Receiver to be Determined by the Payload?* Depending on the business process and its data (described in the canonical data format), the actual receiver(s) might depend on the payload that is processed by an IOF. If so, the rule for the receiver determination need to be defined and the `Receiver Service` needs to be realized and/or configured accordingly.
- *Can Multiple Receivers Exist?* Depending on the information contained in the resource view of a business process, it can be determined whether possible multiple receivers need to be connected to a composite application using one single IOF. This is the case when multiple application systems expose the same functionality and the data needs to be replicated to several (or all) of these systems. This design decision has to be made while considering the possibility of using several services as part of a service coordination (cf. section 5.8). Whenever this option is used, the corresponding IOF acts as a service aggregator. This way, the *Aggregator to Aggregator Re-Use (AAR)* value (cf. section 3.2.1) is increased. Hence, the maintainability of the composite application might be decreased.
  If multiple receivers exist, each delivery is executed in a different, unsynchronized control flow as described by the workflow pattern 12 *Multiple Instances Without Synchronization* of [108].
- *Are Lookups Required?* Whenever the integration flow should be further decoupled from an application landscape, a `Routing Service` can make use of a service registry for looking up appropriate endpoints of a service provider. If this is necessary, both, the lookup information and the service registry need to be described.

A `Routing Service` might be used as a service within an IOF of the DET. In order to realize a `Routing Service`, the platform for the DET needs to support the following patterns:

- **Content-Based Router Integration Pattern 230** of [95]
- **Recipient List Integration Pattern 249** of [95]
- **Workflow Pattern 12 – Multiple Instances Without Synchronization** of [108]
- **Workflow Pattern 14 – Multiple Instances With a Priori Runtime Knowledge** of [108]
- **Dynamic Lookups from a Service Registry**

5.7.6 Integration In-Flow

Purpose and Functionality The integration services (IS) are used in two integration processes that orchestrate the single IS: one for reading data from and one for storing data to connected application systems. Both integration flows are generic references that allow the realization of the required functionality for an actual connection to (an) application system(s). They describe how interactions with application systems are possible while using various communication semantics. These are synchronous, asynchronous and asynchronous communication with acknowledgments. Both integration flows are described as a reference that indicate design decisions (red entities in the diagrams) that have to be taken in order to realize an actual interaction with an application system.
The reader process that describes how data can be transmitted from application systems to a composite application is called Integration In-Flow (IIF). The steps of an IIF are described in the activity diagram of figure 19 and in the following paragraph.

An IIF is either invoked by external application systems or is triggered by the composite application in order to fetch data from an application system. The first design decision is whether a `Data Service` is required in order to interact with an application system (design decision 1). Possible scenarios for the use of the `Data Service` at this place are the retrieval of data and/or to split or aggregate the data that was received to messages that are necessary as the DET relies on messaging (cf. section 5.7.1). Depending on the design, errors during the data retrieval (decision I) can be handled or ignored (design decision 1a). If the respective IIF is invoked within a transactional context, the `reject` activity can abort the current transaction. Next, the validity of the actual message[34] can be checked. If a message is not valid (decision II), an IIF can react accordingly (design decision 2a). By including such a validity check, the application systems and the composite application are decoupled (cf. [6, p. 192]).
If a message was received and (potentially) verified, an IIF can inform the calling party about the success if there is no transactional context present. In these cases, a synchronous call might be closed or an asynchronous acknowledgment might be sent back to the initiator (design decision 3).
Depending on initial design decisions, the incoming message might need to be transformed into the canonical data format of a composite application. Whenever the application system uses the canonical data format as its own data format this step is not required (design decision 4).
The next step is usually to trigger a composite application by using the `Trigger Service` (design decision 5). This step is optional, as an IIF can also be used in a request from the composite to back-end applications (cf. section 5.7.8). Depending on the actual requirements (design decision 6), an IIF can close the synchronous request, terminate and also acknowledge an asynchronous request. If a transactional context is present, these activities can issue a pre-commit for the current transaction. If the IIF is called by an IOF in a scenario that requires a coordination service to request an external service, the asynchronous acknowledgment needs to either include the correlation identifier (cf. [95, 163]) or to address the appropriate call-back endpoint (cf. section 5.8).

Design Decisions and Realization Requirements The design decisions for an IIF, apart from the decisions required for the single IS, are driven by the interaction require-

[34] If multiple messages are processed, multiple instances of the IIF exist.

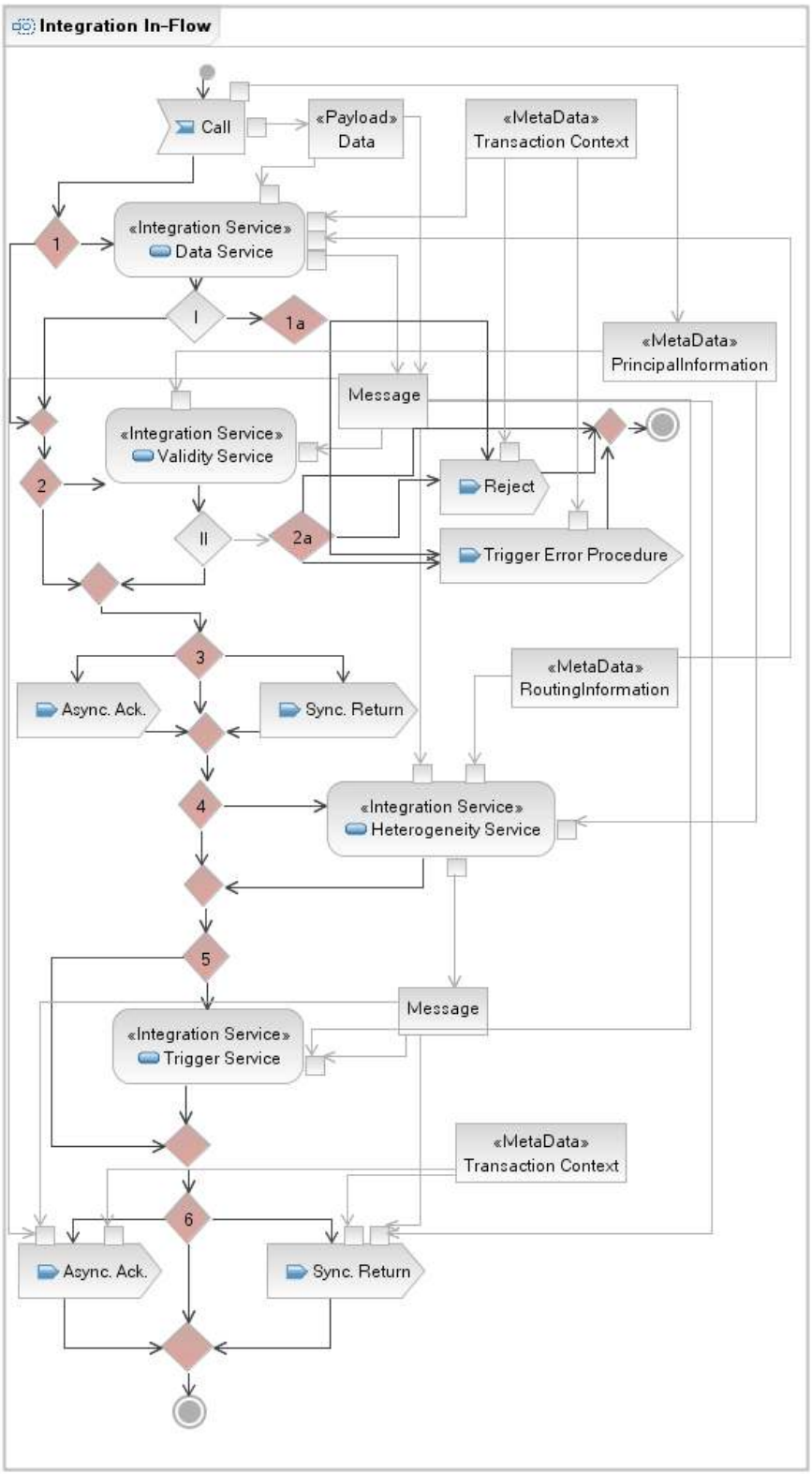


Figure 19: Activity Diagram of the `Integration In-Flow`

ments of a composite application. The interaction of a composite with heterogeneous application systems is realized using both, an IIF and an IOF. Hence, a part of the interaction is realized by an IIF.
The following list provides an operational overview of the design decisions that have to

be taken if an IIF is designed and upon which factors these decisions depend on.

- *For which Communication Semantics is the IIF Used?* An IIF can be used in synchronous and asynchronous scenarios. Both acknowledgments as well as a two-phase commit protocol (2PC) within a distributed, atomic transaction can be used to inform the respective party about the success of a communication. Knowing the manner of communication is important as it influences other design decisions.
  If a 2PC is used, the transactional context is established by the calling party (which is usually the composite application's service coordination layer). In such cases, first the participating application systems and then the communication protocol for the composite application and the protocol used to communicate with application systems need to support 2PC. The use of 2PC has only some minor implications for the design of an IIF as it basically relies on the communication protocol and just forwards the transactional context to the applicable parties.

- *Is Data Contained in the Request?* If it is required to load data within an IIF (rather than having it passed to it), a `Data Service` is required and needs to be designed accordingly. This design decision is indicated as design decision 1 in figure 19. This decision depends on the interaction an IIF is used in. The actual way of interacting can depend on technical constraints exposed by an application system (cf. section 5.7.8). If it was decided for a `Data Service`, the following design decision has to be taken as well:

  - *How Should Communication Errors be Handled?* If the communication with an application system is erroneous, an IIF can handle this situation by either rejecting the initial call (both for synchronous and asynchronous scenarios with and without a 2PC) or by triggering an error procedure. The handling of this error procedure is, however, out of scope of the IIF and should be defined independently.

- *Does Incoming Data Need to be Validated?* The data that was either fetched or received could be validated. Whenever the overall scenario implies that erroneous data could be present (e.g., during communication with external parties) or if the validity of a request depends on actual user credentials, a `Validity Service` might be used in order to filter out invalid messages. Using a `Validity Service` implies another design decision:

  - *How to Handle Invalid Data?* If the validity of a message is checked by a `Validity Service`, it has to be decided at design-time how the IIF should react to invalid messages. An IIF can handle invalid data either by rejecting the initial call (both for synchronous and asynchronous scenarios with and without a 2PC) or by triggering an error procedure. The handling of this error procedure is, however, out of scope of the IIF.

- *Does an Incoming Request Need to be Closed or Acknowledged if Data is Received (and Potentially Valid)?* Based on the actual communication semantics in which an IIF is used, it may be required to inform the calling party about the success of processing the message. Depending on the scenario, the earliest possible point for closing the request is after the data was received and (potentially) validated.

- *Does Incoming Data Need to be Transformed into the Canonical Data Format?* Whenever an application system that is connected via an IIF does not support the canonical data format of the composite application, an `Heterogeneity Service` is required as part of an IIF in order to transform the data (cf. section 5.7.3).
- *Does the IIF Need to Trigger the Composite Application?* If an IIF is not used for receiving a response that was made using an IOF (cf. section 5.7.8), a `Trigger Service` might be required in order to trigger the computation within the composite application.
  - *Does the IIF need to Store Data into the Data Repository?* If a `Trigger Service` is used, it might also be required to store the received or fetched data in the data repository (cf. section 5.7.4).
- *Should the Communication be Closed After the Data Was Successfully Received?* Based on the actual communication semantics in which the IIF is used, it might be necessary to inform the calling party about the success of processing the message. Depending on the scenario, the latest possible point for closing the request is after the composite application was triggered.
  Depending on the realization of the service coordination, the extraction (out of the request's context) and propagation of a correlation identifier might also be required.

In order to realize an IIF, the platform for the DET needs – in addition to the requirements for the used IS – to support the following functionalities:

- **Queueing** With exactly-once and exactly-once-in order semantics
- **Reliable Messaging**
- **Synchronous Messaging**
- **Asynchronous Messaging**
- **Distributed Transaction with the Two-Phase Commit Protocol**
- **Service Orchestration** Basic workflow support for orchestrating the integration services is required
- **Workflow Pattern 1 – Sequence** of [108]
- **Workflow Pattern 4 – Exclusive Choice** of [108]
- **Workflow Pattern 5 – Simple Merge** of [108]

5.7.7 Integration Out-Flow

Purpose and Functionality Integration out-flows (IOF) are used by composite applications for the sake of communicating with back-end application systems and with external partners. As for the IIF, an IOF is an orchestration of integration services. Several design choices have to be taken in order to realize an actual IOF.

An IOF is called using the service protocol that is used throughout the composite application. Hence, a message in the canonical data model can be expected to be received. As the first step in the process, a pre-conversion can be performed by an `Heterogeneity Service`. This IS could transform the data independently of the actual receiver (design decision 1). An example of such a pre-conversion is to remove confidential data that must not be sent to external parties.
An IOF can be designed in a way to terminate an incoming call (closed for synchronous calls or a rejection reply to an asynchronous message) and/or to trigger an error compensation procedure if a transformation is not successful (design decision 1a).
If no pre-transformation was performed or if it was successful, a `Routing Service` is needed to determine the final receiver(s) of a message. Depending on the way the actual `Routing Service` is designed, multiple IOF instances might be spawned off by the `Routing Service` in order to individually handle the communication with each single application system that was identified as a communication partner (see section 5.7.5).
With the routing information available, the message can be transformed so it is receivable by the identified application system (design decision 2). As for the first transformation, conversion errors can be handled (design decision 2a).
Depending on the interaction mode in which the IOF is used, there are two options that can be used to communicate with an application system (design decision 3):
Whenever there is no response expected, an IOF will directly use a `Data Service` that posts the message to the application system. For the `Data Service` there is the consequence that it must not use the `Fetch Data` flow but the `Store Data` flow (design decisions 2 and 4 in figure 14). If there is an error during the communication, an IOF can terminate an incoming call (closed for synchronous calls or a rejection reply to an asynchronous message) and/or trigger an error compensation procedure (design decision 3b). In case of success, an IOF can also close or acknowledge the communication. However, no data from the application system can be transmitted back to the initial sender in such a scenario.
If an IOF is used in a request/response scenario, the IOF will trigger an IIF that handles the response-part of the communication. First, an IIF is determined and then executed in the context of the respective IOF. The used IIF will then use a `Fetch Data` activity of the initial `Data Service` but not the `Store Data` activity (design decisions 2 and 4 in figure 14). This way an IIF can synchronously or asynchronously communicate with the respective application system. The IIF also needs to be designed to transmit the reply to the initial requester. Additionally, the IIF will close (if required) the initial call to the IOF. This is possible if the IIF is executed in the context of the IOF as a sub-flow. The acknowledgment/closing of the request implies that the IOF passes either the correlation identifier or the endpoint reference to the IIF (cf. section 5.8).
The generic superset of IOFs with all necessary design decisions is described in the activity diagram of figure 20.

Design Decisions and Realization Requirements The design decisions that have to be considered prior to implementing an IOF, apart from the decisions for the single IS, are driven by the interaction requirements of the composite application. The interaction of a composite with heterogeneous application systems is realized using both, the IIF and the IOF. Hence, a part of the interaction is realized by an IOF.
The following list provides an operational overview of the design decisions that have to be taken for the IOF and upon which factors what they depend.

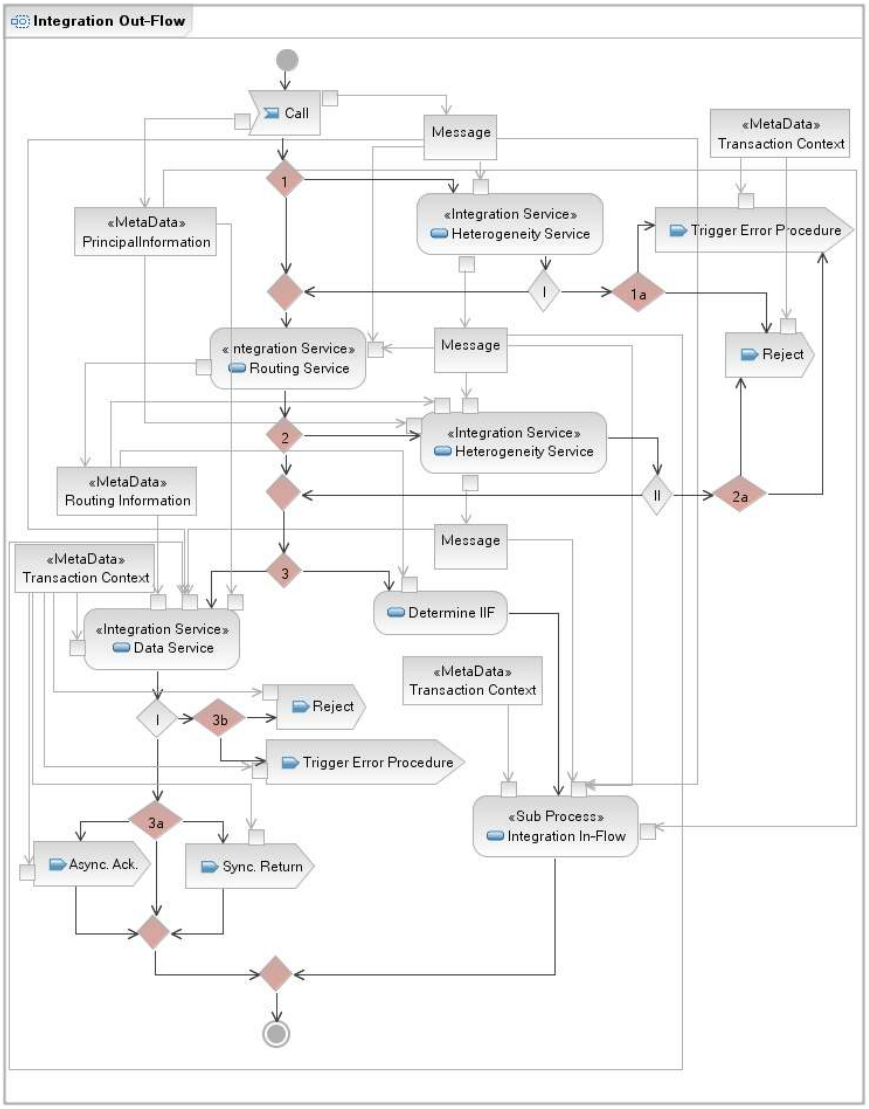


Figure 20: Activity Diagram of an `Integration Out-Flow`

- *Is a Data Transformation Required that is Independent of the Receiver?* Depending on the business data and the business process, a `Heterogeneity Service` might be used to transform an actual message independent of any application systems. This might be useful if common pre-conversions increase the re-usability of the single `Heterogeneity Services` and decrease the complexity of the receiver-dependent `Heterogeneity Services`.
  This option is indicated as design decision 1 in figure 20.
  - *How Should Transformation Errors be Handled?* If a message is transformed, it has to be decided at design-time how the IOF should react to erroneous message translations. An IOF can handle such errors either by rejecting the initial call (both for synchronous and asynchronous scenarios with and without a 2PC) and/or by triggering an error procedure. The handling of this error procedure is, however, out of scope of the IOF.
    This option is indicated as design option 1a in figure 20.
- *Who Should Receive the Message?* It is necessary that a `Routing Service` is de-

signed in a way that supports the actual interaction requirement of an IOF.

- *Is a Receiver-Dependent Data Transformation Required?* Depending on the business data and the actual application system, a `Heterogeneity Service` might be used to transform the actual message so that it can be received by the application system that was determined as the receiver.
  This option is indicated as design option 2 in figure 20.

  - *How Should Transformation Errors be Handled?* Also for this transformation step it has to be decided how possible transformation errors should be handled. An IOF can handle such errors either by rejecting the initial call (both, for synchronous and asynchronous scenarios with and without a 2PC) and/or by triggering an error procedure. The handling of this error procedure is, however, out of scope of the IOF.
    This option is indicated as design option 2a.

- *Which way of Updating Application Systems is Required?* Depending on the required interaction, the IOF might directly use a `Data Service` or invoke an IIF. The first alternative is chosen if no response from the application system to the composite application is required. The second option allows the sending of application data as a reply back to the composite application. This option is indicated as design option 3. According to the decision that is taken, a `Data Service` or an IIF needs to be designed respectively.

  - *What IIF Should be Used?* If the previous decision implies the application of an IIF, it must be determined which IIF should be used and the IOF needs to be configured/realized accordingly.
    The execution of the chosen IIF takes place in the thread of control of the calling IOF. The IIF is considered an ordinary service provider. If an IOF calls an IIF, the `Message Splitter` of the IIF's `Data Service` must not be used in order to avoid synchronization issues. Therefore, the call of the IIF can not create additional threads of control.
    If an IOF is called in the context of a distributed transaction, the IOF might pass this context to the IIF. Also user credentials might be propagated.
  - *What Communication Semantic is Required?* An IOF can support synchronous communication, asynchronous communication and asynchronous communication with acknowledgment. Depending on the required interaction, one option has to be chosen per IOF if a `Data Service` rather than an IIF is used for updating an application system. For both options that interact with a sender (asynchronous with acknowledgment and synchronous communication) no business data can be transmitted. Only a technical acknowledgment is possible. Supplying business data is only possible by using an additional IIF.
    This option is indicated as design option 3a in figure 20.

  If the `Data Service` of an IOF is directly used, it must be decided how successful and unsuccessful interactions with the application system should be treated (cf. section 5.7.1).

  - *How Should Successful Communication be Treated?* If a `Data Service` is used, an IOF possibly needs to close the request made by its requester (usually the composite application) or send an acknowledgment to the requester. The actual

way is use case dependent. This option is indicated as design decision 3a in figure 20.

– *How Should Unsuccessful Communication be Treated?* An IOF also needs to deal with errors that occur during the execution of the `Data Service`. Options are to reject the initial (synchronous or asynchronous) call and/or to trigger an error handling procedure. This option is indicated as design decision 3b.

In order to realize an IOF, the platform for the DET platform needs to support the following functionality (in addition to the requirements of the single integration services):

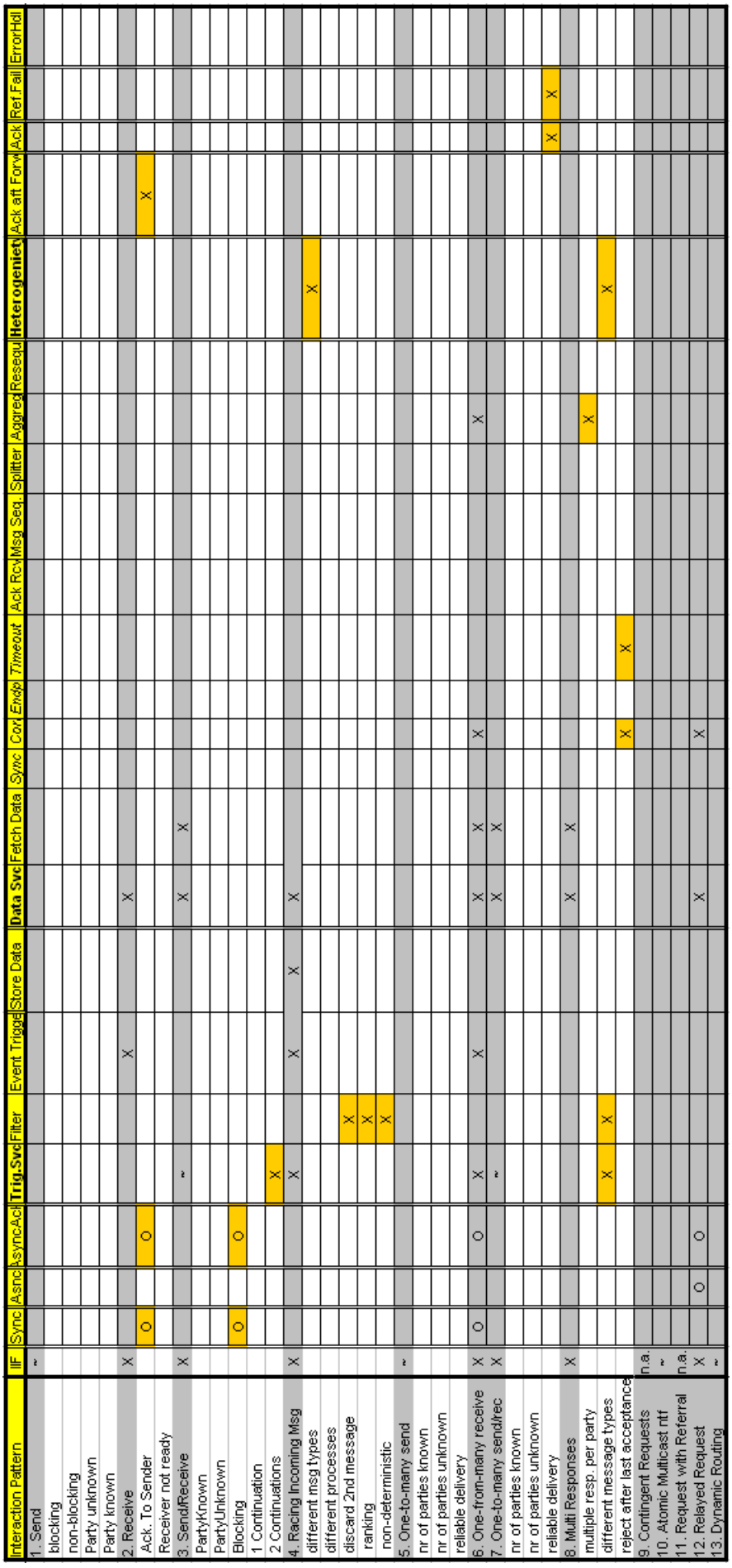
- **Queueing** With exactly-once and exactly-once-in order semantics
- **Reliable Messaging**
- **Synchronous Messaging**
- **Asynchronous Messaging**
- **Distributed Transaction with the Two-Phase Commit Protocol**
- **Service Orchestration** Basic workflow support for orchestrating the integration services is required
- **Workflow Pattern 1 – Sequence** of [108]
- **Workflow Pattern 4 – Exclusive Choice** of [108]
- **Workflow Pattern 5 – Simple Merge** of [108]
- **Workflow Pattern 14 – Multiple Instances With a Priori Runtime Knowledge** of [108] Additionally, the DET platform needs to support services to spawn off new threads in a way that creates multiple instances of subsequent activities.

5.7.8 Realizing Service Interactions with Heterogeneous Applications Using Integration Flows

The DET exposes its functionality using two integration flows that can be combined in order to realize more complex interaction scenarios. The DET exposes its functionality to the composite application (normally to the coordination layer) and addresses the concern of heterogeneity. The coordination layer addresses issues of service granularity and coordinates the interaction of the composite application with application systems. These interactions are usually routed through the integration flows that need to support the different models of service interaction.
This section outlines which forms of service interaction can be supported by a DET. Additionally, descriptions about how certain interaction models influence the design decisions for the DET layer are included. As a consequence, the design of the DET can be facilitated if the interaction models (that might arise out of business logic-imposed requirements or application-specific constraints) are defined for the level of service coordination.
The interactions that are analyzed are the service interaction patterns described in [98]. By using a DET, the interaction schemes are not only possible between services, they are also possible between the service consumers of the composite application and heterogeneous application systems that are not natively acting as service providers. Supporting these patterns leverages the realization of composite applications in heterogeneous application landscapes very efficiently.

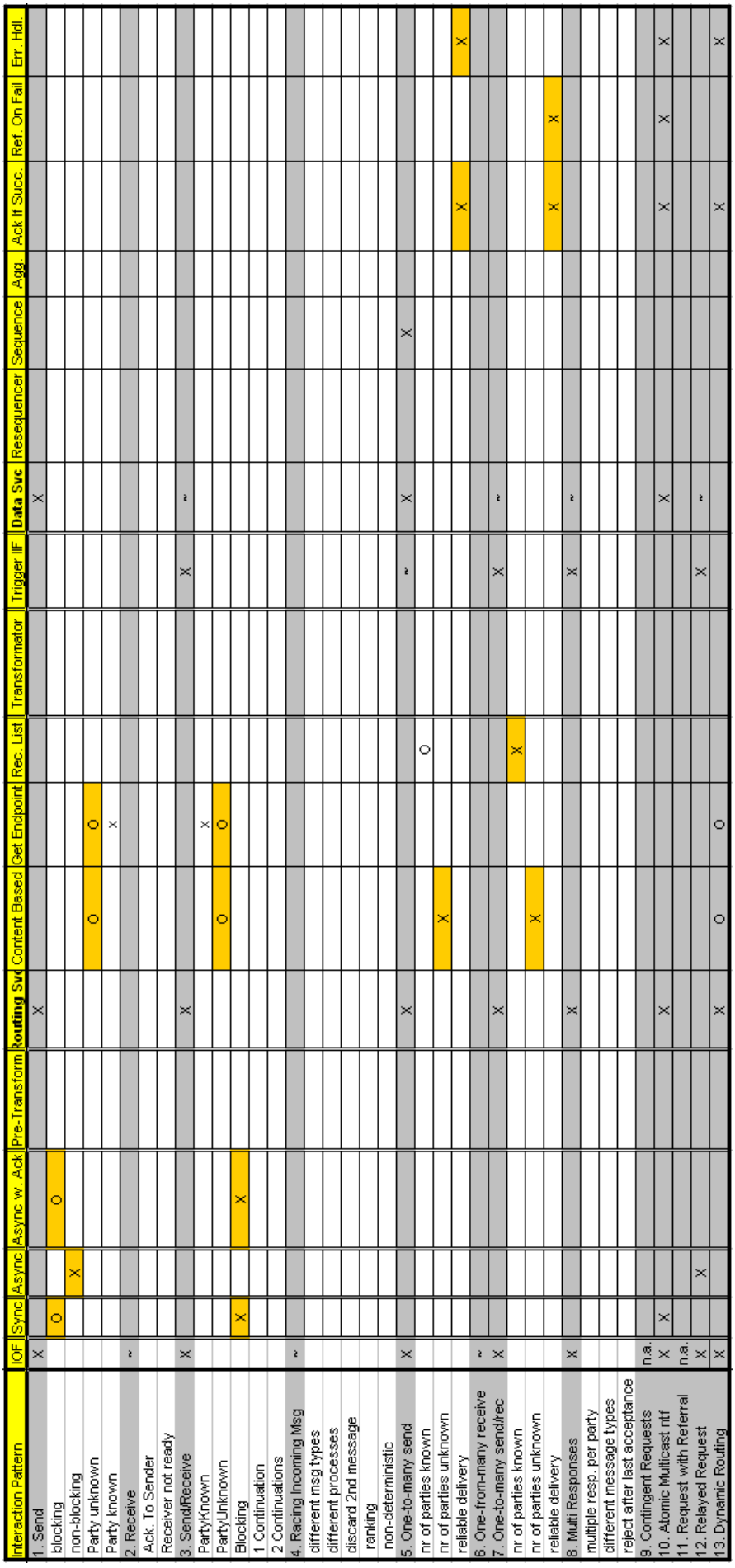
As a first indicator of how the interaction patterns can be supported with a DET, tables 17 and 18 give an overview of applicable service interaction patterns and the decisions they impact on the DET. The different interaction patterns might require the deployment of an IIF and/or an IOF. Additionally, they influence which integration services are required and how they should be configured. The implications from interaction patterns to an IIF are shown in table 17.

The integration patterns and their related design choices are indicated vertically in the first column. The headline lists the design options for the DET ordered by the integration service for which they have to be taken.
Crosses indicate that a certain interaction pattern must be realized by using a certain integration flow, integration service and potentially one of its design options. Swung dashes indicate that the actual pair of patterns can not be found together. The use of the respective integration pattern can not be realized using the indicated DET artifact. If two fields in one row are marked with an o, one of the marked options has to be used. All un-marked pairings are optional.
Similar to table 17, table 18 describes the relations between interaction patterns and the artifacts of the IOF.



Interaction Pattern	IIF	Sync	Asnch	syncAck	Trig.Svc	Filter	Event Trigge	Store Data	Data Svc	Fetch Data	Sync	Cor	Endpl	Timeout	Ack Rcv	Msg Seq.	Splitter	Aggreg	Resequ	Heterogeniety	Ack aft Forw	Ack	Ref.Fail	ErrorHdl
1. Send	~																							
blocking																								
non-blocking																								
Party unknown																								
Party known																								
2. Receive	X						X		X															
Ack. To Sender		O		O																	X			
Receiver not ready																								
3. Send/Receive	X				~				X	X														
PartyKnown																								
PartyUnknown																								
Blocking		O		O																				
1 Continuation																								
2 Continuations					X																			
4. Racing Incoming Msg	X				X		X	X	X															
different msg types																				X				
different processes																								
discard 2nd message						X																		
ranking						X																		
non-deterministic						X																		
5. One-to-many send	~																							
nr of parties known																								
nr of parties unknown																								
reliable delivery																								
6. One-from-many receive	X	O		O	X		X		X	X		X						X						
7. One-to-many send/rec	X				~				X	X														
nr of parties known																								
nr of parties unknown																								
reliable delivery																						X	X	
8. Multi Responses	X								X	X								X						
multiple resp. per party																								
different message types					X	X														X				
reject after last acceptance												X		X										
9. Contingent Requests	n.a.																							
10. Atomic Multicast ntf	~																							
11. Request with Referral	n.a.																							
12. Relayed Request	X		O		O				X			X												
13. Dynamic Routing	~																							

Table 17: How Service Interaction Requirements affect an IIF



Interaction Pattern	IOF	Sync	Async	Async w. Ack	Pre-Transform	Routing Svc	Content Based	Get Endpoint	Rec. List	Transformator	Trigger IIF	Data Svc	Resequencer	Sequence	Agg.	Ack If Succ.	Ref. On Fail	Err. Hdl.
1. Send	X					X						X						
blocking		O		O														
non-blocking			X															
Party unknown							O	O										
Party known								x										
2. Receive	~																	
Ack. To Sender																		
Receiver not ready																		
3. Send/Receive	X					X					X	~						
PartyKnown								x										
PartyUnknown							O	O										
Blocking		X		X														
1 Continuation																		
2 Continuations																		
4. Racing Incoming Msg	~																	
different msg types																		
different processes																		
discard 2nd message																		
ranking																		
non-deterministic																		
5. One-to-many send	X					X					~	X		X				
nr of parties known									O									
nr of parties unknown							X											
reliable delivery																X		X
6. One-from-many receive	~																	
7. One-to-many send/rec	X					X					X	~						
nr of parties known									X									
nr of parties unknown							X											
reliable delivery																X	X	
8. Multi Responses	X					X					X	~						
multiple resp. per party																		
different message types																		
reject after last acceptance																		
9. Contingent Requests	n.a.																	
10. Atomic Multicast ntf	X	X				X						X				X	X	X
11. Request with Referral	n.a.																	
12. Relayed Request	X		X								X	~						
13. Dynamic Routing	X					X	O	O								X		X

Table 18: How Service Interaction Requirements affect an IOF

In order to facilitate the understanding of these tables an example is provided:

Use Case:
On the coordination layer of a composite application, there might be a coordination according to the RosettaNet PIP 3A3 [110] – *Request Price and Availability*. This PIP is – according to [98] – captured by the means of the service interaction pattern 7 *One-to-Many Send/Receive*. The first design choice for this pattern is whether the parties that receive the messages are known or unknown. The second decision is whether reliable delivery is required. Let's assume that the list of recipients is known and that reliable delivery is required.
Configuration: first, it can be identified that the realization of the interaction pattern 7 always requires the IIF as well as the IOF. Concerning the IOF that leverages the *One-to-Many Send*-Part of the pattern, the functionality of a `Routing Service` is required. As the recipients are known, the `RecipientList` pattern is also identified. In order to support reliable delivery, the `Acknowledge if Success` and the `Refuse on Failure` parts of the IIF (that realizes the actual communication with the back-end system) need to be activated in order to give a feedback to the composite for each message. This is necessary for each message that was triggered by the *RecipientList* of the `Routing Service`.

The tables 17 and 18 support the design of the DET by providing an overview of the possible ways to realize the single service interaction patterns using integration flows and services. The following list provides a more detailed description of design implications and how the design of the DET can be facilitated. For all forms of service interaction it is assumed that the service consumer is the composite application while the service providers are the back-end application systems.

- **Send - Service Interaction Pattern 1** of [98]

  The *send* pattern describes that "a party sends a message to another party" [98, p. 306]. Whenever the receiving party does not use the canonical data format nor does not provision services as the composite application requires them, an IOF is required. As there is no response foreseen in this pattern, design decision 3 of the IOF (cf. figure 20) has to be decided in a way that the `Data Service` is used for updating an application system and not the IIF and its `Data Service`. Depending on the design decisions that are taken for the interaction pattern, different implications on the IOF exist. These implications are:

  - *Blocked Sender for Reliable Messaging* Reliable messaging is a requirement for the platform of the DET and does not need to be realized by an IOF. If the service consumer, however, blocks, it needs to be notified by the IOF. This might be realized by using an acknowledgment for the asynchronous message or by terminating a synchronous call.
  - *Non-Blocked Ssender for Reliable or Ordinary Messaging* If a reply to a message is required, design decision 3a has to be taken in a way that the IOF terminates after an application system is updated.
  - *A Priori Known Receiver* If the (final) receiver of a message is a priori known, the `Routing Service` of the respective IOF has to be designed in a way that the endpoint of that receiver is available (design decision 1 in figure 18).

  - *A Priori Unknown Receiver* If the (final) receiver of a message is a priori not known, the final receiver can only be determined by applying a `Content-Based Router` as routing mechanism for the `Routing Service` or by looking up a registry entry.

- **Receive - Service Interaction Pattern 2** of [98]

  The *receive* pattern describes that "a party receives a message from another party" [98, p. 307]. Whenever the party that sends a message either does not potentially use the canonical data format, does not consume services as the composite application provisions them or (and this is a very probable scenario) does not stick to the event-driven approach to trigger a composite application, an IIF is required.
  If an IIF is used to support the *Receive* pattern, the `Trigger Service` is required. Storing data, however, is optional since the message that is to be received might not contain a payload. Depending on the design decisions that are taken for the interaction pattern, different implications exist for the IIF. These implications are:

  - *Acknowledgement for Sender* If the sender of the respective message requires an acknowledgment, the IIF has to close a successful request by sending a signal. This might be realized either by closing a synchronous request or by sending an acknowledgment for an asynchronous message. Both options can not contain any payload.
    If the `Validity Service` is used to check whether the message that has to be received is valid, and the message is not valid, the message has to be rejected by the IIF.
  - *Receiver not ready* This option does not require any design choice at the IIF level. It is assumed that the DET platform uses queuing and buffers messages if a recipient is not ready.

- **Send/Receive - Service Interaction Pattern 3** of [98]

  The *send/receive* pattern describes that "a party X [(the composite application)] engages in two causally related interactions: in the first interaction X sends a message to another party Y (the request), while in the second one X receives a message from Y (the response)" [98, p. 308]. This service interaction pattern involves both, an IIF and an IOF. The part of this pattern that describes the sending is realized using an IOF. The IOF is designed as described for the *Send* pattern except that a `Data Service` does not update the application system but rather an IIF is triggered which must first call a `Data Service`. In this `Data Service`, the `Fetch Data` micro-flow uses the incoming message (that was sent by the composite application) in order to retrieve the response. The `Retrieve Data` step of the `Fetch Data` micro-flow needs to be configured according to the *request/response* scenario the integration flows are used in. The single implications of the design choices of this interaction pattern are:

  - *Unknown Counter-Party* According to the *Send* pattern, an IOF can be configured to use a `Content-Based Router` or to lookup a service registry within its `Routing Service`. However, this only determines the respective IIF that has to be used. As the IIF has no routing option included, the IOF has to trigger an appropriate IIF that is configured for sending to the appropriate application system. This mechanism implies the constraint that the possible receivers need to be known in advance. The decision of the actual receiver can be made during run-time.

  - *Correlation of Outgoing and Incoming Message* Even if [98] relates this as a design choice with the *send/receive* interaction pattern, the correlation is only required if asynchronous messaging is used. If this is the case, the `Data Service` of the IIF needs to be configured as described in the design decision "*correlation or fixed call-back endpoint?*" that influences the `Retrieve Data` activity.
  - *Fault of Either of the Messages* Communication errors might occur during the execution of the `Data Service` of the IIF. The IIF needs to be designed in a way to handle errors (design decision 2a of figure 19) if so required.
  - *Blocking Sender* Whenever the sender is blocked, the IIF needs to close the communication that was opened by calling the IOF (This is indicated by the design decisions 3 and 6 of figure 19).

- **Racing Incoming Messages - Service Interaction Pattern 4** of [98]

  The *racing incoming messages* pattern describes that "a party expects to receive one among a set of messages. These messages may be structurally different (i.e. different types) and may come from different categories of partners. The way a message is processed depends on its type and/or the category of partner from which it comes" [98, p. 309]. This pattern is realized by using an IIF. The basic functionality is realized using a `Trigger Service`. It maps incoming requests into the according event types. This mapping is determined by the business logic. According to the chosen type of event, the processing of the single message can vary. The data that is transmitted in the incoming messages is stored by a `Trigger Service` into the data repository.
  The single implications of the design choices of this interaction pattern are:

  - *Incoming Messages of Different Types* If the incoming messages are not of the canonical data format, the `Heterogeneity Service` is required.
  - *Incoming Messages Require Different Processing* Through the basic usage of events, this is supported by defining different event types for messages that need different processing.
  - *After One Message Was Processed, Following Messages Need to be Discarded* By using a `Message Filter` of a `Trigger Service`, messages can easily be discarded. The `Message Filter` has – in such cases – to be stateful.
  - *Simultaneously Available Messages* This behavior can be prohibited by using queueing (with exactly-once-in order semantics) as the transport mechanism of the DET. A filter of a `Trigger Service` might also be used.

- **One-to-Many Send - Service Interaction Pattern 5** of [98]

  The *one-to-many send* pattern describes that "a party sends messages to several parties. The messages all have the same type (although their contents may be different)" [98, p. 310]. This pattern can be realized by using an IOF. It requires the use of a `Data Service` (instead of a send via the IIF) and the use of a `Routing Service`. The `Routing Service` has either to be used with a `Recipient List` or a `Content-Based Router`. If required (even if not included in the pattern) the message could be transformed specifically for each determined receiver. The single implications of the design choices of this interaction pattern are:

  - *Unknown Number of Parties* A `Content-Based Router` is the only mechanism to dynamically define multiple recipients of a message. Whenever the receivers are not determined by actual message payload, the scenario can not be supported by the IOF.
  - *Notification of Delivery* The composite application can be notified of successful deliveries of all messages if the DET supports the synchronization of control flows that are opened for the single receivers as described by the workflow pattern 14, *Multiple Instances With a Priori Runtime Knowledge*, of [108]. In all cases, for each failed message, an error procedure can be triggered (design decision 3b in figure 20).

- **One-from-Many Receive - Service Interaction Pattern 6** of [98]

  The *one-from-many receive* pattern states that "a party [(the composite)] receives a number of logically related messages that arise from autonomous events occurring at different parties. The arrival of messages needs to be timely so that they can be correlated as a single logical request. The interaction may complete successfully or not depending on the set of messages gathered" [98, p. 312]. As described by [98], the *One-from-many receive* pattern can be realized by the use of aggregation with correlation. Thus, the aggregator of an IIF is required and its `Data Service` needs to be designed in a way to support correlation (design decision 3a of figure 14). A timeout or another stop-condition of the aggregator needs to be set accordingly. In case of unsuccessful aggregation, an error handling procedure might be triggered (design decision 3b of figure 14). In order to trigger the composite application with the received message, a `Trigger Service` is required.

- **One-to-Many Send/Receive - Service Interaction Pattern 7** of [98]

  The *one-to-many send/receive* pattern describes that "a party sends a request to several other parties, which may all be identical or logically related. Responses are expected within a given time frame. However, some responses may not arrive within the time frame and some parties may not even respond at all. The interaction may be completed successfully or not depending on the set of responses gathered" [98, p. 311]. Realizing this pattern requires both, an IOF that is used to address messages to multiple parties and an IIF that is used to gather the responses. Hence, the IOF needs to be configured to not use a `Data Service` by itself but to use an IIF and its `Data Service`. This `Data Service` can synchronously fetch responses from the application system or communicate in an asynchronous way using correlation or call-back endpoints.
  The single implications of the design choices of this interaction pattern are:

  - *Unknown Number of Parties* A `Content-Based Router` is the only mechanism to dynamically identify multiple recipients of a message. Whenever the receivers are not determined by the actual message payload, the scenario can not be supported by an IOF. The `Routing Service` of the IOF is required to determine the IIF for each application system. If multiple application systems are identified, multiple IIFs are initiated.
  - *Correlation of Responses* The `Retrieve Data` activity of the `Data Service` of the IIF that connects to the actual application system is required to use correlation in order to support this scenario (design decision 1 of figure 16)

- *Avoid Indefinite Waiting Period* A timeout can be used in the `Retrieve Data` part of the `Data Service` in order to avoid the IIF waiting for an indefinite period for answers from the application systems. Accordingly, an error behavior needs to be specified for the timeouts (design decision 1a of figure 16).
- *Reliable Delivery* Despite the reliable communication that is supported by the platform of the DET (queueing with exactly-once (in-order) delivery capabilities), it might be required to inform the composite application about the success or error of an interaction.
  The composite application can be notified about the successful delivery of all messages if the DET supports the synchronization of control flows that are opened for the single receivers as described by the workflow pattern 14, *multiple instances with a priori runtime knowledge*, of [108]. In all cases, for each failed message, an error procedure can be triggered (design decision 3b in figure 20). Single notifications for single application systems are not supported.

- **Multi-Responses - Service Interaction Pattern 8** of [98]

  The *multi-responses* pattern describes that "a party X [(the composite)] sends a request to another party Y. Subsequently, X receives any number of responses from Y until no further responses are required. The trigger of no further responses can arise from a temporal condition or message content, and can arise from either X or Y's side. Responses are no longer expected from Y after one or a combination of the following events: (i) X sends a notification to stop; (ii) a relative or absolute deadline indicated by X; (iii) an interval of inactivity during which X does not receive any response from Y; (iv) a message from Y indicating to X that no further responses will follow. From this point on, no further messages from Y will be accepted by X" [98, p. 312f.]. This interaction pattern is supported by using both types of integration flows. As described by the *racing incoming messages* pattern and its solutions, different incoming messages are forwarded to the application system by the notion of an IIF and its `Trigger Service`. A filter can be used to discard unnecessary incoming messages.
  Outgoing messages are initially distributed using an IOF. Such an IOF does not directly use a `Data Service`. Furthermore it uses an IIF for receiving the messages. Only asynchronous communication is possible if the DET is used to support this pattern. This way both, the determination of the appropriate request to close and the restriction to only receive one response are avoided.
  The single implications of the design choices of this interaction pattern are:

  - *Reception of Multiple Messages from One Party* This design option can only be supported by aggregating the responses from one application system by the notion of an `Aggregator` that is used as part of the IIF's `Data Service`.
  - *Incoming Messages of Different Types* If the incoming messages are not in the canonical data format, a `Heterogeneity Service` is required.
  - *Notify Application System if no More Messages Are Accepted* If no more messages are accepted, the `Retrieve Data` activity of the IIF's `Data Service` must be configured to reject incoming messages. This is only possible by using a timeout (design decision 1a in figure 16). If the composite application needs to actively notify the application systems before an error can occur, a *one-to-many send* scenario needs to be realized separately as an error handling procedure.

- **Contingent Requests - Service Interaction Pattern 9** of [98]

  The *contingent requests* pattern describes that "a party X makes a request to another party Y. If X does not receive a response within a certain time frame, X alternatively sends a request to another party Z, and so on" [98, p. 314]. This pattern is not directly supported by the DET. Whenever this is required, multiple request/response scenarios with different receivers should be realized. The actual logic that initiates a new request has to reside at the level that controls the DET – the service coordination layer.

- *Atomic Multicast Notification - Service Interaction Pattern 10* of [98]

  The *atomic multicast notification* pattern describes that "a party sends notifications to several parties such that a certain number of parties are required to accept the notification within a certain time frame. For example, all parties or just one party are required to accept the notification. In general, the constraint for successful notification applies over a range between a minimum and maximum number" [98, p. 315]. This interaction pattern is supported in an all-or-nothing fashion by an IOF and its `Data Service` by including a transactional context in the request. The actual transaction handling (commit/abort) needs to be realized by the actual platform. The IOF simply starts the transaction by including the transactional context in the calls to the application systems. Hence, the actual support of the atomicity lies in the platform and not within the flow logic of the IOF. The single implications of the design choices of this interaction pattern are:

  – *Unknown Number of Parties* A `Content-Based Router` is the only mechanism to dynamically identify multiple recipients of the multicast message. Whenever the receivers are not determined by actual message payload, the scenario can not be supported by an IOF.
  The `Routing Service` of an IOF is required to provide required routing information to the `Data Service`. If multiple application systems are identified, multiple `Data Services` are started. The transactional context needs to be transmitted to every application system.

- **Request with Referral - Service Interaction Pattern 11** of [98]

  The *request with referral* pattern states that "party A sends a request to party B indicating that any follow-up response should be sent to a number of other parties (P_1, P_2, ..., P_n) depending on the evaluation of certain conditions. While faults are sent by default to these parties, they could alternatively be sent to another nominated party (which may be party A)" [98, p. 316]. As the DET solely describes the integration of application systems under the central control of a composite application, there is no need for the composite application to refer the control to another party. This pattern is not supported by the DET.

- **Relayed Request - Service Interaction Pattern 12** of [98]

  The *relayed request* pattern states that "party A makes a request to party B which delegates the request to other parties (P_1, ..., P_n). Parties P_1, ..., P_n then continue interactions with party A while party B observes a 'view' of the interactions including faults. The interacting parties are aware of this 'view' (as part of the condition to interact)" [98, p. 317]. In asynchronous *send/receive* scenarios with correlation, the application system(s) can defer the communication with the composite application

to other application systems. In order to create a "view" of the communication, the communication protocol used between the DET and the application systems has to be the same. The application systems as well as the respective adapters of the connectivity layer need to support "Cc-ing" other communication parties (cf. [98, p. 318]).

- **Dynamic Routing - Service Interaction Pattern 13** of [98]

  The *dynamic routing* pattern states that "a request is required to be routed to several parties based on a routing condition. The routing order is flexible and more than one party can be activated to receive a request. When the parties that were issued the request are finished, the next set of parties are passed the request. Routing can be subject to dynamic conditions based on data contained in the original request or obtained in one of the 'intermediate steps'" [98, p. 317f.]. The basic notion of this interaction pattern can be realized by using an IOF with a `Content-Based Router`. Whenever a feedback from the application systems is required to send the message(s) to more application systems, the DET can not handle this logic alone. The coordination layer needs to support this behavior. Hence, an IOF might use an IIF according to the *one-to-many send* with an unknown number of parties and reliable delivery. If the service coordination layer is informed about the delivery, it might interact with more application systems using another IOF. Optionally, atomicity can be realized using distributed transactions.

5.8 Service Coordination Layer

Purpose and Functionality From a top-down perspective the integration flows provide, in addition to the connectivity layer, a standardized mechanism for interacting with application systems. This is irrespective of communication or computational semantics and provides homogeneous data access as well.
As technical heterogeneity is addressed by the presented mechanisms, the services that are exposed by the integration flows can easily be orchestrated by using an orchestration engine. Without further concepts, however, the functionality that is provided by these services would be determined by the functionality offered by the application systems. Such services are aligned with neither the business requirements nor the business tasks of a company. Hence, the business processes of an organization could either not be used to generate service orchestrations or the business processes would be restricted by the actual application systems. In order to apply the paradigm of service orientation and compose new business-centric functionality out of existent application functionality or to enrich functions within a specific context, it might be appropriate to combine the application specific functionality with new functionality. Expressed differently, application services might be needed to be aggregated to more problem-oriented services (*enterprise services*). The service coordination layer addresses this issue also referred to as *service mediation* (cf. [111]). It can be used in order to invoke 2 to n basic services using the integration flows in order to form the enterprise services that are then orchestrated. This is similar to the description of *business-driven service* pattern in [81] where services are orchestrated in so-called micro-flows in order to correspond with the services that are orchestrated by a macro-flow. This composition of low-level services to *business-driven services* is considered rather static (cf. [81, pp. 44f.]).
[112] describes patterns that demonstrate how the gap between application services and a

business process-centric orchestration can be dealt with using a so-called *process support layer* as a mediator. In particular, granularity problems and interdependency problems that prohibit the direct use of application systems from orchestrations are addressed. The *Composition*, *Decomposition* and *Bulk Service* patterns describe how differing granularity can be dealt with. The patterns *Sequentializing* and *Reordering* describe how interdependency problems can be addressed. Realizing such patterns is the purpose of the coordination layer.

In contrast to the *business-driven service* pattern of [81] and the *process support layer* of [112], the coordination layer is recursive. This means that an aggregated service that is composed at this layer might be aggregated again with services from this or lower layers to expose other high-level services. The benefit of this approach is that the aggregations themselves can remain flexible and their re-usability is increased.
A service coordination layer might aggregate both, company internal and external services to services that are in turn usable both company internally and externally. This also introduces the need to support business protocols. These business protocols are sets of actions that have to be performed by multiple parties in order to allow successful execution of certain business functionality (cf. e.g. [110]). They can be realized at the layer of service coordination.

An aggregation of services at the service coordination layer might also be required due to technical reasons. The service coordination layer exposes services to the service orchestration layer of a composite application (cf. section 5.9.1). Such orchestrations are designed in alignment with business processes and not with "technical" constraints in mind. Whenever a multi-resource interaction is required that ensures consistent state transitions, this might be an indicator for the need of a technically motivated service aggregation. Consistent state transitions can be ensured by two means of transaction handling. Rather short-term transactions fulfilling the ACID properties by locking and rollback mechanisms or more long-term transactions without locking and with compensation actions (cf. section 3.2.2). Of course, the orchestrated functionality is offered by the application systems and the consistent state transition is also assured by these systems.
Orchestrating the services of these systems does, however, raise the need for a cross-service transactional coordination. Even if the application systems have to support the transactional coordination by appropriate compensation operations and/or by supporting transactional protocols, the coordination itself has to be controlled at this layer of the composite application. The service coordination layer controls distributed transactions by initiating them and passing the transactional context to application systems (possibly via the DET) as well as to the data repository.
According to [73], transactional coordination can consist of two layers. One layer for so-called *local transactions* with ACID properties and one for *global transactions* with relaxed transactional properties. The latter one uses ACID transactions as black-boxed functionality to form long-term global transactions. By distinguishing these two layers of transaction handling, the idea of separating concerns of different transactional properties is incorporated.
Composite applications that are implemented using the presented reference architecture realize local transactions at the service coordination layer as it acts as the controlling instance for ACID transactions .
Meeting the long-term characteristics of global transactions, the isolation and atomicity properties can be relaxed and so-called safepoints can be used (cf. [73]). Relaxing the iso-

lation property is realized by publishing intermediated results to the global context (for the presented architecture this is the data repository). Atomicity is relaxed by introducing compensating transactions that "undo" other transactions. Both, context publication and compensation transactions are local transactions.

Safepoints are local transactions that are marked by this special property of being a safepoint. Thus, the fundamental support for global transactions is formed by local transactions. This point of view is in line with the concept of a recursive service coordination layer.
[73] also proposes a way to specify transactional properties (such as the safe point properties for local transactions) and an execution model that supports global transactions based on these specifications. This execution model dynamically calculates workflow paths for partial or complete compensation of global transactions, if required. At the given point this is, however, not seen as a mandatory feature for a composite application. This is why only the notion of short-term and long-term transactions and the notion of compensating transactions are considered a necessity for composite applications.
Another necessity for deploying a service coordination layer is to realize complex interaction patterns. As described in section 5.7.8, a DET offers means for supporting most of the known service interaction patterns. However, the *contingent request* can not be implemented by solely using the DET. The service coordination layer is required here as well.

From a design point of view, the service coordination layer introduces a layer of aggregators. On one hand, this increases modifiability as it decouples application systems from the business process logic of the service orchestration layer. An indicator of this positive mechanism of aggregators is the *System's Service Coupling (SSC)* metric (cf. section 3.2.1) that indicates less complex systems in terms of coupling if aggregators are deployed in a system.
On the other hand an, additional layer of aggregation besides the centralized control of the process orchestration decreases the ability to deal with a system's complexity . This is because modifications have to not just be performed at one single component of a composite application. The decreased control centralization that occurs if a service coordination layer is introduced, is indicated by a decreased SCZ-value. The *System's CentraliZation (SCZ)* metric is sensitive in terms of deploying aggregates exhaustively (cf. section 3.2.1). There is a trade-off between the SSC and SCZ (that is captured by the *Aggregator CentraliZation* metric ACZ). The service coordination layer is introduced since the increased modifiability of the service orchestration layer and the decreased overall coupling is advantageous over a fully centralized control. The level of modifiability is, however, not optimal. The only solution would be to directly orchestrate application services. If these application services are designed accordingly, both modifiability and complete control centralization become possible.
As the latter scenario will hardly be realizable today in a real-life context, the "amount" of control logic that resists outside the central orchestration layer should be minimized. The consequence for the layer of service coordination is that the contained control logic needs to be kept as simple as possible. An indicator that addresses this idea from a service-external viewpoint is the ACZ metric. This metric underlies the idea that service mediation should not blur a centralized control model.

Design Decisions and Realization Requirements The layer of service coordination performs service aggregation of application services that are (possibly) mediated by the integration flows of the DET. [81] proposes the notion of a *Microflow Engine*. Such an engine basically performs service invocations in a defined sequence (as described by the workflow patterns 1 *Sequence*, 4 *Exclusive Choice* and 5 *Merge* [108]). Additionally, it manages the state of the flow. However, since this reference includes a central context repository for a composite application (cf. section 5.6), the platform for the service coordination layer does (and must not) not keep a context.
The patterns for granularity and interdependency of [112] also require just a sequence of service invocations with state management and basic branching facilities. This eases the realization of this layer. Hence, from this point of view service coordination should be nothing more than a sequence of service invocations with some basic branching facilities. However, the service coordination layer is the central control instance for distributed transactions. "Safepoints" for long-running transactions are simply exposed as services, too. ACID transactions, however, need to be supported by the service coordination layer's platform. Distributed ACID transactions are initiated at this layer and passed to the data repository as well as to the DET. This means that the service coordination platform has to support transactional protocols both for communicating with the data repository as well as with the application systems and the DET. If these protocols are different (e.g., due to performance aspects), this also needs to be considered.
The service coordination layer needs to access the data repository and to expose the aggregated services (the so-called enterprise services) to the service orchestration layer. To accomplish this, the service coordination layer "speaks" to the service orchestration as well as to the DET using the common protocol of the composite application. Usually, the platform of the service coordination layer will use a notion of client proxies for interacting with the DET and the application systems. In order to integrate the state management that is offered by the data repository, the service coordination platform needs to provide a means to integrate access to the data repository's smart proxies, as well. As the data repository is proprietary to the presented reference architecture, it is (most likely) not possible to use standard service aggregator platforms. Also, exchanging interactions with services is very likely to require additional overhead. This is because not only service connections need to be established but also the smart proxies to be included. Changes to this layer are likely to not be very frequent, though.
As the service coordination layer interacts with external application systems as well as with the DET, it needs to deal with the actual endpoints of these service providers, too. As a consequence, the platform of the service coordination layer needs to offer the possibility of managing the actual endpoints of the services.
In contrast to a `Routing Service`, no routing logic is required. However, it needs to be decided whether the service coordination layer should use static service endpoints or dynamic lookups of the defined services (using a service registry – cf. section 5.10).

In order to realize the *contingent request* service interaction pattern the service coordination layer needs to register timeouts. Hence, the actual platform needs to either provide means to perform service calls with timeouts or to provide some sort of a timer mechanism that can interrupt service calls (or wait for replies).

From a communication semantics point of view, the DET exposes all necessary means to homogenize the applied communication semantics at the service coordination layer. This means that a service coordination could rely completely on synchronous communication while the interaction with the back-end systems is asynchronous. There are some de-

sign considerations, however. Some necessary service interactions (e.g., the *one-to-many send/receive* with dynamic routing) might require the receipt of more replies than requests that were made (while the actual number of replies is determined within the DET). Additionally, synchronous communication might be inappropriate since replies might be only expected "late" and will require active "waiting" for the response and thus, might require runtime resources. Also synchronous communication couples service consumers (the coordination layer) and service providers (the application system) in terms of availability. This is because short service interruptions of a service provider can be potentially masked by using asynchronous communication.
This is why a service coordination layer needs to support asynchronous messaging. In turn, a call-back mechanism needs to also be included. Additionally the notion of call-back interfaces either requires means for correlating messages or exposing call-back endpoints. As a request and a response are not necessarily related, the platform of the coordination layer either needs to add correlation fields into request messages or provide a mechanism for (stateful) call-back endpoints. Both solutions also imply the need for the DET to support the chosen approach. This might be realized either by keeping correlation identifiers or by using an addressing protocol that supports multiple endpoints.
A good candidate for a correlation identifier (cf. [95, 163]) is the actual `Event` that is being computed.

The service coordination layer and the DET usually communicate by using "complete" service interfaces that describe the complete structure of transmitted data. This means that the mediation of service providers by a DET is transparent to the service coordination. In order to "enrich" the composite-internal, event-based communication, a smart proxy is required. This way, the spaces-based communication can be connected with the messaging-based external world.
By using a smart proxy, necessary data is loaded from the data repository and passed to the DET. If there is a reply involved, there are two different design options. The first one is to pass data from the DET back to the service coordination and the service coordination storing the data (back) into the data repository. This is the recommended method. As a DET can also utilize a `Trigger Service`, that can interact with the data repository directly, multiple replies can be put into the data repository while only one call is closed. By choosing the second approach, interactions that involve *multi responses* can be realized while the coordination layer stays agnostic of the actual number of communication partners. In order to realize such an approach, `Events` need to be used as correlation identifiers (as they are required by the data repository) and the platform of both, the DET and the service coordination layer needs to be able to pass the transactional context from the coordination layer to the DET. This might be required if the coordination layer opens a transaction that needs to be closed by the IIF of the DET.

To summarize, the platform that is used to realize a service coordination needs to support the following functionality:

- **Asynchronous messaging** Correlation and or callback endpoints
- **Endpoint Management**
- **Integration of proprietary** `Smart Proxies`
- **Coordination capabilities for distributed transactions**

- **Workflow pattern 1 – Sequence** of [108]
- **Workflow pattern 4 – Exclusive Choice** of [108]
- **Workflow pattern 5 – Simple Merge of** [108]
- **Timeouts in requests**

The following list provides an operational overview of the design decisions that have to be taken for the service coordination layer and upon which factors they depend.

- **Communication semantics with external service providers** Both, asynchronous and synchronous communication is technically feasible. However, asynchronous communication should be the default. Therefore synchronous communication should only be used when needed. This decision might be based on quality-of-service requirements.
- **Way of realizing asynchronous interactions** Different ways of asynchronous communication are possible but are more dependent on the actual platform than on the actual use case. Nevertheless, use case-specific requirements might determine the way asynchronous communication is realized. The possible approaches to asynchronous communication are:
  - **Correlation identifier** Asynchronous communication via a callback-identifier (cf. [95, 163]) imposes less constraints to a platform than call-back endpoints do. This is because correlation does not rely on addressing protocols but on identifiers within the communication. If, however, common identifiers within request and response messages do not exist, call-back endpoints might be necessary.
  - **Call-back endpoints** Call-back endpoints rely on an addressing protocol that realizes stateful communication among certain instances of service agents. If no correlation identifier is applicable, this might be the only solution for realizing asynchronous communication.
- **Dynamic or static endpoints** The service coordination layer might communicate both, with the DET and application systems. The endpoint references to the respective service providers might either be fixed during design-time or looked-up during run-time using a registry. Usually, there are no reasons for dynamically looking up integration flows of the DET. However if no integration flow is used for communicating with a certain (class of) application system(s), dynamic lookups of the actual endpoint might be required. If so, this has an impact on the service registry that needs then to collaborate with the service coordination layer.
- **Classical request/response or reply via data repository** As described above, there is the possibility of using a `Trigger Service` for realizing multi-response patterns. If the exact number of responses is not known during design-time or if the service coordination layer should be kept agnostic to the actual realization of the interaction, this design decision should be made accordingly. The decision influences the design of the DET.

5.9 Business Process Orchestration Layer

The BPIOAI approach of [4] describes that the control flow that executes distributed functionality should be designed with reference to a business process. Since one major benefit of the service-oriented architectural style is the centralization of the control over distributed functionality, the aim of this reference architecture is to allow for a control flow that is described by the means of business processes. This way control is not only centralized but also aligned with business requirements. Only small technical constraints should prohibit the direct deployment of business process descriptions. The place within the reference architecture to deploy these processes is the *Business Process Orchestration Layer*.
Business processes can be described by workflows in an imperative way using a workflow description language. Workflows have several aspects or perspectives that together form the description of a workflow. These perspectives are the control flow, data, resource and operational perspective (cf. [108]). The control flow "describes activities and their execution ordering through different constructors, which permit flow of execution control, e.g. sequence, choice, parallelism and join synchronization" [108, p. 2]. The data perspective describes business and processing data of the workflow as well as pre- and post-conditions for the tasks of the workflow. The resources and the operational perspective describe how workflows are executed in terms of their organizational support and of supporting application systems.
The *Business Process Orchestration Layer* of a composite application should consist of two elements. First, a workflow engine that executes the control flow by orchestrating application services and service coordinations is required. There a business process is deployed as the central control flow in a composite application.
Second, decisions within the control flow should be controlled by a dedicated `Decision Service` that operates on the data of such a workflow. It is necessary to define a dedicated service as the process context is kept outside the actual workflow engine. These two elements of the process orchestration layer are described in the following sections.

5.9.1 Workflow System for Service Orchestration

Purpose and Functionality The workflow system for service orchestration is the part of the *Business Process Orchestration Layer* that provides a runtime-environment to execute workflows that coordinate services following the control flow of an actual business process. The orchestration needs to be described in a workflow description language that is deployed to the workflow system. However, the actual design and validity of such processes must be checked during design-time.
The services being executed are provided by application systems that are possibly mediated by integration flows of the DET and the service coordination layer. The (data) context of the process is kept in the data repository (cf. section 5.6). This is because of architectural considerations that involve consistency and simplification but also the simple necessity to expand a process' context throughout a composite application. The workflow system that is used for service orchestration does therefore solely invoke external service providers. The context handling is performed by `Smart Proxies` and the data repository. These proxies might be incorporated into the workflow engine, though. In order to realize conditional expressions, the workflow engine utilizes `Decision Services` that are connected with the data repository. Such services are used to decide on conditional

expressions in a given context.
Business processes do not only involve application systems, though. Human interaction is often also part of such processes. From a software architecture point of view, human interactions are realized as the interaction with back-end systems (cf. section 5.3). Hence, humans are considered service providers that use a user-interface to receive input and provide output of a certain functionality. As a consequence, there are no constraints imposed to the layer of process orchestration.

Design Decisions and Realization Requirements The *Business Process Orchestration Layer* is realized as a workflow engine that uses both, resources that are exposed as services and data that is kept in a data repository that is also accessible via services. Hence, the functional requirements that are imposed by this layer include the need to allow for communication with the other elements and the need to be capable of executing workflows that are described in a workflow description language. In order to execute workflows, the platform and the workflow description language should provide means for realizing the basic control flow patterns, advanced branching and synchronization patterns as well as structural workflow patterns that are described in [108].

The actual design of the business process determines several design decisions of a composite application. The decisions that are needed for the design of the actual business process are not, however, part of a composite application's design. They are prerequisite. The actual design of a composite application using a business process is described in chapter 6.
The actual decisions that must be made for a composite application regarding the service orchestration are independent of the actual business process that has to be realized. They concern the interaction of the process orchestration layer and the other components of a composite application. Hence, they are necessary for incorporating platform constraints into an actual virtual machine (in MDA terms. see [91]) rather than designing an actual composite application.
Again, the communication semantics must be decided upon. Asynchronous communication is preferable. As business processes are long-running transactions without atomicity requirements, synchronous communication is likely not required. Whenever the orchestration layer can interact directly with application systems while these systems require synchronous interaction, the DET should be used as a mediator. However, whether correlation or end-points are used for the realization of the asynchronous communication needs to be decided. In any case, the communication between the services must be reliable. Thus, the platform for the workflow engine needs to support the *Guaranteed Delivery* pattern (cf. [95, pp. 122ff]).
The next decision concerns the lookup of service endpoints. As discussed for the service coordination layer in section 5.8, static references or dynamic lookups are possible. Usually, static endpoints are preferable. The actual determination of appropriate services should be part of the control flow. The actual management of endpoints should be possible during design-time without changing the workflow description, though.
As the process context is kept in the data repository, the process orchestration does not manage hardly any data. As a consequence, the service orchestration can be used to simply dispatch `Event` objects to the appropriate endpoints. If this mechanism is applied, the respective endpoint needs to apply `Smart Proxies` in order to interact with the data repository. Alternatively, `Smart Proxies` can also be included in the workflow execution platform.

In sum, the platform used to realize a service coordination needs to support the following functionality:

- **Execution of Process Orchestrations**
- **Asynchronous Messaging** Correlation and/or callback endpoints
- **Endpoint Management Facilities**
- **Guaranteed Delivery** (cf. [95, pp. 122ff])
- **Workflow Pattern 1 – Sequence** of [108]
- **Workflow Pattern 2 – Parallel Split** of [108]
- **Workflow Pattern 3 – Synchronization** of [108]
- **Workflow Pattern 4 – Exclusive Choice** of [108]
- **Workflow Pattern 5 – Simple Merge** of [108]
- **Workflow Pattern 6 – Multi-Choice** of [108]
- **Workflow Pattern 7 – Synchronizing Merge** of [108]
- **Workflow Pattern 8 – Multi-Merge** of [108]
- **Workflow Pattern 9 – Discriminator** of [108]
- **Workflow Pattern 10 – Arbitrary Cycles** of [108]
- **Workflow Pattern 11 – Implicit Termination** of [108]

The following list is a summary of the decisions that are necessary in order to incorporate platform-specific constraints into an actual virtual machine.

- **How is the inter-layer interaction realized?** The workflow engine of the service orchestration layer controls the invocation of external services at any layer of a composite application. The communication semantics that is used by these interactions should be simple yet reliable. If asynchronous messaging is used, it needs to be decided whether correlation-identifiers or call-back interfaces should be used.
- **Dynamic or static endpoints?** The services that are invoked by the service orchestration are deployed to distributed agents. Looking up endpoint information of these agents is possible at design-time or during runtime. Design-time lookup with caching is the preferable approach (cf. section 5.10).
- **Integration of `Smart Proxies` into the workflow platform?** In order to connect the workflow engine with the data repository, `Smart Proxies` are required. They could be either part of the platform of the workflow engine or used at the side of the service provider that are invoked by the service orchestration. In the latter case, it is sufficient to pass `Event`-objects between the service orchestration and the (proprietary) service providers.

- **How should `Decision Services` be integrated?** As a `Decision Service` requires access to the context of a process, a connection between the service and the data repository of a composite application needs to be established. A `Decision Service` might either be deployed as part of the workflow platform and integrated via special annotations within a (standard) workflow description language or deployed as an external service that is invoked using the standard service invocation mechanisms.

5.9.2 Decision Service

Purpose and Functionality The layer of process orchestration forms the central component that controls the overall execution of a composite application. The actual decisions within the control flow (exclusive choice, multi-choice) and structural workflow patterns that require such decisions (e.g., exit conditions in loops) are often based on the context of a certain process instance.
Workflow descriptions that are used in such service orchestrations are monolithic blocks that usually involve several business rules. A business rule "is a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control the behavior of the business" [113, p. 30].
Managing or changing the single rules that are embedded in such blocks is difficult and time-consuming (cf. [114]). In order to increase the maintainability of composite applications that apply the presented architecture beyond the possibilities of the service-oriented architectural style, business rules are managed by `Decision Services`. They are used to determine the actual control flow of a service orchestration. According to the classification of [114], a `Decision Service` decides on *reaction rules* for a business process. It uses the data repository to check whether certain conditions apply. Based on the output of the `Decision Service`, the service orchestration may invoke different services.
By using a `Decision Service`, the business logic that underlies such decisions can be described independently from the used business process description language in a separate service. This way, the decision logic becomes reusable.
In order to reduce complexity, handle transactions and allow for a multi-layered architecture, a generic data description kept inside the process environments by the means of the `Data Repository` is part of this reference architecture (cf. section 5.6). Since the constructs proposed are quite complex and independent from a certain process orchestration language, it could occur that an actual language is not capable of using the generic business data. A `Decision Service` also provides an interface from the service orchestration layer to the `Data Repository` to increase the possible platforms with which the reference architecture could be realized.

According to [115], a business rules engine consists of a *rule base*, a *working memory*, a *pattern matcher* and an *inference engine*. "The *working memory* holds the data on which the rule engine operates" [115, p. 36]. In the reference architecture, this data is kept in the data repository. The *rule base*, the *pattern matcher* and the *inference engine* internal components and their specification is out of the scope of the reference architecture for composite applications.
From an architecture point of view, a `Decision Service` needs to be able to decide *reaction rules*. The actual reaction to such a rule is, however, performed by the workflow

engine as the central control instance of a composite application.[35]

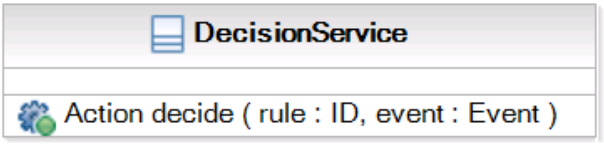


Figure 21: Public Interface of the `DecisionService`

Design Decisions, Realization Requirements and Syntactical Definition A `Decision Service` needs to provide a sub-set of the functionality that is usually provided by business rules engines. It simply needs to evaluate a certain rule in a given context and return the required information to the workflow engine. This way, a `Decision Service` is interoperable with arbitrary workflow engines and the integration efforts are limited. A `Decision Service` needs to implement the interface that is shown in figure 21.

The interface prescribes that a `Decision Service` can be called as part of an orchestration by using a rule's `ID` and the actual `Event`. Based on this information, an `Action`-object is returned. Such an `Action` is an identifier that is used by the orchestration's control flow to branch accordingly. The actual workflow is dependent on the `Decision Services` it uses. This is because the rule IDs and the actions need to be aligned. The interaction between a `Decision Service` and a workflow engine is exemplified in figure 22.

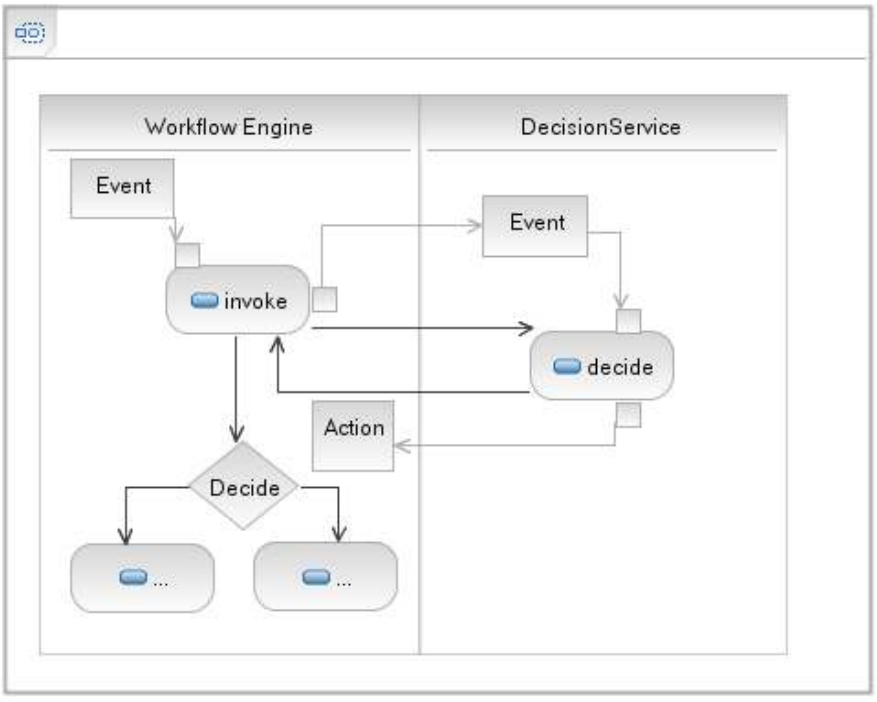


Figure 22: Collaboration between a Workflow Engine and a `Decision Service`

The actual rules are administered using the administration interface of an `Decision Service` that is shown in figure 23.

[35] If the rules engine is integrated with the workflow engine as it is described by [114], this distinction is blurred.

A rule that is added to a `Decision Service` is associated with an `ID`. The rule might be expressed using an arbitrary rule description language (e.g., such as RuleML [116]). Important to note is that variables that are used in a predicate correspond with the accessible data elements that are stored in the data repository for a given `Event` or with the `Event` itself.
In order to retrieve data from the data repository, a `Decision Service` is required to use `Smart Proxies`. This way, the evaluation of rules is included in the consistent state management that is offered by the data repository. The actual description of rules, rule description language and the specification of the other parts of a rule engine that might be required to realize a `Decision Service` are not within the scope of the reference architecture.

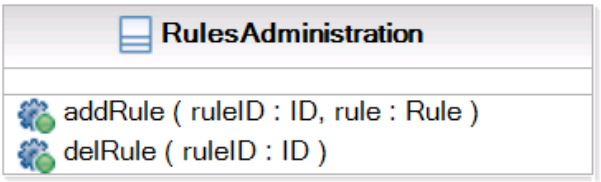


Figure 23: Public Interface of the `RulesAdministration` Service

The following lists considerations that have to be made in order to integrate `Decision Services` into an actual platform.

- **How should `Decision Services` be integrated?** As described in section 5.9.1, a necessary design consideration is whether `Decision Services` are directly embedded into the workflow engine and annotated in the workflow description language or whether they should be treated as external services (as in the example of figure 22). The latter is the simplest approach and requires no integration efforts with the workflow engine. However, in order to realize more complex business rule scenarios, an interceptor-based approach could be realized as it is described in [114].
- **How flexible should the rule engine be realized?** If necessary, an interpreter for rule descriptions should be created. As described above, usable variables within such descriptions need to be the data elements stored in the data repository for a given `Event`. However, if relatively simple decisions need to be realized (e.g., lookup of boolean attributes) the overhead could be reduced by hard-coding such rules in special types of `Decision Services`.

5.10 Service Registry

Purpose and Functionality Services are described by their interface and deployed to an agent (cf. section 2.3). In order to allow a service consumer to invoke a service provider, the physical location of the required service provider's agent is required. This location is referred to as an *endpoint*. Service-oriented landscapes sooner or later consist of many different service providers and agents. Each service provider could also be deployed to several agents. In order to keep such landscapes maintainable, it is necessary to keep track of all available service providers and the agents to which they are deployed. This

is usually achieved by establishing central repositories. Such repositories are also called service registries (cf. [95, pp. 138f.]). "These repositories allow humans (and even service requestors [(consumers)]) to:

- locate the latest versions of known service descriptions
- discover new [...] services that meet certain criteria" [95, p. 138].

This definition describes two possible scenarios in which registries can be used. As described in section 2.3, service registries can be used at runtime to identify an agent for a given service provider by using lookup operations. Additionally, service registries can be used at design time by "humans" in order to retrieve the latest version of a service's definition. This feature is especially important if new services are designed. This is because the service designer can use existent services to decide whether the existent service is applicable for an actual problem or whether an existent service could be redesigned in order to meet the old and new criteria. The design methodology that is presented in section 6 includes an approach to design services with the support of a registry at design-time.
At runtime, the endpoint information of all required service providers needs to be available to all service consumers. As the presented reference architecture consists of several layers, several instances maintain information about their respective service providers. This distributed knowledge decreases the maintainability of the overall composite application. In order to address this issue, the endpoint information for all service providers needs to be centralized (at least) for a composite application.
Dynamic lookups of endpoint information are, however, an additional overhead at runtime. As the service-oriented architectural style centralizes the control of an organization's application systems, the endpoints are considered to be stable and known at design time. This is why the single components of the reference architecture should cache the references and solely offer an interface for managing the endpoint information. This way the endpoint information can be updated based on a registry at design-time, if necessary. In turn, it is not necessary for a service registry that is used as part of a composite application to offer dynamic lookup capabilities. If dynamic lookups are necessary, the `Routing Service` (cf. section 5.7.5) should be used together with integration flows as the connection point to a service registry. This way, not only service providers exposed using the common protocol of a composite application can be dynamically looked up, but so can arbitrary application systems that expose their functionality in any way. Of course, at this point, if a `Routing Service` requires dynamic lookups it makes sense to make use of the lookup-operations of a service registry.

The definition of a registry in [95] includes the notion that services that meet certain criteria can be discovered (cf. [95, p. 138]). Such criteria are usually the syntactical definitions of the required service providers. Quality-of-service as well as semantical information also might be useful for identifying services ("rich specifications", cf. [14]). Application systems are considered to operate on static platforms. The experience during the case study that is presented in chapter 8 demonstrates that semantic descriptions of services are hard to obtain by members of large organizations. In order to allow the building of composite applications in arbitrary environments, the presented reference architecture neither requires a registry to be capable of attributing service with quality-of-service attributes nor includes (structured) semantic information. Leveraging these advanced concepts of service orientation is considered a task after this architectural style has been applied more often and has become natural part of organizations' application landscapes.

Design Decisions and Realization Requirements A service registry required for composite applications that follow the presented reference architecture is a simple catalogue of structural service interfaces and their respective endpoint information. As a matter of course, the information needs to be managed and queried at the design-time of a composite application. Additionally, the following decisions should be made:

- **Are dynamic lookups required?** When creating or choosing a service registry to use within a composite application, it must be decided whether dynamic lookups are required (by the `Routing Service` or by the service coordination layer). If so, the registry needs to be realized/chosen accordingly. If not, the requirements are minimal. A design-time registry could even be realized using a central document. Solely updating endpoint information at the different components of the composite applications should be automated.
- **How should endpoint information be updated?** The single components of a composite application maintain endpoint information. If such information is changed in the service registry, the single components need to be updated. Whether this is done manually or is automated needs to be decided. As the management interfaces of the single components (especially of the `Eventing System`) are likely to be proprietary, updating endpoint information might prohibit the out-of-the box usage of a standard service registry product.

5.11 Summary

In this chapter a reference architecture was proposed that supports the creation of composite applications. The reference architecture is independent from any platforms and is described as a set of requirements for any actual platform. Additionally, core elements are described syntactically and functional specifications are provided.
As a whole, the reference architecture allows for the application of service-oriented principles like control centralization, service aggregation, abstraction and loose coupling while considering trade-offs between optimal design and applicability. Special attention is paid to the necessary means for interacting with heterogeneous application systems. In order to realize a composite application, a design methodology should be applied that uses the reference architecture to design a composite application in a platform-independent manner while incorporating constraints of an application landscape into the design.
By mapping the reference architecture to an actual platform, composite applications can be realized on arbitrary platforms that support the requirements of the reference architecture in a standardized way.

6 Designing Composite Applications

The reference architecture described in chapter 5 standardizes the structure of composite applications so that service-oriented principles can be realized more easily. In order to design a composite application for a given use case, a methodology is required that uses the requirements as its input and derives a suitable design based on this input. An objective of such a methodology is to support the designer(s) of a system to align the actual design with *soft* service-oriented design principles. Those are autonomy, reusability, loose coupling and abstraction (cf. chapter 2).

When defining a design methodology that facilitates the application of the service-oriented architectural style in the context of application integration in large organizations, it is important to restrict the demands towards the requirements engineering. Of course, modeling functional requirements is a must. Actual projects suffer anyway often due to poor requirement engineering. This frequently impacts the go-live of applications that are built. This is because such applications often do not meet the expectations the ordering party had in mind during the requirements engineering phase. And if that is the case, the ordering party often simply refuses to pay for the development of an application.
This, of course, is not a new finding. For this reason, several approaches were defined that aim at holistic requirements engineering. Notable examples are the "Architecture of Integrated Information Systems" (ARIS) from [117] or the "Semantic Object Model" (SOM) from [93]. Both approaches aim at describing a system using different views of a business process (including data, organizations and an organization's resources) and subsequently drilling down on this description.
In practice, it can be observed that these approaches are applied but not exercised completely. This often means that business processes that are derived by such approaches are seen as basis for discussions and are not further drilled-down. Because of this, it seems that process-driven design becomes less important.

The design methodology that is presented in this chapter aims at leveraging the idea of business process-driven integration within the domain of service-oriented architectures. In doing so it incorporates the "wish list" towards a methodology for service construction that was recently developed in [15]. It relies, in terms of ARIS, on the deliverables of a "functional specification" ("Fachkonzept") (cf. [117]) that basically consists of a process model. It uses the description as its input in order to design composite applications. This way, it incorporates the directives of "process dominance", "process scope" and "interface reference" as they are demanded for a design methodology by [15].
Based on the design that is derived from the requirements and a platform-specific mapping of the reference architecture (cf. chapter 7), the realization of the described requirements can be performed.
The point of view towards process modeling and requirements engineering of this approach is rather naive. Neither an actual modeling approach is described or prescribed nor is a specific modeling meta-model required. Discussions about this topic are considered beyond the scope of this thesis. This is because the design methodology aims to apply service-oriented principles rather than describing yet another modeling approach.
Using "Unified Modeling Language" (UML) (cf. [39]) models for the description of the design and the data perspective of the requirements, as well as "Event-Driven Process Chains," (EPC) (cf. [117]) should be considered one example of languages that can be used. These languages were chosen because they were most relevant in the context the

described concepts were developed in. It simply appeared not possible to have domain experts expressing their requirements using different meta-models or developers utilizing different kinds of models. As no automated transformations of models is used, issues with the formal semantics of the process description language EPC (cf. [118]) were not considered as an obstacle.

Based on the above described requirements, a step-by-step approach is defined by the methodology. In these steps, the deliverables of earlier steps are used as input in order to derive new deliverables. The set of all deliverables constitutes the actual design of the composite application.
The design methodology involves the creation of services and their respective aggregators in addition to items for addressing the heterogeneity of application landscapes. This is why a service-meta model is required. After the introduction of this meta-model, the design methodology is discussed.

6.1 A Meta-Model for Services

The design-time reference architecture for composite applications that is described in chapter 5 describes three levels of abstraction. The most abstract description is the business process-based orchestration of services. The less abstract descriptions are the services that are exposed by application systems (in a mediated or un-mediated way). In order to allow for business process-based service orchestration ("process dominance" and "process scope" from [15]), a middle layer is introduced by the notion of the service coordination layer. At this layer so-called coordination services are aggregated to so-called service coordinations.
According to these layers, a meta-model for services is introduced (cf. figure 24).

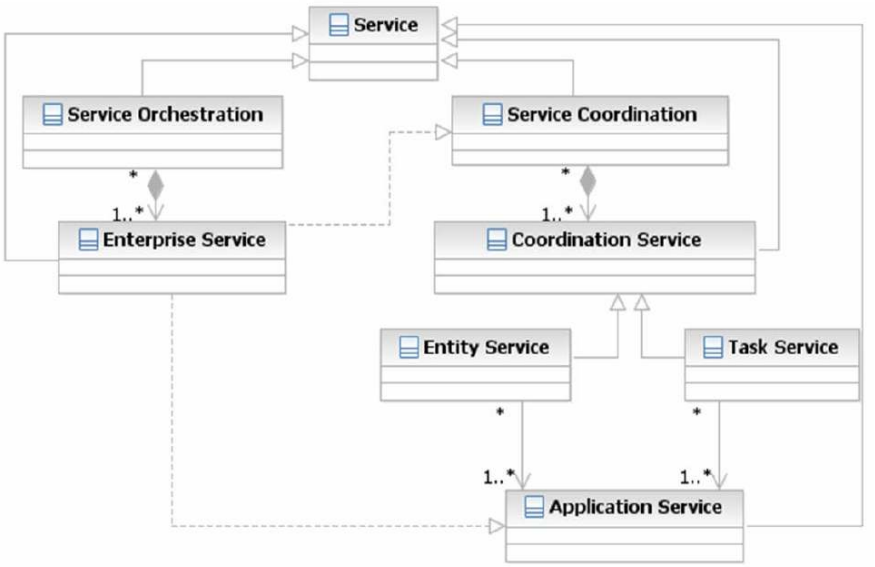


Figure 24: Service Meta-Model

The meta-model describes that a service orchestration is composed of *enterprise services*. Enterprise services are realized as *service coordinations* or as *application services*.
A service coordination is composed of *coordination services*. Two types of coordination

services exist. *Entity services* and *task services*. This categorization stems from the idea of separating services by the type of operation they perform (cf. [11, pp. 390-394] or [90]). Services that deal with the management of business entities are "entity-specific" services whereas services that are designed for the realization of a cross-entity task are considered as "task-centric". A formal categorization of these two types of coordination services is discussed as part of the service design algorithm in section 6.2.4.
Coordination services are abstractions of functionality that are exposed by application systems through application services. If no application service exists that matches an enterprise service, a coordination service can be realized by multiple application services. The standard is that a coordination service is mapped on a one-to-one basis to an application system and that these services are composed by service coordinations. This mapping from coordination services to actual, constrained application systems marks the shift from a top-down to a bottom-up approach of the design methodology that is subsequently described.

6.2 Composite Application Design – A Step-by-Step Process

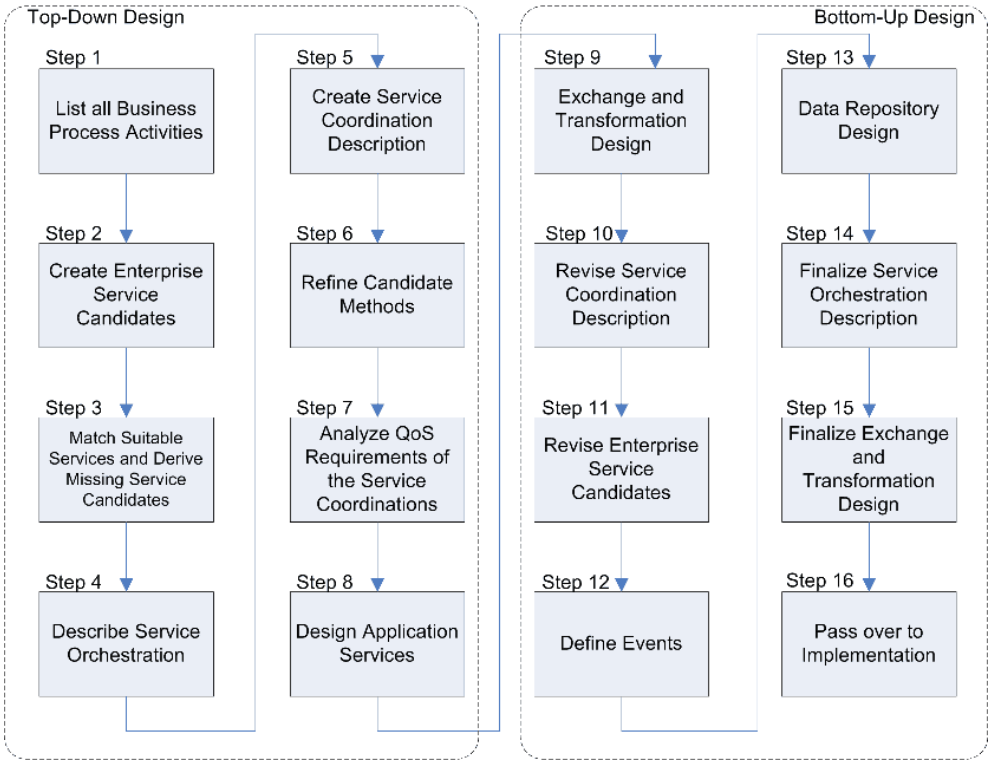


Figure 25: Steps of the Design Methodology

The design approach that is presented in this section comprises 16 steps. If performed correctly, these steps describe how the system design for an actual use case can be derived. The methodology combines a top-down approach with a bottom-up approach. The top-down approach takes the requirement descriptions as its input. Emphasis is placed on deriving services. A service-design approach is included that incorporates the findings of chapter 4 and the service-meta model. Using minimal requirements, the included algorithms can be used to split business process functions into more fine-granular services (principle of "intersection points" from [15]).

After refining the derived service candidates, the link between the design and the actual constraints of an application landscape is performed. From this point forward, a bottom-up approach incorporates the identified constraints into the design of the single components of the reference architecture for composite applications. By doing so, the initial description of the business process is changed in a way that it can be used as the central service orchestration.
The single steps are described as a sequence. However, iterations of certain steps or the reworking of complete branches is possible.

After the introduction of a short example that accompanies the description of the methodology, the actual steps are described in the subsequent sections.

6.2.1 An Example Scenario

In order to illustrate the methodology, a small example case is used. The example accompanies the description of the methodology and demonstrates how the process can be decomposed into several types of services.
Figure 26 shows the model of a short example process. It describes that a purchase order is created and a shipment is scheduled. The processing of the order additionally produces a trade credit insurance. In the example it is assumed that, by default, every order, as soon as it is scheduled to be shipped, also needs to be insured against the buyer failing to pay the invoice.[36]

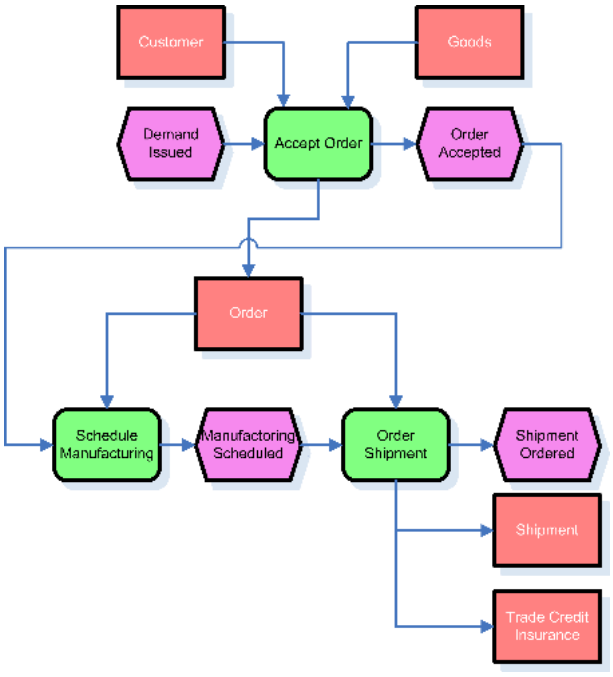


Figure 26: Control-Flow and Data-Flow of the Example Process

The data view of the process is depicted in figure 27. The key attributes of the single entities are marked using the stereotype `key`. The data model references the process model as it describes the dependencies among the single business objects more in-depth.

[36]This should not be confused with an insurance that covers good damages that occur during the shipment.

The data model is designed for the sake of eased understanding. It is not meant to reflect real-life business data. It demonstrates that one customer can have multiple orders for an arbitrary number of goods with the example company (an order could also be about services). Shipment of an order can be scheduled by creating a shipment order with a carrier of choice. Additionally, an order can be insured against a possible failure of payment. Such insurance is represented as a link between an `Order` and a `TradeCreditInsurance`.

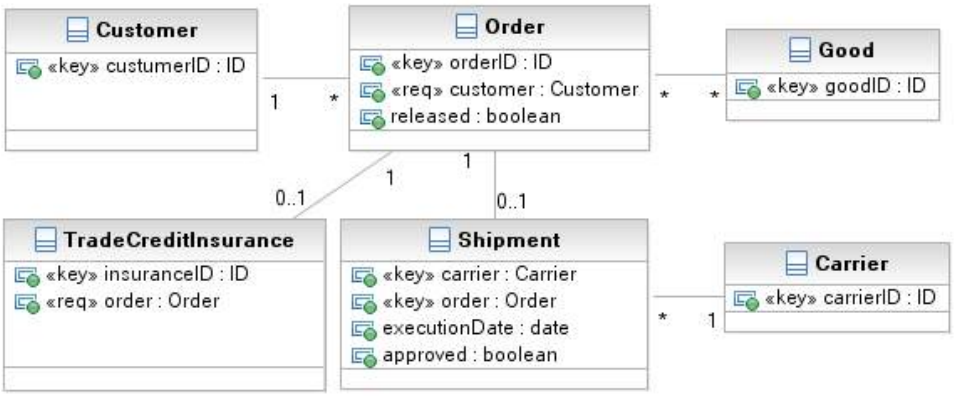


Figure 27: Data Model of the Example Process

6.2.2 Step 1: List all Business Process Activities

The input of the methodology is a complete description of the business process that needs to be realized using a composite application. Based on the described process, the single services that are required by the reference architecture can be derived. This procedure of deriving the services is based on the understanding of a business process as a possible service orchestration and the business tasks as orchestrated services.
Thanks to the reference architecture in chapter 5, the enterprise services can be considered providers of the functionality required by business tasks. The single business tasks are described in the business process description. In contrast to Erl's methodology that begins by refining the actual business process (cf. [11, pp. 397-430]), this approach takes the activities of a business process as an input for later phases. Each function of a business process' control flow is considered to be supported by an enterprise service. This is why it is necessary to extract the description of the respective process' functions to identify these enterprise services.

The functions of a process must not be ambiguous if such a mapping is to be performed. Several approaches exist that support mapping from processes to services that fit into the process. Bentallah et al., for instance, describe in [119] an approach that matches services based on a semantic description. It utilizes a descriptive logic in order to reason about a request for a service lookup. The results of this reasoning are referred to as *best covers*. Best covers are services whose description contain as much information about the request as possible.
A similar approach is presented by Paolucci et al. in [120]. Their approach matches sufficiently similar services for a given requirement. "The degree of match is determined by the minimal distance between concepts in the taxonomy tree" [120, p. 91]. The compared concepts are derived from the input and output of methods as well as from preconditions and effects.

While conducting the case study that is described in chapter 8, it turned out that a semantic approach that relies on formal descriptions in a formal logic or even using an ontology is today not suitable in the environment of an industrial enterprise. This is because process experts that must identify the functions of a business process lack both, the know-how and the time/motivation to create such descriptions. Having descriptions at all is everything but self-evident. Due to this considerable constraint, the presented approach must solely rely on a unique naming for process functions and data. Functions that have the same name, are considered to be semantically equivalent. Thus, the semantics of the respective functions must also be described in an informal way.

The output of this step of the methodology is a list of business functions that includes the name of the function, the input and output data of the function and a rough, informal description of the task. If applicable, the back-end application system that is currently used for such tasks (e.g., order processing) can also be indicated.

For the small example process, this list would look similar to the following list.

- *Accept Order* **Input**: the `Goods` that are demanded by a customer. The `Customer` that is demanding the goods.
  Output: the `Order` that is created by the supplier.
  Functional Description A customer issues a demand for several goods electronically. In order to process the demand, an appropriate `Order` object has to be created.

- *Schedule Manufacturing.* **Input**: the `Order` that should be manufactured.
  Output: *none*
  Functional Description A customer's order has to be manufactured prior to shipping it. The manufacturing process has to be scheduled.

- *Order Shipment.* **Input**: the `Order` for that a shipment should be scheduled and that needs to be insured.
  Output: the `Shipment` that represents the scheduled shipment as well as a `TradeCreditInsurance` that represents the contracted insurance for the respective order.
  Functional Description Based on an `Order` the shipment of the respective order has to be arranged. In order to reduce risks, the order has to be insured with trade credit insurance.

Deliverables Process functions; data flow for each function; informal description of each identified function.

6.2.3 Step 2: Create Enterprise Service Candidates

In this second step, the list of business functions is used to describe enterprise service candidates. If a process was not designed using existent enterprise services and if no services that fit a process description can be found within the actual service registry (i.e., no "perfect matches"), they need to be created.
Usually new enterprise services are built out of existing functionality as well as out of new service methods that are designed using the following steps (especially step three). The composition that realizes an enterprise service by using new and existent (coordination)

services is called service coordination. The coordination services it consists of can only be composed in a way that supports the enterprise services while not being enterprise services themselves. The actual design of the enterprise services is therefore a prerequisite for the design of coordination service methods and the respective service coordination.
A business process can be seen as an event-driven computation of data (cf. [92]). The events do not only determine the task that should follow an event, but also the data that is transmitted during this event. This data is both context data of the process as well as output from preceding enterprise services. Both the events and the data are described in the model of the business process. The data is described in greater detail in the according data perspective of a business process.
As the reference architecture and this methodology were created to directly allow the use of business processes as service orchestration, the single tasks of the business process are considered to be enterprise services. Deriving these enterprise services is simple. Each business function identified in step 1 is considered an enterprise service. The input for this method is the data that is consumed by the business task. The result of a business task is the output of the enterprise service method.[37]

In this methodology, a service method is described as a tuple of data. This is according to Eshuis et al.: "for each service s, input(s) denotes its input message and output(s) its output message. One of these messages is required, otherwise the service does not need to be composed with the other services" [121, p. 102]. Implicitly contained in this description from [121] is an understanding of the methods of the actual service. For the sake of simplification, it is assumed that a service as it is defined by [121] consists of exactly one method. Hence, the enterprise service candidates can easily be derived by considering the list of business tasks that were created in step 1.
The next action is to query a (design-time) service registry with all existent services of an enterprise and analyze whether it contains any suitable services. Lacking a semantic approach, this is performed by considering the name of the process task and the input and output parameters.

In the shipment example, the enterprise service candidates that can be identified are listed below.

- es_1 = `({Customer, Good[]}), {Order}` (*Accept Order*)
- es_2 = `({Order}, ∅)` (*Schedule Manufacturing*)
- es_3 = `({Order}, {Shipment, TradeCreditInsurance})` (*Order Shipment*)

It is assumed that all but the service `({Order}, {Shipment, TradeCreditInsurance})` (*Order Shipment*) are registered with the service registry. This is why (especially for the sake of simplicity), only this enterprise service is used in the following for the illustration of the presented principles.

Deliverables Enterprise service method candidates.

[37] As methods are considered to be ordered sets, a method can have multiple return parameters. How this is actually realized is not the concern at this point.

6.2.4 Step 3: Match Suitable Service Methods and Derive Missing Service Method Candidates

The service design methodology from [11] asserts that services which are used in a process orchestration should fit into the orchestration. In the given context, "fitting" refers to the fact that the service computes a set of input parameters and then provides the described functionality and the required output parameters (cf. [11, pp. 205ff.]). Services that fit into an orchestration can be identified by using the deliverables of step 1 and 2 (process functions; data flow for each function; informal description of each identified function). If, however, no matching services can be identified, it is an informal task to derive the missing services for the identified process functions (cf. [11]). However, this is a complex task. This is largely because a hierarchy of types of services must also be considered. In order to better support deriving missing services, this third step of the methodology is elaborated more in detail.
An approach to the design of services is presented that incorporates the findings that are described in chapter 4. It aims to match existent service methods with an actual business process and to define new candidate methods that might be used to close the gap between an actual process and existing services while being reusable. In this sense, it realizes a *Scatter* operator for an interface adoption as it produces multiple steps (service invocations) that are necessary for realizing a single enterprise service operation in a given landscape (cf. [122, p. 72]). Additionally, the step might conclude that the redesign of existing service methods might be appropriate.

According to [121], there exists a dependency E between two services s and ś with $s \in \Omega.\Psi$ and $ś \in \Omega.\Psi$ that describes that a service consumer and a service provider exchange common data elements.[38] Applied to service methods, E can be defined in the following way:

$$E = \{(s, ś) \in \Omega.\Psi \times \Omega.\Psi \mid s_{output} \cap ś_{input} \neq \emptyset\} \tag{39}$$

This definition implies that the design of services that are consumed by and provided to a service orchestration is the procedure of describing services providers in a way that there exists a suiting service method for every call of an orchestration. A service orchestration is both a service consumer and a service provider – a service aggregator (the set of service consumers is denoted as $\Omega.\mathrm{C} \subseteq \Omega.\Psi$ and the set of service providers as $\Omega.\mathrm{P} \subseteq \Omega.\Psi$). Applying this understanding, service design for service orchestrations is an activity that creates, for a given set of service consumers, a set of service providers (and the corresponding methods of the service) in a way that the two dependencies $\bar{E}$ and $\tilde{E}$ can be fulfilled.

$$\bar{E} = \{(o_c, p) \in \Omega.C \times \Omega.P \mid c_{output} \setminus p_{input} = \emptyset\} \tag{40}$$

$$\tilde{E} = \{(o_p, c) \in \Omega.P \times \Omega.C \mid p_{output} \setminus c_{input} = \emptyset\} \tag{41}$$

These two dependencies could be used as a rule to derive the service method interfaces that fit into a given orchestration. If only these constraints were used, the granularity of single service providers' methods would be solely dependent on the actual service orchestration. As transformed business processes should be used as service orchestration, the services' granularity would be implied by the process design. However, it was identified that a certain level of granularity leverages reuse, though (cf. chapter 4). Additionally, services should be "offered at different levels with different granularity. High-level business process functionality is externalized as coarse-grained services [...]" [79, p. 156] that aggregate

[38] $\Omega.\Psi$ is the set of all services for a given system (cf. section 3.2.1).

more fine-granular services. This is why only satisfying $\bar{E}$ and $\tilde{E}$ is not sufficient to design appropriate services.

According to Reijers, the set of processed information D and the set of operations (methods) processing the information O can be split into more fine granular entities (cf. [86]). He proposes a heuristic of decomposing O into a set of tasks t; $T \in \Pi(O)$ in a way that $\forall o \in O\colon (\exists t \in T\colon o \in t)$ (cf. [86, p. 118]). This heuristic uses a definition of cohesion to determine if a given splitting of an operation into two tasks is preferable or not. The criterion for that decision is whether the two new tasks are more cohesive than the initial method.
The approach that is presented here builds upon this idea of decomposing activities into more fine granular tasks by splitting a method by the information it processes. However, the indicator for determining whether a split is preferable is the only indicator that proofed to be significant for service re-use – service granularity (cf. chapter 4). The appropriate level of granularity should be met at the level of coordination service methods. The identified coordination service method candidates should have a similar granularity of a certain level. Lacking other thresholds, the threshold of a SSM-value close to 13 (that was identified in section 4.4) should be used and refined during subsequent projects. An appropriate level of granularity will be achieved by decomposing service methods and aggregating them afterwards if applicable. In order to check whether joining disjoint methods is applicable, the idea of service categories is incorporated.
As previously stated, according to Erl, services can be categorized in task services and entity services (cf. [11, pp. 390-394]). In the presented methodology this classification is applied to the coordination services (and their method(s)). They are classified into data-centric methods (for entity services) and task-centric methods (for task services). This classification is a simple mechanism to first design the methods (as task-focused or entity-focused) and secondly aggregate these methods to multi-method services, if required. This approach is a simple yet valuable approach because it is easy to understand and can be easily performed – also without tool support.

While entity-specific services have a higher potential for reuse[39] and are supposed to increase modifiability (cf. [11, p. 393]), task-specific services handle process-specific tasks and are explicitly designed for a given process. An example of task services are services that check the validity of entities (cf. [11, p. 392]).
In order to apply this concept to the given design methodology, a formal yet simple categorization of service methods by their respective parameters is proposed. This is required in order to support a service designer and free the design from unnecessary subjective decisions. Additionally, the benefit of such a categorization lies in eased understanding and governance of the resulting services.
In order to categorize service methods, the computed data needs to be categorized first. The set of data that is being computed by a system's services Ω, $\Omega.D$ consists of two subsets. Context data (Γ) and business data (B). So that $\Omega.D = \Gamma \cup B$. Context data is considered to be supplementary data of a process (e.g., validities, keys, counts etc.) while business data contains objects that represent business entities (both master data as well as transactional data).
Whenever a service method accepts business data as its input and produces either context data (that does not reference the business data) or a subset of its input, the method is

[39] Due to the ambiguous definition of the case study's services, the examination in chapter 6 did not include an assessment of a potential significance of service categories on reuse.

considered to be task-specific. The equations (42) and (43) describe the constraint for task-specific service methods.

$$ts_1 = \{(ie, oe) \in B \oplus \Gamma | \ \forall o \in ie : key_o \notin oe \wedge (ie \neq \emptyset \wedge oe \neq \emptyset)\} \quad (42)$$

$$ts_2 = \{(ie, oe) \in (\mathrm{B} \oplus \Gamma) \times (\mathrm{B} \oplus \Gamma) \mid (oe \subseteq ie \vee ie \subseteq oe) \wedge (ie \neq \emptyset \wedge oe \neq \emptyset)\} \quad (43)$$

Consequently, all service methods that do not satisfy these constraints are considered to be entity-specific. These constraints are described by the equations (44) – (46)

$$\epsilon_1 = \{(ie, oe) \in (\mathrm{B} \oplus \Gamma) \times (\mathrm{B} \oplus \Gamma) \mid oe \nsubseteq ie \wedge ie \nsubseteq oe \wedge ie \nsubseteq Req_{oe} \wedge \nexists o \in ie : key_o \notin oe\} \quad (44)$$

$$\epsilon_2 = \{(ie, oe) \in (\mathrm{B} \oplus \Gamma) \times (\mathrm{B} \oplus \Gamma) \mid (\forall o \in ie : key_o \in oe \vee ie \subseteq Req_{oe}) \wedge (oe \nsubseteq ie \wedge ie \nsubseteq oe)\} \quad (45)$$

$$\epsilon_3 = \{(ie, oe) \in (\mathrm{B} \oplus \Gamma) \times (\mathrm{B} \oplus \Gamma) \mid ie = \emptyset \vee oe = \emptyset\} \quad (46)$$

The approach taken here is similar to the semantic service matching approach by Paolucci et al. in [120]. However, due to the reasons outlined in the introduction of this chapter, relying on an ontology is not applicable today. This is why, for the sake of designing service methods, the only suitable information for service matchmaking are the parameters of the methods as well as information about dependencies of the computed data elements.
The data elements of service methods are described as business objects and context data ("interface reference" from [15]). Business objects are sets of attributes and the grouping of attributes to objects ($o = \{a_1, a_2, ...\}$) is subject to the modeling of a business process' data perspective. A subset of the object that determines all other attributes is called key $k \subseteq o$. Hence, there exists a functional dependency $k \rightarrow o$. The key-attribute of an object o it is referred to as key_o. Additionally, keys might be used not as part of an object. This is the case whenever keys are stored as context data.
Keys might also transitively indicate a dependency to associated objects. Thus, despite (non-derived) key values, other objects might be required for the creation of an object o. These objects form the set of "required" objects: $\{o_1, o_2, ...\} \rightarrow o$. The set $\{o_1, o_2, ...\}$ is referred to as Req_o.

Assuming that an enterprise service $E \in \Omega.\Psi$ computes some input data ($\mathrm{E_{input}} \subseteq \Omega.\mathrm{D}$) in order to provide some output ($\mathrm{E_{output}} \subseteq \Omega.\mathrm{D}$), the set M (with $\mathrm{M} = \mathrm{E_{output}} \setminus \mathrm{E_{input}}; M \subseteq \Omega.D$) needs to be provided by additional services that are part of the enterprise service's service coordination. Assuming that there exists a set of service methods that provide the information $\Psi_{\mathrm{exist}} \subseteq \Omega.\mathrm{D}$, the coordination service methods that need to be designed are required to provide the output $\mathrm{C_{output}}$ (with $\{\Psi_{\mathrm{exist}} \cup \mathrm{C_{output}}\} \setminus \mathrm{M} = \emptyset$ and $\mathrm{C_{output}} \subseteq \Omega.\mathrm{D}$). This is why the output of an enterprise service can consist of its input, the output of existing service methods and the output of newly designed coordination service methods:

$$\mathrm{E}_{output} \subseteq \mathrm{E}_{input} \cup \Psi_{exist} \cup C_{output}$$

Using the categorization of service methods as well as the process-specific dependencies among the data entities and the existent service methods, the candidates for new coordination service methods can be derived using a simple algorithm. These service method candidates can be later used to design services that bridge the gap between existent service methods and enterprise services. Such a candidate method (m) is designed to support creating an actual enterprise service's output. It needs to satisfy (47).

$$m = \{(i, o) \mid i \in \mathrm{E}_{output} \setminus o; o \in C_{output}\} \quad (47)$$

Based on the given definitions, the algorithm that is described (for the sake of eased understanding using multiple listings) using the algorithms 1 – 5, can be used to derive candidates for missing coordination service methods. As soon as the algorithm is completed,

the dependencies $\tilde{E}$ and $\bar{E}$ that are implied by the business process are fulfilled using an aggregation of both the newly designed service methods as well as existent methods that are identified in a service registry. Additionally, candidate methods that are categorized into task-specific and entity-specific methods most likely possess the right level of granularity and satisfy the constraint (47). The algorithm describes a semi-automated procedure that requires user decisions.

For the example of the shipment process, the functional dependencies need to be derived from the data model first: the key attribute of a customer is its id: $key_{Customer} = \{customerID\}$. It does not require any other objects: $Req_{Customer} = \emptyset$. An `Order` is determined by its respective identifier: $key_{Order} = \{orderID\}$. However, an `Order` requires a `Customer`-object, too. Therefore, the set of required objects is not empty: $Req_{Order} = \{Customer\}$. A `Shipment` is unique in the context of its respective `Order` and the `Carrier` that is scheduled for executing the shipment. This is why `Shipment` requires both, an `Order` and a `Carrier`: $Req_{Shipment} = \{Order, Carrier\}$. These objects also form the key of `Shipment`-objects, $key_{Order} = \{Carrier.carrierID, Order.orderID\}$. A `Carrier` is identified by its identifier ($key_{Carrier} = \{carrierID\}$) and is independent of the existence of other objects: $Req_{Carrier} = \emptyset$. The two sets for `Good` are $key_{Good} = \{goodID\}$ and $Req_{Good} = \emptyset$. A `TradeCreditInsurance` is dependent from an `Order`. Hence $key_{TradeCreditInsurance} = \{insuranceID\}$ and $Req_{TradeCreditInsurance} = \{Order\}$. These keys and sets of required objects can simply (and possible automated) be deducted from a given data model.
The example that demonstrates the application of the algorithm consists in deriving coordination service candidates for the `Order Shipment` enterprise service (es_3 = `({Order}, {Shipment, TradeCreditInsurance})`). The according enterprise service signature is the starting point for the algorithm: `({Shipment}, {Order, TradeCreditInsurance})`.

The determination of candidate methods initially depends on the actual enterprise service. When describing the steps of algorithm 1, the variable `E` initially has the value `({Order}, {Shipment, TradeCreditInsurance})`. After the initialization of the set `Coordination` that is used to store all service methods that are required to realize the given enterprise service, the required entity-specific (candidate) methods are derived. The algorithm for this (recursive) functionality is described in algorithm 2.
Algorithm 2 is invoked with the input data of the enterprise service candidate method as *available data* and the output of this method candidate as *required data*. In the goods shipping example, the set `Available` would include `{Order}` and the set `Required` would contain `{Shipment, TradeCreditInsurance}`. Within the loop that computes every single required object (type), it is first checked, whether the keys of the actual object is included in the context data. If an object would be directly available, no need for an entity-specific service method would exist. If the key would be accessible, an entity-specific method that loads/creates an object by its key would be required.
If the respective required object (type) is available, it will be used. If there is no directly accessible object, the available context data is checked to see whether it contains a required key (either as an input of the enterprise service or as an output of another coordination service).
Whenever such a key is accessible, an according entity-specific candidate method is added. If such a required object is neither included in the set of available objects nor accessible via its key, it needs to be retrieved or created. In order to determine the required method candidate(s), the algorithm is recursively invoked in order to derive the candidate meth-

Algorithm 1 Derive new Coordination Service Method Candidates for Enterprise Service

E = (E_{input}, E_{output}) {E is the actual (single method) enterprise service}
$Coordination = \emptyset$ {All method candidates for the respective service coordination}
$EntityCandidates$ = getEntityCandidates(E_{input}, E_{output})
$TaskCandidates$ = getTaskCandidates($E_{input}, E_{output}, EntityCandidates$)
$UnusedInput = E_{input} \setminus EntityCandidates_{input} \setminus TaskCandidates_{input} \setminus E_{output}$
$EntityCandidates = EntityCandidates \cup$ getStoreCandidates($UnusedInput$)
$UnprovidedInput = EntityCandidates_{input} \cup TaskCandidates_{input} \setminus E_{input} \setminus (EntityCandidates_{output} \cup TaskCandidates_{output})$
$EntityCandidates = EntityCandidates \cup$ getEntityCandidates($E_{input} \cup EntityCandidates_{output}, UnprovidedInput$)
$RequiredMethods = EntityCandidates \cup TaskCandidates$
for all $m \in RequiredMethods$ **do**
  $PossibleReuse = \emptyset$
  for all $r \in Repository$ **do**
    if $r_{input} \subseteq m_{input}$ **AND** $r_{output} \subseteq m_{output}$ **then**
      $PossibleReuse = PossibleReuse \cup r$ {This might include perfect matches}
    end if
  end for
  for all $p \in possibleReuse$ **do**
    suitable = ask(is p suitable for m)
    if suitable **then**
      $Coordination = Coordination \cup p$
      $RequiredMethods = RequiredMethods \setminus m$
      $RequiredMethods = RequiredMethods \cup \{(m_{input} \setminus p_{input}, m_{output} \setminus p_{output})\}$
      {A method $(\emptyset, \emptyset)$ is considered to not exist}
    end if
  end for
end for
for all $m \in RequiredMethods$ **do**
  $AllOrdersOfMethods = \mathcal{P}(Repository \cup RequiredMethods)$
  for all $Order \in AllOrdersOfMethods$ **do**
    $enlargementPossibility$ = ensureGranularity(m, $\emptyset$, $Order$)
    if User agrees on enlargementPossibility **then**
      newM = m
      for all $enlarge \in enlargementPossibility$ **do**
        newM = $(newM_{input} \cup enlarge_{input}, newM_{output} \cup enlarge_{output})$
      end for
      While ensuring: Include possible combinations only once
      $methodCandidates = methodCandidates \cup \{newM\}$
      $RequiredMethods = RequiredMethods \setminus m$
      BREAK
    end if
  end for
  $methodCandidates = methodCandidates \cup RequiredMethods$
  $Coordination = Coordination \cup methodCandidates$
end for
RETURN $methodCandidates$, $Coordination$
{All candidate methods and the set of all methods that could form the service coordination are derived}

ods that retrieve/create the required object. The recursion either stops as soon as the required objects are available or an object (type) does not require any other objects[40]. As soon as either the independent objects can be made available or there are no independent objects left for an iteration, entity-specific method candidates are created. Those methods could be without any input parameters (for the creation of new objects). Alternatively, methods that accept all required objects as input data and create/retrieve the actual dependent object are created. The realization of both methods is, of course, not included. The algorithm solely determines the methods' interfaces.

Algorithm 2 getEntityCandidates(α, β) – Determine service methods that are required to produce objects in β

```
Available ⇐ α {Parameter 1, set of available objects and keys}
Required ⇐ β {Parameter 2, set of required objects}
candidates = ∅
for all o ∈ Required do
  if NOT o ∈ Available.B then
    if key_o ∈ Available.Γ then
      candidates = candidates ∪ m(key_o, o)
      {Method needed that loads object by its key. ε1-method for entity service}
    else
      RE = Req_o {RE contains all object types that determine o}
      for all k ∈ RE do
        if NOT k ∈ Available.B ∪ candidates_output.B then
          if key_k ∈ Available.Γ ∪ candidates_output.Γ then
            candidates = candidates ∪ m(key_k, k)
            {Method needed that loads required object by its key. ε1-method for
            entity service}
          else
            candidates = candidates ∪ getEntityCandidates(Available, {k})
            {Derive all service candidates that are required for determining k}
          end if
          Available = Available ∪{k}
        end if
      end for
      if RE = ∅ then
        candidates = candidates ∪ m(∅, k)
        {The object k has no required predecessors. It needs to be created.}
        {This might require human-interaction.}
      else
        {At this point, all objects that are required for o can be retrieved}
        candidates = candidates ∪ m(RE, o) {Based on the available objects, o can
        be retrieved/created, too}
      end if
    end if
  end if
end for
RETURN candidates
```

[40]Note that an empty set RE leads to the omission of the inner for all-loop.

In the shipment example, algorithm 2 would check whether a `Shipment` object or its keys are available. As neither is available, the algorithm checks all required objects. Hence, the algorithm is recursively invoked for $RE = \{Order, Carrier\}$. As an `Order`-object is passed to the enterprise service, this recursion terminates without any change to the set of candidate methods.

The recursion for the `Carrier` augments the set of candidate methods with the new method m_1(∅, {Carrier}). Notably, this creation/retrieval method does not include any key information. The creation/determination of the appropriate key is subject to the implementation of the method. m_1 is an entity-specific candidate method and satisfies ϵ_3. After the termination of this recursion, another candidate method for the creation of the `Shipment` is added: m_2({Order, Carrier},{Shipment}). As m_2 satisfies ϵ_2, it is also an entity-specific method. In the second iteration, the algorithm identifies the entity-specific method m_3({Order}, {TradeCreditInsurance}) that satisfies ϵ_2. As algorithm 2 terminates, it returns Candidates = {(∅, {Carrier}), ({Order, Carrier}, {Shipment}), ({Order}, {TradeCreditInsurance})}.

Algorithm 1 continues with a value for EntityCandidates of {(∅, {Carrier}), ({Order, Carrier}, {Shipment}), ({Order}, {TradeCreditInsurance})}. Next, potential task-specific candidate methods are created. This process is described in algorithm 3. It is invoked with three sets of input data: the available objects from the input of the enterprise service, the respective required output of the enterprise service and the already identified entity-specific candidate methods.

Algorithm 3 getTaskCandidates(α, β, γ) – Determine service methods that are required to compute the enterprise service's data

```
Available ⇐ α {Parameter 1, set of available objects and keys}
Required ⇐ β {Parameter 2, set of required objects and keys}
entityCands ⇐ γ {Parameter 3, set of candidate methods}
for all a ∈ Available ∪ entityCands_output do
  for all r ∈ Required ∪ entityCands_input do
    if a = r then
      m = (a, r)
      ask {user involvement}
      if m is OK then
        candidates = candidates ∪ m
      end if
    end if
  end for
end for
RETURN candidates
```

Algorithm 3 uses the notion of the dependency $\bar{E}$ (40) that describes that the output of a service consumer must fit with the methods of the respective service provider. Using this dependency, it iterates over all possible output parameters (of a service consumer) and matches them with all offered input parameters of service consumers. Whenever there is a match, the user is asked whether the respective transmission of entities needs to involve a modification of the transmitted data. The manual involvement could either be realized using a user dialogue or by analyzing a model of the service orchestration that is created using the identified entity-specific candidate methods. Whenever the service designer de-

cides to add a task-specific service, an according method is added to the list of candidate methods. Usually, the constraint ts_2 is satisfied.
In the shipment example, algorithm 3 would be started with the following values: `Available` = `{Order}`, `Required` = `{Shipment, TradeCreditInsurance}` and `EntityCands` = `{(∅, {Carrier}), ({Order, Carrier}, {Shipment}), ({Order}, {TradeCreditInsurance})}`. In the example it is assumed that the `Shipment` needs to be approved. Hence, a task-specific candidate method m_4`({Shipment},{Shipment})` (that satisfies ts_2) is created.

Back in the main algorithm, the set of candidate methods in the example case is now `{(∅, {Carrier}), ({Order, Carrier}, {Shipment}), ({Order}, {TradeCreditInsurance}), ({Shipment},{Shipment})}`.
Based on the identified candidate methods, the set of *unused* data can be derived (algorithm 4). This data is the input data of the actual enterprise service that is neither computed by task-specific or entity-specific methods nor included in the output of the enterprise service. For each of the objects of the unused data, the user is questioned about creating entity-specific methods that either store the respective data or treat them otherwise without producing any output data (e.g., sending them to an external partner). As all data objects of the example are used, this step does not produce any additional candidate methods.

Algorithm 4 getStoreCandidates(α) – Determine service methods that handle unhandled input

```
Unhandled ⇐ α {Parameter 1, set of available objects and keys that are not handled}
for all a ∈ Unhandled do
  m = (a, ∅) {New entity method candidate for storing object a, satisfies ε3}
  ask user if m is ok
  if m is OK then
    candidates = candidates ∪ m
  end if
end for
RETURN candidates
```

Notably, additional functionality might be used to realize the given enterprise service. Especially (entity-specific) methods that store computed data, while the data elements are part of other service methods, into back-end systems are not included in the service coordination. This is for simplification reasons and aims to keep the service coordination simple. As coordination services are realized using (1..n) application services, suitable application services might be used within the coordination services' logic.

In order to accommodate the fact that the input to the algorithm that is taken from the requirements engineering, might be incomplete, the next step of algorithm 1 is to check for data that is required but not provided. To accomplish this, the set `UnprovidedInput` is filled and the `getEntityCandidates` method is invoked. As a result, the necessary steps for deriving the missing data are described by the notion of required entity-specific methods. These methods are added to the set of required methods.

After the set of required (candidate) methods is determined, algorithm 1 continues by matching the required set of methods with the methods that are already available. Such methods are part of services that were created in earlier projects or are shipped by soft-

ware vendors. Such a registry is represented in the algorithm by the set `Repository` that contains all methods.
The used matching algorithm is similar to the one presented by Schaffner et al. in [123]. First, it matches methods that have the same signature. Of course, this matching requires global data types or ontology-based mappings (cf. [124]). If a matching method is identified, the user is (later) asked whether the identified service method fits with the actual requirements. If the user agrees, the candidate method is removed from the set and the set of to-be coordinated service methods is enlarged by the identified method.
If no matching method can be identified, the algorithm checks whether methods exist that are sub-sets of the actual candidate method. All possible methods are kept in a list and the user is consulted which method is best suitable. According to [123], this list might be sorted according to the "match distance". This measure basically describes the missing parameters between a service consumer and a service provider.

Algorithm 5 ensureGranularity(α, β, γ) — Ensure Method α's Granularity

$m \Leftarrow \alpha$ {m is the method the granularity should be optimized for}
$SSMthreshold \Leftarrow 13$ {The minimum granularity of any method}
$PossibleCombination \Leftarrow \beta$ {The set of methods m could be combined with}
$AvailableMethods \Leftarrow \gamma$
if $SSM(m) < SSMthreshold$ **then**
  for all $n \in AvailableMethods \setminus \{m\}$ **do**
    if SSM(n) $< SSMthreshold$ **then**
      if (((*m* satisfies ts_1 **OR** *m* satisfies ts_2) **AND** (*n* satisfies ts_1 **OR** *n* satisfies ts_2)) **OR** ((*m* satisfies ϵ_1 **OR** *m* satisfies ϵ_2 **OR** *m* satisfies ϵ_3) **AND** (*n* satisfies ϵ_1 **OR** *n* satisfies ϵ_2 **OR** *n* satisfies ϵ_3))) **then**
        if ($m_{input} \subseteq n_{input}$ **OR** $n_{input} \subseteq m_{input}$) **OR** ($m_{output} \subseteq n_{output}$ **OR** $n_{output} \subseteq m_{output}$) **then**
          if $\forall i \in n_{input}.\nexists o \in m_{output} : o \in Req_i \vee i = o \wedge$ NOT (*n* satisfies $ts_2 \wedge m$ satisfies ts_2) **then**
            $f = (m_{input} \cup n_{input}, m_{output} \cup n_{output})$
            if $SSM(f) \geq SSMthreshold$ **then**
              $PossibleCombination = PossibleCombination \cup \{n\}$
              RETURN *PossibleCombination*
            else
              $PossibleCombination_f =$
              ensureGranularity(f, $PossibleCombination \cup \{n\}$, $AvailableMethods \setminus \{m\} \setminus \{n\}$)
              RETURN *PossibleCombination*
            end if
          end if
        end if
      end if
    end if
  end for
end if
RETURN *PossibleCombination*

After the completion of the loop for all identified candidate methods and for each possible combination of candidate methods and existing methods, algorithm 5 is invoked in order

to optimize the methods' granularity.
Algorithm 5 is a recursive function that determines for a given method, based on available methods, a set of methods the given method can be combined with. A combination of two methods is possible, iff:

- either the input or the output of two methods are related; and
- both methods are either task-specific or entity-specific; and
- there are no (direct or indirect) dependencies between the methods.

Whenever a method's granularity (measured by its SSM-value) is smaller than a given threshold, the input parameters as well as the output parameters are joined.[41] The algorithm uses the result of the analysis of chapter 4 as its threshold (13).
Whenever the direct combination of two methods does not increase the overall SSM-value sufficiently, the algorithm is invoked recursively until either no more suitable methods are available or the granularity threshold is reached.
Another possibility to manually increase the size of the signature of methods is to include predecessor objects (of the objects that are already part of a method's input set) into the signature of a method. The service design algorithm checks for the existence of the *required* objects within a coordination service. This is why augmenting the interfaces by predecessor objects is not performed automatically. Especially in message oriented systems, augmenting interfaces in this way might increase the reusability of the services as this would lead to a (partial) transfer of the objects that are necessary for object navigations.

Returning to the shipment example, it is postulated that there are no existent service methods available. Hence, the algorithm directly continues optimizing the single method's granularity. Assume that first the granularity of the method $m_1 =$ `{(∅, {Carrier})` is optimized by using the available methods `{(∅, {Carrier}), {({Order}, {TradeCredit-InsuranceInsurance}), ({Order, Carrier}, {Shipment}), ({Shipment}, {Shipment})}`.
Initially, $SSM(m_1) = 1$. Hence, suitable combinations are searched. As the only possible combination of joining m_1 with `({Order, Carrier}, {Shipment})` is forbidden, because the methods are directly dependent (the `Carrier` that is the result of m_1 is required by m_2), m_1 cannot be combined with any of the available methods.
The only possible combination of methods is to merge `({Order}, {TradeCreditInsurance})` with `({Order, Carrier}, {Shipment})`. This is because both methods are entity-specific and {Order} is a subset of `{Order, Carrier}`. The SSM-value of the resulting candidate method `({Order, Carrier}, {Shipment, TradeCreditInsurance})` is 10.[42] As there are no more suitable methods, the algorithm terminates. So the set *possibleCombinations* consists for the iteration of m_2 of the entry `({Order, Carrier}, {Shipment})`.

After the derivation of possible combinations of methods, the user needs to agree on the possible method combinations. As soon as a valid combination was identified, the

[41] Note that in contrast to the satisfaction of `Size.III`, the newly created method's SSM-value is not the sum of the single methods' respective SSM-values. This is because, in contrast to the definition of the property `Size.III`, both methods are not disjoint.

[42] Applying the same position in the $\overline{\mathtt{MV}(\mathtt{m_1})}$-space as it was done in the use case for which 13 was identified to be a suitable SSM-threshold.

algorithm terminates. The result is a set of candidate methods as well as the set of all coordination service methods that are required in order to realize a given enterprise service.
Based on the interfaces, it is a manual task to describe the functional requirements for the identified candidate methods.

For the example, it is assumed that the user agrees to use the proposed combination of methods. So the results of the algorithm are the following three candidate methods:

- m_1 = (∅, {Carrier}). *Functional requirements*: m_1 shall fetch a `Carrier`-object that will represent the company that will be asked to ship the goods of a given order.
- m_2 = ({Order, Carrier}, {Shipment, TradeCreditInsurance}). *Functional requirements*: m_2 shall perform the ordering of a shipment for a given `Order` with a given `Carrier`. Additionally, the order needs to be insured by `TradeCreditInsurance`.
- m_3 = ({Shipment},{Shipment}). *Functional requirements*: m_3 shall check the validity of a newly created `Shipment`. If a `Shipment` is valid, the `approved`-attribute shall be set to `true`. Otherwise `approved` shall be set to `false`.

Deliverables List of coordination service candidate methods; list of coordination service methods that need to be composed; functional requirements for all service candidates.

6.2.5 Step 4: Describe Service Orchestration

As service orientation is a paradigm for control centralization over distributed functionality, the control flow logic needs to be described. According to [11], potential types of logic that need to be considered are:

- "business rules
- conditional logic
- exception logic
- sequence logic" [11, p. 403]

Part of this service orchestration logic is typically included in the initial business process model. This is especially true for the sequence and conditional logic. These parts describe how the preconditions of service invocations can be met, how data is exchanged between services and how post-conditions of services should be computed.
"Business rules are statements about how business is conducted, i.e. the guidelines and restrictions with respect to business processes in an enterprise" [125, p. 9]. Exception logic finally describes how a process should react on exceptional states. Exception logic is normally both, part of a business process and part of a more general exception handling procedure. A process description will usually contain a procedure that will be used to react to business exceptions that are closely related to the process. However, the general part of exception rules describes how exceptions, that are not handled directly by the

process, should be treated. Reasons for not including such rules into processes are that they were not foreseen in the description (e.g., technical routing errors while calling a service).

In the example shipment process, the business rules are included in the realization of the service method m_3 = `({Shipment},{Shipment})`. This service only marks a `Shipment` as valid whenever it complies with company rules. An example would be that the shipment has to be executed within 3 days after creation.
Conditional logic in the example is the check whether a valid `Shipment` exists. If the `Shipment` was marked as not valid, the process continues differently then it would for a valid `Shipment`.
For simplification reasons, the example process does not include business exception logic. An example for general exception logic would be that in the case of the creation of the `Shipment` a communication error (on the service invocation part) would occur. As a company rule, such errors need to be reported to the central middleware help desk.
The sequence logic finally is described by the control flow of the business process. In the example, goods need to be scheduled for manufacturing before their shipment can be initiated.

Despite the description of the logic, the actual process model needs to be translated into a formally correct and possibly executable service orchestration description. In order to achieve this prior to a translation of the actual model, the correctness of the process model needs to be checked (cf. [126]). The actual checking is considered beyond the scope of the presented methodology. Complementary work such as [127] that checks for soundness of a process orchestration should be considered, too. For the combination of event-driven process chains (cf. [117]) and WS-BPEL (cf. [128]), [126] proposes an approach that uses EPC-based process description to derive sound orchestrations.

Due to the simplicity of the example, the soundness of the process is not checked in greater detail.

Deliverables Sound description of the process orchestration; conditional, exception logic and sequence logic; business rules

6.2.6 Step 5: Create Service Coordination Description

From the initial requirements, a service orchestration (with associated rules) as well as method candidates that act as the single service providers for the process orchestration were derived in the previous step. In order to further drill-down the abstraction level and realize the business process implementation, every single enterprise service needs to be described in detail.
This description includes the already known interface of the actual enterprise service as well as the coordination service method candidates that were identified in step 3. In this step of the methodology, the identified candidate methods are composed in a way that the composition provides the realization of the respective enterprise service. Usually, the coordination should be a sequence of all coordination service methods in a way that required data is produced before it is consumed. However, either due to the modeling of the initial business process or due to special requirements, the description of a service coordination might include conditions or parallel tasks, too. This is why the creation of

the coordination description is considered to be a manual (yet simple) step. In conjunction with business process modeling rules, this step could be automated. This might be realized by a simple, data dependency-based approach (that simply links services in a sequential way). An alternative would be to apply more sophisticated automated service composition approaches. [125] is an example for such an automated composition process. More general information about the requirements for automated service composition is given by Meyer et al. in [129].

In the shipment example, the description of the service coordination for the `{Order}, {Shipment, TradeCreditInsurance}` enterprise service, would be a sequential invocation of the identified candidate methods. The coordination is described in figure 28 using an UML activity diagram.

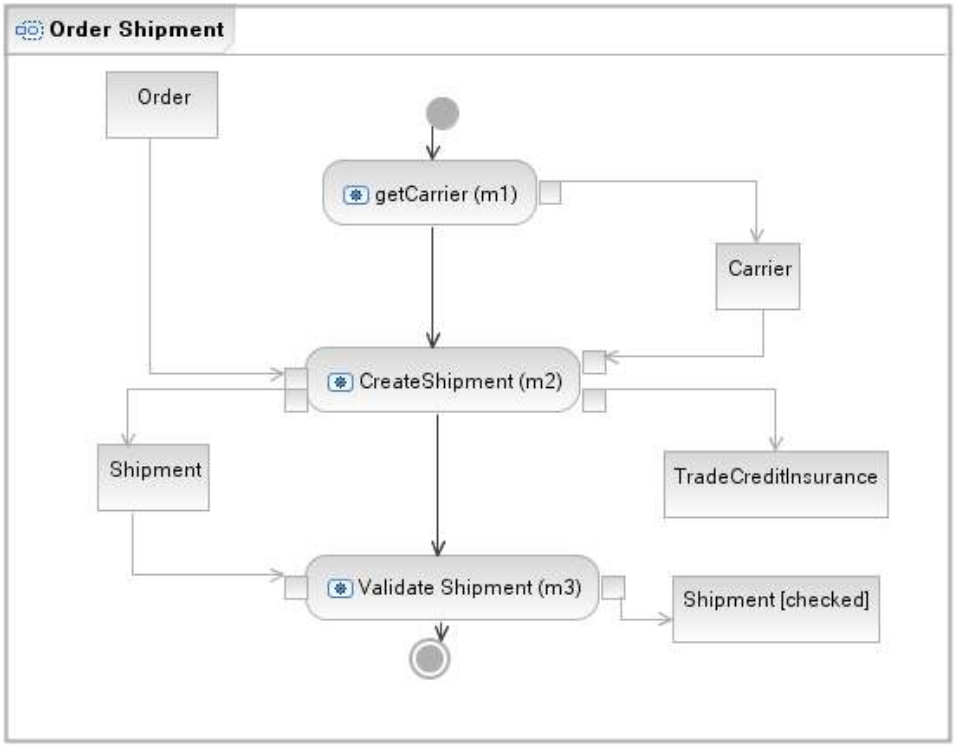


Figure 28: Model of the Service Coordination for the `Order Shipment` Enterprise Service

Deliverables Specification of the service coordination for every enterprise service using the candidate methods that were identified in step 3.

6.2.7 Step 6: Refine Candidate Methods

Depending on the outcome of the previous steps, the requirement to refine the candidate methods of step 3 might arise. Basically, this is a step the designer of a composite can perform in order to manually adjust the shape of the service method candidates.
One possible scenario for the need to adjust the candidate methods is that the assumed interface does not fit well together with the required service coordination. In particular, entity-specific creation methods might not be required for creating new entities. Furthermore, these methods could include manual lookups with human interaction. Whenever a human agent provides a required functionality of a method, context information is needed.

Hence, an example for an adjustment in this step is adding context information to entity-specific candidate methods.
Another possible requirement for the redesign of candidate methods is a too-complex service coordination. A measure for the complexity of a the service coordination candidates can be the *System's Service Coupling (SSC)* metric that was defined in chapter 3.2.1. If this metric is applied to the system as it is defined so far (enterprise services and the respective coordination services), a first indicator about complexity can be obtained. If a high complexity is indicated by a high SSC-value and the designer agrees on the complexity, the candidate methods of the coordination services can be refactored in a way that less complex service coordinations are possible.
A third possible need for a redesign is a poor degree of aggregation through the service coordinations. The *Density of Aggregation (DOA)* metric that was defined in section 3.2.1 is a way to analyze this quality aspect of the design.[43] Positive DOA-values for a complete system indicate an efficient complexity handling via aggregators. Since the metric in this context is applied to an aggregator with one *serviceCall*-port, the DOA value cannot be smaller than zero. This is why a threshold for the use in this context should arise out of experience from different projects. However, DOA for the single services is only a small indicator here. Additionally, the AD measure of the *Aggregator CentraliZation* metric (ACZ) should be identified for each service coordination.
Service coordinations potentially represent a control instance in a composite application. They add a control flow that is independent from the top-level service orchestration. Such aggregations might decrease the complexity (as it is measured by the SSC metric). But there is a trade off between the complexity in terms of coupling and an efficient complexity handling by the means of control centralization. In order to have a contrast to the coupling metrics, the *Aggregator CentraliZation* (ACZ) should also be measured for the overall system during this step.
The ACZ metric indicates a loss of control centralization if *dense* aggregators are part of a system's design (cf. section 3.2.1). Hence, the ACZ and the DOA metrics are in mutual contrast. As a rule of thumb, a DOA value of ≥ 0 and a relatively high ACZ value should not lead to a re-design. Again, the judgment of an experienced architect is only supported by these measures.
Finally, the centralization in terms of the SCZ value should be considered as well. Together with the ACZ value it indicates how the control flow is centralized in the system. By considering differences between the ACZ and the SCZ values, a hint about the centralization by the means of service aggregators is provided (cf. section 3.2.1).

The possibility or the need to reuse certain services that do not match the identified candidate methods is another possible requirement for a redesign of the candidate methods and their service coordination. Even if these methods were not identified by the algorithm in step 3, certain circumstances (from modeling mistakes over budget restrictions to political circumstances of the actual project) might require to definitely reuse certain service (methods). Whenever this is the case, the set of candidate methods needs to be changed accordingly and the affected service coordination needs to be redesigned.
In addition to the given reasons, step 6 is also the step where candidate methods are checked in terms of their degree of incorporating service-oriented principles. Especially the design of stateless services is a service-oriented design principle. Hence, in this step

[43] As the application services for each coordination services are not yet known, it should be assumed that each coordination service aggregates one application service. This way, the most defensive assumption is made.

the single candidate methods are checked whether they allow for a stateless realization. Finally, all candidate methods that will be used in later steps should be given names.

If the designer decides to refactor a candidate method for any reason, steps four and five of this methodology will need to be executed again. In particular, the service coordination will likely require a change.

In the small shipment example the following observation can be made: the DOA-value of the `Order Shipment` enterprise service is 0.61. The enterprise service is aggregated using three coordination services. This is why the density of aggregation is both good and sufficient and does by itself not imply a re-design.
The SSC-value for the overall system as it has been so far is 0.43.[44] Lacking an objective threshold, it can be assumed that an SSC-value of 0.43 does not indicate a high level of complexity. However, this is a subjective judgment of the given example.
The centralization metric ACZ has a value of 0 for the current design (with a central service orchestration and one service coordination. This means that at this point in time the control is not centralized at all. It is equally distributed between the service orchestration and one service coordination. This indicates a low level of modifiability. However, the complexity that is supposed to be handled by control centralization, is not very high in the given example.
In contrast, the centralization in terms of the SCZ metric (cf. section 3.2.1), indicates a relatively high centralization of 0.86. By interpreting the difference between the two centralization metrics ACZ and SCZ, it can be postulated that the control is obviously centralized in few aggregators that are not solely mediating the service providers. It could therefore be analyzed whether the service coordination might be too complex in contrast to the relatively simple service orchestration. However, when sticking to the business process-centric service orchestration, the only choice for a re-design with regards to control centralization would be to define an application service that matches the enterprise service `Order Shipment`. This should be kept in mind for the design of the application services in step 8.

The description of the m_1 method might reveal that the respective `Carrier` should not be created but furthermore retrieved form a list of possible carriers. In order to support this (possibly manual) step, the actual order for which a carrier is searched is required by the method. This is why the signature of m_1 is changed to: `({Order}, {Carrier})`. m_1 remains an entity-specific candidate method.
As the candidate methods were changed, in the example, steps 4 and 5 need to be executed again. While no changes of the service orchestration are required, the service coordination needs to be changed according to the new methods. A revised service coordination with the new `getCarrier` coordination service method is depicted in figure 29. The values of the design metrics for the system are not altered by this change.
Finally, the methods are named. m_1 is called `getCarrier`, m_2 is called `createShipment` and m_3 is called `validateShipment`.

Deliverables A list of stateless, named and probably revised coordination service method

[44] Assuming one central service orchestration that aggregates three service providers. One of these service providers is the `Order Shipment` service coordination that aggregates three service providers itself. At this time the (necessary) consumer of the top-level orchestration is not yet identified, but the orchestration considered an aggregator.

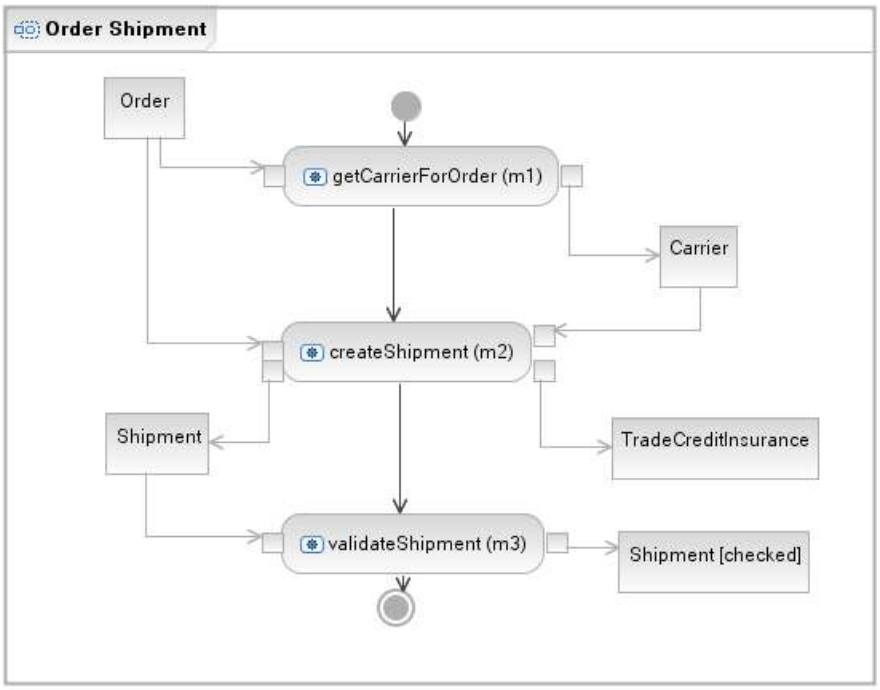


Figure 29: Revised Model of the Service Coordination for the `Order Shipment` Enterprise Service

candidates.

6.2.8 Step 7: Analyze QoS Requirements of Service Coordinations

The previous steps of this methodology produced a service orchestration that orchestrates enterprise service (candidates). Additionally, so-called service coordinations, that describe how coordination services can be used to realize the single enterprise services, were initially defined. In this step of the methodology the quality-of-service (QoS) requirements of the single service coordinations are analyzed.
Two elements are included in this assessment. The first is the availability of the respective enterprise service. Based on the required overall availability of an enterprise service, the required availability of the composed coordination services can later be derived. Enterprise services are service aggregators that, in fact, use service aggregators (coordination services) themselves. The availability of such service aggregators can be calculated using the *Avl* metric that is described in (13) in chapter 3.2.2 (page 44). As a part of step 7, the to-be value of *Avl* is defined for each enterprise service.

The second element that must be considered is the manner in which each service coordination for the single enterprise services deals with the unavailability of the composed services. Basically, this involves the notion of distributed ACID transactions (cf. section 3.2.2) as well as the definition of transactions with relaxed ACID properties.
ACID transactions are defined as a set of service methods of a service coordination that need to be executed together or not at all.
Transactions can also be relaxed in terms of their atomicity and isolation properties (cf.

section 5.8). In order to utilize relaxed, global transactions, (sets of) method calls can be marked as being *safepoints*. The return values of such method calls must be persisted in the data repository before subsequent services are called. In case of exceptions during subsequent steps, these steps need to be compensated. Processing is then restarted after the most recent safepoint with the data that was previously committed by that safepoint method to the data repository.
If safepoints are identified, compensating actions for a set of methods also need to be defined. In order to define such actions, a group of services and different exceptions, that trigger them, are defined. Possible types of exceptions include work item failures, deadline expirations and resource unavailabilities (cf. [130]). In order to realize a compensating activity, a corresponding service method has to be identified for each defined exception. Compensating operations that work on the data repository should be realized as services at the service coordination layer. Additional functionality that might be required needs to be realized as ordinary application services that are part of such an aggregation.
Service coordinations as a whole are always considered safepoints. Hence, they also have to commit their result to the global data repository. The compensating activities for the enterprise services should be defined as part of the business process model.

Using the service coordination that was described in the previous step of the example, the `getCarrierForOrder` method might be marked as being a safepoint. In case of failures, a carrier that has been identified for an order could be kept in order to avoid additional lookups for carriers after a failure. A compensating action for the set {`createShipment`, `validateShipment`} could be `deleteShipment`. An exception that triggers this compensating action could be a timeout of the `validateShipment` method. Consequently, the service coordination will need to be refined accordingly. The result of this is shown in figure 30.

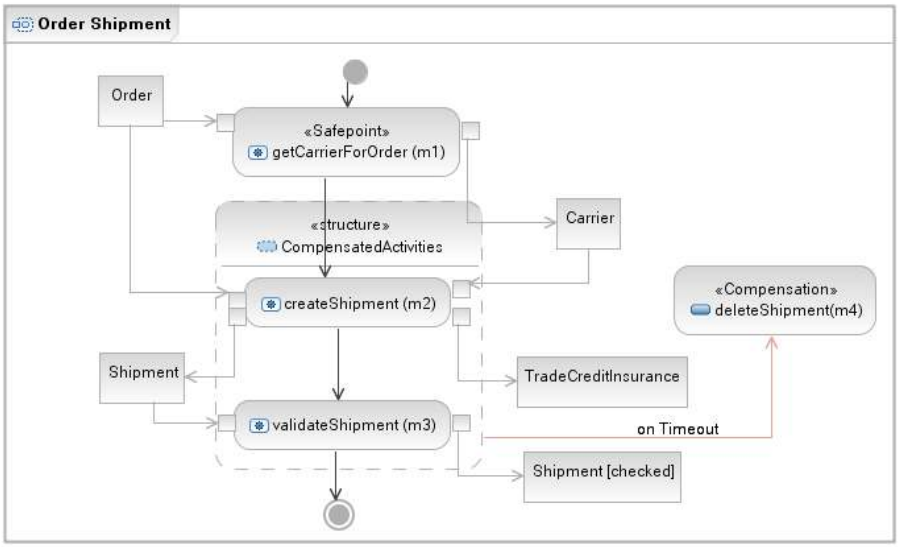


Figure 30: Model of the Service Coordination for the `Order Shipment` Enterprise Service with Transactional Properties

Assuming a required overall availability of this service coordination of $Avl = 99\%$, the three basic services need to have an θ_s-value (failure rate) of 0.3% if the service coordination platform has a θ-value of 0.1%.

Deliverables Sets of coordination service methods for distributed transactions; sets of methods that are marked as safepoints; sets of methods that are to be compensated; method descriptions for realizing compensations; required maximal failure rates for the coordination services for all enterprise services.

6.2.9 Step 8: Design of Application Services

The functionality of the coordination services often already exists within an application system. If so, four options for utilizing them in a composite application exist.

1. **Direct (Re-)use** If an application system exposes the required functionality as a service provider, a composite can directly use it. This is true for both, complete enterprise services and coordination services. In order to find appropriate services, a service registry should be used.
   If required functionality is not realized at all in a landscape, new services have to be created. These services need to implement the signature that was identified in step 3 and potentially revised in step 6.
   If human interaction is required as part of an application service that is to be created, the actual design of the user experience needs to be performed based on the deliverables of this step. However, the actual design approach for user interactions is not within the scope of this methodology. It solely delivers functional requirements to this design in terms of the interface of the service and the resource and organization view of the underlying business process. This is necessary because these views define the actual resource that should interact with an application service.

2. **Mediated Reuse** The required functionality might be implemented by an application system in a remotely accessible way but may not be exposed appropriately. This might be the case if the exposed functionality does not support the service protocol and/or the required service interaction mode of a service coordination flow. There might also be heterogeneity in terms of data structures between the designed coordination service and the remotely accessible application system.
   All these cases require the use of integration flows and the connectivity layer. The actual design of it is performed in the next step. This step's deliverable is the identification of a suitable piece of an application system's functionality that is remotely accessible. If a service cannot be realized only by the identified module, additional modules should be created.
   In established application landscapes this is the most probable scenario of reuse. The identification of suitable function modules of the application systems is likely to require extensive knowledge of the respective application system.

3. **Direct Use of Wrappers** If functionality is existent but not exposed, a wrapper could be created within the respective application system (cf. step 5 from [97]). Such wrappers need to be designed specifically for each application system and implement the interface of the service as it was designed. This approach, however, is based on the assumption that a wrapper can be deployed on an application system. If a governance rule prohibits changing COTS, integration flows are required.
   If there is no problem with implementing a wrapper within an application system, the deliverable of this step is the specification of an appropriate wrapper.

4. **Mediated Access** If the implementation of wrappers is prohibited, the respective application system can be integrated by using connectivity means and integration flows as part of a data integration approach (cf. section 2.1). In such a scenario, a `Data Service` as well as a `Heterogeneity Service` might need to be part of an integration flow.
   This approach is especially suitable for entity-specific methods that store, retrieve or modify entities. Task-specific methods could also be realized. However, the suitability depends on the actual application system.
   If mediated access to an application system must be realized, the data schema that should be accessed is the deliverable of this step. Additionally, the available connectivity options need to be described.

An indicator for the appropriate way of accessing application services can be the values of the re-use metrics *Re-Use Ratio* (RUR) and *Mediated Re-Use Ratio* (MRR) (cf. section 3.2.1). These two metrics indicate to which extent an implementation of a business process is coupled with another composite application. It is a task for a system's designer to choose the appropriate way of using services based on the actual requirements, technical constraints and modifiability issues as they are objectified by these metrics.

If only part of the required functionality can be identified in application systems, additional services will be required. These additional services are added to the list of required coordination services together with coordination services for the functionality that was identified.
According to the identified functionality, previously designed coordination services might be changed or adjusted, too. The quality-of-service requirements should also be identified for the revised service methods.

In the shipment example it is assumed that the required functionality is already implemented in two application systems. The `getCarrierForOrder` method is part of a supply-chain management system that is based on SAP R/3. A remote function call (RFC) `Carrier[] getAllCarriers(salesOrg, qualityReq)` might exist. This RFC requires an organizational code for a sales organization as well as the quality requirements for the actual shipment. Hence, *mediated reuse* is required.
The RFC does not necessarily pick one suitable carrier for a shipment. This is why the service coordination needs to be extended. A branch is required that, if multiple carriers are suitable, invokes a task-specific service method that involves a user interaction and determines the actual shipment. The service method shall be defined as `Carrier chooseCarrier(Carrier[])`. Using this not yet created service means a *direct use* of a service provider.
Concerning the `createShipment` method it is assumed that an Intermediate Document (IDOC) exists that is provided by the sales and distribution (SD) module of another SAP R/3 system. As IDOCs are transmitted using a different protocol and use different data formats, an IOF is required. The usage of such an IDOC represents *mediated reuse*.
The `validateShipment` is assumed to be supported by an RFC `Shipment validate-Shipment(Shipment)`. Also here, *mediated reuse* – via a combination of an IOF and IIF – is required.

Deliverables For *direct reuse*, the respective service methods need to be determined. For the *direct use* of services, the not yet created application services must be described.

For mediated reuse, suitable functional modules need to be identified. Their use must also be described. Further, suitable connectivity options are required and potential new services must be described as services for *direct use*.
If *direct use of wrappers* is applicable, the wrappers need to be specified. If application systems can only be used via *mediated access*, the data scheme that should be accessed needs to be described. Additionally, the available connectivity options need to be determined.

6.2.10 Step 9: Exchange and Transformation Design

Integration flows are required for any method of an application system that needs to be mediated. As the realization of the actual application services and their combination with the data exchange and data transformation layer (DET) are usually heavily constrained by technical circumstances, steps 9 and onward need to be performed while taking the actual target platform and application systems into consideration. This is how the platform-independent design that is created in the first steps is aligned to an actual platform.
In order to design the necessary integration flows, the required interactions with the service providers first need to be identified. This is achieved by specifying the appropriate service interaction pattern (cf. [98]) and its respective design decisions. Based on the identified pattern, the required integration flows, as well as some integration services, can be identified. The mapping from interaction patterns to integration flows is described in section 5.7.8. The identified mapping indicates the required integration services, as well as several design decisions for each integration service.
Based on the identified integration services, the design needs to be completed for each service. For the single integration services this includes:

- `Heterogeneity Service` If heterogeneous data structures or formats are identified, this service must be part of the respective integration flow(s). According to section 5.7.3, it needs to be decided on which levels transformations are required. If existing `Heterogeneity Services` can be reused, a multi-step transformation can be designed. For every level of transformation, the actual transformation needs to be specified completely. This also involves determining whether data lookups for enrichment are required.

- `Data Service` The necessity for a `Data Service` is determined by the respective interaction pattern. The data format of the connected application system is a deliverable of the previous step of the methodology, just like the connectivity options of the application system. Using this input, the connectivity layer of the composite application needs also to be specified.
  In complex interaction scenarios, additional design decisions are determined by the identified interaction pattern (cf. section 5.7.8).

- `Validity Service` If the chosen service provider or a type of realization indicates a high failure rate in terms of data validity, a `Validity Service` is then required. Typical scenarios that require the inclusion of a `Validity Service` are communications with external partners (“business-to-business”) or services that involve user-interaction.
  According to section 5.7.2, the design of a `Validity Service` involves the definition of the data representation as well as the data format. Additionally, possible return

values need to be specified. They also need to be incorporated into the decision logic of the respective integration flow.

- `Routing Service` As described in section 5.7.5, a `Routing Service` is required in order to determine the counter-party(ies) of an interaction. For more complex routing instructions, whether receivers are determined by the actual content of the interaction, whether dynamic endpoint lookups are required and the amount of receivers that are involved in an interaction must be described.

If the design of the integration flow(s) and the respective integration services reveal that the mapping from application services to coordination services cannot be accomplished with integration flows, the application service design needs to be revised. The identified constraint is then required in order to design the application service(s) differently.

This step of the methodology is exemplified by the `getAllCarriers` RFC. Its inclusion in the composite application needs to be mediated by integration flows. The suitable interaction pattern for the communication with the RFC is a *Send/Receive* interaction with a single, known counter-party. As the RFC only supports synchronous communication, no correlation or blocking of the sender is required. In case of communication errors, a manual error handling procedure should be initiated.
According to the description of section 5.7.8, this implies the need for both an IOF and an IIF. The IIF requires a `Data Service` while the IOF does not use such a service. Instead, the IIF is triggered by the IOF.
In order to complete the design of the interaction with the RFC, the single integration services need to be analyzed. Concerning a `Heterogeneity Service`, it was identified that the data types as well as the data structure is not the same inside the composite application and the SAP R/3 system. Additionally, a lookup is required. This is because the RFC requires the indication of an associated sales organization. By using the `Order` object that is passed to the IOF, a value-mapping can be used as part of the `Heterogeneity Service` for retrieving the respective organization for a customer of an order. Additionally, an order needs to be transformed into a `qualityRequirement` object. In the example it is determined that a quality requirement is an integer that indicates the value of an order as a class of order. The respective `Heterogeneity Service` needs to implement this transformation logic.
According to the mapping of the interaction scenario, a `Data Service` with a `Fetch Data` activity is required as part of the IIF for the inclusion of the RFC into the composite application. The necessary `Fetch Data` activity synchronously invokes the RFC and receives a reply. Such a reply is, in turn, transmitted to the requesting coordination service consumer.
The mandatory `Routing Service` is a simple one item list of receivers that points to the identified R/3 system's message server that hosts the RFC.
A `Validity Service` is not required.

Deliverables Complete specification of the integration flows for every mediated service interaction. The description needs to include the specifications for the necessary `Heterogeneity Services`, `Data Service`, `Validity Services` and `Routing Services`.

6.2.11 Step 10: Revise Service Coordination Description

The design of the actual application systems as well as the design of the mediating integration flows reveals the applicability of the top-down design for the coordination services. Based on the deliverable of the two previous steps, the coordination descriptions might be changed. If changes occur, the quality-of-service requirements need also to be analyzed for the new coordination.

The example could largely be realized by mediating existent functionality of application systems. Steps 8 and 9 identified what functionality can be reused and what additional service providers need to be created. The final service coordination for the example case, that includes all the necessary steps, is described in figure 31. It also includes the revised quality-of-service requirements.

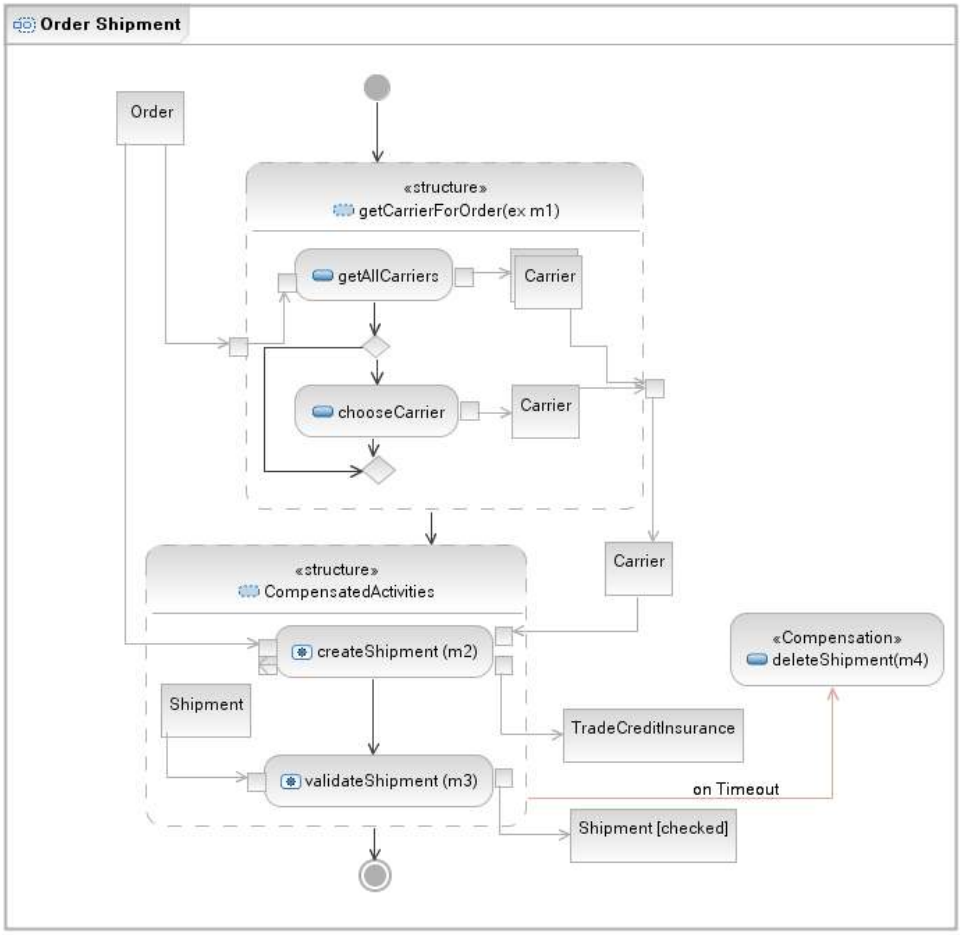


Figure 31: Model of the Revised Service Coordination for the `Order Shipment` Enterprise Service

Deliverables Final description of the single coordination services as well as the service coordination that aggregates the coordination services to enterprise services.

6.2.12 Step 11: Revise Enterprise Service Candidates

During the step of redesigning service coordinations, technical or organizational constraints that prohibit the realization of enterprise services that are aligned with the actual business process might be identified. If this is the case, two possibilities for proceeding exist.
First, the whole process can be started from the beginning. This is a preferable option if the modeled business process does not fit into the landscape of an organization. The identified constraints should then be used in order to define a business scenario that is in alignment with these constraints.
The second option is to revise the actual design of enterprise services in a way that the defined service coordinations can be realized. This will usually involve defining additional input and/or output parameters for the single enterprise services.
If the enterprise services are changed this is likely to also impact the process orchestration. If the enterprise services are changed, the service orchestration will need to be adjusted in a later step (step 14). It must be noted that changing a process orchestration is a violation of the objective of aligning application development with business needs.

In the example there was no indication that the enterprise services needed to be changed. This is because changes to the service coordinations were internal changes that did not affect the interface of the enterprise service.

Deliverables Decision about whether to continue the design or to start over. If the design is continued, the revised enterprise services must be defined. If the decision is to start again, the constraints that led to this decision are required deliverables.

6.2.13 Step 12: Define Events

At this phase of the design, the enterprise services are defined. In addition, how these services can be realized and what transactional requirements exist is described.

The use of many components of the reference architecture relies on the notion of events. They are used as identifiers for data in many steps. In particular, the data repository component relies on events. `Event Types` are used in order to define a consistency model for the data and define the visibility of data (cf. section 5.2.2).
The definition of `Event Types` and relations between them also determine how data visibility can be realized by the notion of workflow data patterns (cf. section 5.2.2). In this 12^{th} step of the methodology, the required data visibility patterns need to be identified and event types and relations need to be defined in accordance with the discussion in section 5.2.

If a workflow requires a more strict definition of visibility boundaries than described by the *Case Data* pattern (cf. [96, pp. 365f.]), several `Event Types` must be defined for one process (type). Additionally, data needs to be passed between the different scopes. In contrast to data visibility patterns, the different modes of data passing, as described by `WFData`, are not immediately supported. Special coordination services need to be included to support, for instance, the *Task to Task* workflow data pattern that describes "the ability to communicate data elements between one task instance and another within the same case" [96, p. 364]. By defining overlapping event type boundaries and allocating

coordination services for the passing of the data, transferring the necessary data from one scope to another can be realized. Alternatively, data can be stored in the application systems and read back by appropriate services (cf. the *Task to Environment - Push-Oriented* and the *Environment to Task - Pull-Oriented* workflow data patterns (cf. [96])).

The definition of the event types needs to comply with the design rules that are described in section 5.2.1. In order to finalize these descriptions, the relations between the identified `Event Types` also need to be described. They are required in order to define the parameter values for the eventing system.

The boundaries of the defined event types as well as the data passing requirements are input to step 14 of this methodology. This is because the respective events must be generated by including interactions with the eventing system into the service orchestration. Further, the invocation of a service coordination that passes data from task to task must be integrated into the process orchestration.

In the example process, there is no need to concurrently compute an `Order` from multiple process instances. There is also no data passing between the single process functions. This is why the data visibility pattern *Case Data* is suitable (cf. [96, p. 365]). According to the discussion in section 5.2.2, this results in one event type for the complete process being defined. In the example the event type *orderProcessing* is identified.

Deliverables `Event Types` and their respective boundaries; relations between `Event Types`; and optionally, the definition of coordination services that pass data between different scopes.

6.2.14 Step 13: Data Repository Design

Based on the `Event Types` that were identified in the previous step, the preconditions for every event type can be defined. Such preconditions might exist in terms of data existence. They describe that an event of a certain type can only be computed if certain data is stored in the data repository of a process. Such preconditions need to be identified in order to appropriately configure a data repository.
On top of data prerequisites, data types from the data perspective need to be incorporated into the design of the data repository (that addresses the principle of "interface reference" from [15]). This is achieved by defining the smart proxy for the given process as well as the transfer objects that are used to access the smart proxy. This information can be found by analyzing the data model of the business process. The business rules that are identified in step 4 should also be analyzed as the data by which they are defined is required to be kept in a data repository. Another input to the definition of smart proxies and transfer objects are the interfaces of the coordination services. The data that is required by those services also needs to be integrated into the data repository and made accessible by the smart proxies.
Based on the identified data and the respective transfer objects, the data objects can then be grouped by the event types that concern them. Together with the event type relations from the previous step, the data repository configuration can thus be completed.

With the artifacts produced so far, it is possible to determine which data in the data repository has to be accessed when and by which component. Additionally, it is determined how these accesses must be protected in terms of transactional properties. As a

result, in this phase it is possible to describe the interaction with the data repository that establishes the context for the composite applications.

The shipment example relies on the single event type *orderProcessing*. As described by the process, this event type can only be computed when a `Customer` object and a set of `Good` objects is accessible.
The data transfer objects that must be defined are `Customer`, `Order`, `Good`, `TradeCreditInsurance`, `Shipment` and `Carrier`. The data repository for the example process must be capable of managing these objects, as they are all *concerned* by *orderProcessing* events.

Deliverables Design of smart proxies and data transfer objects; configuration instructions for the data repository.

6.2.15 Step 14: Finalize Service Orchestration

The deliverables of the previous steps describe all the facets of a composite application. This description is aligned with the description of the business process. However, several constraints could require changing the service orchestration.
If a service coordination cannot be realized as required, this also impacts the service orchestration. Such changes need to be performed during this step. As such requirements for changing the orchestration are usually imposed by application system-specific constraints, the adjustment of the orchestration is not structured further.

A mandatory activity of this step is the design of the necessary `Decision Services`. Based on the business rules that were identified in step 4 and the available data that is represented by the design of the data repository, the rules need to be described and stored into a `Decision Service`. This might either be performed by using a `RulesAdministration` service (cf. section 5.9.2) or by hard-coding the respective rules under consideration of the data repository.
The actual way of integrating the `Decision Service(s)` must also be decided upon. This is dependent upon the actual platform the composite should be realized. Based on the integration mechanism, the orchestration might need to be adjusted (e.g., by implementing a service interaction with a `DecisionService` prior to a *switch* command).

In order to realize the identified types of events with their respective boundaries, the service orchestration needs to be extended with such interactions. This is achieved by adding calls to the `Event Service`. The operations `update` and `finish` need to be invoked at the respective boundaries of event types. If data needs to be passed over such boundaries, the service orchestration needs to be extended with the invocation of the service coordinations that were described in step 12 for this purpose.

The service orchestration naturally includes the invocation of the appropriate services. This involves service coordinations as well as application services of the back-end systems that match with an enterprise service. The invocation of those services needs to be accompanied by a respective callback method.

The example process can be realized as an orchestration of two application services and the service coordination that was defined during the previous steps. A diagram of the final service orchestration, representing the central control instance of the composite application, is depicted in figure 32. Additionally, other associated components are added

to provide an example of the single components' interactions (described more in depth in chapter 5).

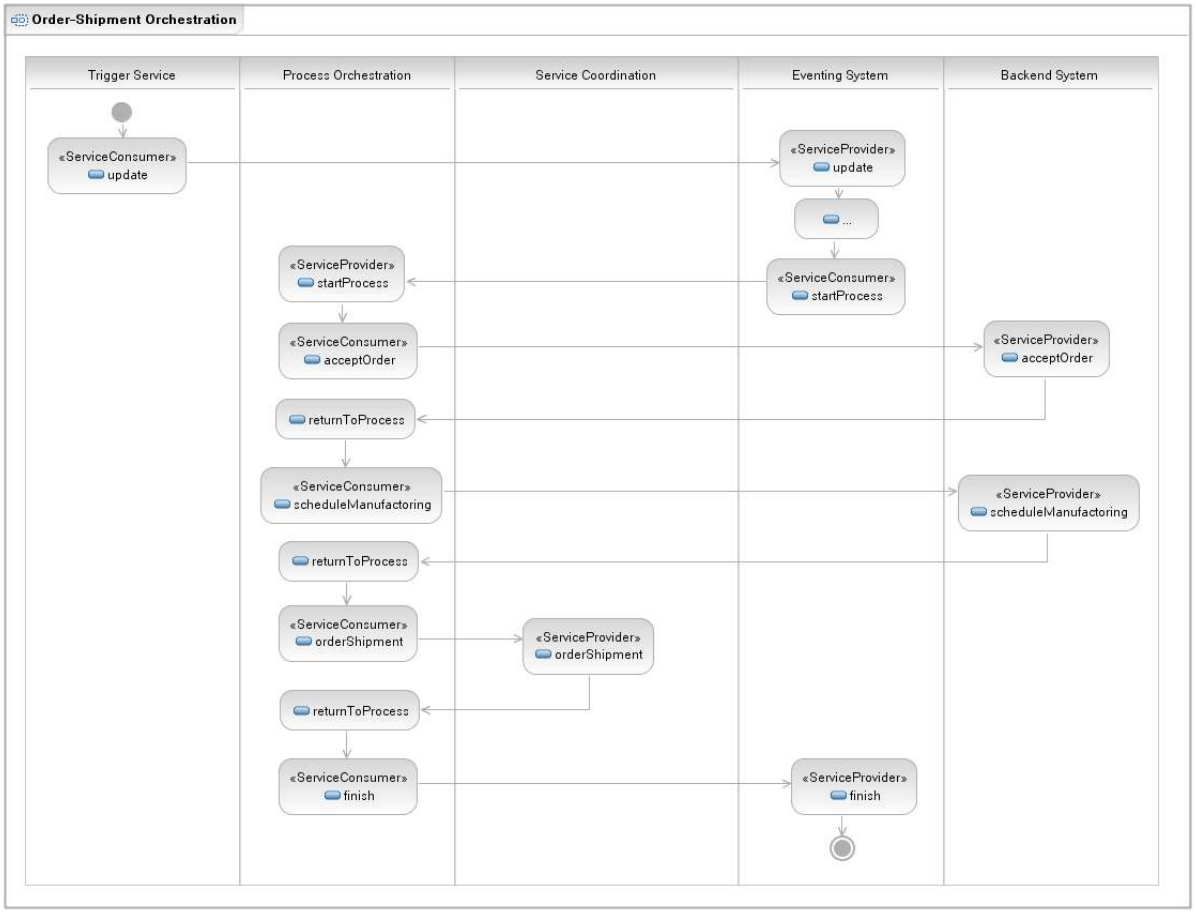


Figure 32: Final Orchestration for the Example Process

Deliverables Business rules that are formulated such that they can be interpreted by the actual `Decision Service(s)`; decision regarding the method of integrating `Decision Service(s)` in a platform-specific way; revised service orchestration that incorporates communication with the `Decision Service(s)`, `Event Service` and service coordinations for data passing; all necessary changes to the orchestration that are necessary due to informal constraints also need to be reflected.

6.2.16 Step 15: Finalize Exchange and Transformation Design

This step of the procedure is required in order to describe the `Trigger Services` that are needed to mediate service interactions.
Based on the event types and the data repository design, the data that is required as a prerequisite for a process is known. By analyzing the actual interaction, the source of this data can also be identified. If the source of such data is an agnostic application system, a `Trigger Service` needs to be integrated. By adapting to the interface of the service consumer (the application system), the interface of the respective `Trigger Service` is determined. Based on the identified event type, the `Heterogeneity Service` of the respective `Trigger Service` can also be designed. Additionally, filter logic needs to

be described. This logic is determined by the scenario as well as by technical constraints of the service consumer invoking a `Trigger Service`. Finally, based on the data repository design, the *write* activity of a `Trigger Service` that stores the data into the data repository can be defined.
The initial design of the data exchange and data transformation performed in step 9 involves the analysis of service interaction patterns as described in section 5.7.8. If the need for a `Trigger Service` was identified, it needs to be designed at this step.
In order to prepare the upcoming implementation it needs to be verified whether the target platform supports all necessary interaction patterns. This is because some interactions that are not natively supported can be realized by adding `Trigger Services` to the model of the data exchange and data transformation layer.

The single event type of the example process requires a `Customer` object and a set of `Good` objects in order to be executed. As a deliverable of step 1 it was described that an order is submitted electronically. The order and customer data are transmitted to the supplier using the interface of the `acceptOrder` method. A `TriggerService` is required that implements this interface, triggers the eventing system to produce an `Event` of the *orderProcessing* type and transmits the data into the data repository of the composite application. No filter logic is required. The `Trigger Service` for the shipment process is depicted in figure 33.

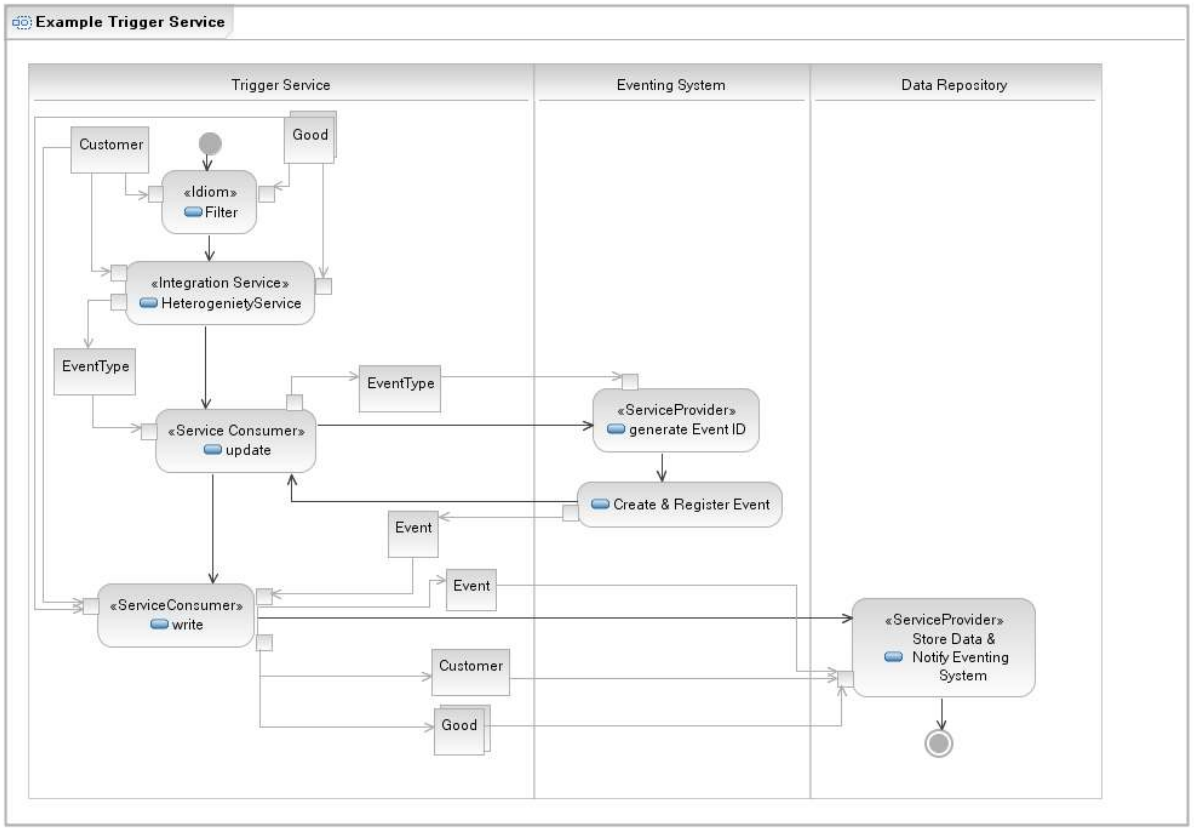


Figure 33: `Trigger Service` for the Example Process

Deliverables Finalized description of the integration flows including the design of all necessary `Trigger Services`.

6.2.17 Step 16: Pass over to Implementation

After step 15, the design of the composite applications and the services they consist of is finished. The design is described in a platform independent manner. Additionally, some constraints of the target platform are also incorporated. In particular, this involves the possibilities of interacting with application systems, a description of the service orchestration and the integration of `Decision Services`.

Based on this design, a composite application can subsequently be implemented. It is not recommended to add additional design artifacts for this phase. In contrast to the idea of a Model Driven Architecture (cf. [131]), this methodology does not aim to transform a platform independent design into an executable, platform-specific application. Experience shows that a design typically does not encompass all issues that occur during an implementation phase. If that were possible, design models that covered all aspects would be as complex as a final implementation. Because of this, the implementation phase of a composite application should be oriented towards the design performed by this methodology. However, the personal expertise and experience of the single developer should also be leveraged.

6.3 Summary

The design methodology discussed in this chapter describes how a business process can be used as input for the design of a composite application that supports the respective process.
The methodology focuses on a business process as the description of the service orchestration that centralizes the control within a composite application. Emphasis is placed on deriving the design of the orchestrated services. In order to allow for business-aligned orchestrations, a service meta-model is applied as part of the methodology. The meta-model reflects a separation of concerns that is achieved by the platform independent reference architecture. According to this meta-model, business functions are considered as enterprise services. These services are realized as aggregations of more basic services (referred to as coordination services). While the enterprise services are specific to a business process, the coordination services are designed in order to allow for their reuse. This is why the overall design methodology includes a detailed approach to the design of these services.
The way services are derived and designed is inspired by several pre-existing service design approaches (cf. [11], [120] and [121]). The approach presented here focuses on two additional objectives: minimizing requirements in terms of input to the methodology and maximizing reasonable reuse while avoiding "over engineering" service design principles. In order to minimize the input requirements, the design approach focuses on functional dependencies and avoids the notion of formal semantics. In order to maximize reuse while keeping the approach applicable, it incorporates the findings of the analysis that was made on service design principles (in chapter 6). The service methods that are the outcome of this service design approach are again aligned with the reference architecture.

The overall design methodology uses the identified service method candidates in order to derive the complete design of a composite application. For this sake, all components of the reference architecture are designed in alignment with those candidates, service-oriented principles and interaction and consistency requirements. Service-oriented principles were

included by using the reference architecture and design metrics for the assessment of intermediate design artifacts. Also, constraints that were imposed by an existent application landscape were considered and incorporated into the design. This is why a large part of the methodology is concerned with the adjustment of the design artifacts that were initially derived top-down. The deliverable of the overall methodology provides the input for the actual development of a composite application.

7 Platform-Specific Reference Architecture: Applying the Concepts to the SOA Platform of BASF IT Services

The reference architecture proposed in chapter 5 describes, in a platform-independent way, how the realization of composite applications can be standardized while service-oriented principles are incorporated. In terms of the Model Driven Architecture (MDA) classification, it represents both, a meta-structure for the design of composite applications as well as a *virtual machine* (cf. [91, pp. 2-6]). The description includes both a design-time aspect of composite applications and a run-time aspect that defines requirements for an execution environment. The design methodology that is presented in chapter 6 describes how an actual business process can be used in order to design a composite application that is based on this architecture. In order to realize a composite application it is necessary to describe how such a design can be realized on an actual platform. In MDA terms, a platform model for the Platform Specific Model (PSM) is required. Such a mapping from the platform-independent reference architecture to a platform-specific architecture is a scenario-independent, generic description of a runtime execution environment that can be used to implement scenario-specific composite applications.

The reference architecture and the development methodology were developed in order to support a project that was conducted by BASF IT Services. This project's aim was to identify how application development could be improved by the service-oriented architectural style and how composite applications could be realized.
BASF IT Services is an IT service provider of industry companies that (among others) focus on chemical products. As the majority of the business activities of BASF IT Services are related with such companies, BASF IT Services incorporates the IT strategy of its main customers into its portfolio.
Without going into specifics, it is obvious that the key differentiator of industry companies is not its application landscape in the first place. This is why such companies follow a vendor-centric IT strategy. Through this strategy it is understood that a vendor is chosen as a preferred supplier for a certain domain. For any new software that is required, it is then usually determined whether the respective product of the chosen vendor fulfills the actual requirements of the company. If so, the product is chosen. If not, other strategies are applied.
This strategy is an alternative to a best-of-breed strategy that aims to choose for each aspect the optimal software on the market. By choosing this strategy, the overall total cost of ownership for the respective application domain should be reduced.

This strategic decision also influences the analysis of the service-oriented architectural style. This is because, as an external supplier, BASF IT Services has to anticipate that main customers are likely to choose a certain supplier for building composite applications. This software vendor is SAP (cf. [132]).
When the project was initiated in 2005, SAP had launched a technology platform called SAP NetWeaver (cf. [133]). This is why an objective of the project that investigated the suitability of the service-oriented architectural style project was to evaluate whether SAP NetWeaver would be suitable to realize composite applications. For this sake, a case study was planned.
In order to use the opportunity of this case study for validating the concepts of this thesis,

SAP NetWeaver was also chosen as the target platform for the runtime framework of the platform-specific reference architecture.

To evaluate the suitability of NetWeaver as a platform for composite applications, two tasks are required. First, the evaluation comprises the general description of a platform-specific reference architecture that implements all elements of the platform-independent reference architecture that was described in chapter 5. Second, the evaluation includes a case study. The aim of such a case study is to informally evaluate "hands-on" experiences with the given platform. The case study that was conducted is described in chapter 8. The description of this platform, its general suitability and the platform-specific reference architecture are described below.

7.1 Elements of the SAP NetWeaver Platform

In contrast to the enterprise resource planning (ERP) products of SAP (e.g., R/3), SAP NetWeaver is a technology platform that is provided in order to enable customers to use open technologies when addressing their business needs (cf. [134]). These open technologies are web services [36] including the *Web Service Description Language* WSDL [135] that is used to describe services, SOAP [27] and the *Hypertext Transfer Protocol* HTTP [136] as the common transport protocol as well as the *Web Services Business Process Execution Language* WS-BPEL [128]. Additionally, SAP included a Java stack that allows for programming applications and services in Java [137].

On top of these open technologies (and some proprietary SAP technology), the SAP NetWeaver suite contains a set of products that can (mostly) be used independently from each other. SAP NetWeaver is not one single tool or product. It is a suite of tools that have a common technology platform from SAP. Additionally, they are all based on the SAP Web Application Server (WAS). At the point when the platform-specific architecture was defined, there were no common components that would transform SAP NetWeaver into a holistic platform.
The single components that are positioned to be potential platforms for composite applications are roughly outlined in the following sections. The described versions are part of the SAP NetWeaver 2004s stack.

7.1.1 SAP Web Application Server

The product Web Application Server (WAS) actually consists of two independent technology stacks. One is the so-called WAS ABAP and the other is called WAS Java. The ABAP stack is an evolution of the SAP R/3 application system that was simplified by removing all ERP components. Hence, the WAS ABAP can be used to program and execute programs that are realized in ABAP language [138]. Besides the various ERP modules of SAP, the WAS ABAP provides a platform for other higher-level technology platforms. One example is the SAP *Business Workflow* [139] that can be deployed on this application server. Communication is enabled by the *Internet Communication Manager* (ICM) that is used for handling HTTP requests for both stacks of the application server. The so-called Java-Stack of the WAS is a Java J2EE 1.3-compliant [140] application server. According to the specification it allows for the execution of Java-based implementations

and provides means for realizing transactional security, naming, reliable messaging and persistent data management. Additionally, the WAS provides a *Software Deployment Manager* (SDM) that is used at design-time for the deployment of applications.
The single components of the SAP WAS are shown in figure 34.

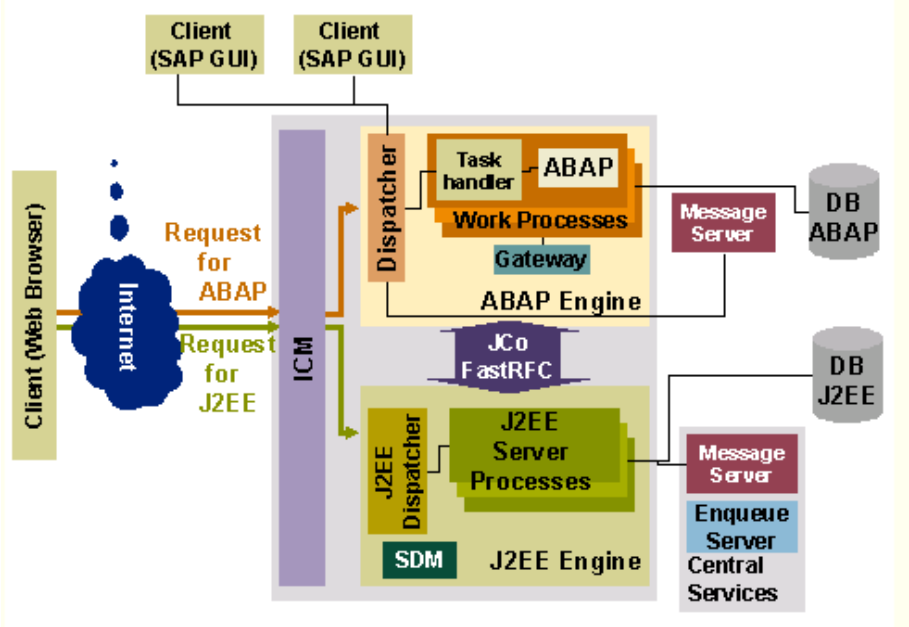


Figure 34: Components of the SAP Web Application Server [141]

Both application servers can be used as service agents. Service consumers as well as service providers can be deployed. The protocol that is used between services is HTTP-based SOAP.
All other components of SAP NetWeaver are based on at least one of the two types of WAS. An installation of a single stack of the WAS is possible.

7.1.2 SAP Exchange Infrastructure

The SAP Exchange Infrastructure (XI) is the enterprise service bus solution of SAP XI and includes a runtime as well as a design-time environment that can be used for reliable communication with arbitrary back-end systems. It is a dual-stack solution as it requires both the WAS ABAP as well as the WAS Java stack and is completely based on XML [142], SOAP and HTTP as communication protocol.
The single components of XI are shown in figure 35.

The *Integration Builder* forms the proprietary design-time environment. It is used for designing and configuring integration relations among various systems. Designing such relations includes the description of interfaces, structural mappings and so-called business processes. Interfaces are described using XML standards like *XML Schema* (XSD) [144] or WSDL [135]. Mappings are SAP-proprietary components that transform messages from one format into another. The so-called business processes are WS-BPEL-based descriptions of service orchestrations.
Configuring relations includes the notion of endpoint referencing, routing and the application of various adapters.

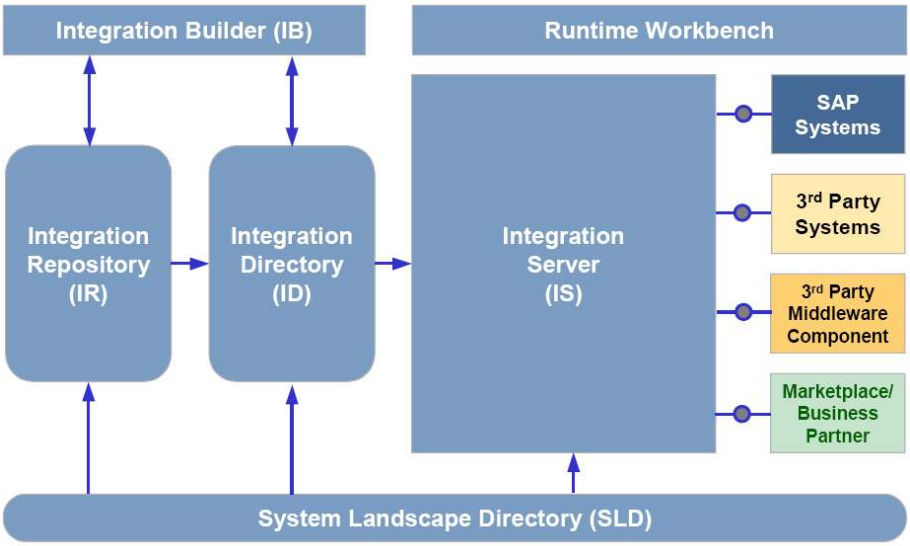


Figure 35: Functional Components of the SAP XI [143]

During run-time, the *Integration Server* (IS) acts as the ESB. It executes the artifacts that were specified during design-time. The single components of the IS are shown in figure 36.

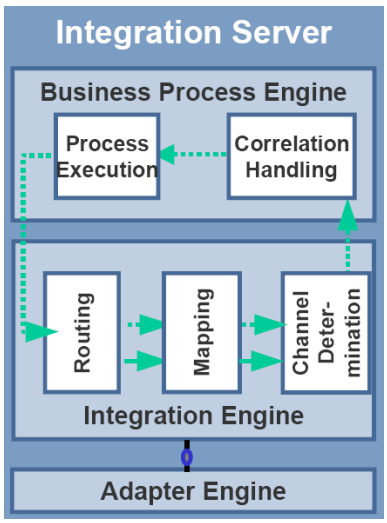


Figure 36: Components of the SAP XI Integration Server [143]

The IS consists of three parts: the JCA [145] compliant adapter engine (AE), the business process engine (BPE) and the integration engine (IE).
The AE allows for the deployment of arbitrary adapters. Such adapters establish the connectivity to back-end systems. Additionally, the AE transforms all data that is forwarded to the IE into an XML-based data format. This is required as XI internally relies on XML-based messages.
The IE establishes a pipe-and-filter architecture. The pipe follows a fixed pattern: incoming messages are routed, transformed (mapped) and forwarded to the actual receiver. Possible receivers are application systems that are connected via adapters or processes that are executed in the BPE.
The BPE is a SAP Business Workflow-based runtime engine for processes that are described using WS-BPEL. It includes a correlation handler that allows for dispatching

incoming messages to the appropriate process instance. All service calls that are performed by a WS-BPEL-based service orchestration are forwarded to the IE that treats those messages. Messages are stored and forwarded using internal queues of the IS that can be prioritized.
XI allows for synchronous messaging, asynchronous messaging and asynchronous messaging with acknowledgments. For both types of asynchronous messaging both exactly-once and exactly-once-in-order semantics are possible. Hence, SAP XI can provide guaranteed delivery and queuing functionality. Version 3.0 does not provide support for distributed atomic transactions.

7.1.3 SAP Composite Application Framework

The SAP Composite Application Framework (CAF) provides a programming model that was introduced by SAP to facilitate the programming of composite applications. It includes a Java-based runtime environment as well as a development environment. It is based on the SAP WAS Java stack.
The CAF provides an architecture that can be used to realize user interfaces or services and provides a persistence framework. All components of the CAF architecture are depicted in figure 37.

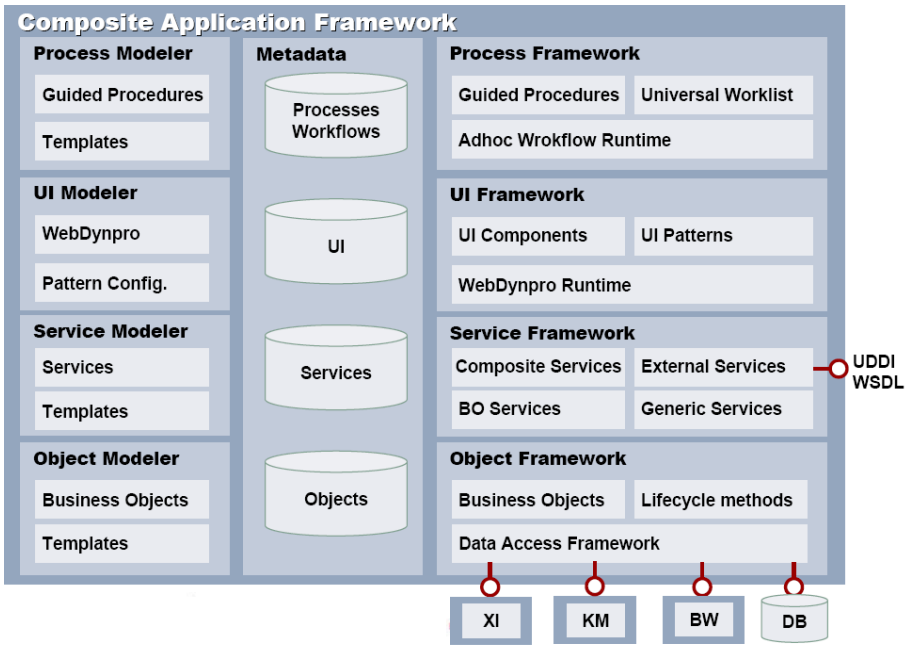


Figure 37: Components of the SAP Composite Application Framework [146]

The single components are structured as layers: the basic providers of persistence and functionality are application systems and (external and internal) databases. Access to this functionality is either possible by exposing the user-interface of application systems directly via a web-based interface or by generating so-called *Entity Services*. Technically, *Entity Services* are realized as J2EE stateless Session Beans that use generated Entity Beans (for database connectivity), web service client proxies or JCO-based[45] RFC client

[45]JCO stands for "Java Connector" and is a proprietary Java-based technology that is used in SAP environments for communicating with ABAP-based, proprietary SAP remote function calls (RFC).

proxies. *Entity Services* form the data model of CAF-based applications.
Application logic is realized in CAF-based applications by so-called *Application Services* that operate on *Entity Services*. *Application Services* are also realized as stateless J2EE Session Beans. *Application Services* can be exposed as web services or as so-called local references if the respective service consumer is deployed on the same WAS. This part of the CAF is also referred to as "CAF Core".
The user interface framework that is used by the CAF is SAP WebDynpro for Java framework (JWD). JWD is a proprietary SAP framework that implements the model-view controller pattern (cf. [37]). Models of the JWD are either local references to *Application Services* or external web services. JWD is a complex framework that can be used to implement complete applications independently of the CAF or the SAP Enterprise Portal.

In the context of the CAF, so-called Guided Procedures (GP) can be used to realize structured user interactions. SAP GP is a proprietary framework for realizing user-centric workflows. Work items are assigned to users by the notion of the *Universal Worklist* (UWL). The execution of the respective work items is realized as a workflow that defines a control-flow on-top of so-called *actions*. *Actions* are wrappers for so-called *callable objects* (CO) and are assigned to roles that can be used to identify users. One action can contain up-to two COs. Different types of COs exist. All different types of COs can be distinguished into two classes: COs with and COs without user interaction. One example of COs with user interaction are JWD COs that allow the use of WebDynpro applications as steps of a GP. Another user-centric type of CO is the so-called "URI-COs" that allow the inclusion of external applications that are accessible via HTTP, identified by a *Uniform Resource Identifier* (cf. [147]) and expose their user interface using the *Hypertext Markup Language* (cf. [148]).
Examples of the class of COs that lack user interaction are the "Web Service CO" that can be used to invoke external web services that are described using WSDL or the "CAF CO" that can be used to invoke *Application Services*.
GPs maintain a process context that is passed from one action to another. The workflow description language that is used by the GP execution engine is undocumented.

7.1.4 SAP Enterprise Portal

The SAP Enterprise Portal (EP) is the portal solution of SAP that aims to unify the interaction of users and application systems. This addresses both, organization-internal users as well as external users. The components of the SAP EP are depicted in figure 38.

Roughly described, the EP offers the functionality to cluster users by assigning roles to users. Each *role* in-turn describes a set of *pages* that form the user-interface that is exposed to the users that are assigned to the according roles. Pages cluster groups of so-called *iViews* that are organized using a defined layout on a *page*. *iViews* are similar to *Java Portlets* (cf. [150]) but do not implement the specification of JSR 168. *iViews* standardize the access from an EP to external content providers. Various types of *iViews* exist. Examples are *iViews* that can be used for accessing GPs or *iViews* that encapsulate JWD applications. By using GPs or JWD applications in the context of EP, user-management capabilities (among others) are added to these components.

The SAP EP consists of many more components, such as facilities for real-time collaboration, content management, access to back-end systems or single-sign-on facilities. As

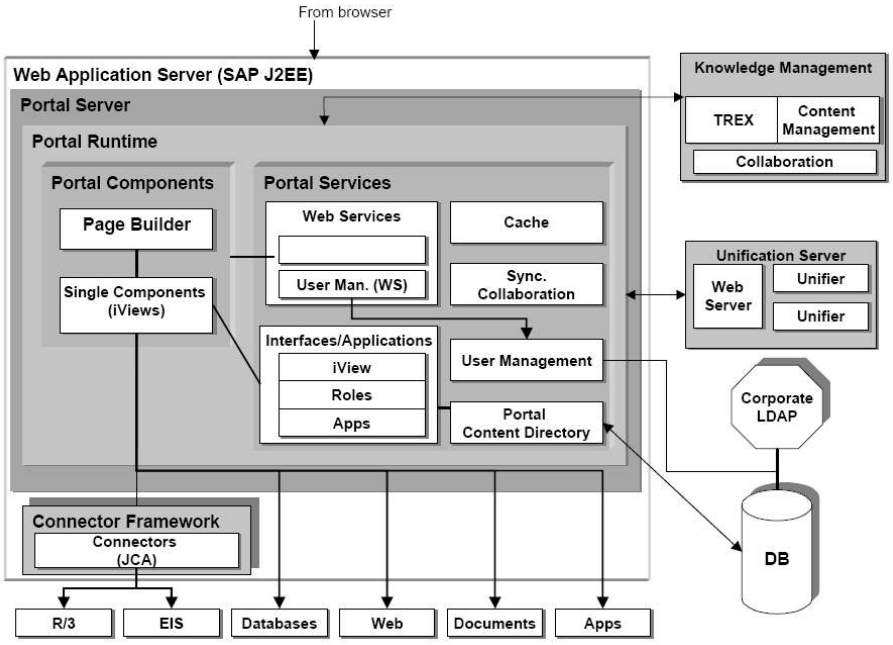


Figure 38: Components of the SAP Enterprise Portal [149]

these components are not in the primary focus of composite applications, their description is omitted.

7.2 Platform-Specific Reference Architecture for SAP NetWeaver

The reference architecture for composite applications described in chapter 5 standardizes composite applications in a way that benefits of the service-oriented architectural style can be utilized with greater ease.
The design methodology that was described in chapter 6 can be used to describe composite applications based on the platform-independent reference architecture. In order to realize and operate a composite application, a platform-specific reference architecture is required. A generic description of such a platform-specific reference architecture facilitates the mapping of platform-independent design artifacts to actual composite applications. In the terms of the Model Driven Architecture, it represents a *virtual machine* that underlies the platform-specific model (cf. [91, p. 2-6]).

As outlined in the introduction of this chapter, SAP NetWeaver was the platform of choice for BASF IT Services in this project and needed to be analyzed regarding the feasibility of realizing composite applications. The subsequent sections describe how all elements of the reference architecture can be realized by using the above-described components of SAP NetWeaver. In doing this analysis, the suitability of SAP NetWeaver will also be analyzed.

7.2.1 Eventing System

The components of an eventing system are described in section 5.5. Basically, an eventing system offers publicly accessible services, maintains a set of persistent data, implements logic for maintaining relations among events and acts as a service consumer when initiating process orchestrations. An eventing system does not necessarily require a user interface. By using the CAF, this broad set of functionality can be realized with NetWeaver.

An `EventService` can be realized as an *Application Service* that is exposed as a web service. It operates memory-internal (using local references) on other *Application Services* that realize the `EventRegistry` and the `EventIdGenerator`. These two services in turn operate on *Entity Services* with local persistence in order to keep track of active events within (all) the composite application(s) and comply with the defined rules between types of events.
The service consumer that triggers service orchestrations can be realized by using an *Entity Service* that is generated based on the WSDL description of the orchestration layer (cf. section 7.2.6). The endpoint that is referenced by this web service proxy can be static. This is because the process orchestration's interface is static and routing can be based on the type of the transmitted event inside the orchestration layer. This way, the drawback of CAF *Entity Services* only having fixed endpoints can be addressed.
By overloading the `setProcessEndpoint`-operation in a way that it accepts names instead of endpoint references, the name of the respective *Entity Service* that should be used for the connection can be provided. This way, a certain flexibility in the reference to the orchestration layer can be included. However, this relies on the deployment of new services which can not be done during configuration time. For this reason, having the routing mechanism inside the orchestration layer is preferable.

How the eventing system can be realized by using CAF services is depicted in the diagram of figure 39.

In the release of the CAF that was available to the project (NW 2004s, SP 8), there was no out-of-the box solution for realizing intrinsic event generation. This is because no timer-mechanism was available. This is why this mechanism has to be implemented differently. In order to activate events that are blocked by other active events, the `ActiveEvent` service is checked from within the `unregister`-operation of the `CAFEventRegistry`. As this operation is invoked whenever an event is being de-registered, other events can be activated based on this invocation rather than periodically checking for available events. The actual dispatching that takes place by invoking the `dispatch`-operation of the `EventRegistry` is initiated by the `notifyaboutData`-call-back operation of the `CAFEventService`. The `dispatch`-operation also manages to comply with the event rules that are stored into the `EventRelation` *Entity Service* by using the locally persisted `ActiveEvent` *Entity Service*.
Administration can take place by using the remote accessible `CAFEventingAdministration` *Application Service*. As no timer mechanism is available, the operations for administering intrinsic event generation are not supported. The relations between event types are stored by the `CAFEventingAdministration` *Application Service* using the `EventRelation` *Entity Service*.

Although exposed as web services, the operations of the `CAFEventService` are mediated

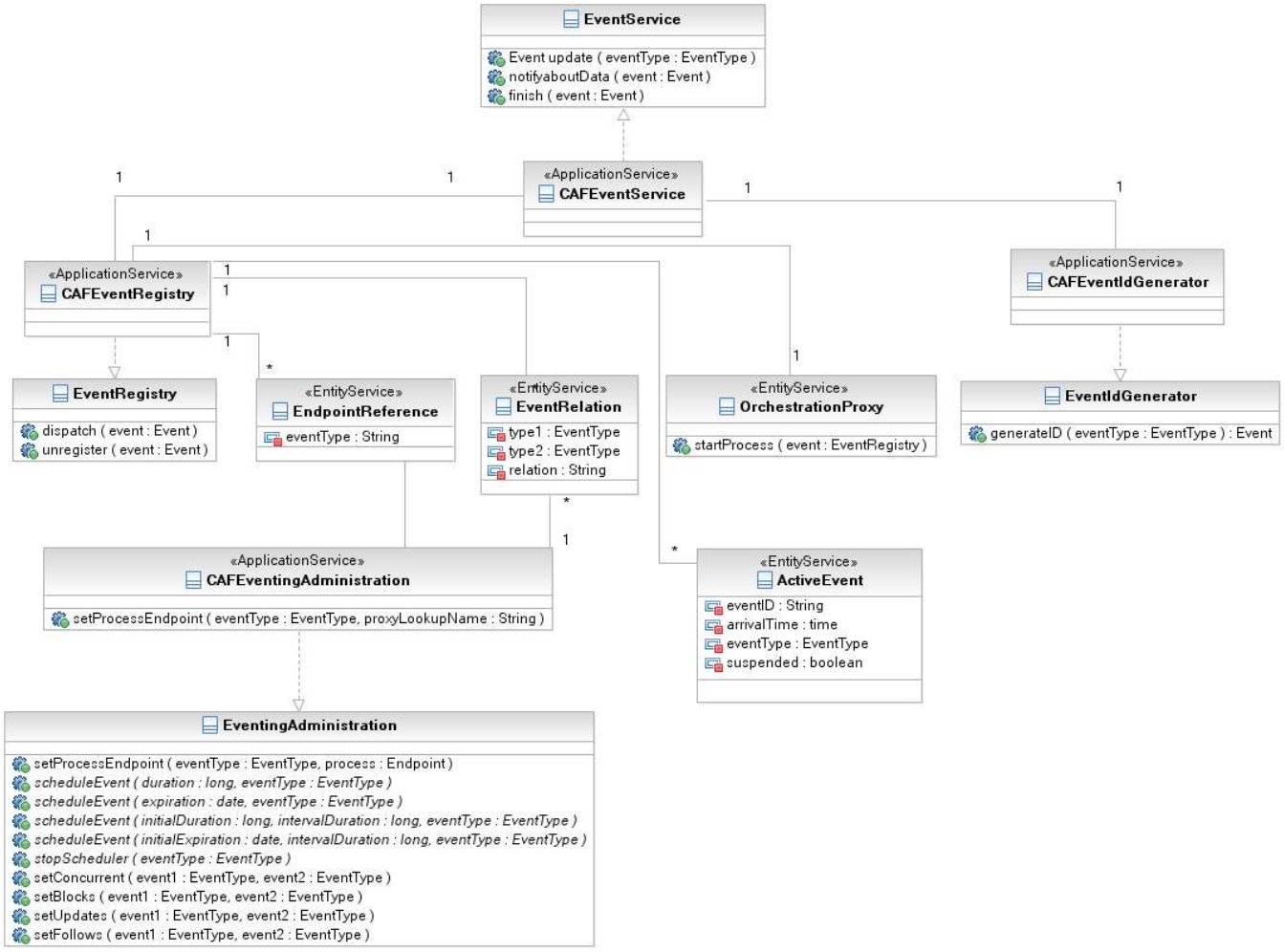


Figure 39: Structure of the Eventing System within the CAF

by SAP XI in order to allow for reliable messaging. As the service orchestration's workflow engine is also realized using SAP XI (cf. section 7.2.6), the dispatching of events is also realized using guaranteed delivery.
As the CAF does not support dynamic call-back references for asynchronous communication, the integration flows (that are also realized using SAP XI – cf. 7.2.4) have to be used if this service interaction pattern is required.

7.2.2 Data Repository

When realizing the data repository by using the CAF in a SAP-centric environment, two major constraints apply. First, for governance reasons there is no possibility to deploy third-party software on the platform. As a consequence, the data repository can not be realized using the JavaSpaces [105] framework that realizes a tuple space for Java. The second constraint is that the CAF only supports transactions within a WAS. Only *Application Services* and *Entity Services* with local persistence profit from the transaction model that is established in the J2EE container. When using *Entity Services*, only container-managed transactions can be used.
Due to some known deficiencies of the CAF persistence layer (that primarily concern per-

sisting complex objects), it is preferable to not implement a generic smart proxy as it is described in section 5.6. Fully typed access should be realized instead. This is why it is not possible to provide a generic implementation for the data repository. However, the basic concepts can be described.
Basically it is required to create *Entity Services* for every data item that is part of the CDM. These *Entity Services* will have two additional attributes: a surrogate key and an event. This event indicates the "owner" of the respective data. The event and the primary key of the CDM element form a secondary key for the *Entity Services*. An additional *Entity Service* is required in order to implement the tuple-space semantics. The `TakenElement` *Entity Service* contains two attributes (that also form the key): an event ID and a foreign key reference to a *Entity Service* that forms a business object. This *Entity Services* is used for tracking the events that are checked-out. If data is written using the event that was used to *take* the data, the respective instance of the `TakenElement` service is deleted. If another event is used for writing taken data, an exception is raised.
Figure 40 exemplifies a data repository with two data objects (`Customer` and `Order`) and an existential dependency between these objects. This examples aims to demonstrate how more complex queries to the data repository can be realized.

Figure 40: Structure of the Data Repository within the CAF

These data objects are realized using *Entity Services*. The dependent object (the `Order`) exposes a finder-operation.

These *Entity Services* are wrapped by an *Application Service*. The `CAFDataRepositoryCustomerOrder` service is exposed both as an internal reference as well as web service[46] and represents a smart proxy. It offers the required get, read and write operations. As the WAS manages transactions, the respective operations that require `Transaction` objects as parameters are not included.
The necessity of realizing a dedicated smart proxy for each scenario arises because of software logistic issues that prohibit augmenting one single *Application Service*. The underlying *Entity Services* must, however, be shared among all smart proxies and data repositories.

In order to use the `CAFDataRepositoryCustomerOrder` with internal references, data transfer objects (cf. [151, p. 401]) are necessary. In the example, these transfer objects are realized with the classes `CustomerTO` and `OrderTO`. The actual business attributes are omitted in order to simplify the diagram.

Because a data repository can only be implemented specifically for certain data types, the administration of the data repository can neither be implemented completely. The `checkForData`-operation is required, though. As the dependencies are static, the `CAFDataRepositoryAdministration` directly looks-up a `Customer` *Entity Service*. If such an entity exists, the callback to the `notifyaboutData`-operation of the `CAFEventService` is invoked. If no entity can be found, an `OpenNotification` is created. This *Entity Service* is checked during every invocation of the `CAFDataRepositoryCustomerOrder`'s `writeCustomer`-operation. If an order is stored for an event the eventing system was not notified, the `notifyaboutData`-operation of the `CAFEventService` is invoked and the respective `OpenNotification` entity is removed.
The `invalidate`-operation of the `DataRepositoryAdministration` can not be implemented using the CAF. This is because invalidating data would require the rollback of all ongoing transactions for a given type of event. If a user-managed transaction could be used, the `CAFDataRepositoryCustomerOrder` (and all other data repositories) could maintain a relationship among events and transactions. As the container-managed transactional model that is used by the CAF prohibits access to the actual `Transaction` object, there is no way of managing concurrent transactions and rolling them back in case of invalidated data. This has the consequence that the *updates* relation can not be realized using the CAF. This is why the `EventRelation` *Entity Service* solely indicates how event relations could be managed in a data repository. Since the `CAFDataRepositoryAdministration` *Application Service* does not implement the `setUpdates`-operation, this entity is not required.

When an event is computed, the eventing system is informed. In turn, the `eventIsFinished`-operation of the `CAFDataRepositoryAdministration` *Application Service* is invoked. This operation checks for taken elements first. If a data object is registered to be taken for an object, an Exception is raised. If no data is checked out, the data objects that are registered for a given object are deleted. Additionally, it is determined whether `OpenNotifications` exist in the database. If there are any and they are outdated (days is the appropriate unit to measure outdated events), the respective entries are removed in order to keep the database clean. If required, special error handling procedures can be realized for outdated events that were never computed.

[46] Due to CAF-restrictions, there is no possibility of including CAF web services into managed transactions.

As the data repository and the eventing system can both be realized using the CAF, the reliable registration and de-registration of events can be assured by using the built-in transaction management of the WAS. No other component of the NetWeaver stack is required.

7.2.3 Connectivity to Application Systems

The common protocol of a composite application realized using SAP NetWeaver is "web service". More precisely, in the context of NetWeaver this means SOAP over HTTP as communication protocol and XML as data representation. This is the common denominator all platforms of the NetWeaver stack are capable of. Whenever an application system offers functionality that allows it to interact using these standards, no additional components are required in terms of connectivity. If application systems do not support this common way of interaction, several components exist in several products of the stack that can be used to address this issue. The EP, CAF and XI provide means for connecting external application systems. The flexibility of the offered mechanisms vary. For instance, the CAF offers mechanisms to connect to relational databases or to invoke SAP Remote Function Calls (RFC). However, there is neither a flexible mechanism to integrate arbitrary application systems nor does the connectivity mechanism of the CAF offer functionality that would allow application systems to trigger a composite application using an arbitrary protocol.
The EP offers a JCA-compliant adapter framework. However, the flexibility that is offered by portal-based adapters is only accessible by user-centric portal interactions. There is also no way for application systems to trigger composite applications as the version of the JCA framework that is used (JCA version 1.1) does not offer a channel from an application to the adapter.
The adapter framework of the XI is the only suitable component that can be used for realizing the connectivity layer of a composite application. In its core, it offers a JCA 1.1 compliant adapter framework in addition to adding proprietary functionality that enables application systems to pro-actively communicate with the adapters. The adapter framework (AF) of the XI can only be used as part of the XI platform. Consequently, the ESB functionality of XI must also be used if deploying the AF.

The AF offers a pluggable architecture that allows for realizing custom adapters. As part of the XI platform, adapters for the following connectivity options are provided: SOAP over HTTP(S) (which is also realized using an adapter), simple mail transfer protocol [152] (SMTP); access to arbitrary relational database management systems, access to flat-files; file transfer via the file-transfer protocol [153] (FTP); arbitrary payload over HTTP(S) ("plain HTTP adapter"); SAP RFCs and SAP Intermediate DoCuments (IDOC). Internally, the XI also relies on SOAP over HTTP. As such, all components of XI require the transmitted payload to be represented in well-formed XML. Hence, the XI adapters need to provide functionality for addressing heterogeneity of the data representation.
As arbitrary adapters might be required for an actual scenario, it is not possible to assess the suitability of the adapters independently of this scenario.
The following list describes, on an adapter-independent level, how the functionality, that the connectivity layer of a composite application has to provide, can be realized with the XI in general and how the natively included adapters address the issues.

- **Sending from Application System to Composite Applications - Service Interaction Pattern 1** of [98].
  The XI AF provides a proprietary framework around the standards-based JCA 1.1 adapter framework that can be used in order to realize inbound communication from application systems into the connectivity layer of a composite application. Additionally, the WAS the XI instance is deployed on offers functionality that allows external application systems to send requests.
  However, there is no generic straightforward solution to inbound communication with XI. If the respective protocol is not supported by the WAS, the only solution is to establish a gateway (cf. [95, pp. 468ff.]) outside the AF. In such a scenario the AF actively connects to the gateway. Also the application systems that will be connected need to be configured to connect to this gateway. Transitively, a connection from the application system to the adapter framework is established. If a application system subsequently sends request to the gateway, the gateway forwards the requests to the AF. This way, the send pattern can be realized in a generic way. The RFC adapter that is shipped with XI uses this paradigm. This method can also be adapted for arbitrary application systems and protocols.
  A simpler but less generic approach is to use the agents that are deployed on the two stacks of the WAS that is used by XI. Such agents open listener-ports for certain protocols and can forward requests to the AF. This approach is limited to HTTP and tRFC-based IDOC communication, though. The SOAP over HTTP, plain HTTP and IDOC adapter that are shipped with XI utilize this approach.
  If application systems are required to be monitored for state transitions, XI adapters can also poll the application systems. The event generation mechanism is not documented and therefore not applicable for arbitrary adapters. The database adapter, which is shipped with XI, uses this mechanism. It polls relational databases and extracts tables as XML messages. The detection of state transitions is application specific and needs to be realized according to the actual requirements.
  Also the plain-file, FTP and SMTP adapters use a polling mechanism in order to receive data from application systems.

- **Send/Receive - Receive/Send - Service Interaction Pattern 3** of [98].
  The XI AF is capable of both synchronous and asynchronous communication. These communication semantics are realized by combining the mechanism for sending and receiving (described above) as required. Synchronous requests (from both an application system and from a composite application) can include calls to the DET if it is realized by the XI. In these scenarios, the DET can also handle correlation-identifiers.
  The AF of XI supports neither dynamic endpoints nor addressing. Consequently, it is not possible to relay requests.

- **Transactional Support** XI and its AF neither internally nor externally support distributed transactions. ACID transactions are supported by the RFC adapter that can invoke transactional RFCs (tRFC) and the database for relational databases that can group both a read and a write command into one ACID transaction.
  These transactions can not, however, be spanned over multiple resources or requests. These types of transactions are also only local transactions as they can only satisfy the ACID properties within a single adapter (that exclusively connects to one back-end system).

- **Conversion of Data Representation** XI adapters can only be used if the data is also routed through XI. The pipe of the IS of XI can only compute data that is represented in XML.[47] This is why XI relies on adapters that forward XML messages to the IS.
  The actual conversion of the data representation is handled by adapter modules that are realized as stateless session beans that are deployed on the Java-stack of the WAS of XI. They are generic and can be reused for any Java-based adapter. They usually require scenario-specific configuration, though.

- **Dynamic Addressing** The AF of XI does not support addressing. Only fixed endpoints can be configured at design time. Using special attributes that are proprietary for each adapter, the outbound communication towards application systems can be realized in a dynamic way. This approach does not allow for relayed requests, though.

7.2.4 Data Exchange and Data Transformation Layer

The DET provides ESB functionality to a composite application. The ESB solution of NetWeaver is the XI. Additionally, the adapters of the XI AF require the use of the XI as integration solution. This is why the DET can not be realized without using the XI if NetWeaver is the actual target platform.
The XI provides the functionality of a reliable messaging system in addition to its AF. The flow through the messaging system is described as a pipe-and-filter architecture. The flow, that is outlined in section 7.1.2, puts basic integration functionality in a fixed and not configurable sequence. In order to realize the more flexible integration flows of the reference architecture for composite applications, it is required to analyze the filters, the flow and the extension possibilities that are provided by the XI. As an objective, only standard configuration means of the XI should be used. This is because modifications of the actual platform are likely to be discarded by a governance organization that manages the operations of a platform such as the XI.

Integration Services The basic functionality of the DET is exposed by the single integration services. Hence, if the integration flows must be described, an approach for realizing the integration services is a prerequisite.
The pipe of the XI includes filters for *Routing* and *Mapping* (cf. section 7.1.2). A pipe is invoked by an external application or by an XI *business process* and sends data to an external application or a business process. A *business process* is a service orchestration that is described in WS-BPEL. In addition, XI *Mappings* can be explicitly included as a step of a business process. Such mapping steps are realized as an extension to the standard WS-BPEL language and are independent of the computation within an XI pipe. Also *Routing* can be included as a step into a business process. Receivers are then treated as values of container elements of the respective process instance.
Both, the *Routing* and *Mapping* can be used as integration services by the notion of the reference architecture:

[47] There are some optimization parameters that allow IDOCs to be tunneled in a binary format. Tunneled IDOCs can neither be transformed nor dynamically routed.

- **Routing Service** A `Routing Service` is the equivalent to the *routing* mechanism of the XI. XI supports *routing* by introducing two components: the *receiver determination* and the *interface determination*. A *receiver determination* routes a message of the XI pipe to an arbitrary amount of receivers. A receiver by the notion of the XI is a *logical system*. A *logical system* is the XI representation of an agent. Based on the payload of the actual message, a set of such receivers can be dynamically determined. However, querying a service registry at runtime for looking up service endpoints is not supported.
  If multiple receivers are determined (by a static rule or dynamically), the respective message is duplicated for each receiver. Based on the actual service consumer (described by its agent and the requesting service interface) and the determined agent of the service provider, the actual service interface of the service provider is determined by an *interface determination*. Both mechanisms together describe a `Routing Service` that is capable of implementing the *Recipient List* pattern (cf. [95, 249]) and the *Content-Based Routing* pattern (cf. [95, 230]). Due to the multiplication of messages based on the amount of receivers, the workflow pattern *Multiple Instances Without Synchronization* can also be realized.
  The XI also supports acknowledgments. Acknowledgments are realized as independent messages that are transparently generated by an adapter. However, acknowledgments are not routed as independent messages. Based on the trace list of the initial message, they are routed back to the initial requester. No mechanism exists to aggregate acknowledgments to overall acknowledgments for all receivers. As a consequence, a `Routing Service` that supports the workflow pattern *Multiple Instances With a Priori Runtime Knowledge* can not be realized with the XI.
  When used within a *business process*, the receiver and interface determinations do solely return a list of receivers. If an *invoke*-activity of a *business process* is used together with a receiver determination from within the process, a message is created and sent to the actual receiver. If multiple receivers are determined, a *block*-activity is required to surround the *invoke*-activity in order to instantiate multiple messages.

- **Heterogeneity Service** A `Heterogeneity Service` is the integration service that handles the data translation from the application-specific data format to the canonical data format that is used internally by the respective composite application. The XI comprises two components that realize a `Heterogeneity Service`: a so-called *message mapping* and an *interface mapping*.
  A *message mapping* is a program that transforms at least one XML message to at least one XML message. It is possible to use several target messages that are represented in several data formats as the source for a message mapping. Also, the target messages can have different formats. Several technologies exist for realizing *message mappings*. The standard approach is a proprietary mapping environment that executes message mappings designed using the XI message mapping editor. Alternatively, XSL Transformation (XSLT) (cf. [154]) style sheets, ABAP programs or Java programs can be used. *Message mappings* that are realized in ABAP or Java have to be programmed against a proprietary application programming interface (API).
  This API does not only describe the interface for an actual mapping. Additionally, a set of classes is provided. These classes can be used from within a *message mapping* in order to access routing information or to query external data providers. Querying external data providers allows for lookups that can be used for enriching message mappings with payload that is dynamically determined either by the actual message

or the context it is used in. Queries are treated as ordinary messages that are routed through the IS of the XI.
Interface mappings group several *message mappings*. Only a sequential ordering of *message mappings* can be realized. *Interface mappings* are assigned at runtime to a certain relation and are executed as a step in the pipe of the XI IE. Alternatively, interface mappings can be used as steps within a *business process*. In such a scenario, input as well as output messages are container elements of the actual *business process*.
The concepts of *message mappings* and *interface mappings* allow for the realization of a `Heterogeneity Service` with the XI. The only constraint of the XI mapping environment is that it relies on XML messages. As a result, if XI is used as DET platform, the transformation of the data representation has to be handled by the connectivity layer of the respective composite application.

These two integration services can be directly realized by the built-in functionality of the XI. The remaining integration services that are required by the integration flows can also be realized. However, they are only partially supported natively by the XI. By combining several native components, it is possible to realize them without modifying the XI platform.

- **Data Service** Realizing a `Data Service` with the XI of NetWeaver is possible by combining the functionality of the AF, the IE and the business process engine. By doing so, the line between the connectivity layer and the DET is blurred. As the AF can not be deployed independently of the IE, this is not considered a drawback of the solution.
  Messages that are computed by the IE must have been computed by an adapter beforehand. The AF handles both, the reception of calls as well as polling for changes in back-end systems.[48] This is why the `Fetch Data` as well as the `Retrieve Data` activities are realized by configuring adapters appropriately.
  If the actual adapter supports it, principal propagation of the requester can be used in order to authenticate against an application system. Alternatively, fixed principals can be configured for a certain *channel*[49].
  The AF can also handle different communication semantics independently from the IE by configuring an actual *channel*. Thus, synchronous as well as asynchronous communication is possible. If required, an *exactly-once in order semantics* of asynchronous calls can also be configured. The configuration applies both, for the AF as well as for further computation within the IE. The only functionality in terms of communication semantics that can not be realized by the AF is asynchronous communication with correlation. If correlation for fetching data is a must, a WS-BPEL process must be used as an aggregator in order to combine the request and the response.
  Such a process is required either way if a `Data Service` must be actively called in order to perform a lookup. Such a scenario requires to use the complete IS for establishing an integration relation with the system the lookup should be performed against. This relation must contain an *interface mapping* that transforms the request into an appropriate lookup. An exemplification of how a `Fetch Data` activity

[48] Both options are not available for all types of adapters.
[49] An XI channel is an adapter that is configured for the interaction with an application system. It can be considered to be an *instance* of an adapter.

that receives a message asynchronously, uses an *interface mapping* for the generation of a lookup and sends the correlated response to (a possible different) service is depicted in figure 41.
The response for the lookup is to be considered as the result of the `FetchData` activity that is forwarded (as messages that were produced by the AF independently) to the IE. When asynchronous lookups are used, the reply can be deferred to another service consumer rather than the initial requester.

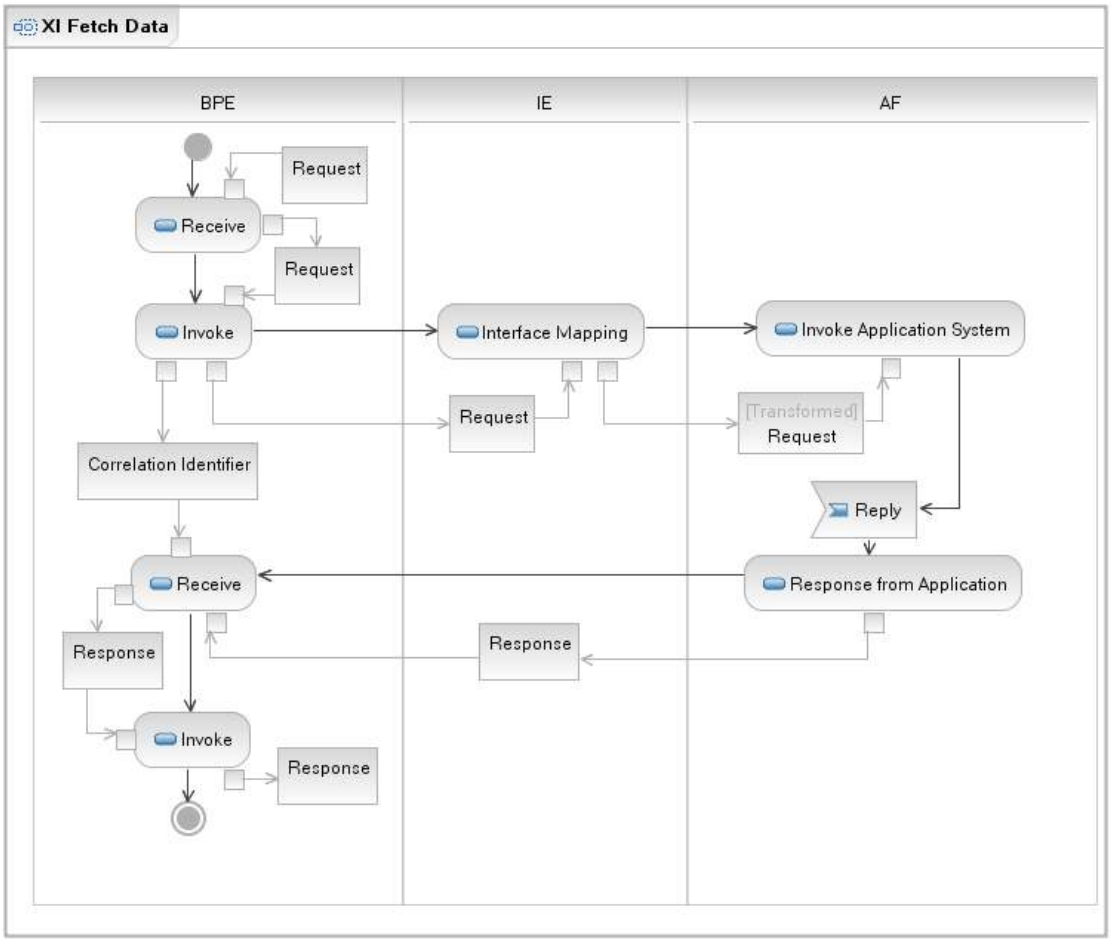


Figure 41: Example of a `Fetch Data` Activity with Correlation and Lookup in XI

Depending on the actual requirements, data that was received or fetched from an application system might require further manipulation prior to forwarding it to an integration flow.
If a message needs to be spilt into several messages (as described by the *message splitter* integration pattern of [95]), a "multi-mapping" can be used within an *interface mapping*. A multi-mapping can create several target messages based on one source message. Each of the created messages is processed independently by the IE. If a multi-mapping is used as a step of a *business process*, a WS-BPEL *loop*-activity is required in order to compute all messages that are produced by the mapping.
If a set of (logical) messages is aggregated into one (envelope) message, a simple mapping can realize a resequencer. This might be possible if the position of a logical message within an envelope message can be changed.
If single messages need to be re-sequenced, the mechanism for realizing this functionality is similar to the mechanism that is used for realizing aggregators (see below).

For both scenarios a WS-BPEL process is required. Additionally, a resequencer requires a correlation identifier in order to assign related messages to the same instance of the business process. Beside the correlation that is encapsulated either by a loop or a time-out block, a mapping and a loop with an enclosed *invoke*-activity is required. The multi-mapping is required for re-ordering the set of correlated messages into a multi-line container of the process. The final block is required as a loop for (sequentially) sending the single messages.
How a resequencer can be realized by using the XI BPE is depicted in figure 42.

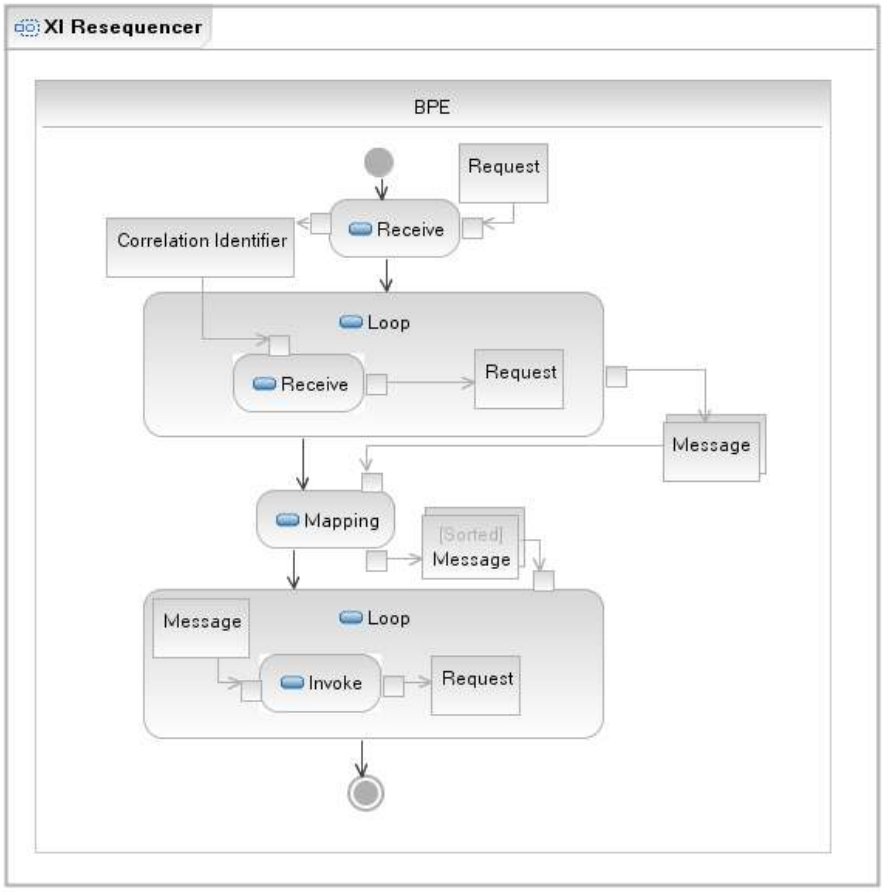


Figure 42: Example of a Resequencer Activity with Correlation in XI

A WS-BPEL-based process with multi-line containers is also required if an aggregator (as described by the *message aggregator* integration pattern of [95]) should be realized with the XI. The asynchronous variation of an aggregator additionally requires a correlation-identifier, too. The only difference with the resequencer which is described above, is the actual realization of the multi-mapping. In contrast to the mapping that is used as part of the resequencer process, the mapping for the aggregator produces one single message that is in-turn sent.
As an alternative, a message aggregator can also actively poll for messages that it aggregates. This polling can both be realized synchronously and asynchronously. In order to realize a synchronous polling, one *invoke*-activity and no correlation-identifier is required as part of the first loop. An asynchronous aggregator that actively polls for messages and correlates the replies is depicted in figure 43.

Triggering an error handling procedure can essentially be realized through two mechanisms of the XI. The first mechanism is to realize such a procedure as a service

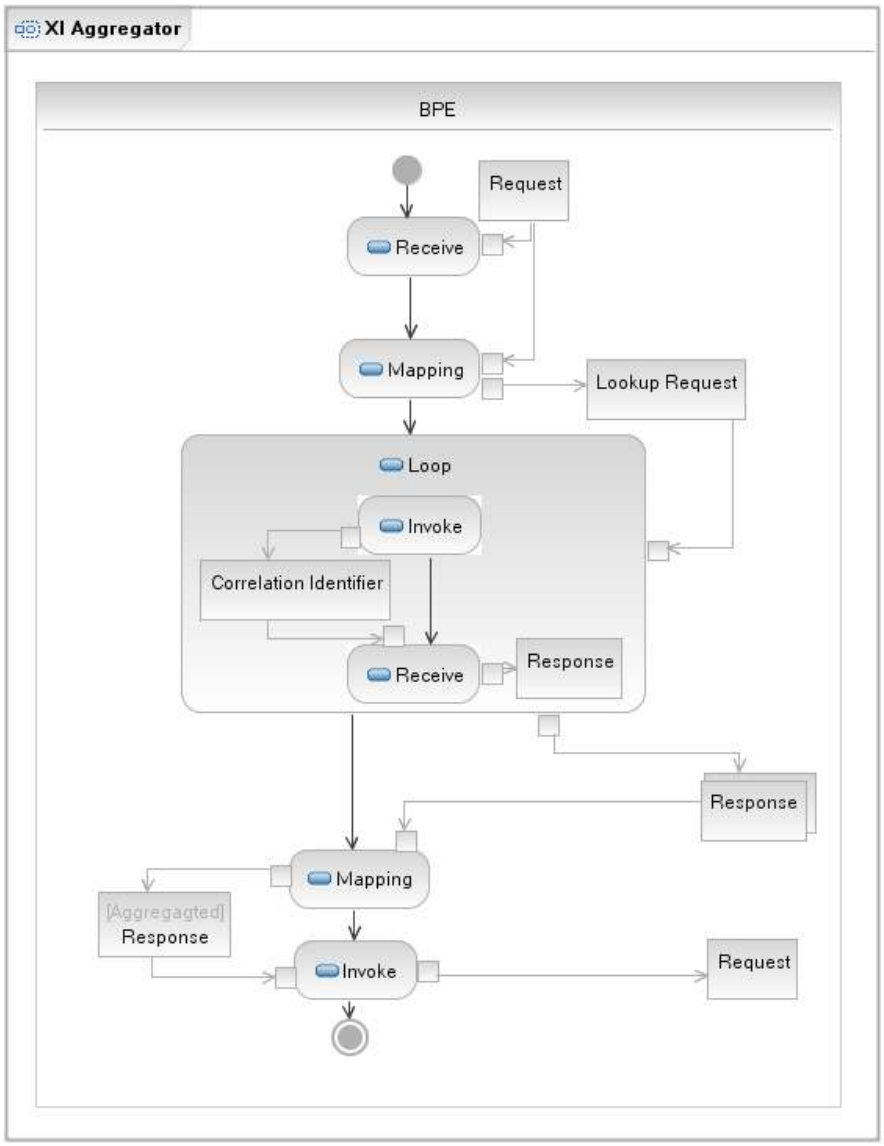


Figure 43: An Active Aggregator with Correlation Realized as a Process for the XI

provider and trigger it by using an *invoke*-activity from within the WS-BPEL *compensation handler*. Such an error handling procedure can involve automated and manual tasks.
A proprietary solution is to trigger a so-called *alert*. Alerts are notifications that are assigned to human administrators of an XI platform in order to allow a manual intervention.

If a `Data Service` is realized by a *business process*, a *Store Data* activity is simple realized as an *invoke*-activity of the respective process. A channel to the actual application system has to be configured accordingly. If no process is necessary, a *Store Data* activity is simply realized by configuring a channel with its appropriate *receiver determination* and *interface determination*.

- **Trigger Service** The `Trigger Service` can be realized by combining the pipe of the IE with the process engine of the XI.
  The initial filter activity of the `Trigger Service` (cf. section 5.7.4) is realized on the XI platform as a routing step of the IE pipe. The same mechanism as for the

realization of the `Heterogeneity Service` apply. In order to discard messages, the content-based router has to return an empty set of receivers. The appropriate XI *receiver determination* needs to be configured in a way to stop the computation of messages without a receiver.
By using the lookup API of message mappings, stateful filters might be realized, too. In this case, a mapping is used as router. By returning routing information that is represented in a special format, realizing stateful routing becomes possible.
In the case of a message not being required to be filtered out, the *receiver determination* needs to determine a *business process* that provides the additional functionality of a `Trigger Service`. This WS-BPEL-based *business process* uses a *message mapping* in order to generate a message that contains the required `EventType`. This transformation can be dependent on the sending system and interface as well as on the payload that was received as routing meta-data is accessible via the mapping API of the XI.
Depending on the actual scenario, a WS-BPEL process that is used as part of a `Trigger Service` must contain between two and four containers: one for the message that is received and one for the `EventType`-message that is generated by the first *message mapping*. The *business process* has as a mandatory next step a *invoke*-activity that sends the message that contains an `EventType` to the eventing system. Usually, this will be realized by a synchronous service invocation. The XI has to be configured in a way that the sent message is routed through the IE to the `Event Service`.
According to the actual requirements the WS-BPEL process might contain two additional steps and two additional containers. The first possible step is a *message mapping* that transforms the received data into a format that can be used by the data repository. The last possible step is an *invoke*-activity that transmits the payload and the `Event` from the eventing system to the data repository.
In contrast to the generic description in section 5.7.4, it is necessary to always use the transformation and send activities when conducting this integration service with XI. This is because XI can only treat whole messages that contain all parameters of a request. Hence, the second *message mapping* has to merge the payload that was received by an application system and the `Event` from the `Event Service` to one message that is sent in-turn to the data repository.

- **Validity Service** At runtime, *message mappings* might fail if a source message does not comply with a defined data structure. This mechanism is neither applicable at any point of the pipe nor is it possible to verify messages without executing a mapping. No dedicated mechanism for verifying messages is foreseen.
  One approach to realize a `Validity Service` with NetWeaver is to realize an external service provider with the CAF that is deployed on a WAS. Such a service provider will be invoked by an *invoke*-activity of an XI *business process* as a regular application system. The interface of such a `Validity Service` is simple. It contains one `validate` operation that receives a message and a schema name. The `CAFValidityChecker` needs a reference to a local *entity service* for looking up a schema by its unique name. By using an XML parser (e.g., the built-in of the WAS Java-stack), the message can be validated. For a valid message the `validate` operation returns `true`.

  In order to use such a `CAFValidityService`, a XI *business process* is required. First, the actual message needs to be transformed into a call to the `validate` operation

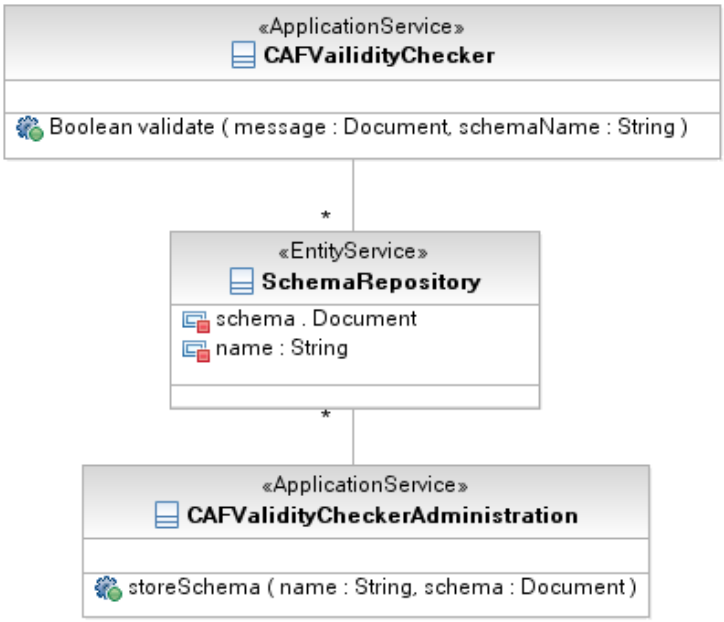


Figure 44: Structure of a `Validity Service` that is Realized Using the CAF

using a `message mapping`. The subsequent step of the *business process* is a synchronous *invoke*-activity. The third step is a branch that is based on the return value and triggers further computation accordingly. These steps are realized as part of the surrounding IIF that is realized using a WS-BPEL *business process*.

Integration Flows The `Routing Service` and `Heterogeneity Service` can be realized by standard elements of the XI IE. Realizing a `Data Service` might require a realization with the XI BPE. If realized by a WS-BPEL process, a `Data Service` can be called like any arbitrary service provider. The `Trigger Service` and the `Validity Service` are always exposed as services.
This impacts the integration flows, such that they need to be (partially) realized as WS-BPEL processes if an integration service that is exposed as a service is required. If all required integration services can be realized with pre-defined parts of the IE, the IE pipe could be used as integration flow. Using the pipe would have positive impact on performance of the integration flows. A drawback of this approach would be that the maintainability decreases as the implementation of the integration flows would not be standardized. Hence, the explicit formulation of integration flows as WS-BPEL process is preferable over the implicit solution by using the IE pipe. If special performance requirements need to be realized, a "pipe-only" solution should be considered, though.

According to [155], WS-BPEL can be used to describe the workflow patterns *Sequence*, *Exclusive Choice*, *Simple Merge* and *Multiple Instances Without Synchronization*. This is why XI and its BPE are suitable for realizing the basic workflow functionality for the integration flows (cf. sections 5.7.6 and 5.7.7) .
Due to the importance of adapters for the XI platform and their necessity for realizing `Data Services`, integration in flows should always be realized on the XI as WS-BPEL processes that are called by a `Data Service`. In contrast to the orchestration that is described in section 5.7.6 as part of the generic reference architecture, an XI-based IIF

should not invoke a `Data Service` but be invoked by one. An advantage of this approach is that this way the *Multiple Instances Without Synchronization* workflow pattern (cf. [108]) can be seamlessly realized.
All integration services are then part of the *business process*. They are either called by an *invoke* activity or included by proprietary means. The latter concept is required for *interface mappings* that realize `Heterogeneity Services`.
An important aspect of the integration flows are the different possibilities to close and acknowledge requests. Acknowledging requests is possible in an XI-based WS-BPEL processes in two ways: first, an *invoke* activity can be used to send a message to a service consumer in order to notify it about the reception of data. Additionally, the XI includes an internal acknowledgment mechanism that can also be triggered like a *reply* activity.
Closing synchronous requests can be realized in an XI process by the notion of *reply* activities. Such replies refer to *receive* activities. The process platform of XI uses the notion of these activities for closing synchronous calls. When placed accordingly, these mechanisms can be used to realize arbitrary boundaries for service calls.
Compensation blocks of the WS-BPEL process can be used to handle arbitrary errors. Both raising *alerts* and invoking arbitrary service providers is possible.
With the difference of the initiator of a `Data Service` an IIF can be realized with the XI as demanded by the reference architecture by simply creating a WS-BPEL process as described in section 5.7.6.

Realizing an IOF as a WS-BPEL process is also possible. As an IOF is always called by a composite application and the actual request contains the payload that needs to be transformed, the notion of a (SOAP) adapter calling an IOF process fulfills the requirements for realizing an IOF.
A `Routing Service` and a *determine IIF* activity might be required for realizing an IOF. Both can be realized as *receiver determinations* that are included as steps of the *business process*. All other steps of the IOF can be realized as described in the discussion about IIFs.

As the XI platform supports reliable messaging with *exactly-once in order* semantics, reliable integration flows can be used to connect composite applications with application systems. As the XI does not allow for distributed ACID transactions, especially no distributed transactions that use a two-phase commit protocol, no transactional security can be established in NetWeaver-based composite applications. This is a major deficit of the platform.
The workflow pattern *Multiple Instances With a Priori Runtime Knowledge*, that is required for aggregating acknowledgments of messages that are multiplied by a *message splitter*, is not supported natively by the WS-BPEL-based process engine of XI (cf. [155]). Hence, the service interaction pattern *one-to-many send* (cf. [96]) that includes a *notification of delivery* needs a *message aggregator* as part of an IIF that collects acknowledgments. In contrast with a platform that natively supports this interaction mode, on the XI platform an IIF is required to overcome the deficit of not supporting aggregated acknowledgments. The IIF, that is required on-top of the `Data Service` of the IOF, aggregates acknowledgments and forwards a positive or a negative acknowledgment to the composite application. As a consequence, proprietary (implicit) XI acknowledgments can not be used in such an interaction scenario.

7.2.5 Service Coordination Layer

In order to integrate transactional handling as well as eased and high-performance integration of the data repository, the service coordination layer should be realized with the CAF.
Realizing the service coordination layer with the XI is also an option. As the XI version that was available to the project does not support transactions, it was decided to realize coordination services programmatically as CAF services.

The Java language of the CAF can be used to realize sequential processes with choices and merges. Additionally, the CAF supports distributed transactions via a J2EE transaction manager. Asynchronous messaging via correlation can be programatically supported. However, the CAF is intended to be used in synchronous scenarios. External web services can be included by the notion of *Entity Services* that represent proxies. The endpoints for the generated proxies can not be dynamically looked up. However, those endpoints can be administered via the WAS on which they are deployed at configuration time.
As the data repository is also realized as CAF services, local references can be used in order to access the data repository in a high-performance and transactional way.

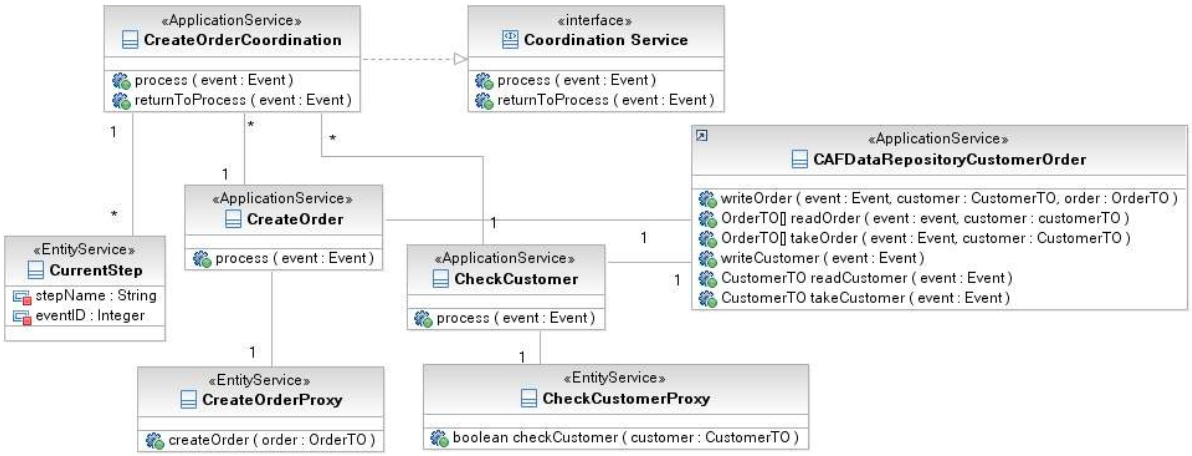


Figure 45: Structure of a Coordination Service within the CAF

The service coordination layer sits in between the service orchestration layer and the DET. When realizing composite applications with the CAF, the interface of a coordination service contains only an `Event`. Identified by its endpoint, a coordination service does not require additional data from an orchestration layer other than the event of its invocation. This way, coordination services and particularly service orchestrations can be implemented in a very simple way. Additionally, the amount of transmitted data is reduced. As a consequence, the messaging approach between the layers is replaced by a *spaces* approach (cf. [94]).
The `process`-operation of a coordination service is invoked by a service orchestration. Coordination services in-turn operate on application services. If asynchronous communication is required, application services (mediated or not) perform callbacks on the `returnToProcess`-operation of a coordination service. All coordination services in the CAF realize the `Coordination Service` interface.
Asynchronous callbacks are realized with a fixed endpoint to the respective service coordination and with a *correlation identifier*. This identifier is realized by the actual `Event`.

In order to allow for error recovery, a coordination services stores "open" correlations into an *entity service* called `CurrentStep`. An entity of that type is a persistent key/value-pair and stores an identifier of the step a coordination service needs to perform when its `returnToProcess`-operation is invoked. The actual identifier is specific to the coordination service that uses it. The `returnToProcess` operation also requires the notion of an `Event`. As a consequence, a service that performs the callback can not be agnostic to the use of `Events`. Additionally, the CAF neither allows for stateful services nor for dynamic callback-endpoints. This is why asynchronous communication with arbitrary service providers always requires integration flows that use a `Heterogeneity Service` to remove and add the actual `Event` from the communication. This increases the complexity of asynchronous applications. Additionally, a `Trigger Service` is required, that stores the returned data into the data repository, prior to returning to the coordination service (exemplified as step 1.1.2.2.2.5.4 in figure 49). Due to that lack of functionality in the CAF, when using the CAF, synchronous interaction with application systems should be considered seriously.
Also specific to the actual use case is the set of external service providers a coordination service aggregates. In the example of figure 45 the example from section 7.2.2 involves orders and customers. In the example, the business process requires the creation of an order for a customer. Not expressed by the business process is that the validity of a customer needs to be checked[50] before an order is created. Checking a customer's validity and creating an order is functionality that is offered by two different service providers that each expose one operation. The remote services are made accessible to the CAF by creating two *Entity Services* – `CreateOrderProxy` and `CheckCustomerProxy` – that are used by two `Application Services` for the sake of including the application services (as exposed by application systems) into the coordination service. These `Application Services` combine the data repository's smart proxy for the scenario as well as the generated service proxies. As a coordination service is always being called from a service orchestration, these CAF `Application Services` (that represent the application services as defined by the reference architecture) always have access to the actual `Event` and can therefore access the data repository to retrieve and store data. Thanks to the `Trigger Service`, necessary data can be stored into the data repository before a coordination service is invoked.
The actual communication with the application systems is realized by the `Application Services` that retrieve the necessary data from the data repository and invoke the respective proxy. A sequence diagram that describes the initial computation of an event by all layers of a composite application is described in figure 49 in the next section.
In contrast to the platform-independent reference architecture, the service coordination layer is not optional for composite applications that are realized with NetWeaver. This is because the integration of the data repository can be achieved in the most facile way by using the data repository from a CAF *Application Service*. In order to keep the control-flow description of the service orchestration simple, the orchestration merely dispatches `Events`.

7.2.6 Business Process Orchestration Layer

NetWeaver offers three process engines that can be used to orchestrate services. The first two are user-centric process builders. These are the guided procedures of the portal and

[50] The actual logic that makes up a *valid* customer is not important at this point.

the so-called visual composer that can generate (besides others) WebDynpro applications. Both engines require user interaction. This is because service providers are exposed to a user that provides input parameters and checks return values from services. This is why the third option – the WS-BPEL engine of the XI – is the only process engine that can be considered a central control instance of a composite application.

Business processes can be transformed into workflow specifications (cf. [126]). In order to execute a workflow specification on the XI, this workflow specification (more precisely, the control flow) has to be expressed in WS-BPEL. A WS-BPEL process is triggered by an *EventRegistry* of the eventing system. This component invokes the `startProcess(Event)`-operation of the process. In the XI this is realized by creating a *receive* activity that references the `startProcess` activity. The *receive* activity additionally needs to have the attribute *createInstance* set to "yes".
Functions (in EPCs) or transitions (in Petri nets) are considered as a pair of *invoke* and *receive* activities of WS-BPEL. An *invoke* activity always invokes a `process`-operation of a coordination service. The routing from the process engine to the appropriate service coordination is based on the type of *business process* and on the `Event` that is transmitted. Specifically for the XI, an `Event` requires an attribute that indicates the actual step of the computation in a process-specific way. In order to modify an `Event` so that it can be used to identify a service coordination, an interface mapping is embedded in between an *invoke* and a *receive* activity. One possibility is to increase a numeric step identifier. This way the same mapping can be used for every step. Of course, branches also need to be handled. An example of an `Event` and the respective `EventType` that can be used in the NetWeaver platform is shown in figure 46.

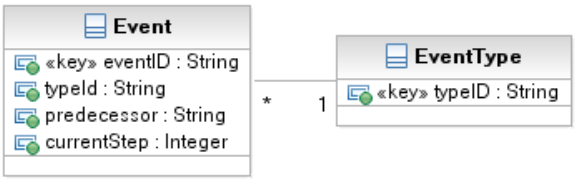


Figure 46: `Event` and `EventType` for NetWeaver

The data and resource view of a process description are used to realize these components. Once a coordination service has been computed, it calls the XI workflow engine by passing an `Event` that is used as a *correlation identifier*. The operation of the *business process* that is invoked is the `returnToProcess` operation.

In order to realize an *exclusive choice* and a *multi-choice*, a (synchronous) *invoke* operation is required as part of the *business process*. Through this activity, the `decide` operation of a `Decision Service` is invoked. The return-value is assigned to a *container* of the WS-BPEL process. Subsequently, a *switch* activity can be used to branch according to the value of this *container* to realize an *exclusive choice*. In order to realize a *multi-choice*, the *container* that contains the return value must be evaluated as a *transitionCondition* of a *link* of the respective *flow* construct (cf. [155]).
A `Decision Service` is realized as an *Application Service* that implements the actual decision logic.[51] The `CAFDecisionService` uses the passed `Event` for querying the data repository according to the passed identifier of the required rule. The `Action` that is

[51] So far, how to implement a business rule engine with CAF has not been investigated. This is because, for the investigated use cases, plain implementations of the decision logic were sufficient. Additionally,

returned, is realized as simple `String`. Once returned to the process, it is evaluated by a WS-BPEL *switch* activity as the subsequent activity of the *invoke* activity. The structure of the (simple) `CAFDecisionService` is described by the class diagram of figure 47.

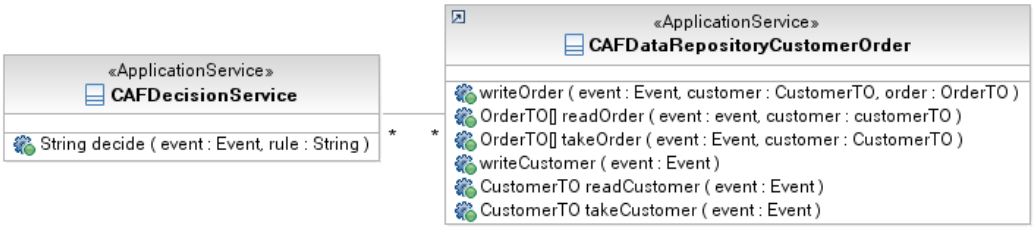


Figure 47: Structure of a `Decision Service` for the CAF

7.2.7 Service Registry

The NetWeaver stack neither supports dynamic lookup of service endpoints at design-time nor does it (in the version described here) include a central service registry. For every product of the stack, the endpoints of the actual services can be specified at design-time. The CAF and the EP additionally allow for (manual) reconfiguration of endpoints after the deployment of the service consumers.
XI keeps a design-time repository that contains all syntactic service descriptions as well as endpoint configurations. Hence, if the service providers that are not mediated by XI are also kept in this repository, XI can be considered as a design-time catalogue of all available service providers.
Future versions of XI (that will be called SAP Process Integration (PI)) are likely to extend the existent functionalities towards a run-time service registry.

7.2.8 Centralizing the User Interface

The reference architecture for composite applications relies on application systems that expose the functionality that is integrated by a composite application. This functionality might or might not involve users. If a composite application must be accessible through a central user interface, the user interface should be realized as service provider.

When using NetWeaver, there is a possibility to optimize the creation of a central user interface. Additionally, there are some deficiencies that create the need for a dedicated mechanism for including a user interface.
The technology for creating (web-based) user interfaces with NetWeaver is WebDynpro. Both ABAP and Java can be used as programming languages. Java is the language of choice for creating user interfaces for composites. This is because it is possible to access CAF *Application Service* natively by using a local protocol of the WAS. This has the advantage of reducing communication and increasing the performance of the composite application.
However, WebDynpro applications can not be exposed as services. There is no mechanism

SAP has announced plans to release a business rules framework that could later be integrated into the platform specific reference architecture for NetWeaver (cf. [156]).

in WebDynpro that allows for creating a service provider in WebDynpro. This is why the SAP EP is required. More specific, the *guided procedures* from the CAF are required to be deployed on an EP.
WebDynpro applications can be used as *callable objects* within guided procedures. This is possible if the respective WebDynpro *component* implements the interface `com.sap.caf.eu.gp.co.webdynpro.IGPWebDynproCO` (cf. [157]). This interface is part of the standard CAF API. It includes two operations: `getDescription` and `execute`. The first operation is used during the design and deployment phase to retrieve meta-data about an application. An application is started by invoking the `execute` operation. Parameters are passed as part of the context that is described by the class `IGPExecutionContext`.
By creating a guided procedure with one action that references the respective (WebDynpro) callable object, WebDynpro applications can be exposed as service providers. This is because the GP API provides a web service called `GPProcessDiscovery` that allows starting guided procedures via a web service interface (cf. [158]).
Using this mechanism, guided procedures (and therefore the WebDynpro applications) are started by invoking the `startProcess` operation. Among others, payload data can be passed as parameters to the WebDynpro in a custom structure (which is autonomously generated based on the meta-data that is accessible via the `getDescription` operation of a callable object/guided procedure). Even if the service interface can not be designed independently, proprietary data can be passed to such a service. As the internal structure of the guided procedure runtime is not documented, the class diagram of figure 48 solely shows the components that are open for implementation. The (complex) structure of the runtime environment is represented by the class `GPRuntime`.

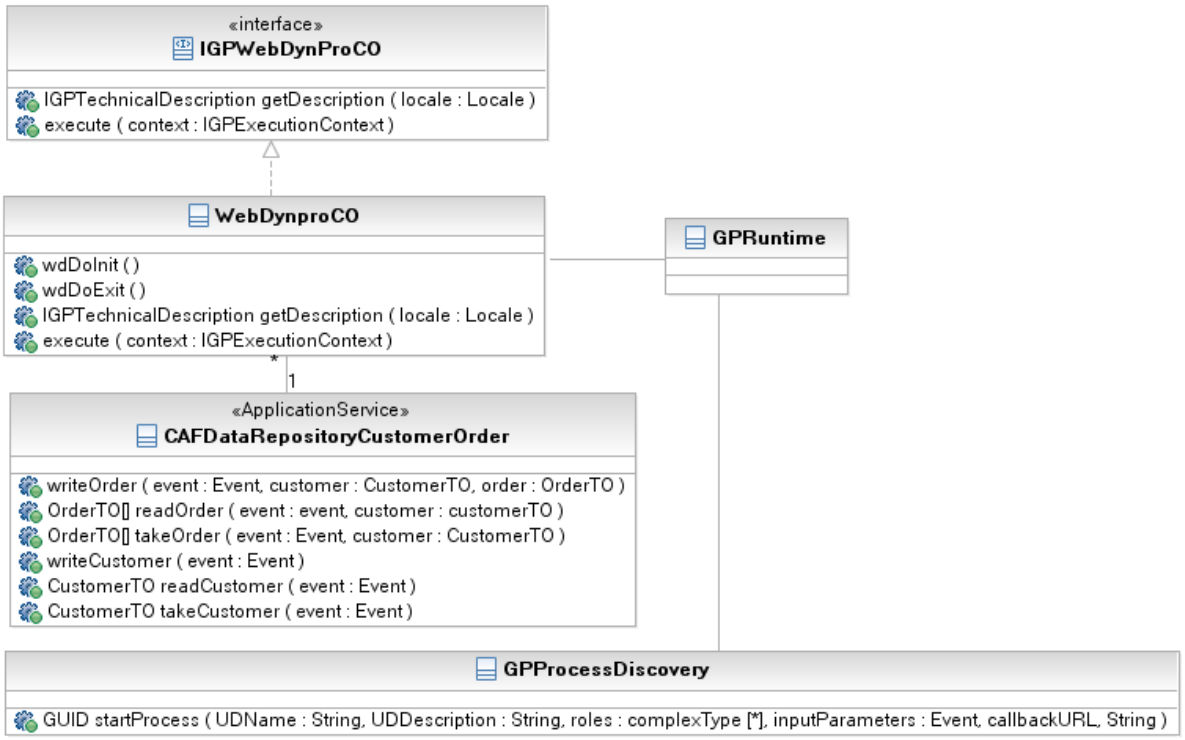


Figure 48: Structure of a WebDynpro Component that can be Invoked Using Web Services

However, in order to accelerate both the development as well as the runtime performance, it is not recommended to pass parameters. WebDynpro applications implement the model-view controller pattern. Models can either be realized as CAF *Application Services* or as

external web services. By using the data repository locally as CAF services, access to the data is accelerated. Additionally, the transaction management for the data repository can be extended to the WebDynpro application. A third advantage is that data that is changed by a WebDynpro application does not need to be checked out during the complete user interaction. A WebDynpro application can `take` data as required.
In order to allow access to the data repository, an according `Event` must be passed to the `startProcess` operation of the `GPProcessDiscovery` service. The `Event` that is passed as part of the context to the WebDynpro's `execute` operation at runtime must be used.

7.3 Summary

The sequence diagram in figure 49 exemplifies the interaction of the CAF components. While the internal communication within the eventing system and the data repository are omitted for simplification reasons, the diagram shows how an agnostic application system (*application system 1*) uses an XI-based `TriggerService` to trigger a composite application (steps 1.1 to 1.1.1.2).
The process orchestration is started by the invocation of the `process` operation. Inturn, the process orchestration asynchronously invokes the respective coordination service (`CreateOrderCoordination`).

The shown example includes one coordination service that uses one application service (`CheckCustomer`) with one operation `checkCustomer`. In order to invoke the application system, the `CreateOrderCoordination` first persists its conversational state. This state indicates that it has been invoked and is now at step one (transition 1.1.2.2.1). Subsequently the coordination service invokes the CAF *Application Service* that encapsulates the actual proxy (transition 1.1.2.2.2.1). In turn, data is fetched from the data repository. As the *validity* attribute of the customer object is to be modified, the data is *taken* and not *read* from the data repository.

After the data has been taken, the actual proxy is instantiated and invoked (1.1.2.2.2.4 and 1.1.2.2.2.5). By using an IOF that is configured as the actual endpoint for the proxy, an IIF is called. Its `Data Service` finally invokes the application system (1.1.2.2.2.5.3.1.1). In order to exemplify asynchronous communication in CAF, the application system asynchronously returns the validity of the passed customer to the IIF. The IIF uses a `Trigger Service` without event generation to modify the customer according to the return value (by using a `Heterogeneity Service`) and *writes* the customer object back to the data repository. As the `Event` is passed to the IIF by the IOF, the data can be written back for the data repository and made accessible by the composite application. Subsequently, the IIF has to return the control to the actual coordination service. This is realized by invoking the fixed callback operation `returnToProcess` of the coordination service (transition 1.1.2.2.2.5.4).
In the simplified example the coordination service looks up the current state for the received `Event` and decides to return the control to the service orchestration (1.1.2.2.2.7).

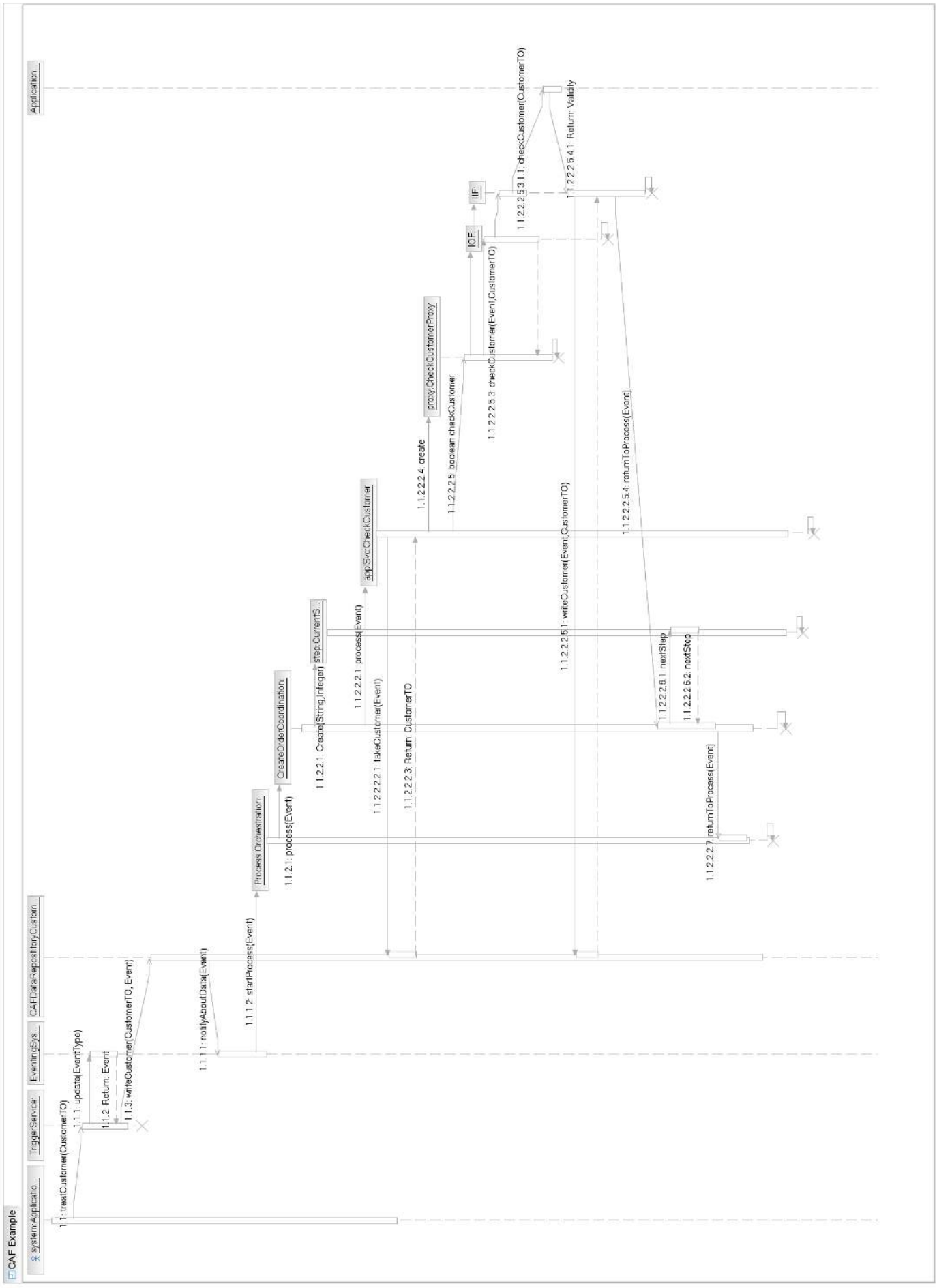


Figure 49: Example of the Interaction of all CAF Components

7.4 Conclusion

SAP NetWeaver is a platform that can be used to implement composite applications. By mapping the reference architecture for composite applications to this platform, the efficient development of composite application seems possible. However, this mapping is not complete. Some features that are demanded by the reference architecture can not be realized with this platform. This is especially the case for distributed transactions that are only supported locally within the CAF. Also asynchronous messaging is only supported in some modules. In order to realize asynchronous communication throughout composite applications, several mechanisms have to be used that increase the complexity of the solution.

Applying the platform classification of section 2.4, the presented reference architecture for SAP NetWeaver allows for dynamic changes of the structure of composite applications. This is thanks to the centralized control that is expressed as a process orchestration. Even if not supported by a service registry, dynamic replacement of services can be realized if the integration flows in XI are configured accordingly. Additionally, the integration flows within the XI realize a failure tolerant communication backbone. Following the classification of [40], the presented architecture would be classified as (D, R, FB, XX). In contrast to the plain usage of the application server that is classified as (D, N, FN, SN)[52] (cf. [40]), the presented architecture utilizes the whole NetWeaver stack in order to realize a platform that is suitable for realizing composite applications.

The fact that asynchronous communication is not natively supported in all components of NetWeaver decreases the modifiability of a composite application. As increasing modifiability is a major objective of the service-oriented architectural style, the SAP platform prohibits the complete leveraging of service-oriented principles.
To fully exploit the modifiability advantages of the service-oriented architectural style, synchronous application can be used more frequently. However, this decreases the reliability of the overall system. In conjunction with the lack of distributed ACID transactions, SAP NetWeaver does not seem to be capable of supporting industry-proof composite applications. However, building smaller applications with minor requirements for reliability and availability is possible. Thanks to the high degree of integration (especially in the CAF and the WebDynpro frameworks), fast implementation and good runtime performance can be expected when building composite applications on NetWeaver. Based on this discussion, SAP NetWeaver is a good platform for quickly implementing small applications that are likely to change often and do not require a high degree of reliability.

[52] Development environments were not investigated. Since there is no integrated development environment, [40] evaluates NetWeaver to *SN*. Due to the non-integrated development environment, we do not evaluate this criterion ("*XX*").

8 A Case Study

The case study that is presented in this chapter demonstrates the applicability of the concepts of the reference architecture as well as the design methodology for composite applications (cf. chapter 5 and chapter 6). A solution description as well as a description of the "look and feel" of the composite application that was based on the design demonstrate the applicability of the platform-specific architecture for SAP NetWeaver (cf. chapter 7). This further shows that the concepts of this thesis can be used to realize solutions for real-life problems.
The observations made during the project are described before the chapter is concluded.

8.1 The Business Case

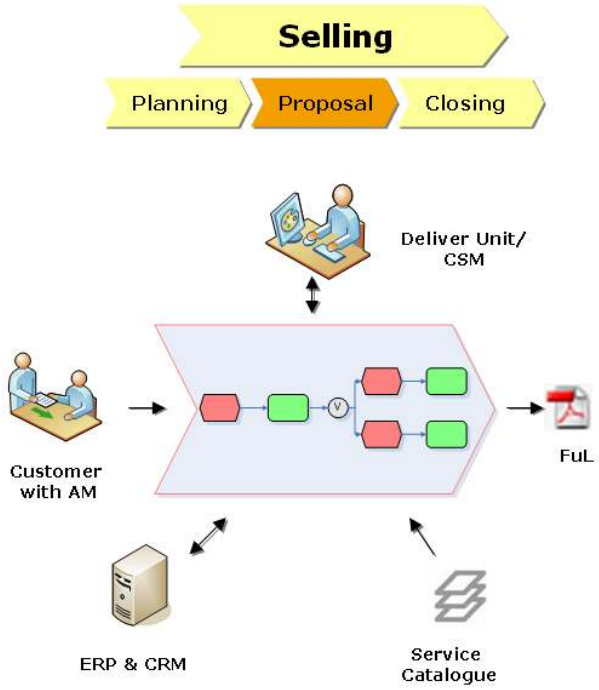


Figure 50: Functional Sketch of the FuL Creation Process

The use case of the case study was chosen by a dedicated working group within the project of BASF IT Services. Lacking a structured methodology, the group analyzed, on a business expert level, several business processes and how steps included in these processes might be used in other processes as well. The advantage of service-oriented architectures that was best understood by the business analysts was reusability across system boarders. The business process ultimately chosen was the so-called "agreement management" business process. This decision was based on the business requirements and the real-life setting.
This process describes the procedure of how the company reacts to customer demands by estimating the efforts the realization of a request might cause and then providing an offer to the customer. In the overall process there is an extract that is concerned with creating an offer for a given demand. The so-called "Funktions-und-Leistungsbeschreibung" (FuL), which can be loosely translated as "service description", is a document that is used to

describe a solution offered in response to a demand. An FuL accompanies an offer and delineates the offered services, contains (among others) a functional description of the solution, and indicates the service-level of the offered services. This description must be aligned with the service portfolio the company offers. Namely, only services that are officially included in the company's portfolio can be proposed. An FuL is the basis for a cost calculation for the proposed solution. Based on this estimated cost, prices are established and an offer, that includes an FuL as well as the calculated prices, is sent to the customer. FuLs are created by different organizational entities of the company. While the need for the creation of an FuL is indicated by the account management for a certain customer, the creation of the FuL is a task that is performed by delivery units or so-called client-service managers (CSM). The functional description of the FuL creation process is outlined in figure 50.

8.1.1 Requirements

The use case was chosen by a working group of process experts. This same working group defined the functional requirements for an FuL creation prototype. The deliverables handed over to the working group concerned with the design and implementation of the composite application were: a process model represented as an EPC and a UML class diagram that describes the data-perspective of the process.

The business process used as a functional requirements specification for the composite application includes the control and data flow, as well as the organizational/system perspective of the process. The process describes a collaborative sequence where different experts from single delivery units work out the single sections of the FuL in addition to cost estimations.
The actual structure of an FuL is dependent on the "FuL template". An FuL template specifies, besides the language of the document, the necessary sections that determine the contract details. Examples of such sections include the service delineation, service-level agreements and so forth. Based on the template that was chosen by the initiator of the FuL creation process, the FuL is written. This complex step is represented in the process model as a single function (*Describe Service*). After the actual FuL is created, it needs to be approved. Based on the decision, it needs either to be reworked or the cost calculation needs to be initiated based on the information contained in an FuL.

In the subsequent phase, account management (AM) uses this information to generate an offer that might lead to a contract with the customer. The extract of the agreement management process that underlies the case study is shown in figure 51. The complete process model can be found in figure 74 of appendix C.

The data perspective was described as a separate UML class diagram. It describes all entities that are concerned by the agreement management process. For the FuL creation process, especially important are the entities `Service Description`, `FuL` and `Calculation`.
The data model that was created by the process expert working group is depicted in figure 52. The single entities of the data model are described more in detail, providing a description of the respective steps of the design that require information about data.

The aim of the case study is to automate this process using the existing system landscape

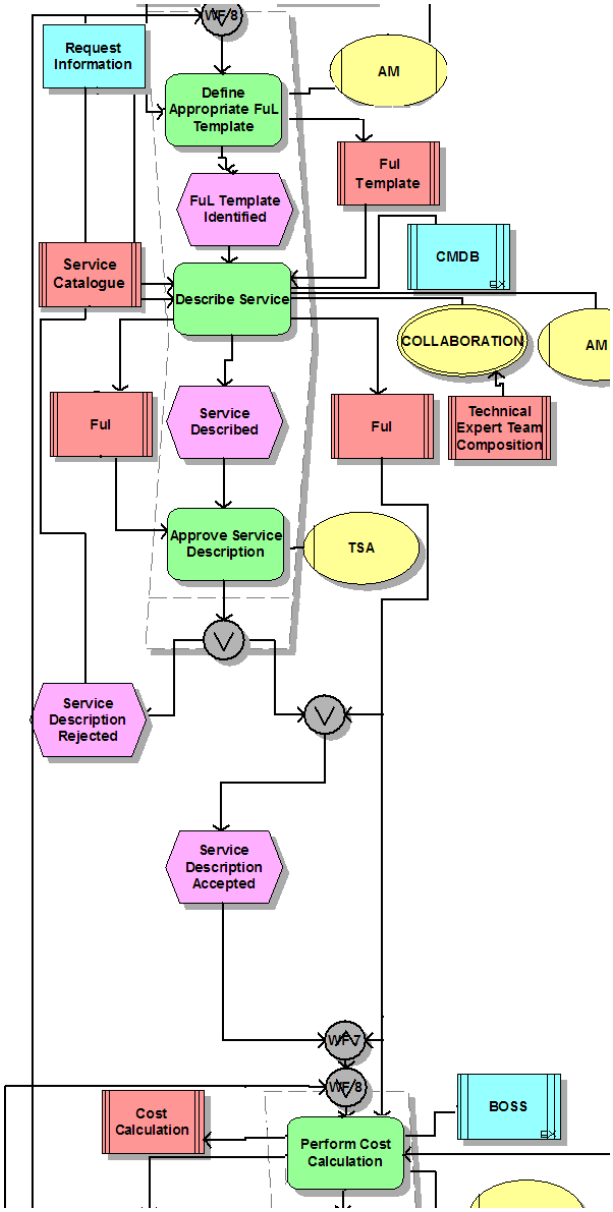


Figure 51: EPC Process Diagram as Part of the Functional Requirements

by building a composite application. In doing so, the FuL creation should be structured into a process and aligned with the company's service portfolio. The composite application should also unify the user access to the systems. There were no additional requirements provided.

8.1.2 On the Suitability of SOA for the Use Case

The use case for the case study was chosen independently of technical considerations. The process experts understood the service-oriented architectural style as a paradigm for remote computing with a web frontend. Reusability was considered to be the benefit compared with other remote-computing paradigms. This group concluded that the FuL creation process is the most suitable of all processes that require a re-engineering of its supporting application(s) at the time of the decision. Unfortunately, alternative use cases

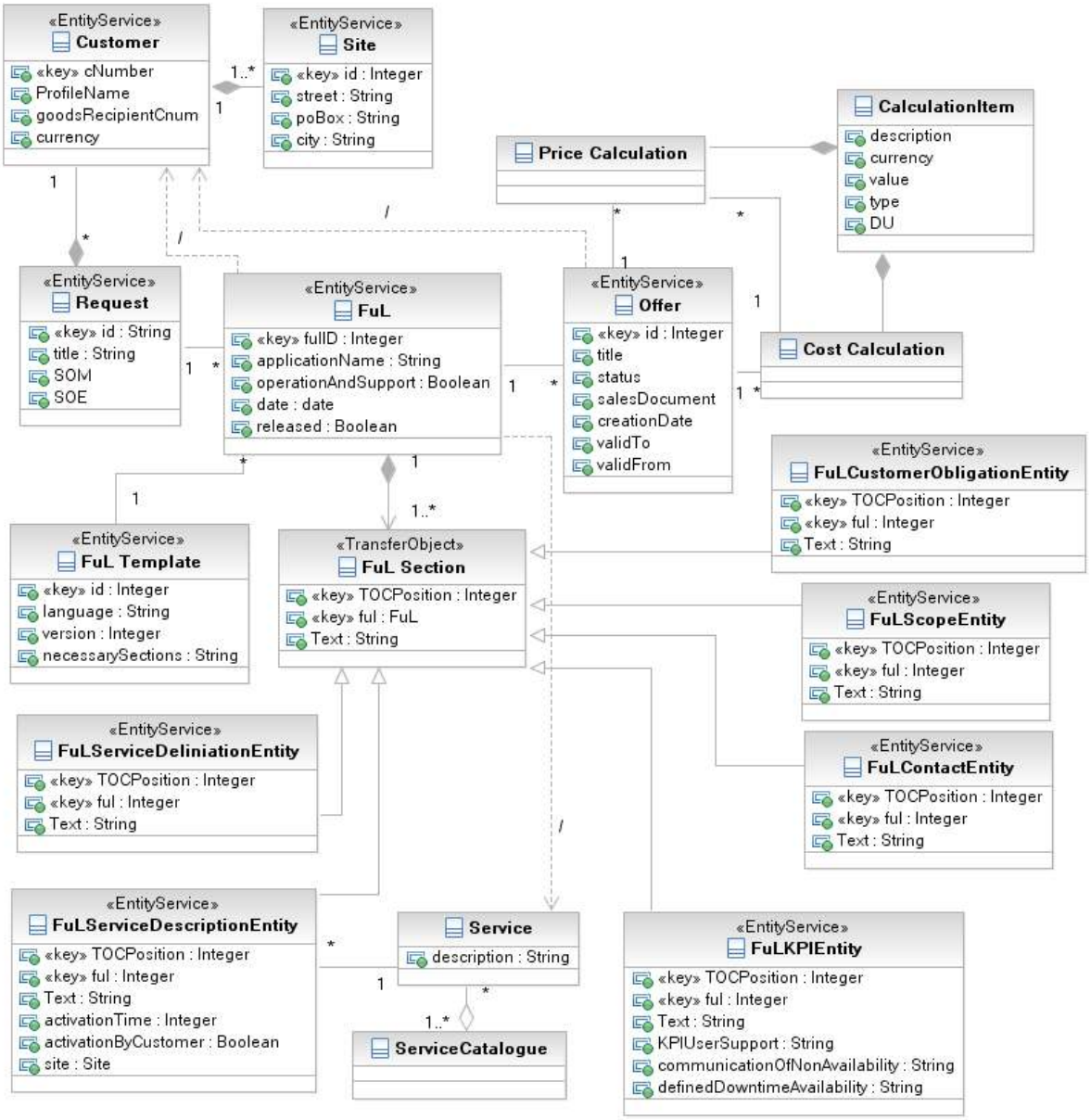


Figure 52: Data Model of the Case Study

were neither documented nor communicated.

In order to verify the appraisal of the process expert community, it was necessary to roughly document the suitability of the service-oriented architectural style for realizing an automation support for the FuL creation process. This verification was constrained by the available budget as well as by the need to use it as a communication means to achieve an agreement for a project budget. This is why the assessment had to be kept simple. The evaluation criteria, that are discussed in section 3.3, were used for the verification and were demonstrated to be suitable for the given objective.
The evaluation of the single suitability criteria are summarized by the following list.

- **Frequent Changes in Processes** The FuL Process itself can be considered quite stable. This, however, is only true for the control flow of the process. Data, as well as the resources that are used for the execution of the process, are changed on occasion.

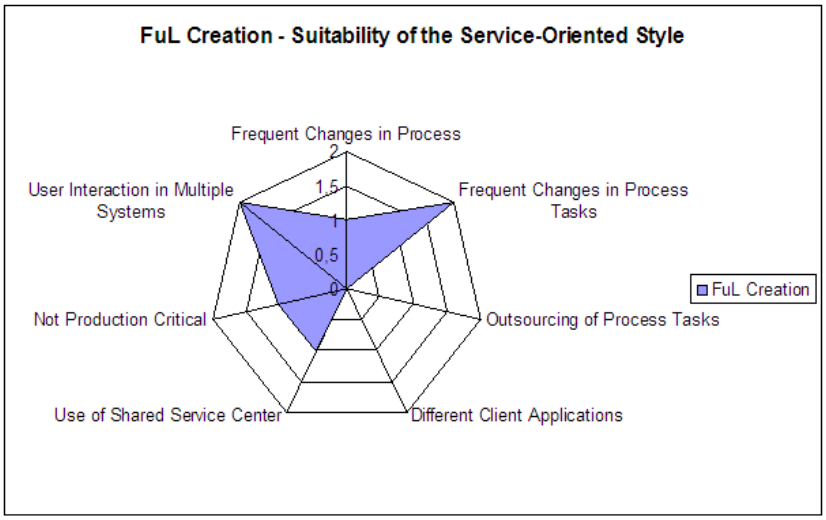


Figure 53: Initial Assessment of the Suitability of SO for the Use Case

Data structures are subject to change especially since the FuL template is reworked from time to time. Resources could be changed as the application landscape might be renewed.

- **Frequent Changes of Process Tasks** The actual way a service is described is expected to be subject to frequent changes in future. This is because the task was performed manually at the time of the project definition. Changing a task from a manual task to a structured, semi-automated task is considered likely to happen. This is expected due to anticipated improvement suggestions from key users.
- **Outsourcing of Process Tasks** All tasks of the FuL creation process are conducted in-house. Also no outsourcing was planned.
- **Different Client Applications** No need was identified to allow for multiple client applications that utilize the FuL creation process. No need was identified for supporting mobile applications.
- **Use of a Shared Service Center** The creation of the FuL is de-centrally organized within the organization. The single delivery units are responsible for the creation of FuLs that concern their business. If multiple delivery units are concerned, an FuL is created collectively. This is why it was considered beneficial to establish a central application the de-central units could use.
- **Production Control** The creation of the FuL does not control the production of the services. It is solely used during an early phase of the selling process. This is why short periods of unavailability are not likely to cause serious harm. Also, data inconsistencies would only cause additional effort. A general unavailability during a long period would, however, seriously harm future business initiatives of the company.
- **Process Requires Human Interaction in Multiple Back-end Applications** Before the project, creating FuLs was realized as a manual process that was supported not only by word processor applications but also by application systems. Several application systems store data that is relevant to the creation of FuLs. They

must be accessed by FuL creators frequently. Hiding the actual complexity of the landscape was considered a major benefit of the composite application. Earlier proposals for automating the FuL were discarded because the integration of the single application systems appeared to be too complex.

Even if the use case assessment showed mediocre suitability results, there were no obstacles for applying the service-oriented style. As the use case was a real-life issue that could not be previously solved by traditional means with a reasonable budget[53], it was decided that an attempt should be made to realize a composite application that supports the FuL creation process. The key argument was to unify the user interface while reusing the functionality of the back-end systems. A constraint for the project was to not re-realize functionality that was already existent.
Further, the way of assessing suitability proved to be beneficial. It provided a quick estimation that was suitable for communicating the objectives of the service-oriented style. The summary of the discussion that was used for the communication is depicted in figure 53.

8.1.3 Application Landscape and Constraints

Before the case study was conducted, the actual FuL creation was a manual task that was mainly based on telephone, word processors and email. The process experts that defined the business process also determined what application systems would be used to manage the creation of FuLs.
So-called "sales objectives" were managed in a CRM application called "SAMS". The management of such sales objectives involved keeping an inventory of requests that were made by customers. Cost calculations, offers and contracts were managed in sales and distribution (SD) as well as project management (PM) modules of an SAP R/3 system called "BOSS".
Additionally, there was a change-management database (CMDB) that was used in the company's different ITIL ("Information Technology Infrastructure Library") [159] processes. This database stored service-level agreements, service descriptions and assets that are required for delivering the services. The process experts roughly estimated that this application could be used to retrieve lists of services the company offers.
The application landscape additionally contained an SAP Exchange Infrastructure and an SAP Enterprise Portal installation on separate hardware. As discussed in chapter 7, the objective was to assess the suitability of the SAP NetWeaver platform rather than choosing a best-of-breed platform. This is why these systems were chosen to be the platform for the composite application that was to be created.

The CRM application was realized as a set of IBM Lotus Domino version 6 (cf. [160]) databases and applications. Domino applications, such as the CRM system, provide user access through so-called views that are created for the single databases. Views can be accessed in various ways. As a company policy, views are only accessible via the Lotus Notes client.
The CRM system was initially purchased as COTS. Due to several requirements, it was modified heavily. These modifications represented a major investment that was to be

[53] An alternative that was discussed before was to integrate the process into the SD module of the ERP system by custom development.

protected. This is why the CRM application should not be replaced by the composite application.

The ERP system was an SAP R/3 4.6c. Among others, the SD, MM and PM modules were installed on the application server. As the central ERP system of the company, the R/3 system was a COTS that was not allowed to be modified. Due to the complex processes in the ERP system it was also an objective to keep the ERP system integrated into the FuL creation process.

The CMDB was also initially purchased as COTS and modified over time due to business requirements. It is a C++ (cf. [161]) client/server application with fat clients. The logic is mainly realized within the relational database. The data schemes grew complex over time and support the requirements of various use cases. The main purpose of the CMDB is the IT asset management. This means that all IT assets of all customers of the company are stored in the database. Additionally, for every asset a corresponding service-level agreement is stored into the database. Various monitoring applications access the database. A service-level agreement is represented in the CMDB data scheme as a set of multiple so-called *key performance indicators* (KPI). The monitoring applications access the threshold values of the respective KPIs in order to create statistics that are in-turn used for billing purposes.
An objective of the case study was to utilize the implicit notion of service-level agreements by KPIs in order to use the CMDB as a service catalogue. As it turned out during the design of the composite application, the data quality was not sufficiently high. This is why there was no possibility of using the CMDB as a service catalogue.

8.2 Design of the Composite Application

The business process (and the data model) was used as the input for the design methodology for composite applications that is described in chapter 6. The deliverable of this phase was the platform-independent design that is aligned with the reference architecture of chapter 5.

8.2.1 Step 1: List all Business Process Activities

There are four process activities that make up the core of an FuL creation. They can be easily derived from the process model. These are:

- ***Define Appropriate FuL Template***
  Input : information about the customer request an FuL should be created for.
  Output : the appropriate FuL Template that can be used to answer to a customer's request.
  Functional Description Based on a given request, a user should define the language of the FuL that should be created. If new FuL templates exist, the user should be able to choose among the available templates.

- ***Describe Service***
  Input : information about the customer request an FuL should be created for, the

FuL template that was chosen for that specific request and a list of services the company offers as part of its service catalogue.
Output : an FuL that provides an answer to a customer's request.
Functional Description Based on a given request and a template, a user should be identified that creates an FuL for the given request. The description of the single elements of an FuL should be structured and not leave a freedom to the creator to introduce terms and conditions or services that are not offered by the company.

- ***Approve Service Description***
  Input : the FuL that was created.
  Output : an FuL that indicates its validity.
  Functional Description Only based on an FuL, an identified user decides whether an FuL is compliant with company rules and political circumstances. It is not checked in this step, whether an FuL meets a specific demand.

- ***Perform Cost Calculation***
  Input : an approved FuL.
  Output : the estimated cost for the realization of the solution as it was described by an FuL.
  Functional Description The costs for the realization of the solution, as it was described by an FuL, are estimated in this step. The actual estimation is a subjective task. Its result needs to be stored into an application system.
  This functionality can already be supported by the ERP system of the company. This functionality should be used as part of the solution.

8.2.2 Step 2: Create Enterprise Service Candidates

As the case study was the first service-oriented project under the control of the company there was neither a service registry available nor existent services accessible. This is why four new enterprise service candidates were necessary:

- $es_1 =$ `({Request}, {FuLTemplate})`
- $es_2 =$ `({Request, FuLTemplate, ServiceCatalogue}, {FuL})`
- $es_3 =$ `({FuL}, {FuL})`
- $es_4 =$ `({FuL}, {CostCalculation})`

8.2.3 Step 3: Match Suitable Service Methods and Derive Missing Service Method Candidates

Due to the lack of existent services, the algorithm that is used during this step could not possibly match any services.
The single steps of the algorithm are described in the following:

1. **Define Entity Service Candidates** The entity candidates that could be identified were (according to the classification in section 6.2.4):

- svc_1 = `(∅, {FuLTemplate})` (result for es_1; satisfies ϵ_3 (cf. (46) on page 145)
- svc_2 = `({ServiceCatalogue}, {Service})` (result for es_2; satisfies ϵ_2 (cf. (45))
- svc_3 = `({Request, FuLTemplate, Service}, {FuL})` (result for es_2; satisfies ϵ_1 (cf. (44))
- svc_4 = `({FuL}, {Offer})` (result for es_4; satisfies ϵ_2 (cf. (45))
- svc_5 = `({Offer}, {CostCalculation})` (result for es_4; satisfies ϵ_2 (cf. (45))

2. **Define Task Service Candidates** The task candidates that were identified were:
   - svc_6 = `({FuL}, {FuL}))` (result for es_3; satisfies ts_2 (cf. (43))
3. **Define Services for Unused Input** After the determination of entity services and task services, the `Request` object was not used for the enterprise service es_1. As a result, a store candidate method was identified.
   - svc_7 = `({Request}, ∅))` (result for es_1; satisfies ϵ_3 (cf. (46))
4. **Define Services for Unprovided but Required Input** After the determination of entity services and task services, no unprovided elements were available.
5. **Ensure Granularity** As there were no services in registries available, the next step was to ensure the granularity for the services svc_1 - svc_7. The only possible combination was to combine svc_3 and svc_7. However, as such a combination neither increases the SSM value of the resulting service, nor was there a functional argument for such a combination, no method candidates were merged.

As a result of this step, the coordination service candidate methods svc_1 - svc_7 were identified. Additionally, the requirement for the service coordinations `{`svc_2`,` svc_3`}` (es_2), `{`svc_4`,` svc_5`}` (es_4) and `{`svc_1`,` svc_7`}` (es_1) could be identified.

8.2.4 Step 4: Describe Service Orchestration

By using the enterprise service candidates from step 2, an orchestration candidate was described. The decision logic of the exclusive-choice branch accesses the `FuL` attribute `released` and checks whether the boolean value equals *true*.
The deliverable that was described during this step is described in figure 54.

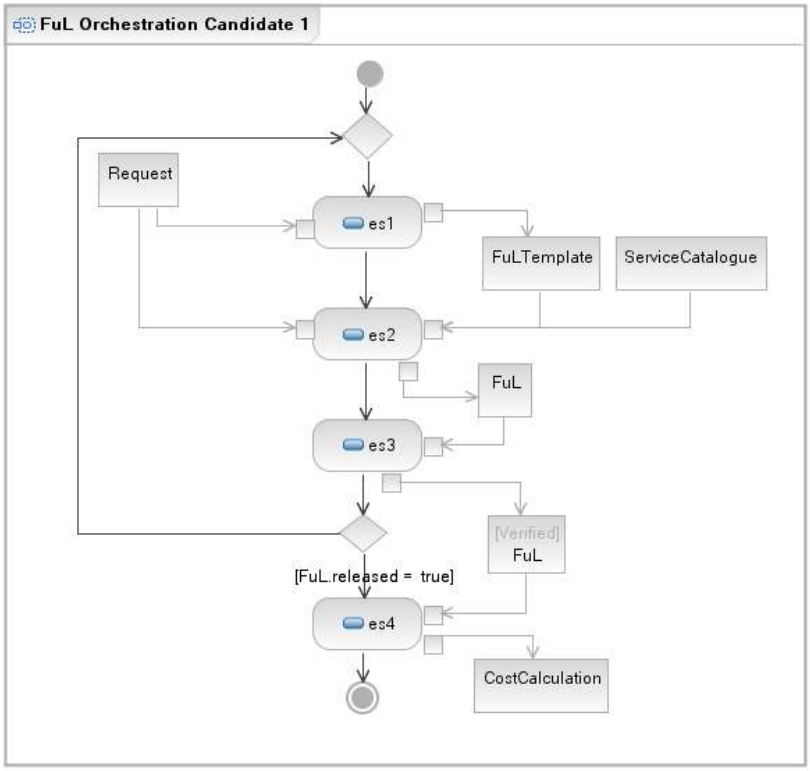


Figure 54: Initial Orchestration Candidate for FuL Creation

8.2.5 Step 5: Create Service Coordination Description

Three enterprise services were identified in step 3 to require a service coordination. The control and data flow of these coordinations was defined in this fourth step. The resulting service coordination candidates for the enterprise services es_1, es_2 and es_4 are shown in figure 55.

8.2.6 Step 6: Refine Candidate Methods

In order to objectively decide on a redesign of candidate methods, the values for the DOA, SSC, ACZ and SCZ metrics (cf. section 3.2.1) were calculated for the single service coordinations and for the overall system as far as it was designed at that stage. The results are shown in the tables 19 and 20.[54]

Service Coordination	DOA	AD
es_1	+0.29	0
es_2	+0.29	0
es_4	+0.29	0
es_1 & es_2 combined	+0.47	1

Table 19: Metrics for Assessing Coordination Design

[54]The metrics are calculated according to the procedure described in section 6.2.7.
The structure of the intermediate system that is analyzed as well as basic size metrics can be found in appendix D.

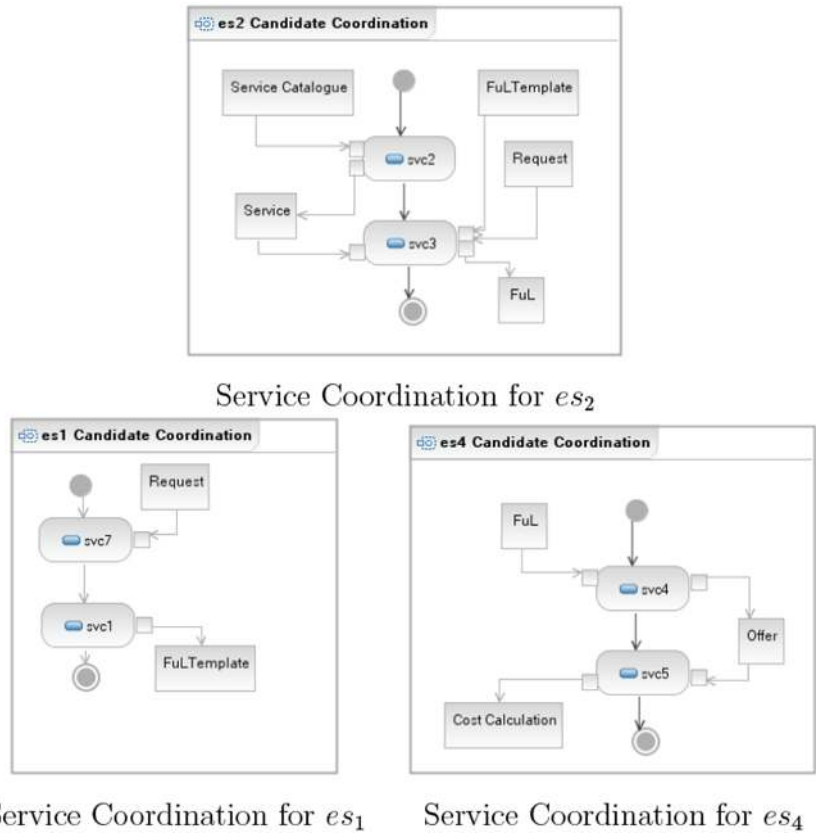


Figure 55: Service Coordination Candidates of the Case Study

The DOA values and the AD values for the single services indicate that the aggregators are sufficiently dense. Hence, no re-design is indicated by these metrics here.

Also the DOA value for the overall system is sufficient (bigger than zero). The SSC values indicate also a relatively low complexity.
The ACZ and SCZ metrics did not motivate a re-design either. The degree of centralization of the system at this stage seemed sufficiently high.
It would have been an option to merge es_1 and es_2 into one service. From a metric point of view, this would have slightly increased the complexity and distributed the centralization in a few non-mediating aggregators. Overall, no necessity for a redesign was indicated by the design metrics.

Overall System (Ω)	DOA	SSC	ACZ	SCZ
$\{es_1, es_2, es_4\}$	+1.33	0.23	0.75	0.53
$\{\{es_1 \cup es_2\}, es_4\}$	+1.16	0.3	$0.\overline{3}$	0.69

Table 20: Metrics for the Overall System Sketches

As the definition of an appropriate FuL template is based on the request of a customer as well as on other circumstances, it was decided to combine svc_1 and svc_7 to one single coordination service that corresponds with the enterprise service es_1. The resulting coordination service svc_8 was defined as follows: $svc_8 =$ `({Request}, {FuLTemplate})`. In exchange, the service candidates svc_1 and svc_7 were discarded. The other service coordinations were not changed.

Finally, the remaining coordination services were named. The following items represent a list of the revised coordination services:

- `Service[] getServiceList(ServiceCatalogue)` (svc_2)
- `FuL createFuL(Request, FuLTemplate, Service)` (svc_3)
- `Offer createOffer(FuL)` (svc_4)
- `CostCalculation createCostCalculationForOffer(Offer)` (svc_5)
- `FuL approveFuL(FuL)` (svc_6)
- `FuLTemplate defineTemplateForRequest(Request)` (svc_8)

All these services do not rely on an internal state. Each service has input values that are used in order to compute a result. Thus, all six candidate services were considered to be stateless.

8.2.7 Step 7: Analyze QoS Requirements of the Service Coordinations

The required availability for the enterprise services was defined with 99% each. Using the *Avl* metric (cf. section 3.2.2), the tolerated failure rate for the single coordination services (θ_{svc}) was deducted. The results are shown in table 21.

Coordination Service	θ_{svc}
Coordination Platform	$< 0.10\%$
`getServiceList`	$< 0.46\%$
`createFuL`	$< 0.46\%$
`createOffer`	$< 0.46\%$
`createCostCalculationForOffer`	$< 0.46\%$
`approveFuL`	$< 1.00\%$
`defineTemplateForRequest`	$< 1.00\%$

Table 21: Tolerated Failure Rates of the Coordination Services

Due to the long-running nature of the coordination services, no need for ACID transactions was identified.
As the service catalogue might be subject to frequent changes, the intermediary result of the queries to it were neither considered as being a safepoint.
The only safepoint that was identified was the method `Offer createOffer(FuL)` for es_4. The created offer should be kept also in case of a failure during the creation of the cost calculation.

8.2.8 Step 8: Design Application Services

Identifying appropriate application services was, by far, the most labor intensive part of the case study. Political discussions about the necessity of single application systems were

as much a part of the discussion as technical discussions were. Identifying appropriate experts for the single applications was a time-consuming task. Finally, key people for the CRM, ERP and CMDB applications were identified.

- **Accessing the CRM System** As the responsibility for the FuL creation process belonged to the same organizational area as the CRM system, support was very good. Experts were identified, necessary changes staffed appropriately and executed in a timely manner. Even if the application was not included into the business process model, discussions revealed that the initial trigger of the composite application was best to be performed by the CRM system. According to the identified process orchestration and the service coordinations, the initial coordination service method that was invoked is `FuLTemplate defineTemplateForRequest(Request)`. The parameter of this method had been identified to be the actual `Request` of the customer. Hence, the CRM system was identified as a service *consumer* for this service provider.
  This functionality (of performing a request) was at that time not implemented as part of the CRM solution. This is why the option of a *mediated use* was identified.

- **Accessing the ERP System** As the ERP system was a standard COTS system it was feasible to identify appropriate functionality for the requirements. According to the process model, the *perform cost calculation* activity was (to be) performed by the ERP system. This meant, the services `Offer createOffer(FuL)` and `CostCalculation createCostCalculationForOffer(Offer)` were candidates for deployment on the ERP system. An analysis revealed that the functionality for creating an offer was existent and accessible as a remote function call. The identified RFC was the (previously self-developed) function module `Y_ISD_QUOTATION_CREATE`. For the method `Offer createOffer(FuL)` *mediated reuse* of the RFC was identified to be an appropriate realization.
  Discussions both with the application experts and the FuL creation process owner revealed that the selling process was implemented in large portions within the ERP system. Because of this, it was decided to end the control centralization after the creation of an appropriate offer. As a consequence, the service method `CostCalculation createCostCalculationForOffer(Offer)` was discarded and not realized.

- **Access to the Service Catalogue** A major objective from a business point of view for the composite application was to allow for a creation of FuLs that are based on a standard service catalogue. The CMDB was identified to be the place that *should* be used to store available services. The idea was that, thanks to the huge amount of already sold services, all services of the company would be stored into that database. This assumption was wrong. Mainly two obstacles hindered the integration of the CMDB. First, the data model of the formal COTS was modified in a way that it was impossible to access the data in a structured way. Also the application itself did not offer remote accessible functionality. Hence, neither *mediated reuse* nor *mediated access* was possible. Additionally it turned out that the data quality of the kept data was poor. Each contract that was contained in the database was created using different descriptions for the same services. Proper key management was neither implemented. As a consequence, a project was launched to establish a central service catalogue within the company. For the composite application the impact was that no service catalogue was available. It is planned to integrate the service catalogue into the composite application as soon as it is finished

(estimated development duration: 15 month).
The discussed issues led to the fact that the `Service[] getServiceList (ServiceCatalogue)` method could not be realized. The idea of using a service catalogue was abandoned. Instead, an FuL was decided to be created still without a list of services available.

- **Actual Creation of the FuL** As the actual creation of FuLs was realized using a word processor before the project was launched, there were no application systems or functional modules that could have been reused. This is why it was decided to realize the service methods `FuLTemplate defineTemplateForRequest (Request)`, `FuL createFuL(Request, FuLTemplate)` and `FuL approveFuL (FuL)` from scratch. As the services were to be created, it was decided to use these services without a mediator.
The functional requirements for these services were analyzed and formulated in collaboration with the process owner. This involved basic screen design and usability tests.

8.2.9 Step 9: Exchange and Transformation Design

In the 8^{th} step the need for two mediators was identified. The first mediation was required for the CRM application to consume the `defineTemplateForRequest` method. The required interaction pattern from the composite application point of view for this step was the *receive* pattern (cf. [98]). No acknowledgment was foreseen. According to the discussion of section 5.7.8, one IIF was required to realize this pattern. Both a `Trigger Service`[55] and a `Data Service` were required to be part of that IIF.
As the budget was limited and there was no functionality implemented for realizing this interaction, the most efficient yet simple solution was chosen: the CRM application triggers the composite while the composite application retrieves the actual data from the request. The lookup describes a *send/receive* pattern (cf. [98]). As a design option of this pattern, it was determined that the counter party was known and that the call should be realized in a synchronous way. This was decided because there was no simple way of realizing a callback from the domino application to the composite application.
The synchronous call was decided to be closed at the earliest possible position in the IIF, as the CRM application was not capable of computing the result. In case of errors during the IIF it was decided that manual support procedures would be triggered.

In terms of connectivity, it was determined to realize the initial trigger using an SAP RFC. As it turned out, this was not feasible.[56] Because of this, the following solution for connectivity was chosen: a menu item was added to the user interface of the CRM. The activation of this menu item would trigger the generation of a file. In order to avoid compliance issues, as well as to increase the reactivity of the solution, it was decided to use a solution called "SemFis" that was based on [162]. This solution provided a means of connectivity to the CRM application. [162] describes a virtual file system that converts file-level operations into service calls. By configuring the core of the "SemFis" solution in a way that invokes the appropriate IIF, certain file operations of the CRM were designed to be redirected to the DET.

[55] Note that `Trigger Services` are not designed in this step but in step 15.

[56] Due to an incompatibility of the Lotus Domino RFC implementation and the SAP XI RFC implementation via message gateways.

Together, the CRM application and the "SemFis" solution, were designed to form a service consumer. The file format that was decided on was an XML file that solely contained the key of the actual `Request`. Based on these circumstances, the integration services were defined as follows:

- **Heterogeneity Service** As the file that was identified to be the service request, a `Heterogeneity Service` was necessary to translate the key-field of the file into an empty `Request` object for the composite application.
- **Data Service** The "SemFis" solution can be considered to realize the connectivity part as well as `Fetch Data` activity of the required `Data Service`. For the `Data Service` of the IIF that had the consequence that the request contained the necessary data. Hence, the `Data Service` was realized as a simple mechanism to forward the request to the integration flow.
- **Validity Service** Due to the simplicity of the file and the absence of special requirements, no need for a `Validity Service` was identified.
- **Routing Service** As solely an IIF was required, no `Routing Service` was necessary.

The IIF candidate after this step for the mediation of the `defineTemplateForRequest` is depicted in figure 56.

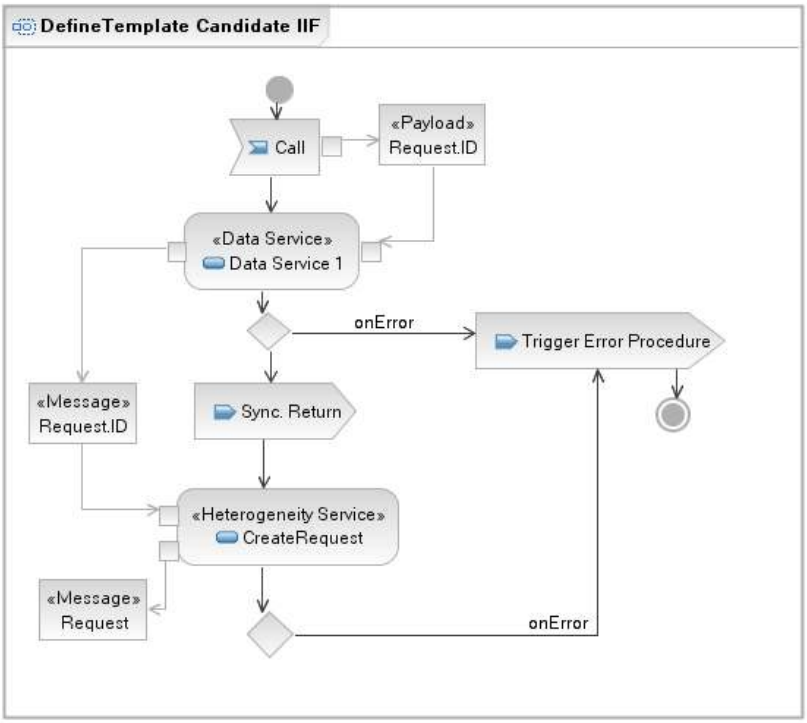


Figure 56: Candidate IIF for the Mediation of the Method `defineTemplateForRequest`

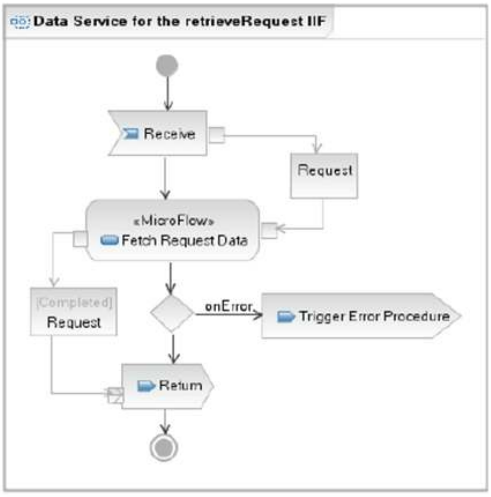
As the data of the request was not completely transmitted, the service coordination for the `defineTemplateForRequest` enterprise service method was extended by a method called `Request retrieveRequest(Request)`.

The definition of this method required an iteration of step 8 in order to describe the realization of this method. It was decided to realize the functionality by exposing a *view* that made all requests of the CRM accessible. By using adapters that access that view, a synchronous *send/receive* interaction (cf. [98]) with a known counter-party was foreseen. According to the discussion of section 5.7.8, both an IOF and an IIF are required to realize this type of interaction. Additionally it could be determined that the IOF requires a `Routing Service`, needs to trigger an IIF and must not use a `Data Service` by its own.
The actual access was realized with a 3^{rd} party adapter for the XI (cf. [163]). The necessary integration services are described below:

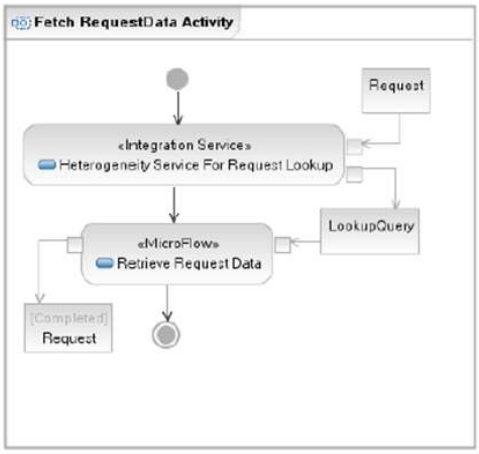
- **Heterogeneity Service** The necessary transformation was – as a lookup was to be performed – part of the `Fetch Data` activity of the triggered IIF (cf. section 5.7.8). This is why there was no **Heterogeneity Service** defined for the IOF.
- **Data Service** The use of a `Data Service` for an IOF of a *request/response* scenario is forbidden.
- **Validity Service** Due to the simplicity of the file and the absence of special requirements, no need for a `Validity Service` was identified.
- **Routing Service** The necessary `Routing Service` was a simple one-item determination of the CRM application as the final receiver.

An IIF for a *request/response* interaction must not use a `Trigger Service` but use a `Data Service` that contains a `FetchData` activity. The integration services that were designed for the required IIF are:

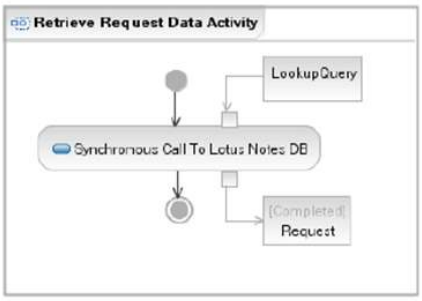
- **Heterogeneity Service** In order to transform the `Request` object that was part of the service method's interface into a lookup to the lotus database, a `Heterogeneity Service` was designed. The actual transformation of this service uses a pre-defined query that is used as a template. In the template, the clause is replaced by the identifier that is extracted out of the `Request` object. This integration service is part of the necessary `FetchData` activity of the `Data Service`.
  In order to transform the result of the query into a `Request` object that can then replace the object stub in the service coordination, another `Heterogeneity Service` was designed. This realizes a structural transformation from the database's data format into the data format of the composite application.
- **Data Service** The `Data Service` for the required IIF uses a `FetchData` activity in order to query the database. The `FetchData` activity was designed to use the above-described `Heterogeneity Service`. A synchronous call was foreseen to realize the functionality for the `RetrieveData` activity (cf. section 5.7.1).
  The `Data Service`, its `FetchData` activity and the respective `RetrieveData` activity are depicted in figure 57.
- **Validity Service** Due to the simplicity of the file and the absence of special requirements, no need for a `Validity Service` was identified.
- **Routing Service** No *Routing Service* is necessary for an IIF.



Data Service



Corresponding FetchData Activity



Corresponding RetrieveData Activity

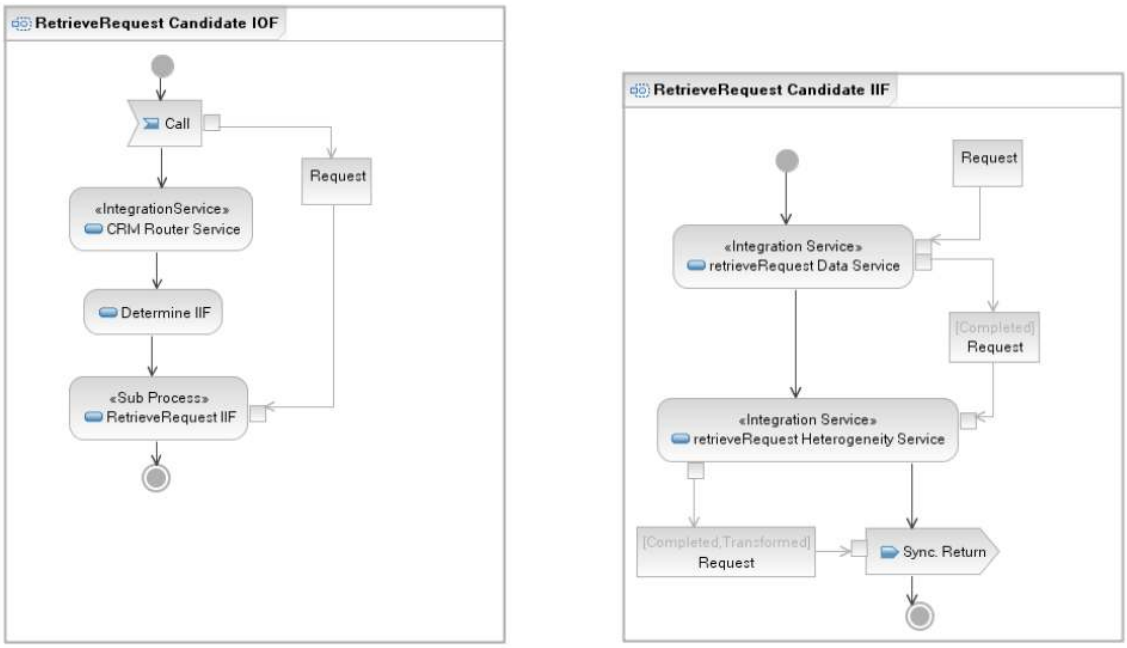
Figure 57: `Data Service` for the IIF that mediates the Method `retrieveRequest`

The IIF and IOF that mediate the `retrieveRequest` service interaction are finally depicted in figure 58.

In order to mediate the usage of the SAP RFC, the DET was also required. The interaction between the coordination service `createOffer` and the RFC `Y_ISD_QUOTATION_CREATE` was designed to be a synchronous *send/receive* interaction. The reception of a response was included in order to allow the composite application to track the changes of the ERP application system. This way future changes were anticipated.
As the function module that realizes the RFC was not allowed to be modified, synchronous communication was chosen. This is because there was no standard mechanism of realizing call-back interaction with RFCs in the SAP R/3 system.

As discussed for the `retrieveRequest` interaction, both an IOF and an IIF were required. The integration services for the `Y_ISD_QUOTATION_CREATE`-IOF were:

- **Heterogeneity Service** In order to transform the `FuL` into a message that suits the RFC, a `Heterogeneity Service` was designed. This service was able to map



Integration Out-Flow Integration In-Flow

Figure 58: Candidate Integration Flows for Mediating the Method `retrieveRequest`

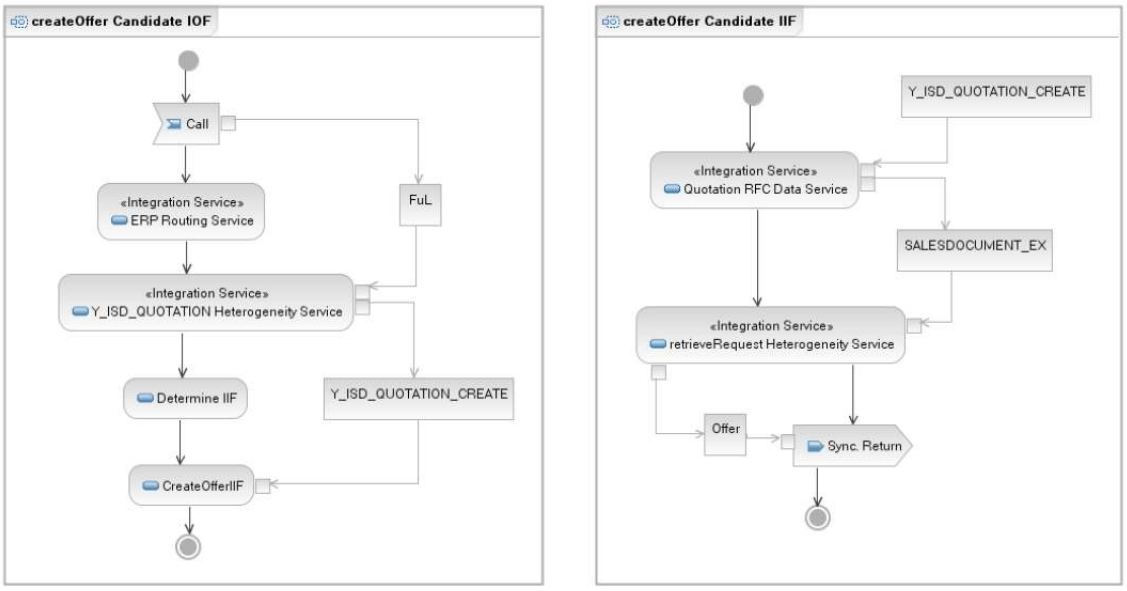
the actual `FuL` into the parameters of the RFC.

- **Data Service** The respective integration service of the IIF was used.
- **Validity Service** Due to the simplicity of the file and the absence of special requirements no need for a `Validity Service` was identified.
- **Routing Service** The necessary `Routing Service` was a simple one-item determination of the SAP ERP application system.

The IIF for the *request/response* interaction with the RFC included a `Data Service` that contains a `FetchData` activity. The following integration services were designed:

- **Heterogeneity Service** In order to transform the result of the RFC call into an `Offer` object that indicates that an offer was created in the back-end system, a `Heterogeneity Service` was included. This service simply created a stub of an `Offer` object that contained the id of the offer in the back-end system.
- **Data Service** The `Data Service` for the required IIF uses a `FetchData` activity that simply invoked a `RetrieveData` activity. A synchronous call was needed to realize the functionality of this activity.
- **Validity Service** Due to the simplicity of the file and the absence of special requirements no need for a `Validity Service` was identified.
- **Routing Service** No `Routing Service` was necessary for an IIF.

Both candidate integration flows that were designed to mediate the access to the ERP application are described by the diagrams of figure 59.



Integration Out-Flow Integration In-Flow

Figure 59: Candidate Integration Flows for the Mediation of the Method `createOffer`

8.2.10 Step 10: Revise Service Coordination Description

Several requirements to change the service coordinations were identified during the design of the DET. This was because, due to technical constraints, several coordination services had to be added.

The method `defineTemplateForRequest` was defined to be aggregated with another service method `retrieveRequest` in order to complete the object stub that was transmitted by the CRM application. This is why the coordination service for the enterprise service es_1 was modified. The result of that redesign is described in the model of figure 60. The name of this service coordination was defined as `defineAppropriateTemplate_SvcCoord`.

Due to the issues with the CMDB system, the creation of an FuL was realized without a list of available services. This is why the remaining coordination service `FuL createFuL(Request, FuLTemplate)` (formerly svc_3) became the single item of the service coordination for the enterprise service es_2 `Describe Service`.

The service coordination for the enterprise service es_4 was simplified as it was decided to pass the control to the ERP system after creating an offer. This is why the coordination service `Offer createOffer(FuL)` (svc_5) became the only step of the service coordination.

The final list of five coordination services for the four enterprise services consisted of:

- `Request retrieveRequest(Request)`
- `FuLTemplate defineTemplateForRequest(Request)`
- `FuL createFuL(Request, FuLTemplate)`
- `Offer createOffer(FuL)`

- `FuL approveFuL(FuL)`

The first two services were aggregated to the service coordination `FuLTemplate defineAppropriateTemplate_SvcCoord(FuL)`.

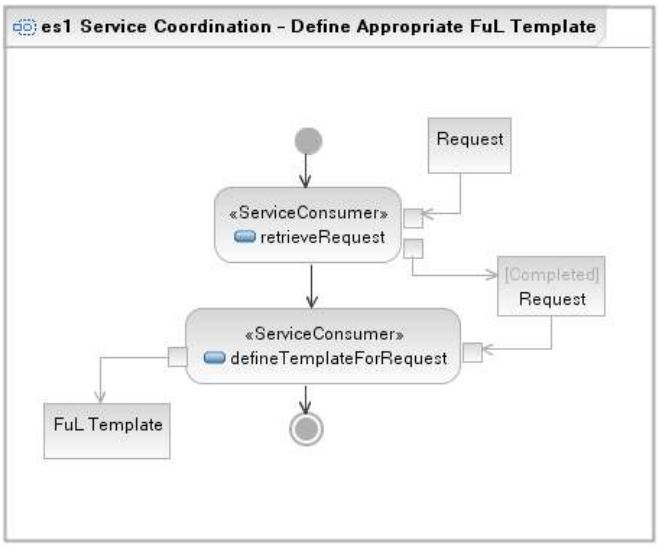


Figure 60: es_1: `defineAppropriateTemplate_SvcCoord`

8.2.11 Step 11: Revise Enterprise Service Candidates

Two constraints for designing the enterprise services were identified. The first was that there was no way of supporting a service catalogue with the given application landscape. The other challenge involved changing the creation of the calculation. As it was decided to not create a calculation but solely an offer, the respective enterprise service was changed accordingly. The other two enterprise services remained unchanged. The final list of enterprise services was:

- es_1 = `FuLTemplate defineTemplate_ES(Request)`
- es_2 = `FuL describeFuL_ES(Request, FuLTemplate)`
- es_3 = `FuL approveFuL_ES(FuL)`
- es_4 = `Offer createOffer_ES(FuL)`

8.2.12 Step 12: Define Events

The data visibility pattern *Case Data* (cf. [96, pp. 363f.]) is sufficient for the creation of FuLs. A factor in this decision was the fact that every enterprise service uses the output of a previous enterprise service. Only one event type was therefore necessary: `fulRequired`. The occurrence of this type of event determines the course of the FuL creation process. As soon as the FuL was passed on to the ERP system (es_4), an event of that type was considered finished.
As there was only one event type, no event relations had to be defined.

8.2.13 Step 13: Data Repository Design

The necessary data for `fulRequired` events are determined by the input parameters of the first enterprise service in the computation. As the analysis of the application systems determined is the transmission of the complete description of a `Request` not possible. This is why a service coordination was defined for es_1.
Nevertheless, a (stub) `Request` object is a prerequisite for the `fulRequired` event type.

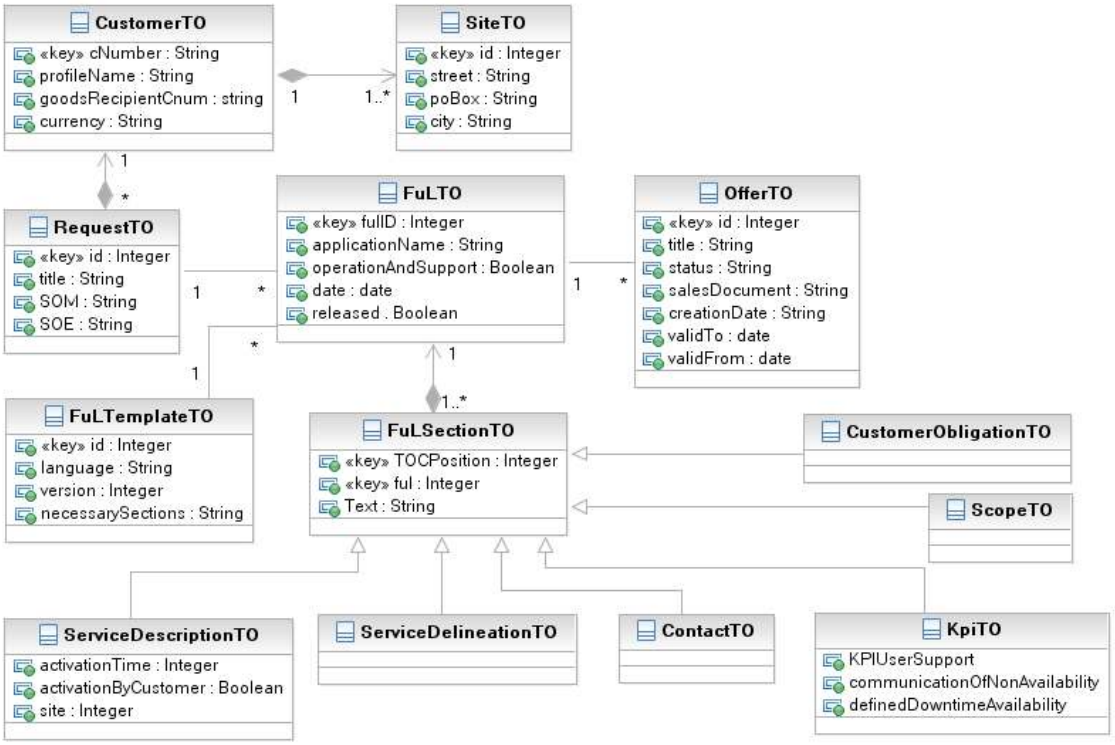


Figure 61: Data Model of the necessary Transfer Objects

By analyzing the single enterprise services and coordination services, the required data transfer objects were determined. They are described in the diagram of figure 61. The indicated compositions convey that the actual associated objects need to be transmitted as a whole.
Since the target platform of SAP NetWeaver was already determined, the platform-specific design of the data repository was also already defined. In accordance with the restrictions that are described in section 7.2.2, the final data repository for the FuL creation process included operations for each accessible data object and *entity services* for persisting the data. The structure of the data repository is shown in figure 62. For simplification reasons, not all 36 data repository operations are included in the diagram.

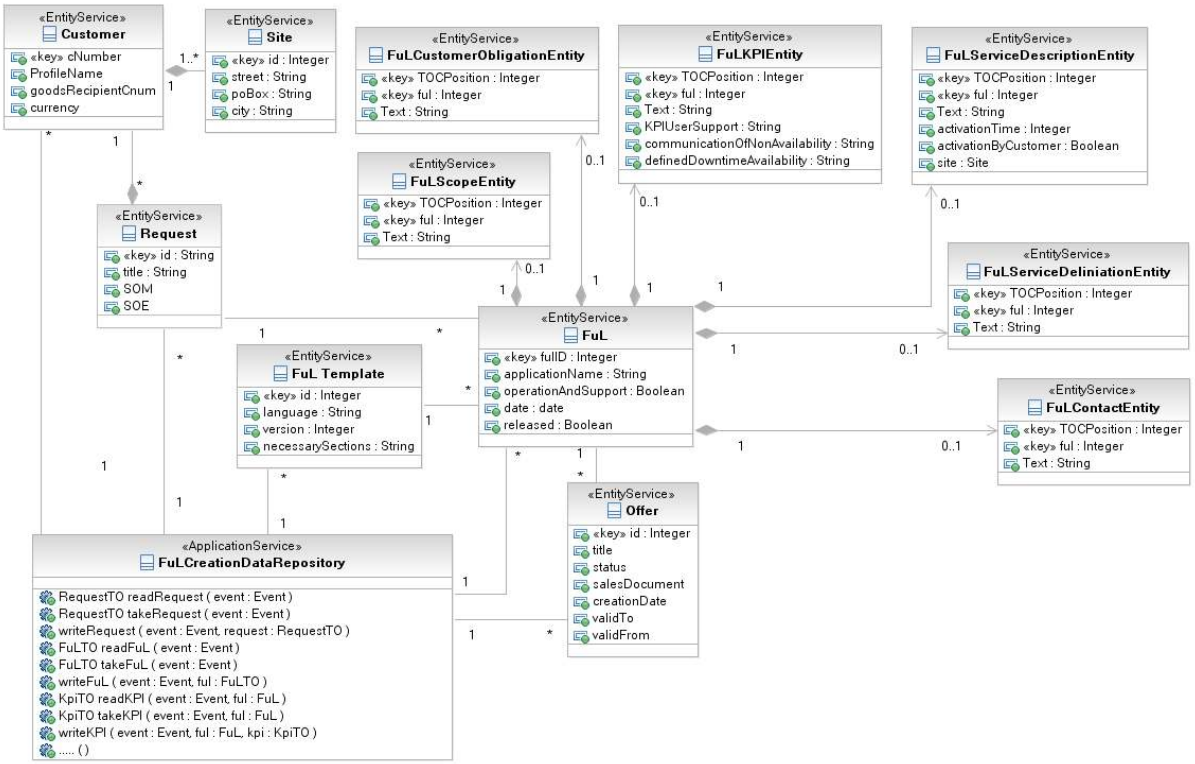


Figure 62: Data Repository for the FuL Creation Process

8.2.14 Step 14: Finalize Service Orchestration

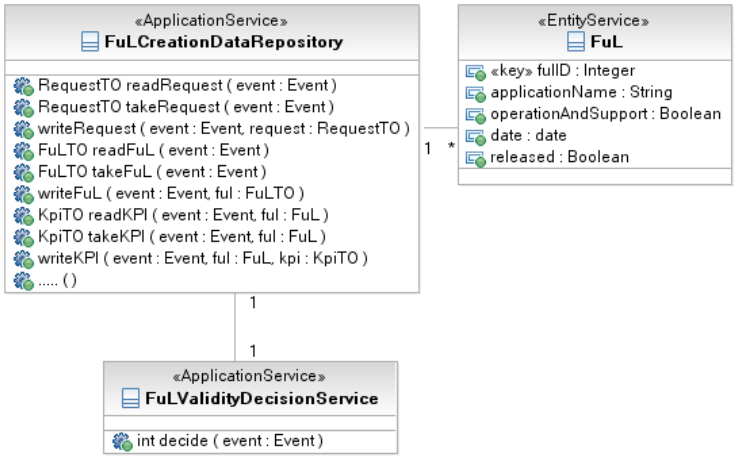


Figure 63: `Decision Service` for the FuL Creation Process

The intent of the first part of the finalization of service orchestration was to define an appropriate `Decision Service` for the FuL creation process. As the decision logic was simple, it was decided to not use a rule engine and to not formulate the decision logic in a business rules engine. The structure of the `FuLValidityDecisionService` is shown in figure 63.

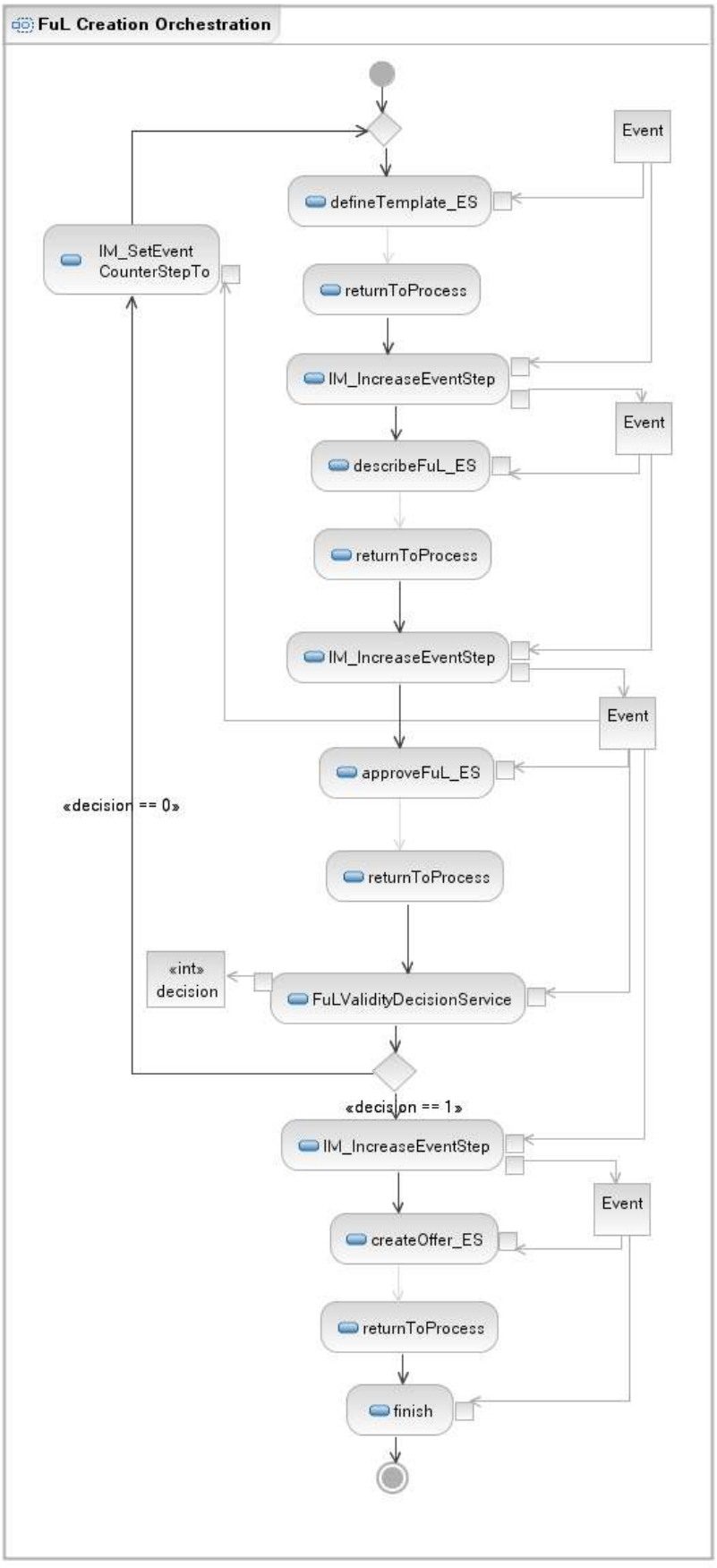


Figure 64: Final Service Orchestration

According to the modified enterprise services, the event type, and the `FuLValidityDecisionService`, the final service orchestration was defined. As it was known that the orchestration will be deployed on the SAP XI, XI `interface mappings` were included in the orchestration. Additionally, a count was added to the actual `Event` message that was passed back and forth within the process. This was necessary in order to allow the XI integration server to dispatch the events accordingly (cf. section 7.2.6).
The service orchestration, that was used to create the WS-BPEL process for the XI, is depicted in figure 64.

The analysis of the interactions within the FuL creation process did not reveal the need for any `Trigger Services`. Since the process can not start its computation without a `Request` object, the `Trigger Service` that also interacts with the eventing system needs to store the according data into the data repository. The `FuLTrigger Service` that was designed for the composite application is depicted in figure 65.

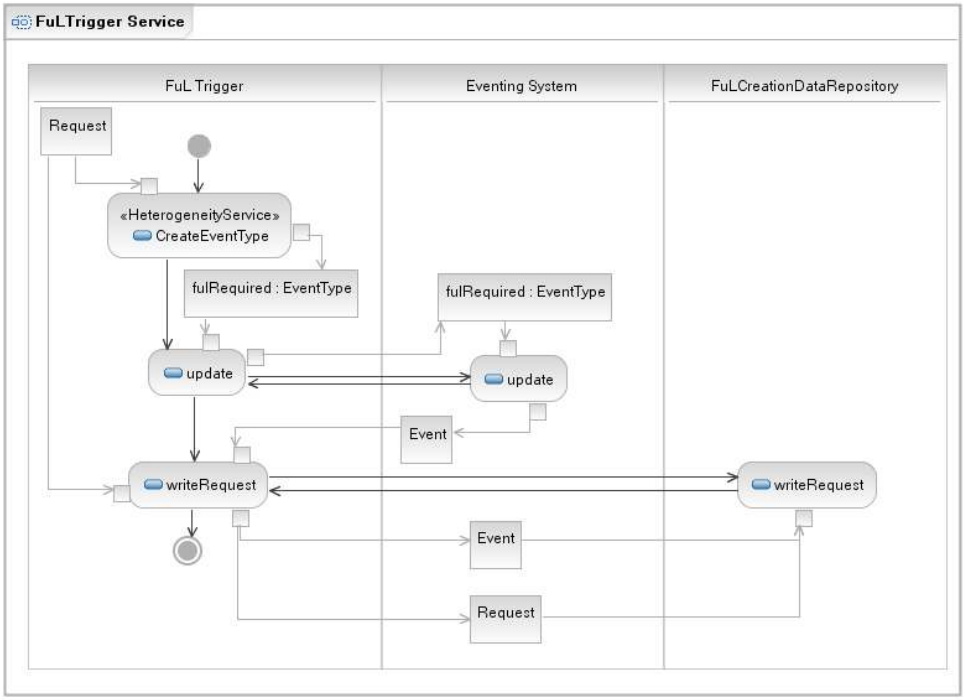


Figure 65: `Trigger Service` for the FuL Creation Process

8.2.15 Step 15: Finalize Exchange and Transformation Design

After the `FuLTrigger Service` was defined, the IIF for the initial call was finished. The other integration flows, that were already defined, were not changed. To conclude the description of the composite application's design, the finalized IIF is depicted in figure 66.

8.2.16 Step 16: Pass over to Implementation

Based on the artifacts that were identified during the steps of the design methodology, the implementation was started. Due to a lack of tool support, all design artifacts were manually translated into code and XI descriptions.

As it turned out, the description of the design was valuable for the implementation. The developers concerned with the realization were able to use them in conjunction with the platform specific model (cf. chapter 7) for the realization of the composite application.

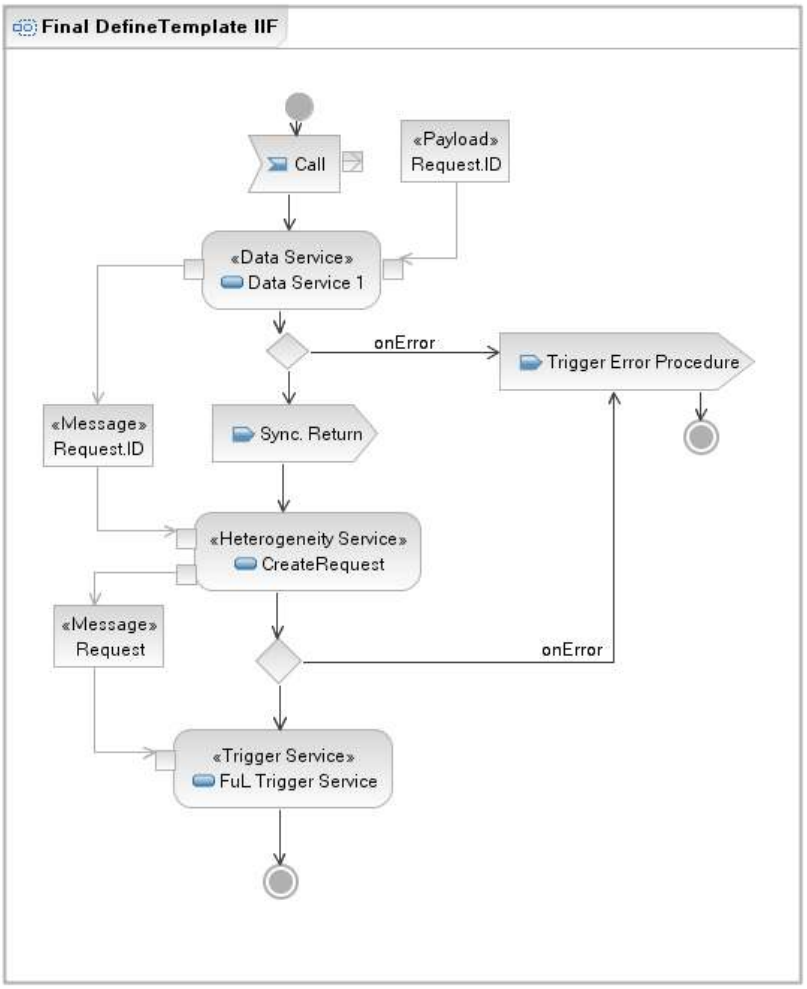


Figure 66: Candidate IIF for the Mediation of the Method `defineTemplateForRequest`

8.3 Analysis of the Design

The reference architecture provides a structure for implementing composite applications. In combination with the design methodology as well as the platform-specific model, it supports the application of the service-oriented style within the context of an IT supplier (and, of course, potentially in other contexts).
The architecture describes the re-use of existent functionality by two main mechanisms: 1) accessing potentially any target application that offers appropriate functionality via flexible integration flows; and 2) centralizing the control over these application systems into one service orchestration. In order to use business processes as a blueprint for these service orchestrations, it is necessary to include service coordinations to aggregate application functionality in a way that a business process is supported.

To approach the quantification of a composite's compliance with service-oriented principles, the design of the FuL creation composite application is analyzed in the following by using the metrics that are defined in chapter 3. This analysis should also provide an estimation for the significance of the metrics. As there is no overall evaluation, this only indicates the single quality aspects of a composite application but does not allow for determining valuable thresholds for the metrics. This is why the following evaluations are considered a first description of the single metrics' "behavior" in the real world. An approach to the interpretation of the single values is also made. However, it is important to note that the single values can only be reliably interpreted if a certain number of applications were analyzed so that estimations for reasonable thresholds exist.

The design that is analyzed is described in the previous section. For the sake of eased understanding, the design is summarized in the component diagram in figure 67. In this diagram, services are modeled as components. Service consumers have *required* interfaces while service providers offer *provided* interfaces. Service aggregators have both types of interfaces. In the given design, the integration flows do not aggregate any services. They are simple *service-enabling* application systems. This is why they are only indicated as circles and not as components and not included in the analysis of the design.

In order to describe the complexity of the composite application for the FuL creation process, the complexity metrics that are discussed in section 3.2.1, are applied to the composite application.

The following analysis will be performed for two cases: 1) considering the data repository as a service and 2) not considering it as a service. The second case is included since, depending on the platform, the data repository is not necessarily realized as a service provider (eg. when using SAP NetWeaver — cf. section 7.2.2). Of course, the data repository adds a certain amount of complexity. However, it does not, e.g. include a portion of a system's control model. This is why it is valuable to measure a system while excluding the data repository. However, its complexity has to be taken into account.
First, the different coupling metrics are identified for every service. The result is shown in table 22.

Based on the metrics of the individual services, the complexity of the overall system is then analyzed. The results of this analysis are shown in table 23.

This analysis shows that the complexity metrics SSC and SCF indicate low values for both cases. This means that the overall complexity (through the notion of coupling) of the

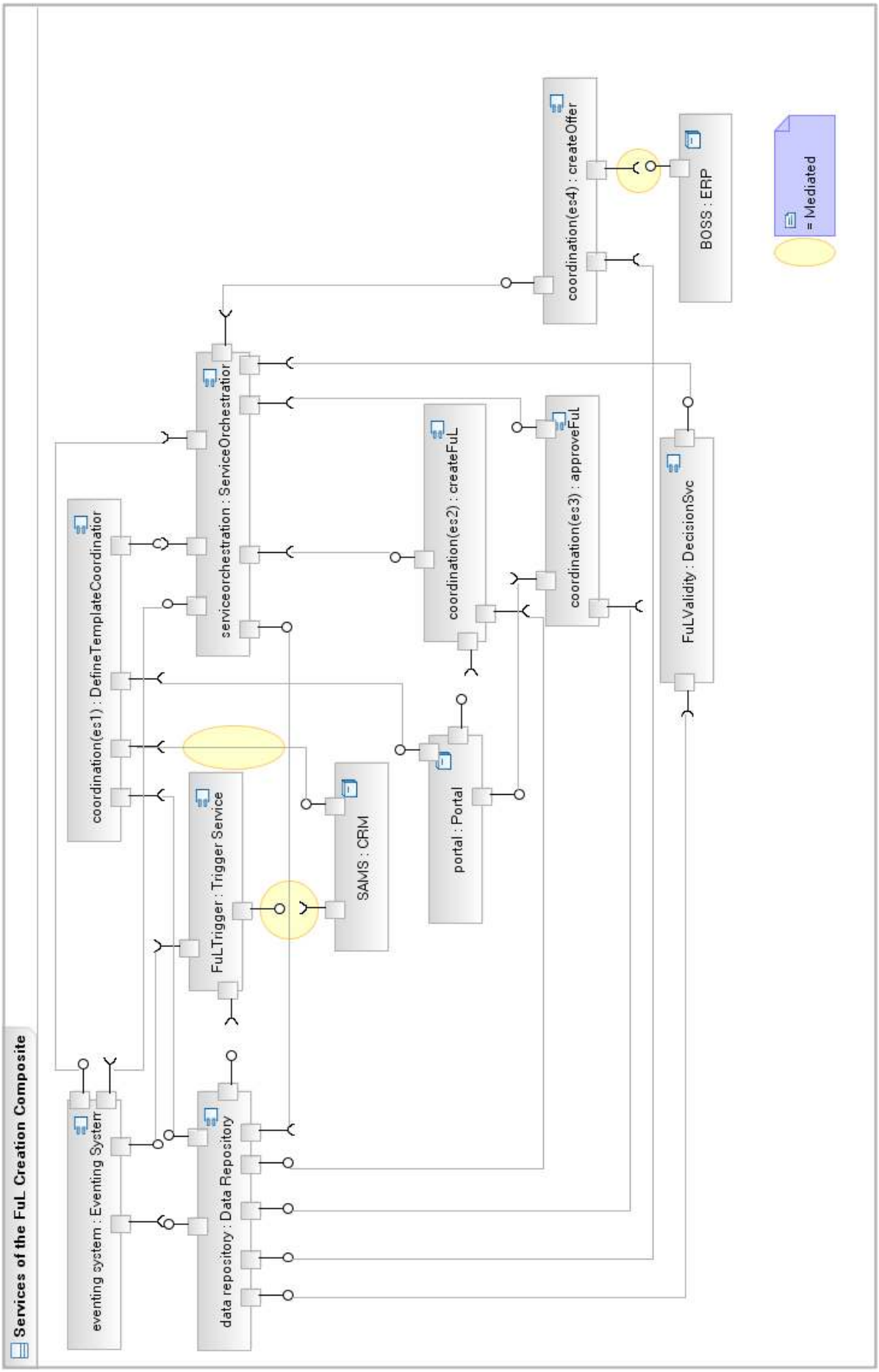


Figure 67: Components and Links of the FuL Application

composite application can be assumed to be relatively low. These values solely indicate the coupling of the services. To get another picture of the composite's complexity, the count of the single services should be considered, too. Those values (11 and 12 respectively) are also an indicator of a relatively low complexity.

In order to provide an overview of the behavior of the complexity handling metrics, they are also described for the FuL creation composite. As the composite application was the first one in the application landscape, the modifiability metrics of chapter 3.2.1 that describe reuse could not be applied.
Interesting to note is that the extent of aggregation is zero in both cases. This is because no pure service consumers are part of the system. This is also why the system's centralization, as far as the SCZ is concerned, is relatively low.
Another reason for a relatively low centralization value is that the application services are mediated by service coordinations with a low density. As the service coordinations do not provide any control logic, the SCZ, that is only based on the notion of aggregators, indicates a distributed control. As discussed in section 3.2.1, the SCZ value should be considered together with the SSC, SCF and ACZ values. As these values indicate a low complexity, the little centralization in terms of the SCZ metric is acceptable.
The control centralization that is indicated by the ACZ slightly varies between the two scenarios. In the scenario without the data repository, ACZ indicates a relatively high degree of centralization. This is because most of the aggregators are used as mediators. As these mediators also interact with the data repository, the ACZ for the other scenario is a little lower. The difference between the values of the metrics that indicate the centralization indicates that the centralization might be sufficient if the aggregators are designed accordingly.

Service s of Ω	$cos(s)$	$cts(s)$	$\gamma(s)$	$\pi(s)$	$AD(\Omega, s)$
Define Template (es_1)	3	1	3	1	1
Define Template (es_1) w/o DR	2	1	2	1	0
Create FuL (es_2)	2	1	2	1	0
Create FuL (es_2) w/o DR	1	1	1	1	0
Approve FuL (es_3)	2	1	2	1	0
Approve FuL (es_3) w/o DR	1	1	1	1	0
Create Offer (es_4)	2	1	2	1	0
Create Offer (es_4) w/o DR	1	1	1	1	0
Central Orchestration	6	2	6	2	1
Central Orchestration w/o DR	6	1	6	1	1
Eventing System	2	2	2	2	0
Eventing System w/o DR	1	2	1	2	1
Trigger Service	2	1	2	1	0
Trigger Service w/o DR	1	1	1	1	0
Decision Service	1	1	1	1	0
Decision Service w/o DR	0	1	0	1	n.a.
SAMS CRM	1	1	1	1	0
Portal	0	1	0	1	n.a.
ERP	0	1	0	1	n.a.
Data Repository	1	7	1	7	1

Table 22: Size Metrics for the Overall Composite Application

The density of aggregation (DOA) is in both cases positive. This means that the aggregators access more service methods than they provide.
The data repository is an example of an aggregator that provides more methods than it consumes. Interesting to note is that, due to the amount of aggregators, this does not heavily influence the overall DOA value of the system.
The values of the complexity handling metrics are shown in table 24.

Application (Ω)	NS	SC	SP	SA	SSC	SCF
FuL Creation	12	10	12	10	0.183	0.167
FuL Creation w/o data repository	11	8	11	8	**0.156**	**0.109**

Table 23: Complexity Metrics for the Overall Composite Application

With regards to the metrics that describe the modifiability of a system it can be summarized, that the overall picture for the FuL creation composite indicates a relatively low complexity of the application that extensively uses (appropriate) aggregators. On one hand, the use of aggregators is a sign of incorporating service-oriented principles. On the other hand, the extensive use of aggregators does endanger the principle of control centralization.
The reason for the extensive use of aggregators is the adaptation of heterogeneous applications to fit into a business process. The fact that aggregators are used as mediators is indicated by the relatively high ACZ value. This value indicates high control centralization within some aggregators. Expressed differently, in the scenario that excludes the data repository, 75% of the aggregators is used as mediators while the control is centralized in 25% of the composite application's aggregators. Seen from this point of view, the FuL creation composite centralizes control on top of a heterogeneous landscape. It can be finally stated that the metrics do not motivate a re-design of the composite application.

Application (Ω)	SCZ	EOA	DOA	ACZ
FuL Creation	0.21	0	+0.58	0.7
FuL Creation w/o data repository	**0.22**	**0**	**+0.4**	**0.75**

Table 24: Complexity Handling Metrics for the Overall Composite Application

8.4 The Composite Application

This section describes the executable example of the design. After a description of the development phase and the observations that were made during the development, the composite application, and its look and feel are described.

8.4.1 Observations from the Development Phase

The composite application was realized using SAP NetWeaver as the platform for the composite application itself. The application landscape that was analyzed was also used to realize the required back-end functionality.
The development was organized into several groups that used the design to communicate. Each of these groups was dedicated to a special area of technology. As a consequence,

the service coordination layer, the data repository, the eventing system, and parts of the orchestration layer were realized by a unit that is specialized in Java and portal developments. Yet another group addressed the integration flows and the actual connectivity. This group was staffed with members of another organizational unit that focuses on enterprise application integration.
Each application system was again managed by different organizational units. Throughout the course of this, it became obvious that expert support of the respective application system is crucial.
Due to the simplicity of the transformation, the actual business process was implemented by the same architecture team that also supervised the collaboration of the single groups.

The intense collaborative nature of the development phase required to have a simple and precise design for the communication between the groups. Since every group was working on a specific platform with specific development tools, communicating diagrams became a major task of the project lead. Here it was helpful that, having a platform-specific model in mind, the platform-independent descriptions were always describing one artifact at a time.

One major issue of the implementation phase was the extreme immaturity of the development environment for the CAF. While the runtime platform proved to be stable, the bug-intense development environment actually consumed a major portion of the project's budget. This "devil in the details" issue made frequent re-installations of the developer workplaces necessary. The deployment of components that were not correctly updated by the development environment also rendered several runtime platforms useless.
The other components and the application systems could be programmed efficiently.

Another observation is that the requirements in terms of know-how the development of a composite application brings with it is extreme. The technology and the variety of products is too complex for single developers. The gap between the technologies is often so big, that the communication between the developers has to be moderated.
Because of this, the standardization of composite applications by the means of a reference architecture is considered to be a major answer to this issue. By restricting the possible solutions, a common base of knowledge among different specialists seems achievable.

8.4.2 Look and Feel

Concerning the result of the implementation, the process owner of the FuL creation process approved the composite application. It was agreed that it satisfies the initial business requirements.
The most apparent gap, both in the design as well as in the implementation, is the generation of an FuL as a document, though. It was agreed that the creation of an FuL in a structured way proves the suitability of the application. The labor-intense, yet unrelated to service orientation, process of creating documents was postponed until the composite would be changed in terms of connecting to a service catalogue.

The process starts in the Lotus Notes application SAMS by creating a `Request`. First, a so-called *Sales Objective* is created (figure 68).

Subsequently, the sales objective is exported to the file system. As the file system is realized as a virtual file system that adheres to the description of [162], the file is immediately

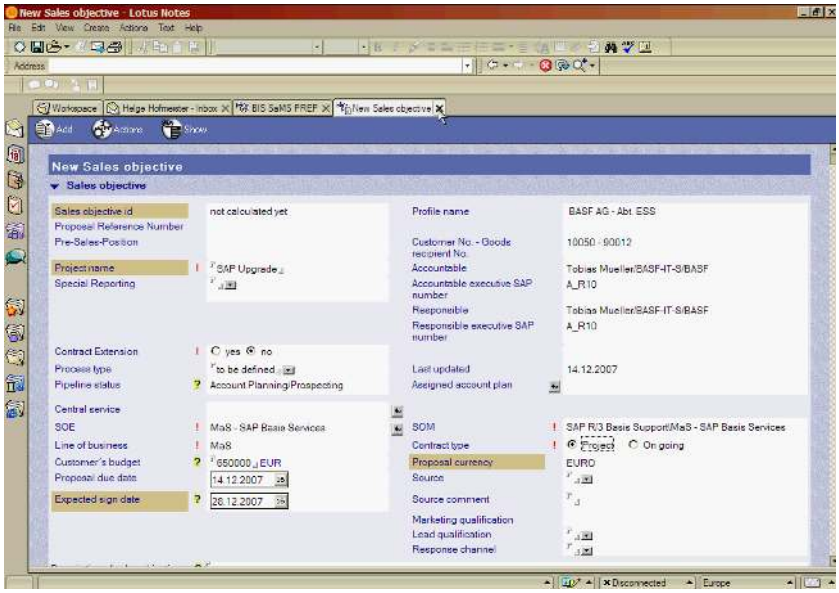


Figure 68: Step 1: Creating a *Sales Objective* in the Lotus Notes Client

transmitted to the composite application via XI, the data repository and the eventing system (figure 69).

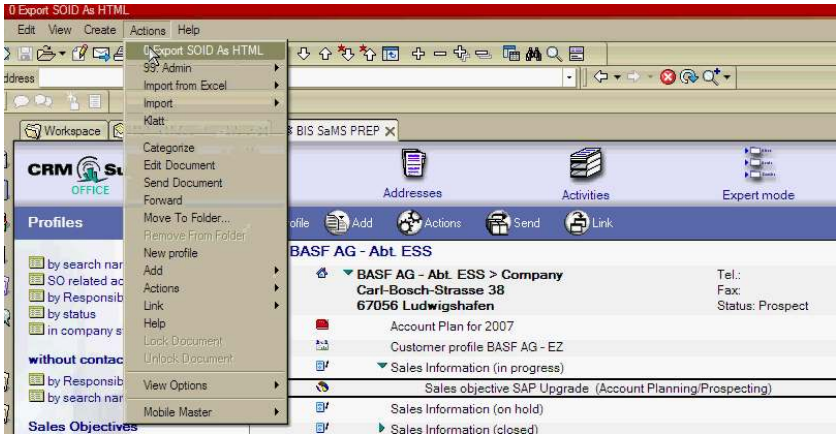


Figure 69: Step 2: Export the *Sales Objective* to the Composite Application

After the XI integration flows have transmitted the information to the composite application, the process orchestration triggers the first coordination service that, in turn, read the complete information out of the Lotus Notes database. After the information is loaded, the `defineTemplateForRequest` application service method is invoked. Such calls that require human interactions are realized as entries in the universal worklist of the respective clerk (figure 70).

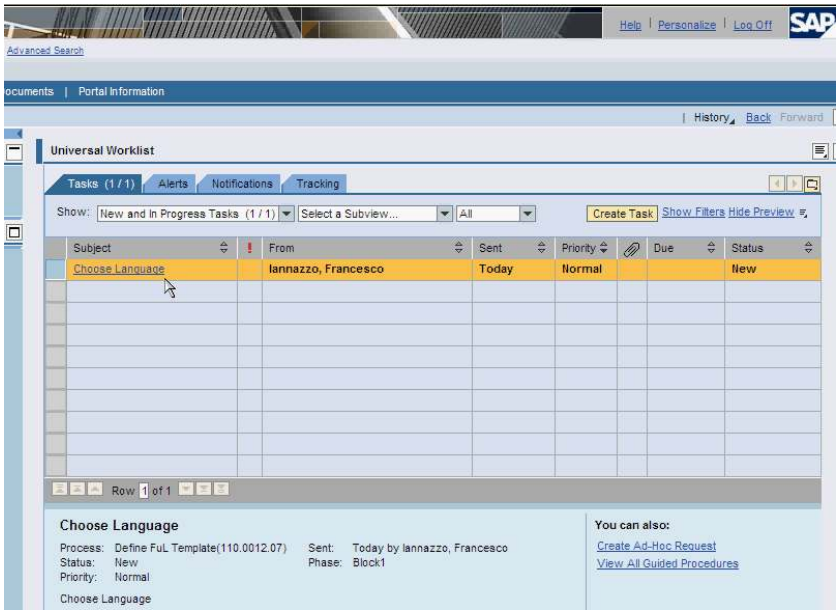


Figure 70: Step 3: User Interaction via Universal Worklist

After the appropriate language (that represents the template) was chosen, the orchestration calls the next service coordination that again invokes the portal. After selecting the entry from the universal worklist, the actual FuL can be described (figure 71).

When the creation is finished, the actual data is used to create an offer by the means of a *Quotation* in the ERP back-end system.
After the *Quotation* is created in the back-end system, the composite application stops. The next step of creating a calculation is performed outside the control of the composite application (figure 72).

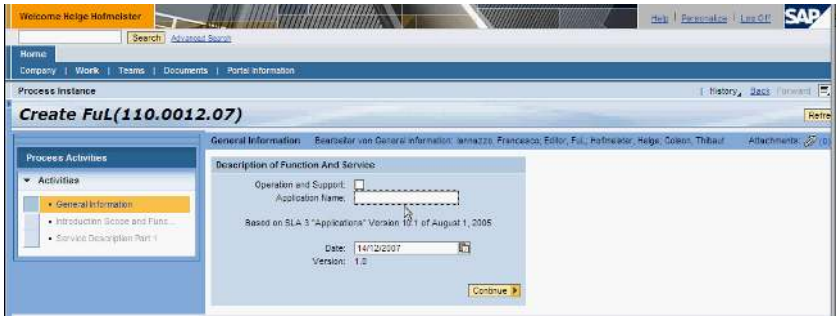


Figure 71: Step 4: Actual Writing of the necessary FuL Description

8.5 Summary and Conclusion

The composite application whose design was described and analyzed in this chapter represents the application of the service-oriented style in a real-life setting. Beginning with a business case that was mostly defined in terms of business needs, it was first necessary to identify the suitability of a composite application for realizing the FuL creation process. By using a simple yet applicable mechanism, this task was performed and the suitability of service orientation was identified.

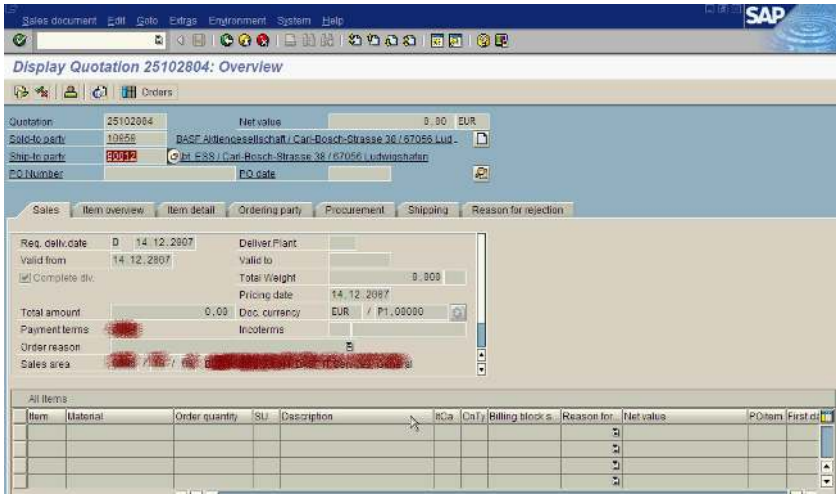


Figure 72: Step 5: Created *Quotation* in the SAP ERP System

Based on the requirements, the development methodology for composite applications was exercised. Taking the previously described top-down approach, the business process was analyzed and suitable services were designed. The business process was also used to derive a service orchestration that defined a control over the designed services. By aligning the top-down design with the actual constraints of the application systems, the design of the services and their compositions had to be adjusted. In the bottom-up phase of the methodology, several components had to be adjusted in order to allow for a composite application that fits into an organization's application landscape and complies with governance rules.

The final top-level service orchestration was slightly different from the initial business process. This was due to the constraints imposed by the application systems. The analysis of the design, which was conducted by calculating and interpreting several key metrics, showed that the structure of the composite determined by the reference architecture was not very complex while incorporating service-oriented principles for complexity reduction and adaptation to the heterogeneous application landscape.

The subsequent implementation of the composite application was influenced by technical platform-related issues. Besides problems with the technology, the design proved to be achievable. As such, the results finally met the expectations of key stakeholders.

The utilized development methodology was designed to minimize the descriptions a requirement engineering has to produce. By using a fully specified process model that included a complete description of the data perspective, a composite application was built.
However, the case study showed that this is still an ambitious demand. The degree of formalization and structure as it is demanded by the concepts could not have been higher. An interesting observation is that the focus on a business process and the actual use of the process as an artifact that is later implemented as specified (if possible), shifts the attention of both requestors and the people realizing a project towards a common understanding. This is why focusing on business processes is considered an advantage in and of itself.

The use of the reference architecture demonstrated that the structure it provides makes service-oriented projects possible. Without the common, limited understanding of service-oriented applications it would have been impossible to realize a composite application that incorporates service-oriented principles that go beyond component orientation. The platform-specific model for NetWeaver proved both applicable as well as supportive in actual development questions.

To conclude, the framework that was presented in this thesis made the application of the service-oriented style possible. In order to increase its value, an integrated tool support that involves both, development tools as well as a library of runtime components would be beneficial. As an IT supplier, BASF IT Services does not consider itself a tool vendor. However, if the status of all components of the preferred target platform matures, the development of such tools is potentially achievable.

9 Related Work

This chapter discusses the contributions of this thesis in comparison to related work.

9.1 Incorporated Work

This thesis is built on a variety of fundamental concepts. These concepts vary from essentials of distributed computing (such as the consistency model of [75] or the spaces approach of [104]) to more recent approaches that describe specific integration concepts or approach a holistic description of a service-oriented architecture (such as e.g. [4] and [11]). All concepts that form the basis of this work are referenced and discussed in the respective passages of this thesis.
In order to give an overview of the incorporated ideas, important concepts are briefly discussed in this section.

9.1.1 Reference Architectures

The discussion in chapters 2 and 5 show that a structure – in addition to the style definition of service orientation – is required to properly apply this architectural style. This can be achieved by creating reference architectures that describe the building blocks that need to be designed for a certain use case. Such a reference architecture can be referred to as a *design-time* architecture.
Regarding design-time reference architectures, related work is available. The Business Process Integration Oriented Application Integration (BPIOAI) approach introduced by Linthicum in [4] is one key concept for the findings of this thesis. This is because it describes how to centralize the coordination in a distributed and service-based application on top of remote-accessible functionality by using business processes. This provides an initial approach to structure as it is required in the described context.
Without explicitly referencing the work of Linthicum, Hentrich and Zdun use BPIOAI to put a service composition layer on top of a service oriented architecture (cf. [81] and [89]). This service-oriented reference architecture emphasizes the distributed nature of SOA by incorporating service invocation, adaptation, request handling and communication into the framework. The service coordination is classified as a macro workflow for business processes and a micro workflow for so-called "more technical" aspects. This way, a business process-centric development of composite applications can be achieved. The reference is built-up as a set of patterns that describe such a layered approach.
Erl has established in [11] a reference architecture that distinguishes a service interface layer, an orchestration layer, a business service layer and an application service layer. The service interface layer is put as a mediator in between the business process layer and the application layer. Emphasis is put on abstraction. Application services describe functionality exposed by the application layer while business services represent functionality used to reach business goals by putting them into a service orchestration that implements a business process.
The outline that is given by [90] also layers business processes, orchestrated services and enterprise components on top of an application landscape. While stating that user interfaces are out of scope for the discussions around a SOA, the reference architecture of [90]

anticipates that a dedicated user interface layer might be needed in the future. It is stated that, however, such a layer will be placed on-top of a business process layer and access services offered by this layer or by the basic service layer.
While agreeing with the fact that the design of user interfaces is not specific to the service-oriented style, the reference architecture of this thesis explicitly considers user interfaces as application services that do not differ from other back-end services. The control model is explicitly defined and user interfaces can, as all other application services, solely consume aggregated functionality as it is exposed by the service business process layer. This way, the integration into established processes and landscapes is facilitated while the user interface and a composite application are decoupled. This way, the modifiability of service-oriented solutions can be increased.

Schelp and Winter state that today's existent work in this area "does not sufficiently address the integration layer and its importance for decoupling business related structures on the one hand, and IT related structures on the other. This decoupling however is a necessary precondition for buffering changes and supporting alignment, hence for contributing to agility on a sustainable level" [15, p. 68].
In order to overcome this identified lack, the presented concepts of this work heavily emphasis the integration layer. This is achieved by introducing an integration layer that regroups and standardizes well known integration patterns as they were described by Hohpe and Woolf in [95]. The pipe-and-filter architecture that is used in [95] is replaced by a more flexible process-based approach. Here, the basic integration patterns are grouped into integration services that are orchestrated to integration flows. This way, the functional requirements as they are described by the means of the patterns are enriched with a functional specification that includes structural interfaces. Additionally, the integration processes, services and patterns are described together with necessary design decisions that facilitate the design of the integration layer for arbitrary interaction requirements.

The reference architectures cited above, however, all share the placement of a business process-like service orchestration on top of services that are distinguished into business task-focused services and "technical" services, though. These approaches all structure composite applications at design time on generic and platform-independent levels as well as address the idea of control centralization. This way, all approaches follow the requirements of "process dominance" as it is demanded by [15].
However, all these models lack a more detailed description of the single layers. Layers are merely described on a verbal basis. For instance, neither interface definitions nor platform requirements are provided. This is why there is necessarily a gap between the design based on the actual reference architecture and the execution model. The structure of the presented concepts addresses this issue by defining platform functionality and runtime requirements already on a platform-independent level that can be transferred to a runtime architecture.
In contrast to a design-time reference architecture, a "run-time reference architecture" can also be referred to as an *execution environment*. In order to close the gap between design and execution, the reference architecture of chapter 5 incorporates the platform-independent specifications of an execution environment.
All necessary components and their interfaces are described. So is the interaction of the components as well as the platform requirements that need to be satisfied in order to use a basic platform for realizing composite applications. In addition to other reference architectures, the description of the presented runtime environment addresses issues of data and context handling within composite applications.

The described distinction of a platform-independent reference architecture and a platform-specific execution environment builds upon the idea of the Model Driven Architecture (MDA) (cf. [131]). "The MDA defines an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform. To this end, the MDA defines an architecture for models that provides a set of guidelines for structuring specifications expressed as models" [164, p. 3].
The platform-independent reference architecture defines a meta-composite application. Based on this structure, a system's functionality can be described by using the design methodology of chapter 6 in a platform-independent manner. This addresses the concept of the MDA separating the description of a systems' functionality from any technology platform.
The platform-specific reference architecture represents, in turn, a *virtual machine* (cf. [91, p. 2-6]) that underlies the platform-specific model of the MDA approach. The deliverable of the design methodology in conjunction with this description of a runtime framework forms a platform-specific model in the sense of the MDA. This way, the MDA idea is incorporated into the concepts that were presented here.
However, there also is a major difference between MDA and these concepts. While the MDA aims to automate transformation from a platform-independent model to a platform-specific model through templates and their transformations, the approach that is presented in this thesis explicitly requires non-automated steps. This occurs during the specification of a design. Here, constraints imposed by an application are incorporated into the initial top-down design. Later, the deliverable of the design methodology presents a blueprint that can be used by the programmer who implements the code of a composite application. This approach is considered more applicable in the context of evolving heterogeneous application landscapes with its different systems, languages and constraints.

9.1.2 Service Design and Design Methodologies

In order to conduct projects within large organizations that are built on a reference architecture, a design methodology with comprehensive, yet objective, design support is required. These aspects are not considered holistically by the reference architectures of [4], [30] and [81] resp. [89] as they do not include a methodology for designing composite applications or single services. However, a methodology that allows the use of a reference architecture in a given use case is crucial.

In [15], the general requirements towards a methodology for service construction are motivated. It is stated that services design should follow a business process design ("process dominance"), the orchestrated services should cover a business process ("process scope"), borders within business processes should also determine service boundaries ("intersection points") and that service interfaces should be derived from higher level business objects ("interface reference") (cf. [15, pp. 68f.]).
The design methodology of [11] incorporates all these principles. In addition, the methodology is in concert with the reference architecture described ibidiem. However, the main aspect is the creation of single services. Other aspects of the design of composite applications (e.g the design of the integration platform, consistency or interaction requirements, incorporation of design principles) are hardly addressed.
The approach described in this thesis is inspired by the methodology of [11]. The top-

down phase of the methodology of chapter 6 describes how a business process can be used as an input for identifying several levels of services. This way, the "reference principles" of [15, pp. 68f.] are all met. However, the rough description of [11] is extended, refined and aligned with the artifacts that are required by the meta-structure of the reference architecture. Design metrics are also incorporated into the methodology. This allows a designer to refine a design in a way that increases modifiability of the composite application by applying service-oriented principles.
The subsequent bottom-up section of the design methodology is defined independent of the methodology of [11] and aims at incorporating platform restrictions into the design of a composite application.

A crucial part of the design methodology is the derivation of services for mediating enterprise services and application services according to the service meta-model that is derived from the platform-independent reference architecture.
Research focusing on the actual design of the services within service-oriented environments concentrates on the intrinsic design of services. The methodologies of Reijers [86] and Erl [11] describe different aspects of service design. Erl introduces the aspect of business process centered design as an enhancement of intrinsic service design. This idea is connected to the concept of BPIOAI since it incorporates business processes as a substantial part of an SOA. This idea is also reflected by the work of Papazoglou in [165] which utilizes business processes for designing services. However, the actual design of services is a subjective and rather unstructured task in all of these approaches.
The approach that is described in [86] proposes the idea of decomposing activities into more granular tasks by splitting a method by the data it processes. It uses the notion of cohesion as the indicator of whether a split is appropriate or not. In the approach that is presented in chapter 6, the described principles of splitting service methods by the data they process is incorporated. However, cohesion is not the basis for such a decision. In fact, the only property that proved to be significant for service re-use – service granularity (cf. chapter 4) – is used.
Based on this idea, an algorithm is defined that structures the task of designing services to mediate business process requirements and application functionality. This algorithm describes an approach for "assisted design-time composition" (cf. [14, pp. 95ff.]). Eased applicability is emphasized in the context of large organizations. This is why the approach exclusively relies on structural service interfaces and functional dependencies of the data processed. It does not require additional descriptions other than the artifacts of a business process model (with all its perspectives). More advanced concepts, such as behavioral interfaces or dynamic and semantic service composition, are therefore complementary to this approach (see below). However, some ideas that underlie semantic service mediation approaches are also incorporated into the service design algorithm. Essentially, this is the idea of service interfaces that "contribute" to a required service mediation (cf. [111])

The distinctive feature of the presented methodology is that it combines a holistic composite application design methodology with objectified principles for the design of single services while not presenting any great obstacles towards its application. This is partially possible because of the quantitative analysis of service design principles in chapter 4 which is the only analysis of that kind known to the author. In this respect, the methodology does not make use of recent principles such as semantic service consumption and automated service composition. However, it addresses these topics in a more "traditional" way in order to pave the way for a later application of more advanced principles in the context of large organizations.

9.1.3 Design Assessment Metrics

Starting in the mid-seventies and lasting until the mid-nineties, intensive research was conducted on software quality metrics. In order to increase the quality of the metrics that were defined, [61], for instance, describes a set of properties that complexity metrics had to fulfill. However, it is observed that the availability of design assessment metrics for object-oriented systems (e.g. [60]) was not extended over the "object-oriented decade" although some of these metrics have been identified to be also applicable in a SOA setting. An example of the definition and application of specialized metrics in the area of service-oriented computing can be found in [166]. Types of granularity for single services are introduced as well as metrics for measuring them. However, these metrics are focused on the analysis of single services and not complete systems. In contrast, [59] proposes a system-wide metric. It applies the concept of coupling to complete *component*-oriented systems.

While the application of metrics to service-oriented systems is sufficiently motivated (e.g. by [166] and [167]), the lack of accepted special metrics might be due to both, the little information provided by their application and the complex way to measure them. In order to allow organizations to apply a new architectural style, such as the service-oriented style, an objective measure can support an organization to make its own experiences. This is why the work presented in chapter 3 builds on the work that was done for object-oriented systems and applies it to service orientation. This is achieved by considering properties of services that can be observed without knowing about a service's source code. A multitude of related metrics with descriptions of their interpretation is provided to enable a designer to assess a system from a more objective base. Additionally, a new class of metrics is proposed that address the control-centralization concept of service orientation. While the concept has been largely described (e.g. by [3] and [4]), the presented metrics are the first approach towards an objective description of this principle that is known to the author. All of the described metrics are complementary to any of the approaches that aim to calculating business values, such as the return-on-investment, for service-oriented systems (e.g. [168]).

9.2 Complementary Work

While this thesis was being researched, the service-oriented architectural style was widely perceived as still being developed for application within large organizations. And at the time this was written, it is still a work in progress.
This thesis aims to pave the way for this architectural style so that it can eventually be actualized and applied to structures within large organizations. This is why the focus of this work has been quite broad. Thus, the delineation of the presented concepts is not an easy task. However, this section discusses a set of related work that was chosen to situate the presented concepts in the complex domain of service-oriented computing.

9.2.1 Reference Architectures

The platform-independent reference architecture of chapter 5 and its platform-specific complement of chapter 7 aim at standardizing how composite applications should be

structured in order to apply all service-oriented principles to an application landscape that exposes its functionality in a heterogeneous way.
A different objective is the motivation behind reference architectures for semantic service provisioning. Notable work is the *Web Service Execution Environment* (WSMX) [169] and the corresponding *Web Service Modeling Ontology* (WSMO) [170] as well as the *Adaptive Service Grid* (ASG) (cf. e.g. [14]). The WSMO/WSMX approach and the core concepts of the ASG are overlapping reference architectures that address both design-time as well as run-time issues (cf. [171, p. 14]). In the following, the ASG will be related to the reference architecture of chapter 5.

The ASG project aims to increase an organization's flexibility by defining a platform for automated service composition and enactment. This objective is addressed by introducing semantic annotations in addition to the syntactic definition of services.
The general principle of the ASG is that "end-user applications or back-end systems act as service consumers. They send a semantic service request to the [ASG] platform. This request is syntactically similar to a semantic service description. However, while the description gives details about a existing service, a semantic service request specifies a desired service. The platform tries to find a service or composition of services which are able to meet a posed request" [14, p. 218]. This objective implies a slightly different understanding of the service-oriented architectural style. Control centralization by the means of a business process is not a major design objective. Furthermore, an agile and flexible collaboration among distributed service providers and service consumers is the postulated interaction scheme.
Implicit in the described interaction scheme is also a slightly different positioning of (back-end) application systems. An assumption that is contained in the ASG positioning is that application systems can be freely used as service consumers that comply with the protocol of the ASG *Facade*. However, in large organizations that have a strong IT governance, the flexible adoption of application systems is typically not allowed.

For the sake of connecting application systems to an ASG platform, part of the ASG definition are so-called *proxies*. Proxies that are deployed to the ASG platform and not to the application systems are introduced to allow for the integration of external (so-called *atomic*) service providers. They both handle the mediation of data types and, if necessary, protocol translation for heterogeneous service agents. Proxies are programmatic, remote representations of external services and can be parameterized by a so-called *mapping document*. Emphasis is put on data transformation via ontology mappings.
It is assumed that "accessible functionalities provide also an according functional interface description" [14, p. 74]. Such interface descriptions are annotated by non-semantic descriptions during a service-enabling process and the interface is described in a way that is suitable for realizing a proxy. Subsequently, a semantic service specification is added to allow for data transformations. Both descriptions are used as input for implementing the respective proxy. Besides the realization of data mappings via ontology mapping, an assumption of this model is that functionality of application systems is evocable by atomic services using an arbitrary service protocol. At this point it becomes obvious that the proxy concept and the concept of data exchange and data transformation are complementary approaches. The ASG proxy concept focuses on semantic description and data transformation, while the data exchange and data transformation layer aims to connect arbitrary application systems regardless of how functionality is exposed. The latter puts emphasis on the standardized and process-based integration of arbitrary application systems. The ASG prerequisite of application systems to expose their functionalities via a

functional interface becomes obsolete for service-enabling application systems by using the data exchange and data transformation layer. Additionally, it allows for the realization of arbitrary interaction requirements.

Aside from the differences that arise from addressing different issues, the two approaches also overlap. Both approaches acknowledge that a service provider that provides a scenario-specific service might not be immediately available in a service landscape. This is why a mechanism for composing application (or atomic) services to more problem-oriented services is included in both approaches. However, the difference between these approaches is that the ASG focuses on automatic service composition while the presented reference architecture solely incorporates the concept of service composition (which are called *service coordinations* in the approach of this thesis) and includes the specification of a runtime environment for such compositions.
The *service coordinations* are not generated ad hoc based on semantic annotations but, rather, are created during design time using a semi-automated algorithm that solely computes non-semantical data such as syntactic interface descriptions and functional dependencies.
In addition to the derivation of the service aggregation, the understanding of an aggregation's purpose is slightly different. While the ASG model understands its service compositions as processes with long-term characteristics[57], the service coordinations of this thesis are considered as multi-resource coordinations with (potentially) short-term characteristics (e.g., ACID transactions). It can be presumed that the ASG service composition serves both the purpose of the service coordination in addition to the service orchestration of the presented approach. As a result, no context handling, as established by the data repository, is required.
This difference is motivated by the different viewpoints that are taken by both projects. The work presented here had to consider today's reality of a large organization. There, business processes are defined outside the IT organization and application systems are heterogeneous, scantily described and limited in accessibility. The ASG project, on the other hand, was free to postulate more optimal circumstances as they *should* be[58]. Of course, automated, failure tolerant and "semantic-aware" service provisioning is preferable over the semi-automated and restricted service coordination used to address constraints. This is why both concepts should be considered as a sequential evolution. By enabling the pervasion of the service-oriented style through the concepts of this thesis, more advanced principles like semantic service provisioning can subsequently be applied. If the ASG reference architecture would be extended towards short-running service aggregation with integrated context handling it could later replace the service coordination layer. By keeping a central control model through a central service consumer for the ASG platform, the service orchestration layer could coexist with the process enactment means of the ASG reference architecture while the data exchange and data transformation layer is used as a provider for the ASG proxies.

Related to the data exchange and data transformation layer is the specification *Java Business Integration* (JBI) of [173]. JBI standardizes an integration platform that manages integration problems rather than solving integration problems. This is achieved by the

[57] This is implied by the fact that the reference implementation is realized with the process description language WS-BPEL [128].

[58] However, as a lesson learned the project did also postulate that obtaining a formal and exact specification of semantics of services is a "laborious task" [172, p. 17]. This supports our argument in chapter 6 that such descriptions are currently not obtainable within large organizations.

definition of a reference architecture for an enterprise service bus (ESB). The JBI standard defines a central messaging system called the *Normalized Message Router* (NMR). Service consumers post requests to that NMR. In-turn, the NMR routes the requests to components that are registered at the NMR. Components can either be deployed to the actual JBI-compliant platform (as so-called *Service Engines* (SEs)) or external services can be integrated via so-called *Binding Components* (BC). The actual functionality and purpose of integrated components is, however, not specified.
The JBI standard defines an architecture of an integration solution that is complementary to the platform-independent reference architecture of chapter 5. In particular, the data exchange and data transformation layer can be *realized* by using the JBI standard. This is because single integration services can easily be implemented as SEs.
In such an implementation scenario, the messaging principle of the JBI standard can be used to deliver messages to the single integration services. The implementation logic of the single integration services can be realized by using an SE-workflow engine. The integration flows could be realized by e.g. the use of an JBI BPEL-SE (cf. [173, p. 14]). The scenario-specific parameterization of the integration flows and services would then be made accessible to the single SEs through *Service Units*.

Additionally, SEs can act not just as service providers but also as service consumers. This is why the other components of the reference architecture can be realized in addition to JBI. In particular, the service coordination, the service orchestration and the eventing system components are candidates for SEs. A complementary standard that can be used for realizing service coordinations is the Service Component Architecture (SCA) (cf. [174]). If integrated by using a special SCA SE, service coordinations can be defined in a declarative way and improve the maintainability of a composite application. A data repository could eventually be realized by realizing a JBI *Shared Library*.
As the JBI standard also includes distributed transaction management, best effort, at-least-once and exactly-once in order communication semantics throughout the communication of all components, the standard seems to support the requirements of the reference architecture better than the SAP NetWeaver platform (cf. chapter 7). However, a more detailed analysis of the standard would be required in order to define a platform-specific reference that is aligned with the reference architecture of chapter 5.

An approach to an execution environment that focuses on transactional aspects and distinguishes an activity, a coordination and a transaction domain is the standard *composite application framework* (WS-CAF) of [30]. Here, transactional security and separation of long-running and short-running processes are incorporated. This approach is, however, composed of a set of protocols that a runtime environment might use. It does not provide a design-time reference architecture for composite applications. Therefore, it might only be used as part of a specification of a platform model that is based on the platform-independent reference architecture of chapter 5.

Hardly noticed, interesting work has been done by the "CBDI Forum" [175] in the area of SOAs. Being a closed forum for organizations to share best practices, the company behind the CBDI forum has developed "A Meta-Model for SOA" (cf. [176]) that was recently released to the public. The presented meta-model incorporates the idea of "process dominance" (cf. [15]) and service categorization into a meta-model that is suitable for building SOA-modeling tools. The holistic model focuses on allowing for the description of relations between business concepts, technical services and their realization. It does not define a reference architecture for software systems, though. However, the described

concepts and their relations provide a holistic structure that is complementary to the concepts described in this thesis.
The service-meta model that underlies the design methodology of chapter 4, for instance, is compatible with the meta-model of [176]. The model of chapter 4 introduces several types or levels of (aggregated) services. These levels of services could be expressed by the concept of CBDI *Service Domains*. Service domains are part of the *Policy Package* of the meta-model and confine architectural layers (cf. [176, p. 15]).
If the concepts and findings that are described throughout this thesis would be used for developing a holistic design tool for composite applications, the CBDI meta-model could be used as a mechanism for describing, representing and persisting models.

An enterprise-wide "quality software architecture" is described by the Quasar Enterprise approach in [6]. It focus on engineering application landscapes, rather than applications themselves, with a service-oriented approach. It describes a governance approach that supports the creation of application landscapes and governance structures for specific business architectures while incorporating service-oriented principles. In this scope, the approach is complementary to the software architecture and design approach described in this thesis. The approach presented here aims to facilitate the creation of composite applications in a given application landscape while the Quasar Enterprise approach aims to influence the application landscape. In this sense, Quasar Enterprise addresses similar needs as other enterprise architecture frameworks such as the *Zachman Framework* of [177], the *Open Group Architectural Framework* (TOGAF) of [178] or the *Integrated Architecture Framework* (IAF) of Cap Gemini Consulting [179] (cf. [6, pp. 82ff.]) do. In addition, it incorporates service-oriented principles.

Both the Quasar Enterprise approach and the concepts of this thesis share the emphasis placed on the significance of service orientation as an integration concept. Both describe the need for a flexible integration platform as an existential part of any service-oriented application landscape. However, the scopes of these approaches are different. While the Quasar Enterprise approach describes the required components of a reference architecture for an integration platform (cf. [6, pp. 234ff.]), this thesis additionally provides functional specifications for the required components. Additionally, it incorporates the integration architecture into a reference architecture for complete composite applications and describes an application design methodology.
In this sense, Quasar Enterprise provides a more abstract descriptive framework that resides "on top" of the concepts described here (cf. [6, p. 86]), while the JBI specification for instance describes a runtime framework "below" this work. If the first composite applications were built using the concepts described here, an approach like Quasar Enterprise's would be used to establish service orientation on an enterprise level. This way, service orientation would be delivered to enterprises through a bottom-up approach. Examining the experiences during this project, this seems to be the only viable approach. This is because organizations with a conservative IT approach will most likely only invest in a service-oriented enterprise architecture if some successful projects have already been conducted with the architectural style for software systems.

9.2.2 Service Design and Design Methodologies

The design methodology of chapter 6 is a methodology that supports the design of service-oriented applications. As outlined in chapter 2, service orientation is considered an ad-

vanced principle for application integration. This is why software engineering methodologies such as the Rational Unified Process (cf. [180]) or the so-called "V-model" (cf. [181]) are complimentary to this integration-centric design approach. While these latter methodologies are meant for steering complex projects for the development of – among others – application systems, is the design methodology of chapter 6 described for allowing the reuse of such application systems.

The methodology proposed in this thesis utilizes simple measures and more complex metrics at defined steps of the development process in order to assess the design as it was described up to the respective step. The aim is to improve the overall design in an iterative way by identifying design issues early in the process.
In [182], a complementary approach is proposed. The approach of [182] "is preferably suitable for measures that are relevant for project controlling. Here often the measures size, time, effort, defects and productivity are mentioned [...]. For these measures our approach supports automatic measurement and reuse" [182, p. 235]. Having a different focus, the design methodology proposed in this thesis does not include a framework for improved project controlling. However, the size measures described in section 3.2.1 could be included into the approach of [182] to allow making the progress of designing an actual composite application more transparent.

The authors of the "OASYS" methodology introduce in [183] an integration methodology that focuses on business process models as an input for the development of integration systems. As described in [184] is this methodology superior to other integration methodologies (such as e.g. ebXML of [185]) in terms of process-alignment and the development procedure.
The OASYS methodology is based on an architecture of "integration sub-systems" that are chosen based on an identified coupling mechanism (cf. [184, pp. 22ff.]). Possible coupling mechanisms include data integration and interface processing (cf. [183, p. 3] and section 2.1). Also a "function-oriented" integration mechanism can be chosen. While this latter principle is comparable to service-oriented application integration (cf. section 2.1) does the methodology not include the design of single services or of a hierarchy of services. Emphasis is put on the analysis of the business process and the identification of according integration sub-systems. However, the analysis and design of a centralized control model is not part of the methodology.
The sub-systems of the OASYS methodology are comparable with the integration services of the data exchange and data transformation layer of section 5.7. But the methodology does not include a list of design decisions or platform requirements for realizing an integration design. This why it can be considered more a methodology for the analysis of business processes with regards to related application integration needs rather than an integration methodology that involves the design of components and interfaces of a composite application.

The methodology of chapter 6 includes a service design algorithm that realizes an "assisted design-time composition" (cf. [14, pp. 95ff.]). This composition aims to mediate enterprise services which are part of a business process-based orchestration, and application services, that expose the functionality of application systems. In this sense the service coordination layer describes how service interface adoption can be realized.
The approach that is thereby taken is again focused on eased applicability. It requires human interaction and does not include a formal check of the mediating service aggregation. This is because no additional requirements in terms of input to the methodology

are defined. As discussed above, the ASG approach is complementary to the work in this matter.

An approach to service interface adoption is made in [122]. There, an algebra and a visual notation are introduced. These concepts address the problem of describing how *behavioral* interfaces can be linked through algebraic expressions. This scope already asserts that this approach does not rely on the structural interface definitions that typically are the only described artifacts. This is why it was considered to be not applicable in this thesis' context.
However, as soon as such descriptions are available and the work that was initiated by [122] is finished, a mechanism will be available that allows for the adaptation of several services in a service landscape with a given business process. In such a setting, business processes that integrate several application systems would be described by a behavioral interface. If a new requirement expressed by a business process is implemented, the mapping from the new process to the established one could be described by these mechanisms. This way the single-service-focused *scattering* of interfaces (cf. [122, p. 72]), as described by the service design algorithm of step 3 of the composite application design methodology, could be extended to include several enterprise services at once.
Additionally, the described mediation could be validated in terms of completeness in a semi-automated way. However, this is currently only a future option as this complementary approach is today still unfinished.

Another very interesting approach to service mediation is semantic service mediation. [111] describes how a one-to-multiple service matching can be automatically derived that can be used as a mediation between service consumers and service providers. While this approach does not rely on a behavioral interface, it derives mediations based on semantic descriptions of the service interfaces. As described above, semantic descriptions are not considered to be applicable in a first step. However, the approach of [111] applies the idea of correlation-based service composition to semantic service mediation. In this approach, a service is included in a service composition if it provides at least one unique output, the service request can provide all necessary input to the service and the requests are correlatable (cf. [111, pp. 493f.]). In this sense, this approach applies similar concepts as the service design methodology of step 3 of chapter 6. But while this approach is based solely on structural information and functional dependencies, [111] applies the concept to semantic descriptions of services. As soon as such descriptions are largely available, the algorithm of chapter 6 could be extended towards semantic service mediation by incorporating the findings of [111].

A completely different approach towards "a methodology for service architectures" is taken by Jones in [186]. It starts at the top of a domain (organizational structure, functions) in order to identify services. In the notion of Jones, "services have been hijacked by technology vendors trying to sell integration and development tools, which most normally focus on 'Business Process', 'Orchestration' or 'Web Services'" [186, p. 4]. Consequently, the understanding of a service is about "*what* the business does and place a boundary which all parties, but predominately the business can agree on" [186, p. 4]. In this notion, the methodology aims at understanding and structuring a company's business and its enterprise architecture. An enterprise architecture is, according to [186], able to deliver the services that have been identified on a business level with the understanding outlined above.
In this notion, the approach of [186] can be more considered a business modeling ap-

proach rather than a design methodology that can be used to align software design with the service-oriented architectural style. In this sense, the approach is more a service-focused complementary to the information system description approaches such as the "Architecture of Integrated Information Systems" (ARIS) (cf. [117]) or the "Semantic Object Model" (SOM) (cf. [93]). In this sense, it might be suitable for delivering the starting point for the design methodology of chapter 6. By also approaching the business-focused design with service-oriented principles, a more clear definition of necessary functionality and processes could be delivered and incorporated into composite applications.

9.3 Summary

This thesis describes the application of service orientation in the context of large corporations. As such, its objective is to analyze existing work in the domain of service orientation and to fill the gaps between the current state of research and real-life requirements.
This 'current state of research' is described above. This thesis adds to this work a unified framework that consists of a triple with a platform independent reference architecture, a design methodology and a platform-specific realization of the reference architecture.

The first element of the framework – the platform independent reference architecture – advances related work in this area towards an holistic applicability of the concepts. This is achieved by combining basic concepts, design patterns with putting the focus on business processes as central control instances. In parallel, the reference architecture puts emphasis on the integration layer which is often not considered in related work.

The second element of the framework – the design methodology for composite applications – is supported by a set of measures and metrics that are based on metrics known from object-orientation. This combination allows for supporting large projects in terms of incorporating service-oriented principles into the solution of real problems.

The last part of this framework is the application of all concepts that are described in this thesis to an enterprise-scale software platform. While more specialized research often focuses on implementing the described findings on basic or 'best-of-breed' platforms, this thesis considered the need of large corporations of using established platforms fitting their needs in terms of IT governance. Together, all these elements propose a unique way of easing the application of the service-oriented architectural style for large corporations.

10 Conclusion

This chapter concludes the thesis by summarizing the results and suggesting directions for future research. Additionally, the findings and experiences made during the dissertation and the associated project are described.

10.1 Summary

The discussion of the service-oriented architectural style has demonstrated that service-oriented principles can only partially be captured by an architectural style. Additionally, the principles that can be formalized are not different than what other architectural styles have incorporated. Namely, there is the component-oriented style that also incorporates the ideas of *composability*, *statelessness*, *contracts* and, partially, *discoverability*. The difference of the service-oriented architectural style, compared with component architectures, is much more based on *how* the components of an architecture are composed. It was determined that these soft design principles are *loose coupling*, *autonomy*, *abstraction* and *reusability*.

In order to analyze how soft design principles can be incorporated in the design of a system, the potential benefits of the service-oriented style were analyzed. It was thus determined that modifiability is a key promise of service orientation. In order to discuss how service orientation can contribute to the modifiability of a system, a set of metrics was introduced. In addition to the discussion of the single elements of the service-oriented style, the described metrics can be used to assess the design of service-oriented systems.

One *soft* aspect of service orientation is the reusability of services. It was demonstrated that large portions of the discussion regarding SOAs focus on reuse and the design of reusable services. In order to identify the significance of this concept, a statistical analysis was performed. This analysis revealed that several mechanisms, which are intended to improve the reusability of services, do not have any significant influence on reusability. The only design principle that proved significant in terms of reusability was service granularity. Based on these findings, recommendations for service-design approaches were formulated.

In order to integrate both the *hard* and *soft* principles of service orientation into a composite application, a reference architecture for composite applications was defined. The presented architecture standardizes the development of service-oriented applications in order to allow organizations to leverage service-oriented principles in actual projects. It describes both a meta-structure for the design of composite applications and a platform-independent specification for a virtual machine a composite application can be deployed on.
The presented reference architecture is aligned around the notion of *abstraction*. It puts a business process at the center and uses *loosely coupled* and *autonomous* services to establish a link between the actual business process and the heterogeneous application landscape. The link to heterogeneous application systems is facilitated by the means of a flexible integration layer. By emphasizing the reuse of application functionality, service orientation is introduced as a beneficiary style for application integration that allows for realizing adaptive applications in established and stable system landscapes. This way, the reference architecture becomes more applicable for organizations that aim at using

standard software.

In order to make the reference architecture applicable in actual projects, a design methodology for composite applications was presented. This methodology combines a top-down and bottom-up approach so as to translate an actual business process into the design of a composite application that is based on the described reference architecture.
The methodology includes a service design algorithm that focuses on minimal requirements in terms of its input while incorporating the findings of the statistical analysis on reusable service design. The methodology also utilizes several of the identified metrics for designing single components of the reference architecture in a given context.
By combining a top-down, business-process driven design with a constraints-driven bottom-up approach that focuses on reusing functionality distributed over heterogeneous application systems, building standardized composites becomes applicable for large organizations.

The reference architecture allows for a composite application design that considers an actual set of application systems while remaining independent from the platform on which a composite will be realized. In order to describe how to implement a design created using the described methodology, a platform-specific mapping for the reference architecture was defined. This mapping targeted SAP NetWeaver as the actual run-time platform. By describing the realization of the single components of the reference architecture on this platform, the design of composite applications can be implemented.

A case study was presented in order to demonstrate the applicability of the presented concepts. Based on a business process that was chosen due to business needs, the study described the suitability of the service-oriented architectural style for the realization of the actual requirements. Subsequently, the design methodology was applied to describe the design of a composite application in the heterogeneous environment in which the business process was used.
Using the previously identified metrics for service orientation, the design was then assessed. The actual realization of the design on the target platform SAP NetWeaver was also described. This demonstrated that the service-oriented architectural style can be applied in the context of heterogeneous real-world application landscapes.

10.2 Future Work

This thesis analyzed how architectural principles can be applied in the constrained, real world. It focused on deriving tangible descriptions of benefits that are associated with the service-oriented architectural style. By defining both a reference architecture and a methodology, practitioners are able to apply the service-oriented style. By developing this during a non-research industry project, the presented concepts had to accept certain pre-conditions and limited possibilities regarding requirement engineering. Notably, there was a lack of formal methods for process modeling. This opens a path for future work.

This thesis should be understood as "grounding" concepts that are discussed in academia today. Its aim is to pave the way for the broad application of service orientation. When the presented concepts have created a certain level of awareness about the concept in addition to a certain pervasion of the concept, a bottom-up loop should be entered. By entering into such an iteration, the described principles can be used to develop further concepts that integrate recent findings in the area of business process management (BPM) and semantic

service provisioning. If combined with a formal modeling and model-checking approach, the described concepts might be used to realize a semi-automated development approach. In particular, the mapping from workflow and interaction patterns to components of the reference architecture seems promising. Integrated into one tool, this combination might be beneficial.

In general, tool support is a broad area of work that should be entered on the basis of this thesis. First, there is the requirement for integrated development environments. By incorporating the notion of modeling methods, the described design methodology, especially its service design algorithm, can be supported by tools. This would certainly ease the application of the service-oriented style even more.

Work that focuses on runtime platforms should also be performed. As described, the reference architecture can be mapped to any arbitrary platform that supports its functional requirements. Experience, however, teaches that using a fragmented set of tools and platforms (even if they are marketed by one vendor under one common name) hinders actual projects. This is not due to conceptual deficits. Few technical issues of a platform can negatively impact projects very heavily. This is why an integrated platform that is aligned with the described reference architecture would increase efficiency and help the service-oriented architectural style to be broadly accepted in the context of industry enterprises. A promising candidate for such a realization would be a JBI-based enterprise service bus.

10.3 Conclusion

Without further structuring, service orientation as a style for software architectures seems to be hardly applicable. In particular if the expected benefits should be incorporated, additional guidelines and structure is needed. This is why the presented concepts are required in the context of big organizations.
It was observed that even small modeling input requirements proved to be a real challenge for an organization. The level of requirements that is necessary for applying the presented principles is achievable, though. This is how the application of the presented principles positively influences projects. If, for instance, the reference architecture should be used, there is no way to not explicitly describe a business process.
The fact that this description is then even used (in a modified way) during runtime, greatly influences the value organizations assign to modeling and structured requirements engineering. For this reason, **the application** of the service-oriented architectural style generates advantages that exceed the benefits that can be achieved by the style itself.

References

[1] B. Zrimsek and D. Prior, "Comparing the TCO of centralized vs. decentralized ERP," Gartner, Tech. Rep., January 2003, last accessed: December 10, 2008. [Online]. Available: http://www.bsg-global.com/assets/autodocs/insight/030124_Comparing%20the%20TCO%20of%20Centralized%20vs.%20Decentralized%20ERP.pdf

[2] A. Nori and R. Jain, "Composite applications: Process based application development." in *TES*, ser. Lecture Notes in Computer Science, A. P. Buchmann, F. Casati, L. Fiege, M. Hsu, and M.-C. Shan, Eds., vol. 2444. Springer, 2002, pp. 48–53.

[3] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch or why it's hard to build systems out of existing parts," in *Proceedings of the 17th International Conference on Software Engineering (ICSE'1995), Toronto; Canada*, 1995, pp. 179–185.

[4] D. S. Linthicum, *Next Generation Application Integration*. Boston, MA; USA: Addison-Wesley, 2004.

[5] M. P. Papazoglou, "Service-oriented computing: Concepts, characteristics and directions." in *Proceedings of the 4th International Conference on Web Information Systems Engineering (WISE'2003), 10-12 December 2003, Rome; Italy*. IEEE Computer Society, 2003, pp. 3–12.

[6] G. Engels, A. Hess, B. Humm, O. Juwig, M. Lohmann, J.-P. Richter, M. Voß, and J. Willkomm, *Quasar Enterprise - Anwendungslandschaften serviceorientiert gestalten*. dpunkt Verlag, 2008, vol. 1.

[7] SAP, "SAP Enterprise SOA," Website, 2007, last accessed: December 10, 2008. [Online]. Available: http://www.sap.com/platform/esoa/index.epx

[8] International Business Machines (IBM), "IBM SOA platform," Website, 2007, last accessed: December 10, 2008. [Online]. Available: http://www-306.ibm.com/software/solutions/soa/

[9] Oracle, "Oracle SOA Platform," Website, last accessed: December 10, 2008. [Online]. Available: http://www.oracle.com/technologies/soa/index.html?SC=NA05060039C0.GCM.7001.SOA.soa.br

[10] IONA, "IONA SOA Suite," Website, 2007, last accessed: December 10, 2008. [Online]. Available: http://open.iona.com/?gclid=CNOwhZrpi44CFQ0eEgodmhV7EQ

[11] T. Erl, *Servcie-Oriented Architecture*, ser. The Prentice Hall service-oriented computing series. Upper Saddle River, NJ; USA: Prentice Hall, Inc., February 2006, vol. Fourth Printing.

[12] H. Cervantes, L. Imag, and F. Hall, "Technical Concepts of Service Orientation," *Service-Oriented Software System Engineering: Challenges and Practices. Idea Group Publishing*, vol. 47, 2005.

[13] P. Fremantle, S. Weerawarana, and R. Khalaf, "Enterprise services," *Commun. ACM*, vol. 45, no. 10, pp. 77–82, 2002.

[14] D. Kuropka, P. Tröger, S. Staab, and M. Weske, Eds., *Semantic Service Provisioning*. Berlin, Germany: Springer, 2008.

[15] J. Schelp and R. Winter, "Towards a methodology for service construction," in *Proceedings of the 40th Hawaii International International Conference on Systems Science (HICSS-40 2007), , 3-6 January 2007, Waikoloa, Big Island, HI; USA*. IEEE Computer Society, 2007, pp. 64–70.

[16] H. Hofmeister and G. Wirtz, "A pattern taxonomy for business process integration oriented application integration," in *Proceedings of the 18th International Conference on Software Engineering & Knowledge Engineering (SEKE'2006), San Francisco Bay, CA; USA*, K. Zhang, G. Spanoudakis, and G. Visaggio, Eds., July 5-7 2006, pp. 114–119.

[17] ——, "Approaching a methodology for designing composite applications integrating legacy applications using an architectural framework," in *Proceedings of the EMISA Conference 2006 – Methoden, Konzepte und Technologien für die Entwicklung von dienstbasierten Informationssystemen, Beiträge des Workshops der GI-Fachgruppe Entwicklungsmethoden für Informationssysteme und deren Anwendung (EMISA'2006), Hamburg; Germany*, ser. LNI, M. Weske and M. Nüttgens, Eds., vol. 95. GI, October 2006.

[18] ——, "Using patterns to design composite applications," in *Proceedings of the International Conference on Enterprise Information Systems and Web Technologies (EISWT'2007), Orlando, FL; USA*, 2007.

[19] ——, "A multi-layered framework for pattern-aided composite application design," in *Proceedings of the 11th World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI'2007), Orlando, FL; USA*, vol. 1, 2007, pp. 54–60.

[20] ——, "Designing a platform-independent use-case for a composite application using a reference architecture," in *Proceedings of the 19th International Conference on Software Engineering & Knowledge Engineering (SEKE'2007), Boston, MA; USA*, July 2007.

[21] ——, "Supporting service-oriented design with metrics," in *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC'2008), Munich; Germany*. IEEE Computer Society, September 2008, pp. 191–200.

[22] ——, "Applying service-orientation through a reference architecture," *Journal of Systemics, Cybernetics and Informatics*, vol. 6, no. 1, pp. 80 – 90, 2008.

[23] G. D. Abowd, R. Allen, and D. Garlan, "Formalizing style to understand descriptions of software architecture," *ACM Trans. Softw. Eng. Methodol.*, vol. 4, no. 4, pp. 319–364, 1995.

[24] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages." *IEEE Trans. Software Eng.*, vol. 26, no. 1, pp. 70–93, 2000.

[25] M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plasil, G. Pomberger, W. Pree, M. Stal, and C. A. Szyperski, "What characterizes a (software) component?" *Software - Concepts and Tools*, vol. 19, no. 1, pp. 49–56, 1998.

[26] M. Fowler, "Inversion of Control Containers and the Dependency Injection pattern," vol. 23, 2004, last accessed: December 10, 2008. [Online]. Available: http://www.itu.dk/courses/VOP/E2006/8_injection.pdf

[27] "SOAP version 1.2 part 1: Messaging framework," W3C, April 2007, last accessed: December 10, 2008. [Online]. Available: http://www.w3.org/TR/2007/REC-soap12-part1-20070427/

[28] K. P. Birman, "Like it or not, web services are distributed objects," *Commun. ACM*, vol. 47, no. 12, pp. 60–62, 2004.

[29] W. Vogels, "Web services are not distributed objects," *IEEE Internet Computing*, vol. 7, pp. 59–66, November/December 2003.

[30] D. Bunting, M. Chapman, O. Hurley, M. Little, E. N. J. Mischkinsky, J. Webber, and K. Swenson, "Web services composite application framework (WS-CAF) ver. 1.0," Arjuna Technologies Limited, Fujitsu Software, IONA Technologies PLC, Oracle Corp and Sun Microsystems, Tech. Rep., 2003.

[31] C. Peltz, "Web services orchestration and choreography." *IEEE Computer*, vol. 36, no. 10, pp. 46–52, 2003.

[32] D. Garlan, R. T. Monroe, and D. Wile, "ACME: an architecture description interchange language." in *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative*, J. H. Johnson, Ed. IBM, 1997, p. 7.

[33] J. S. Kim and D. Garlan, "Analyzing architectural styles," January 2007, submitted to Elsevier in Jan. 2007; last accessed: December 2008. [Online]. Available: http://acme.able.cs.cmu.edu/pubs/uploads/pdf/jss2006.pdf

[34] *The ACME Project*, Carnegie Mellon University's ABLE Project, 2007, last accessed: December 10, 2008. [Online]. Available: http://acme.able.cs.cmu.edu/index.html

[35] A. Yanchuk, A. Ivanyukovich, and M. Marchese, "A lightweight formal framework for service-oriented applications design." in *Proceedings of the 3rd International Conference on Service-Oriented Computing (ICSOC'2005), Amsterdam; The Netherlands*, ser. Lecture Notes in Computer Science, B. Benatallah, F. Casati, and P. Traverso, Eds., vol. 3826. Springer, 2005, pp. 545–551.

[36] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, and C. Ferris, "Web service architecture," 2004, last accessed: December 10, 2008. [Online]. Available: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/wsa.pdf

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison Wesley, 1996.

[38] D. Garlan, S.-W. Cheng, and A. J. Kompanek, "Reconciling the needs of architectural description with object-modeling notations," *Sci. Comput. Program.*, vol. 44, no. 1, pp. 23–49, 2002.

[39] *OMG Unified Modelling Langauge Specification*, Object Management Group, 2003, last accessed: December 10, 2008. [Online]. Available: http://www.omg.org/docs/formal/03-03-01.pdf

[40] W.-T. Tsai, C. Fan, Y. Chen, R. A. Paul, and J.-Y. Chung, "Architecture classification for SOA-based applications." in *Proceedings of the 9th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'2006), April 2006, Gyeongju; Korea*. IEEE Computer Society, April 2006, pp. 295–302.

[41] C. Herault, G. Thomas, and P. Lalanda, "Mediation and enterprise service bus: A position paper," in *Proceedings of the First International Workshop on Mediation in Semantic Web Service (MEDIATE'2005), Amsterdam; The Netherlands*, 2005.

[42] M.-T. Schmidt, B. Hutchison, P. Lambros, and R. Phippen, "The enterprise service bus: making service-oriented architecture real," *IBM Syst. J.*, vol. 44, no. 4, pp. 781–797, 2005.

[43] G. Wiederhold, "Mediators in the architecture of future information systems." *IEEE Computer*, vol. 25, no. 3, pp. 38–49, 1992.

[44] M. Dumas, T. Fjellheim, S. Milliner, and J. Vayssière, "Event-based coordination of process-oriented composite applications." in *Business Process Management*, W. M. P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, Eds., vol. 3649, 2005, pp. 236–251.

[45] A. Einstein, "Not everything that can be counted counts, and not everything that counts can be counted." last accessed: December 10, 2008. [Online]. Available: http://www.quotationspage.com/quote/26950.html

[46] *IEEE Standard 729-1983: IEEE Standard Glossary of Software Engineering Terminology*. IEEE Computer Society, 1983.

[47] *IEEE Standard Glossary of Software Engineering Terminology/IEEE Std 610.12-1990*. IEEE Computer Society, 1991.

[48] L. Dobrica and E. Niemelä, "A survey on software architecture analysis methods." *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 638–653, 2002.

[49] ISO/IEC, "Information technology – software product evaluation," International Organization of Standardisation and International Electrotechnical Commission, Tech. Rep., 2004.

[50] K. Lee and S. J. Lee, "A quantitative evaluation model using the ISO/IEC 9126 quality model in the component based development process." in *ICCSA (4)*, ser. Lecture Notes in Computer Science, M. L. Gavrilova, O. Gervasi, V. Kumar, C. J. K. Tan, D. Taniar, A. Laganà, Y. Mun, and H. Choo, Eds., vol. 3983. Springer, 2006, pp. 917–926.

[51] P. Bengtsson, N. H. Lassing, J. Bosch, and H. van Vliet, "Architecture-level modifiability analysis (ALMA)." *Journal of Systems and Software*, vol. 69, no. 1-2, pp. 129–147, 2004.

[52] F. Cristian, "Understanding fault-tolerant distributed systems," *Commun. ACM*, vol. 34, no. 2, pp. 56–78, 1991.

[53] A. Abran, A. Khelifi, W. Suryn, and A. Seffah, "Usability meanings and interpretations in ISO standards," *Software Quality Control*, vol. 11, no. 4, pp. 325–338, 2003.

[54] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," *Open Grid Service Infrastructure WG, Global Grid Forum, June*, vol. 22, p. 2002, 2002.

[55] L. Bratthall and P. Runeson, "A taxonomy of orthogonal properties of software architecture," *Proc. 2nd Nordic Software Architecture Workshop. Ronneby, Aug*, 1999.

[56] N. F. Schneidewind, "Software metrics model for quality control." in *IEEE METRICS*. IEEE Computer Society, 1997, pp. 127–136.

[57] T. J. McCabe, "A complexity measure." *IEEE Trans. Software Eng.*, vol. 2, no. 4, pp. 308–320, 1976.

[58] L. C. Briand, S. Morasca, and V. R. Basili, "Property-based software engineering measurement," *IEEE Trans. Softw. Eng.*, vol. 22, no. 1, pp. 68–86, 1996.

[59] H. Washizaki, T. Nakagawa, Y. Saito, and Y. Fukazawa, "A coupling-based complexity metric for remote component-based software systems toward maintainability estimation," in *Proceedings of the 13th Asia Pacific Software Engineering Conference, Bangalore; India*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 79–86.

[60] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design." *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[61] E. J. Weyuker, "Evaluating software complexity measures," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, pp. 1357–1365, 1988.

[62] J. S. Poulin, "The business case for software reuse: Reuse metrics, economic models, organizational issues, and case studies," in *ICSR*, ser. Lecture Notes in Computer Science, M. Morisio, Ed., vol. 4039. Springer, 2006, p. 439.

[63] S. Karunanithi and J. Bieman, "Candidate reuse metrics for object oriented and Ada software," *Proceedsings of the 1st International Software Metrics Symposium, Baltimore, MD; USA*, pp. 120–128, May 1993.

[64] N. F. Schneidewind, "Analysis of error processes in computer software," in *Proceedings of the international conference on Reliable software*. New York, NY, USA: ACM Press, 1975, pp. 337–346.

[65] W. T. Tsai, D. Zhang, Y. Chen, H. Huang, R. A. Paul, and N. Liao, "A software reliability model for web services," in *Proceedings of the 8th IASTED International Conference on Software Engineering and Applications (IASTED'2004), Cambridge, MA; USA*, M. H. Hamza, Ed. IASTED/ACTA Press, 2004, pp. 144–149.

[66] D. Hamlet, D. Mason, and D. Woit, "Theory of software reliability based on components," in *Proceedings of the 23rd International Conference on Software Engineering (ICSE'2001), Toronto; Canada*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 361–370.

[67] H. Singh, V. Cortellessa, B. Cukic, E. Gunel, and V. Bharadwaj, "A bayesian approach to reliability prediction and assessment of component based systems," in *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'2001), Hong Kong; China*. IEEE Computer Society, 2001, pp. 12–21.

[68] R. Tripathi and R. Mall, "Early stage software reliability and design assessment," in *Proceedings of the 12th Asia Pacific Software Engineering Conference (APSEC'2005), Taipei; Taiwan.* IEEE Computer Society, 2005, pp. 619–628.

[69] A. Dimov and S. Punnekkat, "On the estimation of software reliability of component-based dependable distributed systems," in *QoSA/SOQUA*, ser. Lecture Notes in Computer Science, R. Reussner, J. Mayer, J. A. Stafford, S. Overhage, S. Becker, and P. J. Schroeder, Eds., vol. 3712. Springer, 2005, pp. 171–187.

[70] V. Cortellessa, H. Singh, and B. Cukic, "Early reliability assessment of UML based software models," in *Proceedings of the 3rd international workshop on Software and performance (WOSP'2002), Rome; Italy*, 2002, pp. 302–309.

[71] A. S. Tanenbaum and M. van Steen, *Distributed Systems - Principals and Paradigms.* Upper Saddle River, NJ USA: Prentice Hall, Inc., 2001.

[72] J. Gray and A. Reuther, *Transaction Processing: Concepts and Techniques.* San Mateo, CA USA: Morgan Kaufmann, 1992.

[73] P. Grefen, J. Vonk, and P. Apers, "Global transaction support for workflow management systems: from formal specification to practical implementation," *The VLDB Journal*, vol. 10, no. 4, pp. 316–333, 2001.

[74] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, 1985.

[75] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Commun. ACM*, vol. 17, no. 11, pp. 643–644, 1974.

[76] Y. Liu, I. Gorton, L. Bass, C. Hoang, and S. Abanmi, "MEMS: A method for evaluating middleware architectures." in *QoSA*, ser. Lecture Notes in Computer Science, C. Hofmeister, I. Crnkovic, and R. Reussner, Eds., vol. 4214. Springer, 2006, pp. 9–26.

[77] R. Kazman, M. H. Klein, M. Barbacci, T. A. Longstaff, H. F. Lipson, and S. J. Carrière, "The architecture tradeoff analysis method." in *Proceedings of the 4th International Conference on Engineering of Complex Computer Systems (ICECCS'1998), Monterey, CA; USA.* IEEE Computer Society, 1998, pp. 68–78.

[78] M. Papazoglou and W. Van Den Heuvel, "Service-oriented design and development methodology," *International Journal of Web Engineering and Technology*, vol. 2, no. 4, pp. 412–442, 2006.

[79] A. Erradi, S. Anand, and N. N. Kulkarni, "SOAF: An architectural framework for service definition and realization," in *Proceedings of the 2006 IEEE International Conference on Services Computing (SCC'2006), Chicago, IL; USA*, 2006, pp. 151–158.

[80] L. H. Etzkorn, W. E. Hughes, and C. G. Davis, "Automated reusability quality analysis of OO legacy software," *Information & Software Technology*, vol. 43, no. 5, pp. 295–308, 2001.

[81] C. Hentrich and U. Zdun, "Patterns for process-oriented integration in service-oriented architectures," in *Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPLoP'2006),Irsee Monastery; Germany*, 2006.

[82] S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," in *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM Press, 1991, pp. 197–211.

[83] L. C. Briand, J. W. Daly, and J. Wüst, "A unified framework for cohesion measurement in object-oriented systems," *Empirical Software Engineering*, vol. 3, no. 1, pp. 65–117, 1998.

[84] B. Henderson-Sellers, *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1995.

[85] L. Walton, "Lack of cohesion in methods," last accessed: December 10, 2008. [Online]. Available: http://eclipse-metrics.sourceforge.net/descriptions/LackOfCohesionInMethods.html

[86] H. Reijers, "A cohesion metric for the definition of activities in a workflow process," in *CaiSE/IFIP8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Desing (EMMSAD'2003), Velden; Austria.*, 2003.

[87] C. Legner and T. Vogel, "Design principles for B2B services - an evaluation of two alternative service designs," in *Proceedings of the 2007 IEEE International Conference on Services Computing (SCC'2007), Salt Lake City, UT; USA*. IEEE Computer Society, 2007, pp. 372–379.

[88] R. Winter and R. Fischer, "Essential layers, artifacts, and dependencies of enterprise architecture," in *Workshops of the 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'2006), Hong Kong; China*. IEEE Computer Society, 2006, p. 30.

[89] U. Zdun, C. Hentrich, and S. Dustdar, "Modeling process-driven and service-oriented architectures using patterns and pattern primitives," *ACM Trans. Web*, vol. 1, no. 3, p. 14, 2007.

[90] A. Arsanjani and A. Allam, "Service-oriented modeling and architecture for realization of an SOA," in *Proceedings of the 2006 IEEE International Conference on Services Computing (SCC'2006), Chicago, IL; USA*, 2006, p. 521.

[91] J. Miller and J. Mukerji, "OMG, MDA Guide Version 1.0.1," Whitepaper, 2003, last accessed: December 10, 2008. [Online]. Available: http://www.omg.org/docs/omg/03-06-01.pdf,

[92] S. Lippe, U. Greiner, and A. Barros, "A survey on state of the art to facilitate modelling of cross-organisational business processes," in *Proceedings of the 2nd GI Workshop XML4BPM*, M. Nüttgens and J. Mendling, Eds., March 2005, pp. 7–22.

[93] O. K. Ferstl and E. J. Sinz, *Grundlagen der Wirtschaftsinformatik*, 4th ed. München, GER: Oldenburg Verlag, 2001.

[94] D. Martin, D. Wutke, T. Scheibler, and F. Leymann, "An EAI pattern-based comparison of spaces and messaging," in *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC'2007), Annapolis, MD; USA*. IEEE Computer Society, 2007.

[95] G. Hohpe and B. Woolf, *Enterprise Integration Patterns*, ser. The Addison Wesley Signature Series. Pearson Education Inc., 2004.

[96] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst, "Workflow data patterns: identification, representation and tool support," in *ER*, ser. Lecture Notes in Computer Science, L. M. L. Delcambre, C. Kop, H. C. Mayr, J. Mylopoulos, and O. Pastor, Eds., vol. 3716. Springer, 2005, pp. 353–368.

[97] W.-J. van den Heuvel, J. van Hillegersberg, and M. P. Papazoglou, "A methodology to support web-services development using legacy systems," in *Proceedings of the IFIP TC8 / WG8.1 Working Conference on Engineering Information Systems in the Internet Context*. Deventer, The Netherlands, The Netherlands: Kluwer, B.V., 2002, pp. 81–103.

[98] A. P. Barros, M. Dumas, and A. H. M. ter Hofstede, "Service interaction patterns," in *Business Process Management*, W. M. P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, Eds., vol. 3649, 2005, pp. 302–318.

[99] P. Vilím, "Batch processing with sequence dependent setup times: new results," in *Proceedings of the 4th Workshop of Constraint Programming for Decisionand Control (CPDC'2002), Gliwice; Poland*, 2002.

[100] R. Kossmann, "An architectural framework for semantic inter-operability in distributed object systems," in *Business Object Design and Implementation*, J. Sutherland, D. Patel, C. Casanave, G. Hollowll, and J. Miller, Eds. Springer- Verlag London, 1995.

[101] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-oriented software architecture: A system of patterns*. John Wiley & Sons, Inc. New York, NY, USA, 1996.

[102] S. K. Stegemann, B. Funk, and T. Slotos, "A blackboard architecture for workflows," in *CAiSE Forum*, ser. CEUR Workshop Proceedings, J. Eder, S. L. Tomassen, A. L. Opdahl, and G. Sindre, Eds., vol. 247. CEUR-WS.org, 2007.

[103] B. Hofreiter, C. Huemer, and K.-D. Naujok, "UN/CEFACT's business collaboration framework - motivation and basic concepts," 2004. [Online]. Available: citeseer.ist.psu.edu/hofreiter04uncefacts.html

[104] N. Carriero and D. Gelernter, "Linda in context," *Commun. ACM*, vol. 32, no. 4, pp. 444–458, 1989.

[105] *JS - JavaSpaces^{TM} Service Specification*, Sun Microsystems, Inc., 2002, last accessed: December 10, 2008. [Online]. Available: http://www.cs.princeton.edu/courses/archive/fall99/cs597b/docs/jxpdoc1_0/specs/js-spec/js.pdf

[106] R. Lämmel, "A semantical approach to method-call interception," in *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'2002), Twente; The Netherlands*. ACM Press, Apr. 2002, pp. 41–55.

[107] G. Kaufman, "Pragmatic ECAD data integration," *SIGDA Newsl.*, vol. 20, no. 1, pp. 60–81, 1990.

[108] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.

[109] D. D. Chamberlin and R. F. Boyce, "Sequel: A structured english query language," in *Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, May 1-3, 1974, 2 Volumes*, R. Rustin, Ed. ACM, 1974, pp. 249–264.

[110] *RosettaNet Partner Interface Processes*. [Online]. Available: http://www.rosettanet.org/RosettaNet/Rooms/DisplayPages/LayoutInitial?Container=com.webridge.entity.Entity[OID[9A6EEA233C5CD411843C00C04F689339]]

[111] L. Zeng, B. Benatallah, G. T. Xie, and H. Lei, "Semantic service mediation," in *ICSOC*, ser. Lecture Notes in Computer Science, A. Dan and W. Lamersdorf, Eds., vol. 4294. Springer, 2006, pp. 490–495.

[112] G. Decker, "Bridging the gap between business processes and existing it functionality," in *Proceedings of the First International Workshop on Design of Service-Oriented Applications (WDSOA'2005)*, 2005.

[113] A. Charfi and M. Mezini, "Hybrid web service composition: business processes meet business rules," in *Proceedings of the 2nd International Conference on Service-Oriented Computing (ICSOC'2004),New York, NY; USA*, M. Aiello, M. Aoyama, F. Curbera, and M. P. Papazoglou, Eds. ACM, 2004, pp. 30–38.

[114] F. Rosenberg and S. Dustdar, "Business rules integration in BPEL - a service-oriented approach," in *CEC*. IEEE Computer Society, 2005, pp. 476–479.

[115] C. Nagl, F. Rosenberg, and S. Dustdar, "VIDRE - a distributed service-oriented business rule engine based on RuleML," in *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'2006), Hong Kong; China*. IEEE Computer Society, 2006, pp. 35–44.

[116] G. Wagner, G. Antoniou, S. Tabet, and H. Boley, "The abstract syntax of RuleML – towards a general web rule language framework," in *Web Intelligence*. IEEE Computer Society, 2004, pp. 628–631.

[117] A. W. Scheer, *ARIS - Vom Geschäftsprozess zum Anwendungssystem*, 3rd ed. Berlin, GER: Springer, 1998.

[118] W. M. P. van der Aalst, J. Desel, and E. Kindler, "On the semantics of EPCs: A vicious circle," in *EPK*, M. Nüttgens and F. J. Rump, Eds. GI-Arbeitskreis Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, 2002, pp. 71–79.

[119] B. Benatallah, M.-S. Hacid, A. Léger, C. Rey, and F. Toumani, "On automating web services discovery," *VLDB J.*, vol. 14, no. 1, pp. 84–96, 2005.

[120] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara, "Semantic matching of web services capabilities," in *International Semantic Web Conference*, ser. Lecture Notes in Computer Science, I. Horrocks and J. A. Hendler, Eds., vol. 2342. Springer, 2002, pp. 333–347.

[121] R. Eshuis, P. W. P. J. Grefen, and S. Till, "Structured service composition." in *Business Process Management*, ser. Lecture Notes in Computer Science, S. Dustdar, J. L. Fiadeiro, and A. P. Sheth, Eds., vol. 4102. Springer, 2006, pp. 97–112.

[122] M. Dumas, M. Spork, and K. Wang, "Adapt or perish: algebra and visual notation for service interface adaptation," in *Business Process Management*, ser. Lecture Notes in Computer Science, S. Dustdar, J. L. Fiadeiro, and A. P. Sheth, Eds., vol. 4102. Springer, 2006, pp. 65–80.

[123] J. Schaffner, H. Meyer, and C. Tosun, "A semi-automated orchestration tool for service-based business processes," in *ICSOC Workshops*, ser. Lecture Notes in Computer Science, A. Dan and W. Lamersdorf, Eds., vol. 4294. Springer, 2006, pp. 50–61.

[124] D. Kuropka and M. Weske, "Towards a service composition and enactment platform," *International Journal of Business Process Integration and Management*, vol. 2, no. 2, pp. 102 – 108, 2007.

[125] L. Zeng, B. Benatallah, H. Lei, A. H. H. Ngu, D. Flaxer, and H. Chang, "Flexible composition of enterprise web services," *Electronic Markets*, vol. 13, no. 2, 2003.

[126] J. Dehnert and W. M. P. van der Aalst, "Bridging the gap between business models and workflow specifications." *Int. J. Cooperative Inf. Syst.*, vol. 13, no. 3, pp. 289–332, 2004.

[127] F. Puhlmann and M. Weske, "Interaction soundness for service orchestrations," in *ICSOC*, ser. Lecture Notes in Computer Science, A. Dan and W. Lamersdorf, Eds., vol. 4294. Springer, 2006, pp. 302–313.

[128] *OASIS WSBPEL Technical Committee: Web Services Business Process Execution Language 2.0*, 31.02.2007. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/

[129] H. Meyer and D. Kuropka, "Requirements for automated service composition," in *Business Process Management Workshops*, ser. Lecture Notes in Computer Science, J. Eder and S. Dustdar, Eds., vol. 4103. Springer, 2006, pp. 447–458.

[130] N. Russell, W. M. P. van der Aalst, and A. H. M. ter Hofstede, "Workflow exception patterns," in *CAiSE*, ser. Lecture Notes in Computer Science, E. Dubois and K. Pohl, Eds., vol. 4001. Springer, 2006, pp. 288–302.

[131] R. Soley, "Model Driven Architecture," Whitepaper, November 2005, last accessed: December 10, 2008. [Online]. Available: ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf

[132] "SAP AG," 2007, last accessed: December 10, 2008. [Online]. Available: http://www.sap.com

[133] "SAP NetWeaver." [Online]. Available: http://www.sap.com/platform/netweaver/index.epx

[134] S. Bruckert and D. Grasman, "The benefits of SAP NetWeaver," April 2003, last accessed: December 10, 2008. [Online]. Available: http://www.sap.info/resources/RFILE216463f265dd300c38.pdf

[135] *Web Services Description Language (WSDL) 1.1*, W3C, March 2001. [Online]. Available: http://www.w3.org/TR/wsdl

[136] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1," RFC 2616, June 1999, last accessed: December 10, 2008. [Online]. Available: http://tools.ietf.org/html/rfc2616

[137] Sun, "The Java™ language specification," last accessed: December 10, 2008. [Online]. Available: http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf

[138] SAP, "ABAP," Website, last accessed: December 10, 2008. [Online]. Available: https://www.sdn.sap.com/irj/sdn/abap

[139] SAP, "SAP Business Workflow," SAP AG, 1997, last accessed: December 10, 2008.

[140] Sun Microsystems, "Java 2 platform enterprise edition specification, v1. 3." *Mountain View, CA*, 2001, last accessed: December 10, 2008. [Online]. Available: http://java.sun.com/j2ee/j2ee-1_3-fr-spec.pdf

[141] SAP, "SAP NetWeaver Manual," Website, 2004, last accessed: December 10, 2008. [Online]. Available: http://help.sap.com

[142] W3C, "Extensible Markup Language (XML) 1.0 (Fourth Edition)," Specification, August 2006, last accessed: December 10, 2008. [Online]. Available: http://www.w3.org/TR/2006/REC-xml-20060816/

[143] SAP, "SAP Exchange Infrastructure 3.0 – technical infrastructure," SAP AG, Tech. Rep., 2004, last accessed: December 10, 2008. [Online]. Available: http://help.sap.com/bp_bpmv130/Documentation/Planning/TechnicalInfrasture.pdf

[144] W3C, "XML schema part 0: Primer second edition," Standard, October 2004, last accessed: December 10, 2008. [Online]. Available: http://www.w3.org/TR/xmlschema-0/

[145] *JSR 112: J2EE Connector Architecture 1.5*, Sun Microsystems Inc. [Online]. Available: http://www.jcp.org/en/jsr/detail?id=112

[146] M. Herger, "Composite application framework – building blocks for realizing the ESA," in *America's SAP User Group Conference*, April 2003.

[147] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform resource identifier (URI): Generic syntax," RFC 3986, January 2005, last accessed: December 10, 2008. [Online]. Available: http://tools.ietf.org/html/rfc3986

[148] T. Berners-Lee and D. Connolly, "Hypertext markup language - 2.0," RFC 1866, November 1996, last accessed: December 10, 2008. [Online]. Available: http://tools.ietf.org/html/rfc1866

[149] P. Schler, "SAP enterprise portal architecture and technology," Documentation, Nov 2002, last accessed: December 10, 2008. [Online]. Available: https://www.sdn.sap.com/irj/sdn/go/portal/prtroot/docs/library/uuid/e308bc90-0201-0010-0ba8-b4bd3c9c6f62

[150] A. Abdelnur and S. Hepper, "The Java community process, JSR 168," 2003, last accessed: December 10, 2008. [Online]. Available: http://www.swe.uni-linz.ac.at/teaching/lva/ws04-05/seminar/Java%20Specification%20Request%20168.pdf

[151] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[152] J. Postel, "Simple mail transfer protocol," RFC 821, August 1982, last accessed: December 10, 2008. [Online]. Available: http://tools.ietf.org/html/rfc821

[153] M. Lottor, "Simple file transer protocol," RFC, September 1983, last accessed: December 10, 2008. [Online]. Available: http://tools.ietf.org/html/rfc913

[154] J. Clark, "XSL transformations (XSLT)," W3C Recommendation, November 1999, last accessed: December 10, 2008. [Online]. Available: http://www.w3.org/TR/xslt

[155] P. Wohed, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede, "Analysis of web services composition languages: the case of BPEL4WS," in *ER*, ser. Lecture Notes in Computer Science, I.-Y. Song, S. W. Liddle, T. W. Ling, and P. Scheuermann, Eds., vol. 2813. Springer, 2003, pp. 200–215.

[156] SAP, "Business Rules Management Roadmap," in *SAP TechEd 2008*, 2008, last accessed: December 10, 2008. [Online]. Available: https://www.sdn.sap.com/irj/sdn/nw-rules-management

[157] SAP Help, "Implementing web dynpro callable objects," Manual, 2007, last accessed: December 10, 2008. [Online]. Available: http://help.sap.com/saphelp_nw70/helpdata/en/43/e085d6421a4d9de10000000a155369/frameset.htm

[158] ——, "Starting a process using web services," Manual, 2007, last accessed: December 10, 2008. [Online]. Available: http://help.sap.com/saphelp_nw04s/helpdata/en/fd/afb4429027da11e10000000a155106/frameset.htm

[159] British Government Office of Government Commerce, "ITIL IT-Infrastructure Library," last accessed: December 10, 2008. [Online]. Available: http://www.itil.co.uk/

[160] International Business Machnines Corporation, "IBM Lotus software," Website, December 2007, last accessed: December 10, 2008. [Online]. Available: http://www-306.ibm.com/software/lotus/

[161] ECMA, "C++/CLI language specification," Whitepaper, 2005, last accessed: December 10, 2008. [Online]. Available: http://www.plumhall.com/C++-CLI%20draft%201.14.pdf

[162] H. Hofmeister, "A server with a core using a virtual file system and a method for securely redirecting a persistent storage device operation to a middleware infrastructure," European Patent Filing; publication number 1988473, 2008, application No. 07009044.4-1225.

[163] Konplan GmbH, "XInotes," Product Description, Feb. 2007, last accessed: December 10, 2008. [Online]. Available: http://www.konplan.com/public1/downloads/konplan_xinotes_en.pdf

[164] Architecture Board ORMSC, "Model Driven Architecture," Whitepaper, July 2001, last accessed: December 10, 2008. [Online]. Available: http://www.omg.org/docs/ormsc/01-07-01.pdf

[165] M. P. Papazoglou and J. Yang, "Design methodology for web services and business processes." in *TES*, ser. Lecture Notes in Computer Science, A. P. Buchmann, F. Casati, L. Fiege, M. Hsu, and M.-C. Shan, Eds., vol. 2444. Springer, 2002, pp. 54–64.

[166] D. Rud, S. Mencke, A. Schmietendorf, and R. Dumke, "Granularitätsmetriken für serviceorientierte Architekturen," in *DASMA Software Metrik Kongress (METRIKON'2007)*, 2007.

[167] M. Stutz and S. Aier, "Vorgehensmodell zur fachlichen bewertung serviceorientierter architekturen," in *Multikonferenz Wirtschaftsinformatik*, M. Bichler, T. Hess, H. Krcmar, U. Lechner, F. Matthes, A. Picot, B. Speitkamp, and P. Wolf, Eds. GITO-Verlag, Berlin, 2008.

[168] E. A. Marks and M. Bell, *Service-Oriented Architecture (SOA): A Planning and Implementation Guide for Business and Technology.* New York, NY, USA: John Wiley & Sons, Inc., 2006.

[169] A. Haller, E. Cimpian, A. Mocan, E. Oren, and C. Bussler, "WSMX – a semantic service-oriented architecture," in *Proceedings of the 2005 IEEE International Conference on Web Services (ICWS'2005), Orlando, FL; USA.* IEEE Computer Society, 2005, pp. 321–328.

[170] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel, "Web service modeling ontology," *Applied Ontology*, vol. 1, no. 1, pp. 77–106, 2005.

[171] H. Meyer, M. Eisenbarth, G. Laures, and K. Jank, "Adaptive services grid deliverable d6.v-1 – reference architecture: Requirements, current efforts and design," March 2007, last accessed: December 10, 2007. [Online]. Available: http://tb0.asg-platform.org/download/downloadrequest.php?asgdocument=D6.V-1.pdf

[172] D. Kuropka and M. Weske, "Implementing a semantic service provision platform - concepts and experiences," *Wirtschaftsinformatik*, vol. 50, no. 1, pp. 16–24, January 2008.

[173] R. Ten-Hove and P. Walker, "The Java community process JSR, 208 - java business integration (JBI)," 2005, last accessed: December 10, 2008. [Online]. Available: http://www.jcp.org/en/jsr/detail?id=208

[174] M. Beisiegel, H. Blohm, D. Booz, M. Edwards, O. Hurley, S. Ielceanu, A. Miller, A. Karmarkar, A. Malhotra, J. Marino, M. Nally, E. Newcomer, S. Patil, G. Pavlik, M. Raepple, M. Rowley, K. Tam, S. Vorthmann, P. Walker, and L. Waterman., "SCA Service Component Architecture - assembly model specification v1.0," Whitepaper, March 2007, last accessed: December 10, 2008. [Online]. Available: http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications

[175] Everware-CBDI Research, "CBDI Forum," 2007, last accessed: December 10, 2008. [Online]. Available: http://www.cbdiforum.com/

[176] ——, "CBDI-SAE meta model for SOA version 2.0," 2007, last accessed: December 10, 2008. [Online]. Available: http://www.cbdiforum.com/public/CBDI_SAE_META_MODEL_FOR_SOA_V2.0.pdf

[177] J. A. Zachman, "A framework for information systems architecture," *IBM Systems Journal*, vol. 38, no. 2/3, pp. 454–470, 1999.

[178] The Open Group, *The Open Group Architecture Framework*. The Open Group, 2006, last accessed: December 10, 2008. [Online]. Available: http://www.opengroup.org/architecture

[179] Cap Gemini Consulting, "Architecture and the integrated architecture framework," Brochure, August 2006, last accessed: December 10, 2008. [Online]. Available: http://www.capgemini.com/resources/thought_leadership/architecture_and_the_integrated_architecture_framework/?d=1

[180] P. Kruchten, *The Rational Unified Process: An Introduction*. Addison-Wesley Professional, 2003.

[181] A. Bröhl and W. Dröschel, "Das V-Modell," *München, Wien: Oldenburg-Verlag*, 1995.

[182] B. Daubner, B. Westfechtel, and A. Henrich, "Towards anchoring software measures on elements of the process model," in *Proceedings of the 1st International Conference on Software and Data Technologies (ICSOFT'2006), Setúbal, Portugal, September*, 2006, pp. 232–237.

[183] S. Mantel, S. Eckert, M. Schissler, C. Schäffner, O. K. Ferstl, and E. J. Sinz, "Eine Entwicklungsmethodik für die überbetriebliche Integration von Anwendungssystemen," in *Überbetriebliche Integration von Anwendungssystemen - FORWIN-Tagung 2004*, D. e. a. Bartmann, Ed., Aachen, GER, 2004.

[184] M. Schissler, S. Mantel, S. Eckert, O. K. Ferstl, and E. J. Sinz, "Entwicklungsmethodiken zur Integration von Anwendungssystemen in überbetrieblichen Geschäftsprozessen," in *Wirtschaftsinformatik 2005*, O. Ferstl, E. J. Sinz, S. Eckert, and I. T., Eds. Heidelberg, GER: Gesellschaft für Informatik, 2005, pp. 1463–1482.

[185] *ebXML, ebXML Business Process Specification Schema, Version 1.01*, 2001, last accessed: December 10, 2008. [Online]. Available: http://www.ebxml.org/specs/ebBPSS.pdf

[186] S. Jones, "A methodology for service architectures," *Oasis Draft*, pp. 1–32, 2005, last accessed: December 10, 2008. [Online]. Available: http://www.oasis-open.org/committees/download.php/15071/A%20methodology%20for%20Service%20Architectures%201%202%204%20-%20OASIS%20Contribution.pdf

[187] ABLE Project, "ACME language in BNF 2.0," Website, 2006, last accessed: December 10, 2008. [Online]. Available: http://www.cs.cmu.edu/~acme/html/ArmaniParser.html

[188] W. M. P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, Eds., *Business Process Management, 3rd International Conference, BPM 2005, Nancy, France, September 5-8, 2005, Proceedings*, vol. 3649, 2005.

[189] S. Dustdar, J. L. Fiadeiro, and A. P. Sheth, Eds., *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5-7, 2006, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4102. Springer, 2006.

[190] A. P. Buchmann, F. Casati, L. Fiege, M. Hsu, and M.-C. Shan, Eds., *Technologies for E-Services, Third International Workshop, TES 2002, Hong Kong, China, August 23-24, 2002, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2444. Springer, 2002.

[191] A. Dan and W. Lamersdorf, Eds., *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4294. Springer, 2006.

Index

List of Figures

List of Tables

APPENDIX

A BNF of the ACME Language

Listing 4: BNF Definition of the ACME Language (taken from [187])

```
NON-TERMINALS

 parse_AcmeDesign  ::=  ( <IMPORT> ( Filename ";" | <STRING_LITERAL> ";" ) )* (
TypeDeclaration | FamilyDeclaration | DesignAnalysisDeclaration | PropertyDeclaration |
PropertiesBlock | SystemDeclaration )* <EOF>
 Filename  ::=  ( "$" | "%" )? <IDENTIFIER> ( ( "."
      | ":" | "-" | "+" | "\\" | "\\\\" | "/" | "$" | "%" ) )+
      <IDENTIFIER> )*
 FamilyDeclaration  ::=  ( <FAMILY> | <STYLE> )
      <IDENTIFIER> ( ";" | ( "=" FamilyBody ( ";" )? ) | (
<EXTENDS> lookup_SystemTypeByName ( "," lookup_SystemTypeByName )*
<WITH> FamilyBody ( ";" )? ) )
 FamilyBody  ::=  "{" "}"
  | "{" ( TypeDeclaration | SystemStructure )+ "}"
 TypeDeclaration  ::=  ElementTypeDeclaration
  | PropertyTypeDeclaration
 ElementTypeDeclaration  ::=  ElementProtoTypeDeclaration
  | ComponentTypeDeclaration
  | GroupTypeDeclaration
  | ConnectorTypeDeclaration
  | PortTypeDeclaration
  | RoleTypeDeclaration
 ElementProtoTypeDeclaration  ::=  ( <ELEMENT> <TYPE>
      <IDENTIFIER> ( "=" parse_ElementProtoTypeDescription ( ";" )? | ";" ) |
<ELEMENT> <TYPE> <IDENTIFIER>
      <EXTENDS> lookup_ComponentTypeByName ( ","
lookup_ComponentTypeByName )*
<WITH> parse_ElementProtoTypeDescription ( ";" )? )
 ComponentTypeDeclaration  ::=  ( <COMPONENT> <TYPE>
      <IDENTIFIER> ( "=" parse_ComponentDescription ( ";" )? | ";" ) |
<COMPONENT> <TYPE> <IDENTIFIER>
      <EXTENDS> lookup_ComponentTypeByName ( ","
lookup_ComponentTypeByName )*
 <WITH> parse_ComponentDescription ( ";" )? )
 GroupTypeDeclaration  ::=  ( <GROUP> <TYPE>
      <IDENTIFIER> ( "=" parse_GroupDescription ( ";" )? | ";" ) |
 <GROUP> <TYPE> <IDENTIFIER>
      <EXTENDS> lookup_GroupTypeByName ( ","
lookup_GroupTypeByName )*
 <WITH> parse_GroupDescription ( ";" )? )
 ConnectorTypeDeclaration  ::=  ( <CONNECTOR> <TYPE>
      <IDENTIFIER> ( "=" parse_ConnectorDescription ( ";" )? | ";" ) |
<CONNECTOR> <TYPE> <IDENTIFIER>
      <EXTENDS> lookup_ConnectorTypeByName ( "," lookup_ConnectorTypeByName )*
 <WITH> parse_ConnectorDescription ( ";" )? )
 PortTypeDeclaration  ::=  ( <PORT> <TYPE>
      <IDENTIFIER> ( "=" parse_PortDescription ( ";" )? | ";" ) |
 <PORT> <TYPE> <IDENTIFIER>
      <EXTENDS> lookup_PortTypeByName ( "," lookup_PortTypeByName )*
 <WITH> parse_PortDescription ( ";" )? )
 RoleTypeDeclaration  ::=  ( <ROLE> <TYPE>
      <IDENTIFIER> ( "=" parse_RoleDescription ( ";" )? | ";" ) |
<ROLE> <TYPE> <IDENTIFIER>
      <EXTENDS> lookup_RoleTypeByName ( ","
lookup_RoleTypeByName )*
 <WITH> parse_RoleDescription ( ";" )? )
 lookup_SystemTypeByName  ::=  <IDENTIFIER>
 lookup_ComponentTypeByName  ::=  ( <IDENTIFIER> "." )?
    <IDENTIFIER>
 lookup_GroupTypeByName  ::=  ( <IDENTIFIER> "." )?
    <IDENTIFIER>
 lookup_ConnectorTypeByName  ::=  ( <IDENTIFIER> "." )?
    <IDENTIFIER>
 lookup_PortTypeByName  ::=  ( <IDENTIFIER> "." )?
    <IDENTIFIER>
```

```
 lookup_RoleTypeByName ::= ( <IDENTIFIER> "." )?
    <IDENTIFIER>
 lookup_PropertyTypeByName ::= ( <IDENTIFIER> "." )?
    <IDENTIFIER>
 lookup_arbitraryTypeByName ::= ( PropertyTypeDescription |
 <SYSTEM> | <COMPONENT> | <GROUP> | <CONNECTOR> |
      <PORT> | <ROLE> | <PROPERTY> | <REPRESENTATION> |
 NonPropertySetTypeExpression )
 SystemDeclaration ::= <SYSTEM> <IDENTIFIER> ( ":" lookup_SystemTypeByName ( ","
lookup_SystemTypeByName )* )? ( "=" SystemBody ( ";" )? | ";" )
 SystemBody ::= ( <NEW> lookup_SystemTypeByName ( ","
lookup_SystemTypeByName )* |
 "{" "}" | "{" ( SystemStructure )+ "}" )
( <EXTENDED> <WITH> SystemBody )?
 SystemStructure ::= ComponentDeclaration
   | ComponentsBlock
   | GroupDeclaration
   | ConnectorDeclaration
   | ConnectorsBlock
   | PortDeclaration
   | PortsBlock
   | RoleDeclaration
   | RolesBlock
   | PropertyDeclaration
   | PropertiesBlock
   | AttachmentsDeclaration
   | RepresentationDeclaration
   | DesignAnalysisDeclaration
   | parse_DesignRule
 parse_ElementProtoTypeDescription ::= "{" ( PropertyDeclaration |
 PropertiesBlock | RepresentationDeclaration )* "}"
 GroupDeclaration ::= <GROUP> <IDENTIFIER>
( ":" lookup_GroupTypeByName ( "," lookup_GroupTypeByName )* )?
 ( "=" parse_GroupDescription ";" | ";" )
 parse_GroupDescription ::= ( <NEW> lookup_GroupTypeByName
( "," lookup_GroupTypeByName )* | "{" ( MembersBlock | PropertyDeclaration |
 PropertiesBlock | parse_DesignRule )* "}" ) (
<EXTENDED> <WITH> parse_GroupDescription )?
 ComponentDeclaration ::= <COMPONENT> <IDENTIFIER> ( ":"
lookup_ComponentTypeByName ( "," lookup_ComponentTypeByName )* )? ( "="
parse_ComponentDescription ";" | ";" )
 ComponentsBlock ::= <COMPONENTS> "{" ( <IDENTIFIER>
      ( ":" lookup_ComponentTypeByName ( "," lookup_ComponentTypeByName )* )? ( "="
parse_ComponentDescription ";" | ";" ) )* "}" ( ";" )?
 parse_ComponentDescription ::= ( <NEW> lookup_ComponentTypeByName ( ","
lookup_ComponentTypeByName )* | "{" ( PortDeclaration | PortsBlock |
 PropertyDeclaration | PropertiesBlock | RepresentationDeclaration |
parse_DesignRule )* "}" ) ( <EXTENDED> <WITH> parse_ComponentDescription )?
 ConnectorDeclaration ::= <CONNECTOR> <IDENTIFIER> ( ":"
lookup_ConnectorTypeByName ( "," lookup_ConnectorTypeByName )* )? ( "="
parse_ConnectorDescription ";" | ";" )
 ConnectorsBlock ::= <CONNECTORS> "{" ( <IDENTIFIER>
      ( ":" lookup_ConnectorTypeByName ( "," lookup_ConnectorTypeByName )* )?
( "=" parse_ConnectorDescription ";" | ";" ) )* "}" ( ";" )?
 parse_ConnectorDescription ::= ( <NEW> lookup_ConnectorTypeByName ( ","
lookup_ConnectorTypeByName )* | "{" ( RoleDeclaration | RolesBlock |
 PropertyDeclaration | PropertiesBlock | RepresentationDeclaration |
 parse_DesignRule )* "}" ) ( <EXTENDED> <WITH> parse_ConnectorDescription )?
 PortDeclaration ::= <PORT> <IDENTIFIER> ( ":" lookup_PortTypeByName ( ","
lookup_PortTypeByName )* )? ( "=" parse_PortDescription ";" | ";" )
 PortsBlock ::= <PORTS> "{" ( <IDENTIFIER> (
      ":" lookup_PortTypeByName ( "," lookup_PortTypeByName )* )? ( "="
parse_PortDescription ";" | ";" ) )* "}" ( ";" )?
 parse_PortDescription ::= ( <NEW> lookup_PortTypeByName
( "," lookup_PortTypeByName )* | "{" ( PropertyDeclaration |
PropertiesBlock | RepresentationDeclaration | parse_DesignRule )* "}" )
( <EXTENDED> <WITH> parse_PortDescription )?
 RoleDeclaration ::= <ROLE> <IDENTIFIER> ( ":" lookup_RoleTypeByName ( ","
lookup_RoleTypeByName )* )? ( "=" parse_RoleDescription ";" | ";" )
 MembersBlock ::= <MEMBERS> "{" ( QualifiedReference ( ";" ) )* "}" ( ";" )?
 QualifiedReference ::= <IDENTIFIER> ( ( "."
      <IDENTIFIER> ) )*
 RolesBlock ::= <ROLES> "{" ( <IDENTIFIER> (
      ":" lookup_RoleTypeByName ( "," lookup_RoleTypeByName )* )? ( "="
parse_RoleDescription ";" | ";" ) )* "}" ( ";" )?
```

```
 parse_RoleDescription ::= ( <NEW> lookup_RoleTypeByName
( "," lookup_RoleTypeByName )* | "{" ( PropertyDeclaration
| PropertiesBlock | RepresentationDeclaration | parse_DesignRule )* "}" ) ( <EXTENDED>
 <WITH> parse_RoleDescription )?
 AttachmentsDeclaration ::= ( ( <ATTACHMENTS> "{" (
      <IDENTIFIER> "." <IDENTIFIER> "to" <IDENTIFIER> "."
      <IDENTIFIER> ( "{" ( PropertyDeclaration | PropertiesBlock )* "}" )?
 ";" )* "}" ( ";" )? ) | ( <ATTACHMENT> <IDENTIFIER>
      "." <IDENTIFIER> "to" <IDENTIFIER> "." <IDENTIFIER> (
      "{" ( PropertyDeclaration | PropertiesBlock )* "}" )? ";" ) )
 PropertyDeclaration ::= <PROPERTY> parse_PropertyDescription ";"
 PropertiesBlock ::= <PROPERTIES> "{" ( parse_PropertyDescription ( ";"
parse_PropertyDescription | ";" )* )? "}" ( ";" )?
 parse_PropertyDescription ::= (
<PROPERTY> )? <IDENTIFIER> ( ":" PropertyTypeDescription )?
( "=" PropertyValueDeclaration )?
( <PROPBEGIN> parse_PropertyDescription
( ";" parse_PropertyDescription | ";" )*
 <PROPEND> | <PROPBEGIN> <PROPEND> )?
 PropertyTypeDeclaration ::= <PROPERTY> <TYPE>
      <IDENTIFIER> ( "=" ( <INT> ";" | <FLOAT> ";" |
      <STRING> ";" | <BOOLEAN> ";" | <ENUM> ( "{"
      <IDENTIFIER> ( "," <IDENTIFIER> )* "}" )? ";" | <SET> (
      "{" "}" )? ";" | <SET> "{" PropertyTypeDescription "}" ";" |
 <SEQUENCE> ( "<" ">" )? ";" | <SEQUENCE>
      "<" PropertyTypeDescription ">" ";" | <RECORD> "["
 parse_RecordFieldDescription ( ";" parse_RecordFieldDescription | ";" )* "]"
 ";" | <RECORD> ( "[" "]" )? ";" | <IDENTIFIER>
      ";" ) )
 PropertyTypeDescription ::= <ANY>
  | <INT>
  | <FLOAT>
  | <STRING>
  | <BOOLEAN>
  | <SET> ( "{" ( PropertyTypeDescription )? "}" )?
  | <SEQUENCE> ( "<" ( PropertyTypeDescription )? ">" )?
  | <RECORD> "[" parse_RecordFieldDescription
( ";" parse_RecordFieldDescription | ";" )* "]"
  | <RECORD> ( "[" "]" )?
  | <ENUM> ( "{" <IDENTIFIER> ( ","
      <IDENTIFIER> )* "}" )?
  | <ENUM> ( "{" "}" )?
  | lookup_PropertyTypeByName
 parse_RecordFieldDescription ::= <IDENTIFIER> ( "," <IDENTIFIER>
      )* ( ":" PropertyTypeDescription )?
 PropertyValueDeclaration ::= <INTEGER_LITERAL>
  | <FLOATING_POINT_LITERAL>
  | <STRING_LITERAL>
  | <FALSE>
  | <TRUE>
  | AcmeSetValue
  | AcmeSequenceValue
  | AcmeRecordValue
  | <IDENTIFIER>
 AcmeSetValue ::= "{" "}"
  | "{" PropertyValueDeclaration ( "," PropertyValueDeclaration )* "}"
 AcmeSequenceValue ::= "<" ">"
  | "<" PropertyValueDeclaration ( "," PropertyValueDeclaration )* ">"
 AcmeRecordValue ::= ( "[" RecordFieldValue
( ";" RecordFieldValue | ";" )* "]" | "[" "]" )
 RecordFieldValue ::= <IDENTIFIER> ( ":" PropertyTypeDescription )? "=" PropertyValueDeclaration
 RepresentationDeclaration ::= <REPRESENTATION> ( <IDENTIFIER>
      "=" )? "{" SystemDeclaration ( BindingsMapDeclaration )? "}" ( ";" )?
 BindingsMapDeclaration ::= <BINDINGS> "{" ( BindingDeclaration )* "}" ( ";" )?
 BindingDeclaration ::= ( <IDENTIFIER> "." )?
      <IDENTIFIER> "to" ( <IDENTIFIER> "." )? <IDENTIFIER> (
      "{" ( PropertyDeclaration | PropertiesBlock )* "}" )? ";"
 DesignAnalysisDeclaration ::= ( ( <EXTERNAL> ( <DESIGN> )?
      <ANALYSIS> <IDENTIFIER> "(" FormalParams ")" ":"
( PropertyTypeDescription | <COMPONENT> |
<GROUP> | <CONNECTOR> | <PORT> |
      <ROLE> | <SYSTEM> | <ELEMENT> | <TYPE> ) "=" JavaMethodCallExpr ";" ) |
 ( ( <DESIGN> )? <ANALYSIS> <IDENTIFIER> "(" FormalParams ")" ":"
( PropertyTypeDescription | <COMPONENT> | <GROUP> | <CONNECTOR> | <PORT> |
      <ROLE> | <SYSTEM> | <ELEMENT> | <TYPE> ) "=" DesignRuleExpression ";" ) )
```

```
 parse_DesignRule  ::=  ( <DESIGN> )? ( <INVARIANT> |
      <HEURISTIC> ) DesignRuleExpression ( <PROPBEGIN> parse_PropertyDescription (
 ";" parse_PropertyDescription | ";" )* <PROPEND> )? ";"
 DesignRuleExpression  ::=  QuantifiedExpression
  | BooleanExpression
 QuantifiedExpression  ::=  ( ( <FORALL> | <EXISTS> (
      <UNIQUE> )? ) <IDENTIFIER> ( ( ":" | <SET_DECLARE> ) ( Type |
lookup_arbitraryTypeByName ) )? <IN> ( SetExpression | Reference ) "|"
DesignRuleExpression )
 BooleanExpression  ::=  OrExpression ( <AND> OrExpression )*
 OrExpression  ::=  ImpliesExpression ( <OR> ImpliesExpression )*
 ImpliesExpression  ::=  IffExpression ( <IMPLIES> IffExpression )*
 IffExpression  ::=  EqualityExpression ( <IFF> EqualityExpression )*

 EqualityExpression  ::=  RelationalExpression ( <EQ>
RelationalExpression | <NE> RelationalExpression )*
 RelationalExpression  ::=  AdditiveExpression ( "<" AdditiveExpression | ">"

AdditiveExpression | <LE> AdditiveExpression | <GE> AdditiveExpression )*
 AdditiveExpression  ::=  MultiplicativeExpression

( <PLUS> MultiplicativeExpression | <MINUS> MultiplicativeExpression )*

 MultiplicativeExpression  ::=  UnaryExpression (

<STAR> UnaryExpression | <SLASH> UnaryExpression | <REM> UnaryExpression )*

 UnaryExpression  ::=  <BANG> UnaryExpression
  | <MINUS> UnaryExpression
  | PrimitiveExpression
 PrimitiveExpression  ::=  "(" DesignRuleExpression ")"
  | LiteralConstant
  | Reference
  | SetExpression
 Reference  ::=  <IDENTIFIER> ( ( "."
      <IDENTIFIER> ) | ( "." <TYPE> ) | ( "." <COMPONENTS> ) |
      ( "." <CONNECTORS> ) | ( "." <PORTS> ) | ( "." <ROLES> )
      | ( "." <MEMBERS> ) | ( "." <PROPERTIES> ) | ( "."
      <REPRESENTATIONS> ) | ( "." <ATTACHEDPORTS> ) | ( "."
      <ATTACHEDROLES> ) )* ( "(" ActualParams ")" )?
 JavaMethodCallExpr  ::=  <IDENTIFIER> ( "." <IDENTIFIER>
      )* "(" ActualParams ")"
 LiteralConstant  ::=  ( <INTEGER_LITERAL> )
  | ( <FLOATING_POINT_LITERAL> )
  | ( <STRING_LITERAL> )
  | ( <TRUE> )
  | ( <FALSE> )
  | ( <COMPONENT> )
  | ( <GROUP> )
  | ( <CONNECTOR> )
  | ( <PORT> )
  | ( <ROLE> )
  | ( <SYSTEM> )
  | ( <ELEMENT> )
  | ( <PROPERTY> )
  | ( <INT> )
  | ( <FLOAT> )
  | ( <STRING> )
  | ( <BOOLEAN> )
  | ( <ENUM> )
  | ( <SET> )
  | ( <SEQUENCE> )
  | ( <RECORD> )
 ActualParams  ::=  ( ActualParam ( "," ActualParam )* )?
 FormalParams  ::=  ( FormalParam ( "," FormalParam )* )?
 ActualParam  ::=  DesignRuleExpression
 FormalParam  ::=  <IDENTIFIER> ( "," <IDENTIFIER>
      )* ":" ( <ELEMENT> | <SYSTEM> | <COMPONENT> |
      <CONNECTOR> | <PORT> | <ROLE> | <TYPE> |
      <PROPERTY> | <REPRESENTATION> | <ANY> | NonPropertySetTypeExpression |
PropertyTypeDescription )
 NonPropertySetTypeExpression  ::=  <SET> "{" ( <ELEMENT> |
      <SYSTEM> | <COMPONENT> | <CONNECTOR> | <PORT> |
      <ROLE> | <TYPE> | <PROPERTY> | <REPRESENTATION> |
      <ANY> ) "}"
```

```
SetExpression ::= ( LiteralSet | SetConstructor )
LiteralSet ::= ( "{" "}" | "{" ( LiteralConstant | Reference )
( "," ( LiteralConstant | Reference ) )* "}" )
SetConstructor ::= ( "{" <SELECT> <IDENTIFIER> (
    ":" lookup_arbitraryTypeByName )? <IN> ( SetExpression | Reference ) "|"
DesignRuleExpression "}" | ( "{" <COLLECT> <IDENTIFIER> "." <IDENTIFIER> ":"
lookup_arbitraryTypeByName "." lookup_arbitraryTypeByName <IN>
( SetExpression | Reference ) "|" DesignRuleExpression "}" ) )
RecordType ::= <RECORD> "[" RecordItem ( "," RecordItem )* "]"
RecordItem ::= <IDENTIFIER> ":" Type
SetType ::= <SET> "{" Type "}"
SequenceType ::= <SEQUENCE> "{" Type "}"
Signature ::= Type "<->" Type
Type ::= ( <IDENTIFIER> ( "."
    <IDENTIFIER> )* )
PrimitiveType ::= <COMPONENT>
  | <GROUP>
  | <CONNECTOR>
  | <PORT>
  | <ROLE>
  | <SYSTEM>
Element ::= ( <IDENTIFIER> ( "."
    <IDENTIFIER> )* )
  | CompoundElement
CompoundElement ::= Set
  | Record
  | Sequence
Set ::= "{" Element ( "," Element )* "}"
Record ::= "[" <IDENTIFIER> "=" Element ( "," <IDENTIFIER> "=" Element )* "]"
Sequence ::= "<" Element ( "," Element )* ">"
```

B Raw Data For the Analysis of Chapter 4

The table can be found on page A-7

Name	Reuse	UM	NUM	RUM	NRUM	IOCM	NIOCM	RIOCM	NRIOCM	OICM	NOICM	ROICM	NROICM	NTCM	NRTCM	$SM	$SDM	pubBean_LC	pubBean_LCOM*	DAO_LCOM	DAO_LCOM*
canBeAffectedPerson	1	1	0,2500	35	0,2932	1	0,6667	18	0,0556	1	0,0000	0	0,0000	0,5000	0,0417	4	6,0238	0	0	0	0
checkAccess	0	2	0,2857	38	0,1728	2	0,6000	30	0,0529	2	0,0000	1	0,0018	0,4286	0,0383	7	0,5218	0	0	0	0
deleteMigrateOrKeepMailbox	1	6	0,7500	32	0,1420	5	0,2857	18	0,0278	1	0,0000	0	0,0000	0,2500	0,0243	8	0,1038	0	0	0	0
generateWorkitemId	0	1	0,2500	18	0,0556	1	0,6667	8	0,0247	1	0,0000	0	0,0000	0,5000	0,0185	4	6,0238	0	0	0	0
generateWorkitemId_54	0	2	0,3333	63	0,1296	2	0,6000	14	0,0288	1	0,0000	0	0,0000	0,5000	0,0240	6	1,4124	0	0	0	0
getAssignmentTypesForms	0	2	0,1818	173	0,1942	0	1,0000	20	0,0224	4	0,5000	33	0,0438	0,6364	0,0380	11	0,3367	0	0	0	0
getCurrentAndFutureDeputiesOfUser	0	1	0,1000	160	0,1375	2	0,3333	14	0,0173	6	0,1429	55	0,0679	0,2000	0,0527	10	0,1186	0	0	0	0
getFormOrgFormLookupValues	0	1	0,0769	119	0,1130	2	0,6000	21	0,0139	6	0,2500	6	0,0057	0,3846	0,0112	13	1,2861	0	0	0	0
getFormTemplateIdByNumber	1	0	0,0000	100	0,2469	0	1,0000	20	0,0434	2	0,0000	1	0,0025	0,6000	0,0306	5	3,0534	0	0	0	0
getHelpdesks	0	0	0,0000	85	0,3438	0	1,0000	14	0,0576	2	0,0000	1	0,0041	0,3333	0,0213	3	11,6471	0	0	0	0
getHROrgUnitPickerFormLookupValues	0	1	0,1000	112	0,1383	0	1,0000	20	0,0247	5	0,2857	7	0,0086	0,5000	0,0135	10	0,1186	0	0	0	0
getLanguages	0	1	0,2500	6	0,0185	1	0,0000	0	0,0000	3	0,0000	0	0,0000	0,0000	0,0000	4	6,0238	0	0	0	0
getLookupValues	1	1	0,5000	66	0,4074	0	1,0000	14	0,0864	1	0,0000	0	0,0000	0,5000	0,0432	2	23,8817	0	0	0	0
getMaxEntitlementValidityDate	1	1	0,2500	110	0,3395	0	1,0000	23	0,0710	1	0,0000	0	0,0000	0,7500	0,0532	4	6,0238	0	0	0	0
getNotificationConfig	0	0	0,0000	97	0,3992	0	1,0000	14	0,0576	2	0,0000	2	0,0082	0,3333	0,0247	3	11,6471	0	0,8	0	0
getOrgData	1	1	0,2000	129	0,3185	0	1,0000	30	0,0741	1	0,0000	0	0,0000	0,8000	0,0593	5	3,0534	0	0	0	0
getOutOfOfficeOfUser	0	2	0,2857	82	0,1446	1	0,5000	14	0,0247	5	0,0000	3	0,0053	0,1429	0,0108	7	0,5218	0	0	0	0
getPeoplePickerFormLookupValues	0	1	0,5000	67	0,4136	0	1,0000	14	0,0864	1	0,0000	0	0,0000	0,5000	0,0432	2	23,8817	0	0,85	0	0
getPersonInfo	1	13	0,7037	115	0,0526	1	0,5000	14	0,0064	21	0,1600	25	0,0114	0,1852	0,0111	27	12,1188	0	0	0	0
getSIAMPositionLoadData	1	5	0,4545	112	0,1257	1	0,5000	14	0,0157	7	0,2222	17	0,0131	0,2727	0,0185	11	0,3367	0	0	0	0
getSIAMRolesForPerson	0	0	0,0000	77	0,3169	0	1,0000	14	0,0576	1	0,5000	6	0,0247	0,6667	0,0357	3	11,6471	0	0	0	0
getSiamRolesTodo	0	0	0,0000	28	0,1152	0	1,0000	16	0,0658	0	1,0000	6	0,0247	1,0000	0,0521	3	11,6471	0	0	0	0
getSiamUser	1	0	0,0000	93	0,1435	0	1,0000	5	0,0077	5	0,2857	60	0,0926	0,3750	0,0820	8	0,1038	0	0	0	0
getTimezones	0	2	0,6667	17	0,0700	1	0,0000	0	0,0000	1	0,5000	3	0,0123	0,3333	0,0082	3	11,6471	0	0,85	0	0
getUserContext	0	0	0,0000	181	0,2793	0	1,0000	5	0,0077	5	0,2857	60	0,0926	0,3750	0,0820	8	0,1038	0	0	0	0
isFormAlreadyFilled	1	1	0,1667	131	0,2695	0	1,0000	25	0,0514	1	0,6667	8	0,0165	0,8333	0,0340	6	1,4124	0	0	0	0
loadEntitelRoles	0	1	0,0556	200	0,1372	0	1,0000	35	0,0240	10	0,2308	16	0,0110	0,4444	0,0146	18	4,5893	0	0	0	0
loadEntiteledFPRoles	0	1	0,0500	211	0,1302	0	1,0000	35	0,0216	10	0,3333	25	0,0154	0,5000	0,0170	20	6,1482	0	0	0	0
loadFormDataset	1	0	0,0000	91	0,2809	0	1,0000	21	0,0648	1	0,5000	2	0,0062	0,7500	0,0355	4	6,0238	0	0	0	0
loadRequestById	1	0	0,0000	272	0,1232	0	1,0000	16	0,0076	10	0,5833	101	0,0480	0,6154	0,0449	26	11,2319	0	0	0	0
loadRequestByWorkItemId	1	1	0,0385	269	0,1277	0	1,0000	19	0,0090	11	0,5417	101	0,0480	0,5769	0,0450	26	11,2319	0	0	0	0
loadRolesForForm	0	1	0,1000	151	0,1864	1	0,8000	27	0,0333	2	0,6000	16	0,0198	0,7000	0,0265	10	0,1186	0	0	0	0
saveRequestAsDraft	1	1	0,0400	281	0,1388	6	0,7500	38	0,0484	1	0,0000	0	0,0000	0,7200	0,0465	25	10,3541	0	0	0	0
SearchOrg	1	6	0,2857	74	0,0435	4	0,5000	14	0,0082	11	0,1538	4	0,0024	0,2857	0,0046	21	6,3531	0	0	0	0
searchPerson	0	2	0,6667	67	0,2757	1	0,5000	14	0,0576	1	0,0000	0	0,0000	0,3333	0,0384	3	11,6471	0	0,85	0	0
sendRequest	1	0	0,0000	263	0,1353	6	0,7331	32	0,0473	1	0,0000	0	0,0000	0,7083	0,0454	24	3,4864	0	0	0	0
setAssignmentTypeValidity	0	1	0,1429	150	0,2646	0	1,0000	37	0,0653	1	0,0000	0	0,0000	0,8571	0,0553	7	0,5218	0	0	0	0
setDeputiesOfUser	0	0	0,0000	35	0,1466	5	0,2857	19	0,0233	1	0,0000	0	0,0000	0,2500	0,0257	8	0,1038	0	0	0	0
setEntitlemementValidity	0	1	0,1667	120	0,2469	0	1,0000	30	0,0617	1	0,0000	0	0,0000	0,8333	0,0514	6	1,4124	0	0	0	0
setFormularValidity	0	1	0,1667	35	0,1355	0	1,0000	28	0,0576	1	0,0000	0	0,0000	0,8333	0,0480	6	1,4124	0	0	0	0
setOutOfOfficeOfUser	0	1	0,1429	84	0,1481	5	0,1667	17	0,0300	1	0,0000	0	0,0000	0,1429	0,0257	7	0,5218	0	0	0	0
setPreferredUserLanguage	0	0	0,0000	83	0,3416	0	1,0000	27	0,1111	1	0,0000	0	0,0000	0,6667	0,0741	3	11,6471	0	0	0	0
setUserNotificationContext	0	1	0,2500	97	0,2994	0	1,0000	42	0,1296	1	0,0000	0	0,0000	0,7500	0,0972	4	6,0238	0	0,8	0	0
setUserPreferredHelpDesk	0	1	0,3333	82	0,3374	0	1,0000	28	0,1152	1	0,0000	0	0,0000	0,6667	0,0768	3	11,6471	0	0	0	0
setUserTimezone	0	1	0,3333	84	0,3457	0	1,0000	28	0,1152	1	0,0000	0	0,0000	0,6667	0,0768	3	11,6471	0	0,85	0	0

Figure 73: Raw Data for the Analysis of Chapter 4

C Complete Agreement Management Process Model

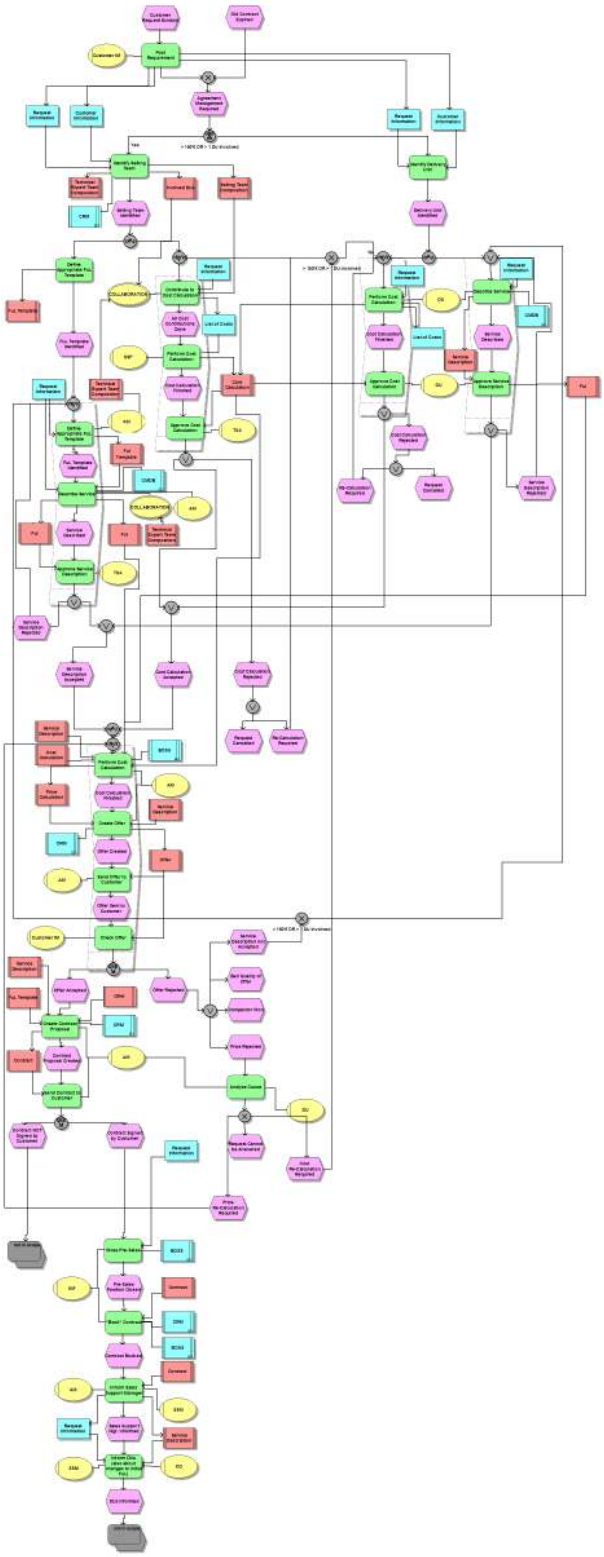


Figure 74: EPC Process Diagram for Functional Requirements

D Metrics for Step 6 of the Case Study

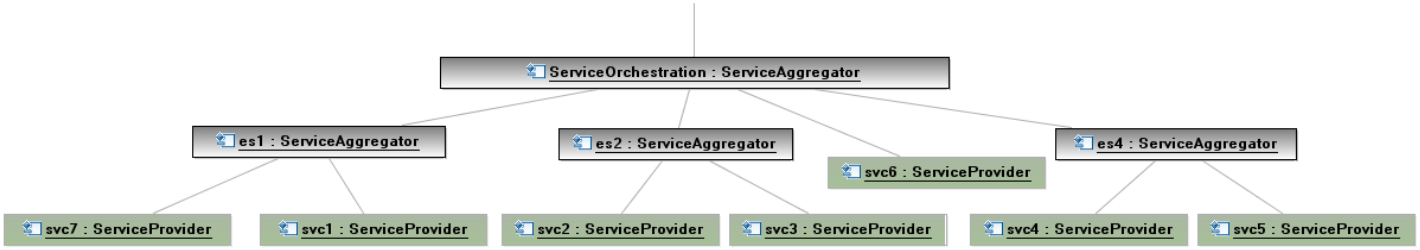


Figure 75: Structure of the Intermediate System Design after Step 6

Service s of Ω	$cos(s)$	$\gamma(s)$	$\pi(s)$	$AD(\Omega, s)$
ServiceOrchestration	4	4	1	1
es_1	2	2	1	0
es_2	2	2	1	0
es_4	2	2	1	0

Table 25: Size Metrics for the Intermediate Composite Application