Institut für Wirtschaftsinformatik der Universität Bern
Abteilung Informationsmanagement

Cases of
Software Services Design in Practice

Arbeitsbericht Nr. 216
2009-05

Susanne Patig

Dagstuhl Seminar Proceedings 09021
Software Service Engineering
http://drops.dagstuhl.de/opus/volltexte/2009/2047

Cases of Software Service Design in Practice

Susanne Patig[1]

[1] University of Bern, IWI, Engehaldenstrasse 8, CH-3012 Bern, Switzerland
susanne.patig@iwi.unibe.ch

Abstract. During the last years, several approaches for the design of software services in service-oriented architectures (SOA) have been proposed. Often these approaches are too rough or too academic to provide guidance for real world SOA projects. Moreover, since the existing SOA design approaches are often not sufficiently validated, their successfulness in practice can be doubted. The research presented here aims at learning from successful SOA projects. Two cases of such projects are described. In the cases similarities show up that are distinct from existing SOA design approaches (mainly the purely academic ones) and, thus, point to necessary enhancements of these approaches.

1 Motivation

Since the initiation of service-oriented architecture (SOA) in the 1990s, several guidelines for SOA design have been published (see Section 2). For companies intending to implement SOA, the mere diversity of design directions is confusing. Moreover, it is often not really clear whether some published design approach results from an academic effort, and, thus, does not necessarily work outside the scientific clean room, or whether the approach represents 'best practices' of some number of SOA projects (e.g., [1], [15]). Even in the latter case it can be questioned whether domain-independent best practices of *application* software service design exist at all. Additionally, most SOA design approaches are not validated (e.g., [1], [9]), only validated by small examples (e.g., [13], [2]) or by 'industrial experience' (e.g., [1], [4]), whose details naturally must remain opaque. In any case, validation is often not reliable because it was done by the authors themselves and sometimes even in the same setting from which the approach emerged. So, sound methodical support for the design of software services in 'real world' application scenarios is hard to find.

Finally, probably none of the proposed SOA design approaches is complete and universally correct. Improving these approaches requires knowledge about facets of real SOA projects that are currently not covered or insufficiently solved.

In response to these doubts and questions, a case study was conducted to investigate how companies design the service-oriented architecture underlying their application systems; first results are presented here. Roughly, the case study tries to find out, (1) which services have been design and implemented for a particular application system or systems landscape (*descriptive research objective*), and (2) why have the services been designed in a particular way (*exploratory research objective*)? *Exploratory* means that practically relevant software service design criteria and processes should be discovered to possibly adapt the existing SOA design approaches.

From the research objectives listed above, Section 3 derives the *research design*, i.e., the plan for the investigation that links the data to be collected to the research objectives [12]. The collected data currently consists of two cases that are presented and compared in Section 4. The overall conclusion and the next steps in our research can be found in Section 5.

2 Current Software Service Design Approaches

Basically, approaches to design software services for SOA fall into two groups (see Fig. 1): principles-driven approaches and hierarchical ones. Hybrid approaches (e.g., [2], [7], and [6]) combine these groups.

Principle-driven software service design approaches are directly linked to the heart of SOA: There is a common understanding that service-orientation is not primarily tied to specific technologies, but to a set of principles that must be obeyed in designing such architectures [3], [8]. The main principles are *abstraction* (services hide information on technology and logic from the outside world), *standardized contract* (services provide technical interfaces by which they can be accessed and which keep to some contract definition standard*)*, *loose coupling* (a service contract is independent of the implementation of the service, and services are independent of each other), *cohesion* (the functionality provided by a service is strongly related), *reusability* (the logic encapsulated by a service is sufficiently generic for numerous usage scenarios and consumers), *autonomy* (service exercise as much control as possible over their runtime execution environment), *statelessness* (between consecutive service calls, no information must be kept within the service) and *discoverability* (service contracts contain meta data by which the services can be found and assessed). Principle-driven approaches (e.g., [3]) provide guidelines (sometimes bundled in patterns [4]) how to realize the SOA design principles – without defining and ordering the necessary steps. Additionally, hierarchical service design approaches in parts involve SOA design principles (see below).

Hierarchical software service design approaches prescribe a series of steps from some level of abstraction to a set of software services, either at the design stage (e.g., [7], [10], [13]) or including further stages of the service life cycle such as deployment, billing, execution and monitoring (e.g., [8], [1]). It can be distinguished between *top-down approaches* that proceed from abstract information at the business level to the detailed technical levels of service design and implementation and *bottom-up approaches* that increase the level of abstraction during design.

Common starting points for *top-down software service design approaches* are business goals (e.g., [5]), functional business areas (e.g., [14]) or business processes (e.g., [9], [7], [10], and [6]). *Goals* describe what should be achieved by a software service, *functional areas* are sets of related tasks referring to, e.g., departments or products, and *business processes* additionally consider the roles that perform these tasks as well as the order of tasks (to derive service orchestrations [1], [8]). Some of the top-down approaches rely on several types of business information or even link, e.g., goals to functional areas and then to business processes [1].

Current *bottom-up software service design approaches* (e.g., [13]) try to achieve service-orientation by wrapping existing application systems. They use reverse engineering techniques such as clustering to identify cohesive components, which form service candidates.

The direction top-down vs. bottom-up mainly refers to the *identification of service candidates*. Often these initial candidates are *refined* before they are specified. The following refinement steps can be found in both top-down and bottom-up approaches:

- *Grouping*: Fine-grained services that have some kind of logical affinity (in terms of, e.g., functions or communication [8]) are grouped into more coarse-grained services. Grouping is unavoidable for bottom-up approaches. In top-down approaches, it guarantees high cohesion, loose coupling and the autonomy of services (e.g., [1], [10], [2]).
- *Verification*: Software services are checked for their conformance to the SOA design principles and adapted if necessary (e.g., [7], [2]).

In contrast to these common steps, only top-down approaches require *asset analysis:* It maps the identified and refined services either to existing application systems that can provide these services or to service implementation projects [2], [1], [7], [8].

The final step of *service specification* is always necessary. It defines the service interface (operations and their signatures, given by message types for inbound and outbound messages) and the conversations between services [1], [2], [14], [8]. Service specification aims, once more, for loose coupling, high cohesion, reusability and standardization. Fig. 1 summarizes the existing approaches to software service design. For the hierarchical approaches, the superset of proposed steps is shown.

Principle-Driven Approaches	**Hierarchical Approaches**			
— (no steps) *Design to*: Abstraction, standardized contract, loose coupling, cohesion, reusability, autonomy, statelessness, discoverability	**Service Identification**			
	Top-Down			*Bottom-Up*
	Goals	Functional Areas	Processes	Existing Applications
	Service Refinement			
				Service Grouping
	Service Verification			
	Asset Analysis			
Service Specification				

Fig. 1: Design Steps in Current Software Service Design Approaches

3 Research Design

Since the challenges of software service design result from the complexity of the underlying application context, the research design must consider real-life scenarios where companies have developed *services-oriented solutions* (application systems or system landscapes). When the boundaries between the phenomenon to be observed and its context are not evident, case studies are an appropriate research strategy [11]. A *case study* is a qualitative empirical investigation of contemporary phenomena within their real-life context without the possibility to control the situation [12].

To increase the external validity (generalizability) of this research, a *multiple-case study* is conducted where conclusions are drawn from a group of cases, which represent a variety of situations [12]. Once a phenomenon has been shown to occur in two to three cases, it can be generalized to develop some general explanation [11]. Then, further cases can be selected that either lead to contrasting results for predictable reasons or that state more precisely the conditions under which the observed phenomenon occurs [11]. This paper presents the first part of the research, namely two cases that form the basis for a generalization of software service design in practice.

The research objectives directly lead to the data to be collected: A *case* in the sense of this investigation is any real-life (as opposed to academic) project where SOA is realized for some application software by implementing software services. This case definition excludes pure SOA middleware projects from the investigation as well as projects where services are not yet implemented or only composed out of existing ones. To count as '*service-oriented*', a system's *architecture* must consist of physically independent software packages that (1) provide well-defined functions by standardized interfaces in a discoverable way and (2) that primarily communicate via message exchange [3]. Though not presupposed by the definition of service-orientation [3], web services are currently the predominant implementation technology. For that reason the comparability of the cases is guaranteed by requiring that at least some of the service interfaces are specified by the Web Services Description Language (WSDL).

The *phenomena* to be observed within each case are the software services and the process of their design. A *software service* is defined by its interface, which groups operations. Each operation has a signature consisting of the types of the inbound and outbound messages as well as optional information on conversations. A *service design process* usually comprises phases in some chronology and criteria for service design.

The *context* of the phenomena to be investigated includes all information that potentially influences SOA design, namely:

- *General context*: the company, its branch, organization, IT department (size and experience) and existing systems landscape
- *Project context*: SOA motivation, project size and duration
- *Requirements context*: needed functionality, quality characteristics as well as technical or organizational constraints
- *Development context*: degrees of freedom (descendent from green-field development over reengineering and migration to wrapping), service provisioning (only within some controllable service inventory vs. to unknown consumers)

In the investigation presented here, the information on phenomena and context was gathered by interviews (both face-to-face and by phone) with several persons representing distinct roles (e.g., software architect, software developer, project manager), by document analysis (design guidelines, service specifications, models) as well as by systems analysis (sample services). The final case report was checked for correctness by the contact persons of the companies. Very condensed versions of the case reports are contained in the following Section 4.

4 Case Study

4.1 Case 1: Mail Order Company

Context: The case stems from a leading trading and services corporation; its 123 companies employ around 50,000 persons in nearly 20 countries. The IT department has circa 230 employees for development. The application systems landscape consists of 200 application systems and includes a central mainframe application that is responsible for, e.g., customer data management, invoicing and order picking. Since the mainframe application was written in assembly language (current in-house software development uses Java and J2EE), it should be replaced by a new application system.

In 2002/2003 it was decided to base reengineering of the mainframe application on SOA for two reasons: First, SOA is able to cope with the unavoidable heterogeneity (in terms of functionality, technology, life cycle and controllability) of the corporation's systems landscape, which is a consequence of continuous acquisitions of companies. Secondly, SOA increases the flexibility to react to new requirements such as seasonal business (Christmas etc.), promotions and changing legal regulations.

The new service-oriented application system should cover the functionality to manage customer orders (i.e., order management in the narrow sense of the word, stock management, customer data management, invoicing and order picking), which is currently covered by several application systems including the mainframe. The consumers of the intended services are several types of clients (e.g., call center clients, web shop clients B2C and B2B) that are mostly under the control of the corporation. Especially the management of customer data is subject to strict security restrictions, and the web shop client for consumers (B2C) is performance-sensitive.

The **service design process,** summarized in Fig. 2 using the Business Process Modeling Notation (BPMN), is hybrid: *Top-down,* business processes and their subprocesses are identified. A *business process* is defined as a set of logically related activities that are chronologically ordered, started by events and lead to results. For IT-supported business (sub-) processes, use cases are derived. A *use case* (e.g., 'create order') is an interaction sequence between a role and a software system to solve some business task. Use cases are modeled as UML activity diagrams. Indivisible (atomic) interaction sequences within a use case that have a business goal are called *application functions* (e.g., 'find shipment with items'). Application functions are candidate application services.

Simultaneously, object-oriented analysis is performed bottom-up: First, *domain classes* are identified, i.e., application-independent objects of the real world with attributes and functionality (e.g., 'article' or 'order'). Then, from the domain classes, *analysis classes* are derived that specialize the domain class within the context of a particular application system. For example, 'stocked article' and 'orderable article' are analysis classes of the domain class 'article'. Each *method* of an analysis class (e.g., 'create', 'release', 'cancel') corresponds to an application function, and an application function can be realized by one or more methods of an analysis class.

Application services (e.g., `GetShipmentWithItems` or `DeliveryCondition-Operations` – changing the type of the delivery to 'urgent' or specifying the delivery address) are the interfaces of software components that provide methods to realize application functions. They result from candidate application services by applying *design rules.* These rules comprise the SOA design principles 'abstraction' and 'statelessness' as well as the requirement that an application service should correspond to the smallest application function. The last requirement leads, for example, to a separation between the application services `CreateCustomer`, which, among others, includes name and address, and `ChangeCustomerAddress`.

Subsequently, the designed services are *evaluated* by the SOA design principle 'reusability' and the criteria 'similarity' and 'stability'. *Similarity* means that existing services must be extended if they provide at least 50 % of the functionality of some new application service. *Stability* calls for the separation of unstable application functions (e.g., country-specific ones, known from domain experience) from stable ones.

So far, around 250 web services have been designed and implemented within a custom-made J2EE framework; execution partly relies on an adapted Oracle BEA WebLogic Application Server[1]. The interfaces of many web services comprise 1 or 3 operations; the largest interface consists of 9 operations. According to their type, 80 % of the web services are application services, 20 % are technical ones supporting software development (e.g., testing web services) or technical process execution (e.g., starting batches). Many application services are centered at analysis classes, e.g., `CreateOrder` or `CreditOperations` (all methods related to order items locked after credit analysis). Additionally, separate application services have been defined for application functions that cannot uniquely be assigned to a particular analysis class. For example, the application service `OrderDetailOperations` groups all the heterogeneous functions that are possible for constituents of an order, e.g., cancellations or substitution of order items, priority adjustments or stock allocations.

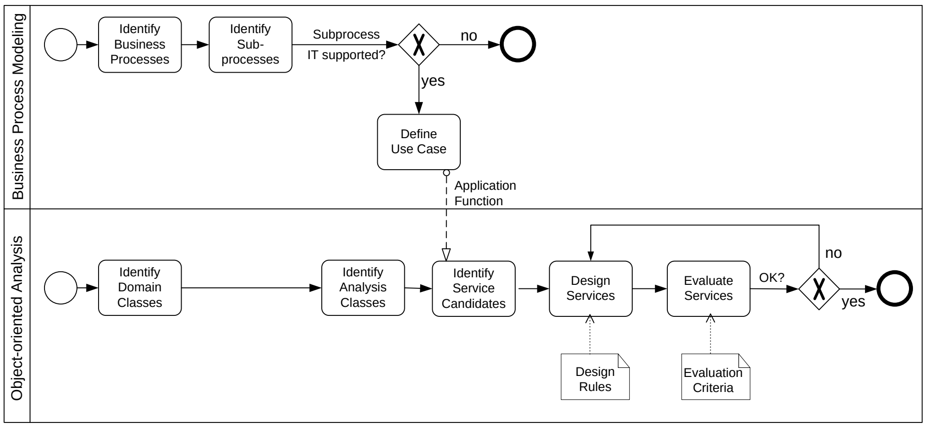

Fig. 2: Software Service Design Process in Case 1

4.2 Case 2: Oil- and Gas-Producing Company

Context: The company, headquartered in Norway, is one of the world's largest crude oil and gas suppliers with about 29,500 employees in 40 countries and more than 30

[1] Even if not explicitly stated, all trademarks mentioned in this article are registered trademarks of the respective companies.

years domain experience. Its IT department has circa 800 employees. The data-intensive application systems for the oil and gas core business (e.g., planning of lifting and cargo, trading, managing contractual documents) have been developed in-house in PL/SQL. Current software development relies on JAVA/J2EE for service providing and on .NET for clients. In addition to the custom-made application systems, the company also uses standard software (SAP R/3) for domain-independent business tasks such as accounting, invoicing and human resources. Moreover, the systems landscape integrates external information providers (e.g., Reuters and pricing agencies), information systems of business partners (e.g., ocean carriers) and the quite independent partner application systems at the offshore field sites.

The SOA project was initiated in conjunction with reengineering: The oil and gas core application systems, 20 years old, were to be replaced, since the old PL/SQL code became increasingly difficult to maintain. Service-orientation was chosen as architecture because of its openness and scalability, which makes it appropriate to cope with the developments on the oil and gas market (e.g., increased activities in new markets including China and India, more and smaller trades by private investors). Hence, some of the later service consumers cannot be anticipated.

Functionally the intended service-oriented solution should cover the core of the oil and supplies 'wet' supply chain (transport by ship). Later the functional coverage was extended to the gas core. Technological constraints resulted only from the systems landscape. Software qualities (e.g., security, performance) were not an issue.

The first web services were implemented around 2001 in the gas domain. Before (since around 1997), service-orientation had been realized in redesigning the PL/SQL functions and their interfaces. The following explanations refer to software service design principles and steps common to both situations.

Service design process (see Fig. 3): 'Application services' have been identified both top-down and bottom up. *Top-down* development started from high-level *business processes* that are collections of activities to achieve a business goal (e.g., 'plan transport', 'forecast replenishment'). These business processes were already available due to governance requirements. In contrast to workflows, the activities in business processes are ordered in strict sequence, and events only exist at the start or end of the business process. Both business processes and workflows are modeled by BPMN.

Business processes within a function area (e.g., 'shipping', 'trading') are grouped and analyzed to find similar activities, e.g., 'update cargo', 'calculate price' and 'calculate volume'. Abstractions of similar activities form candidate application services.

Perceiving similarities between activities is relieved by simultaneous *bottom-up* service design starting from information concepts. An *information concept*, e.g., 'line item' (a position of a deal), 'cargo' or 'pricing formula' (describes how the price for an oil deal should be set), is strongly interdependent information with similar life cycle. Information concepts reside on the logical level of system design, but may correspond to *business objects*, which are input or output of activities. Common operations on information concepts (often CRUD - create, retrieve, update, delete) form one type of bottom-up candidate application service; another type are functions provided by existing application systems and gathered by reverse engineering. Information concepts are modeled by UML class diagrams.

Application services such as `UpdateCargo` or `CalculationEngineService` (for price and volume calculations) express the contribution of a software system to a business process on a logical level. They represent functions with so high interdependencies (in terms of data usage, user interaction or business rules) that they are normally implemented together.

After their identification, the candidate application services are *refined* (grouped or split) by using additional information and the following heuristics:

- Application services are often required to handle start or end events of business processes (e.g., 'lifting program published').
- An application service must refer to the same information concept in the same context. For example, distinct loading operations and facilities are needed to handle the information object 'cargo' in the context 'ship' or 'dock', respectively. Hence, separate services must be defined.
- An application service must stick to the same business rules (for example, calculations and derivations).

Refinement leads to (grouped) functions (and their consumption of information) that form the base of *service specification*. In the beginning of the SOA project, there were no service specification guidelines - except for the principle that a service interface should be small and contain only stable operations. However, identification and refinement brought about three types of software services: (1) entity services (e.g., `TradeService;` mainly CRUD on information concepts corresponding to business objects), (2) task services (execution of process activities; related to no or several information concepts; e.g., `CalculationEngineService`) and (3) technology services that are not related to business (and, thus, no application services), but needed for the systems to operate (e.g., `SmXDataService` that provides a system with data from a data base). Currently, specification guidelines for these service types are prepared in the form of patterns.

At the time of writing more than 70 web services exist; each one has 3-5 operations. The web services run on IBM WebSphere Application Server 6.0 and communicate either through this platform or via Microsoft BizTalk. Non-web services exchange messages via data base read/write or the Oracle Messaging Gateway.

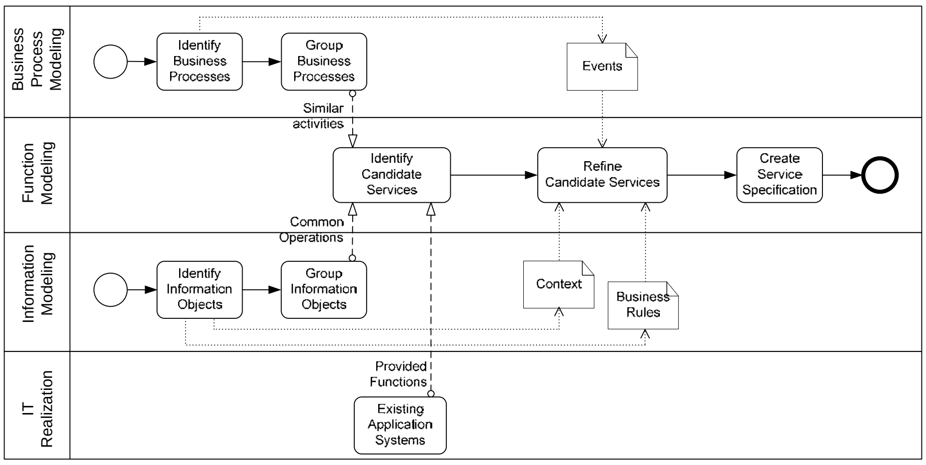
Business Process Modeling
Identify Business Processes
Group Business Processes
Events
Similar activities
Function Modeling
Identify Candidate Services
Refine Candidate Services
Create Service Specification
Common Operations
Information Modeling
Identify Information Objects
Group Information Objects
Context
Business Rules
IT Realization
Provided Functions
Existing Application Systems

Fig. 3: Software Service Design Process in Case 2

4.3 Comparison of the Cases

Both cases have a common *context*: (C1) They relate to SOA-based reengineering of existing application systems in non-IT companies, which have comprehensive domain experience. (C2) Both projects started nearly at the same time. (C3) Web services have been designed, implemented and deployed. (C4) Not all of the potential consumers of these web services can be anticipated or controlled.

Concerning *software services design*, the following commonalities can be stated: (D1) The service design processes were hybrid, i.e., both top-down and bottom-up. (D2) Top-town design started from ‘business processes’, bottom-up design from information objects. (D2a) Closer examination of the ‘business processes’ shows that the most distinctive process characteristic, the control flow between activities, was not considered. Instead, both companies used processes for a functional decomposition of their domains. Hence, top-down service design actually started from functional areas (see Section 2). (D2b) The information objects underlying bottom-up service design express main concepts of the domain (domain ontology) and were gathered from the domain knowledge of experienced employees. (D2c) The predominant way of thinking was bottom-up, i.e., increasing abstraction by grouping functions that refer to information objects or business tasks. (D3) Distinct types of services have been designed whose functionality is related to either business (objects or cross object tasks) or technology. (D4) The SOA design principles ‘reusability’, ‘cohesion’, ‘loose coupling’ and ‘abstraction’ were involved in either design process; additionally, stability (of the operations that form an interface) was used as a criterion for service design.

5 Conclusions and Future Work

This case study has shown that, in contrast to most of the proposed approaches, software service design in practice is always hybrid, i.e., it proceeds both top-down (decreasing abstraction) and bottom-up (increasing abstraction). None of the hierarchical approaches explicitly considered bottom-up service design starting from information objects[2], which is, however, common practice; some of the recently presented service design patterns (e.g., ‘utility abstraction’, ‘entity abstraction’, ‘process abstraction’ [4]) follow this tack. As opposed to the academic top-down approaches, strict top-down derivation of software services from business processes was not observed; instead, processes were used for a functional decomposition of the domain (alike [14]) - the control flow within processes was ignored. Thus, functions and not processes drive service design. Functionality and software quality are the only ‘goals’ considered during service design.

Finally, service design in practice ultimately leads to *service layers*, i.e., services that can be grouped according to the type of functionality they provide (business object services, business task services, technical services). Service layering as a special form of grouping is neglected in purely academic approaches, but mentioned in hierarchical service design approaches that originate from practice (e.g., [7], [6]) and in principle-driven service design approaches [3]. Altogether, focusing on functions

[2] Only very simple entity services are suggested in [6], [7].

and information objects as well as layering services point to required extensions of existing SOA design approaches.

Since both cases considered here stem from distinct branches, and the SOA projects were conducted independently of each other, the results are generalizable. To increase the validity of the results, additional cases will be analyzed that have a slightly different context (e.g., green-field development, development by IT companies). Differences in context can lead to contradicting results, which are needed to specialize the conditions for particular service design processes and steps.

References

1. Arsanjani, A., Ghosh, S., Allam, A., Abdollah, T., Ganapathy, S., Holley, K.: SOMA: A method for developing service-oriented solutions. IBM Systems Journal 47 (2008) 377 -396
2. Erradi, A., Anand, S., Kulkarni, N.: SOAF: An Architectural Framework for Service Definition and Realization. In: Proc. IEEE Int. Conf. on Service Oriented Computing (SCC 2006). IEEE, Los Alamitos (2006)
3. Erl, T.: SOA Principles of Service Design. Prentice Hall, Upper Saddle River et al. (2008)
4. Erl, T.: SOA Design Patterns. Prentice Hall, Upper Saddle River et al. (2008)
5. Kaabi, R.S., Souveyet, C., Rolland, C.: Eliciting service composition in a goal driven manner. In: Aiello, M. et al. (eds.): Proc. of the Second Int. Conf. on Service Oriented Computing (ICSOC 2004). ACM Press, New York (2004) 308-305
6. Klose, K., Knackstedt, R., Beverungen, D.: Identification of Services - A Stakeholder-based Apporach to SOA development and its application in the area of production planning. In: Österle, H. et al. (eds.): Proc. of the 15th European Conf. on Information Systems (ECIS 2007). St. Gallen (2007) 1802-1814
7. Kohlmann, F.: Service identification and design - A Hybrid approach in decomposed financial value chains. In: Reichert, M., Strecker, S., Turowski, K. (eds): Proc. of the 2nd Int. Workshop on Enterprise Modeling and Information Systems Architecture (EMISA ’07). Koellen-Verlag, Bonn (2007) 205-218
8. Papazoglou, M.P., van den Heuvel, W.-J.: Service-oriented design and development methodology. Int. Journal of Web Engineering and Technology 2 (2006) 412-442
9. Papazoglou, M.P., Yang, J.: Design Methodology for Web Services and Business Processes. In: Buchmann, A. et al. (eds.): Third Int. Workshop on Technologies for E-Services: (TES 2002). LNCS, Vol. 2444, Springer, Berlin et al. (2002) 175-233
10. Quartel, D., Dijkman, R., van Sinderen, M.: Methodological support for service-oriented design with ISDL. In: Aiello, M. et al. (eds.): Proc. of the Second Int. Conf. on Service Oriented Computing (ICSOC 2004). ACM Press, New York (2004) 1-10
11. Yin, R.K.: The Case Study as a Serious Research Strategy. Knowledge: Creation, Diffusion, Utilization. 3 (1981) 97 - 114
12. Yin, R.K.: Case Study Research: design and Methods. 4rd ed., SAGE Publications, Thousand Oaks (2004)
13. Zhang, Z., Liu, R., Yang, H.: Service Identification and Packaging in Service Oriented Reengineering. In: Chu, W.C., Juristo Juzgado, N., Wong, W.E. (eds.): Proc. of the 17th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE '2005). Skokie (2005) 620-625
14. Levi, K., Arsanjani, A.: A Goal-driven Approach to Enterprise Component Identification and Specification. Communications of the ACM 45 (2002) 45-52
15. Zimmermann, O., Craes, M., Milinski, S., Oellermann, F.: Second Generation Web Services-Oriented Architecture in Production in the Finance Industry. In: Companion to the 19th annual ACM SIGPLAN Conf. on Object-oriented programming systems, languages, and applications (OOPSLA 2004). ACM Press, New York (2004) 283-289