The Service-Oriented Metaphor Deciphered

Dirk Draheim
IT Center, University of Innsbruck, Innsbruck, Austria
draheim@acm.org

Received 1 November 2010; Revised 6 December 2010; Accepted 13 December 2010

In this article we review the metaphor of service-oriented architecture for enterprise computing. In typical definitions service-oriented architecture appears as a single message and a consistent roadmap for building flexible software system landscapes. But it is not. Different communities have elaborated different SOA (service-oriented architecture) concepts to address different problem areas, i.e., enterprise application integration, business-to-business, business process management, and software productizing. If software architects and software managers are aware of these strands of SOA when talking about SOA in their projects they can avoid misunderstandings and detours better. This article contributes a clarification of the different strands of SOA concepts and technologies and their mutual dependencies and identifies particular SOA concepts as instances of more general software engineering principles.

Categories and Subject Descriptors: Computing Milieux [**Management of Computing and Information Systems**]: Software Management

General Terms: Distributed Systems, Object-oriented Programming, Software Architectures, Reusable Software, Communications Applications

Additional Key Words and Phrases: Service-Oriented Architecture, Enterprise Application Integration, Electronic Data Interchange, Business Process Management, Software Process, Software Reuse, Software as Service, Cloud Computing

1. INTRODUCTION

In the last decade service-oriented architecture has been a leading metaphor in software architecture, both in industry and academia. This means many software products shipped with the label service-oriented, many technologies were designed around a notion of service orientation, know-how on service orientation was requested by many IT stakeholders from IT consultants in IT and software development projects, many research projects were conducted to elaborate service-oriented concepts and so on. We review the promises and actual usages of service-oriented architecture in the realm of enterprise computing. In typical definitions service-oriented architecture

appears as a single message and a consistent roadmap for building flexible software system landscapes. But it is not. Different communities have elaborated SOA concepts to address different problem areas.

The motivation of the article is the following. If software architects and software managers are aware of the concrete strands of SOA when talking about SOA in their projects they can avoid misunderstandings and detours better. The method of the article is to identify the well-distinguishable visions of SOA in the domain of enterprise contributes and to characterize their key objectives and rationales with respect to each of them. With respect to the identification of SOA visions the article's result is that we need to distinguish between SOA as a paradigm for approaching the following problem areas – see also Figure 1:

– Enterprise application integration.
– Business-to-business integration.
– Business process management.
– Software productizing.

As a by-product, the article wants to empower the reader we to answer questions like the following. Is service-oriented architecture actually architecture? Is it an architectural style, a kind of systems thinking, a software development paradigm or even a quality management approach?

In [Bernstein 1996] Phil Bernstein has identified or envisioned a trend of enterprise computing facilities becoming a kind of utility in enterprise system landscapes. The vision is about decoupling information services from appliances that can consume these services. It is about overcoming silo system landscapes and creating a new degree of freedom in using enterprise applications and databases. The information utility vision is discussed from the viewpoint of an implementing middleware. Similarly, in [Schulte and Natis 1996; Schulte 1996] the concept of a service-oriented architecture is described as being about avoiding the tight coupling of processing logic and data. Later, a concrete pattern of service-oriented architecture emerged with the target to create some kind of enterprise-wide information utility. In another strand of work service-oriented architecture evolved into the vision of creating not only enterprise-wide information utilities but also a world-wide information utility, i.e., an inter-enterprise-wide information technology.

In early discussions, service-oriented architecture was rather about the principle of decoupling services from appliances, i.e., the observable trend of some kind of information utilities emerging in large scale enterprise architectures. Some definitions of service-oriented architecture, e.g., the one given in [Gartner Group 2004] boil down to an explanation of modularization. However, the objective of a certain level of decoupling of enterprise applications is a unique selling point of service-oriented architecture one should be aware of. Over time it has become common sense that there are certain design rules and key characteristics that make up service-oriented architecture. To these belong [Brown et al. 2002] coarse-grainedness, interfacebased design, discoverability of services, single instantiation of components, loose coupling, asynchronous communication, message-based communication. Further such characteristics are strict hierarchical composition, high cohesion, technology independency,

idempotence of services, freedom from object references, freedom from context, stateless service design – see, e.g., [Colan 2004; Erl 2007]. These SOA principles can add value to enterprise applications, e.g., discoverability can greatly improve reuse, targeting coarse-grainedness can guide code productizing efforts and message-orientation is the proven pattern in enterprise application integration. The problem is, however, that the SOA principles are by no means silver bullets [Brooks 1975; 1987] for enterprise application architecture, least of all the combination of them. This means, the circumstances under which the SOA principles add value and how they can be exploited must be analyzed carefully. It is important to recognize that with respect to enterprise system architecture the promise of SOA, i.e., the creation of an information utility, is an objective rather than a feature of the approach. Best practices have to be gathered in order to make SOA work in projects. There is also a demand for heuristics like SEI's (Software Engineering Institute) SMART (Service-Oriented Migration and Reuse Technique) technique [Lewis et al. 2005] to estimate and mitigate risks of using SOA and SOA technology in concrete projects.

Some of the key characteristics usually connected with service-oriented architecture are know from other paradigms like component-based development or objectoriented design, others are unique selling points of service-oriented architecture and others explicitly distinguish service-oriented architecture from other approaches. For example, single instantiation of components explicitly distinguishes serviceorientation from component-orientation. Loose coupling, high cohesion and interfacebased design are also targeted by object orientation. However, in the context of service-oriented architecture, loose coupling is often understood differently from how it is understood in the object-oriented community. In the object-oriented community it stands for the general pattern of minimizing the overall number of any kind of dependencies between components in a system. In service-oriented architecture it is often understood as the availability of a dynamic binding mechanism. The issue of discoverability of services is often seen as crucial for service-oriented architecture. What is meant is special support for discovering in the form of appropriate registries and tools [Brown 1997]. Discoverability of services touches the assembly and eventually the organization of development and maintenance of services, a strand of work that is currently addressed by SOA governance. On the other hand, discoverability is at the core of the business-to-business vision of service-oriented architecture that is discussed in due course.

One of the mainstream perceptions of service-oriented architecture is that it is a technology independent meta architecture for large-scale enterprise computing [Natis 2003]. There are other communities that use the service-oriented architecture metaphor for their technology, for example, OSGi – note that OSGi stand for Open Service Gateway initiative. Standard applications of OSGi technology are in the area of embedded systems. Another such example is Jini [Waldo 1999]. Jini technology has discoverability of services, which is considered as a key characteristics of service-oriented architecture, as a crucial asset and indeed Jini is often said to establish a service-oriented architecture [Mahmoud 2005]. Jini targets the level of network-connected devices, whereas service-oriented architecture as discussed in this book targets the creation of information utilities in large-scale enterprises. We believe that

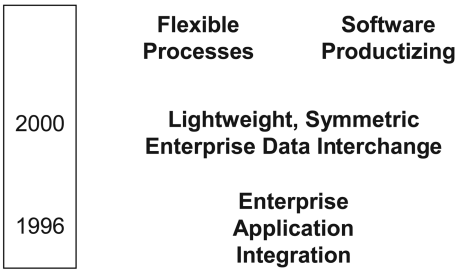

Figure 1. The evolution of SOA paradigms and visions.

an architectural paradigm should be discussed always with its domain objectives in mind in order to have a context to appreciate or criticize the several design principles it imposes. The currently discussed OASIS (Organization for the Advancement of Structured Information Standards) reference model for service-oriented architecture [MacKenzie et al. 2006] mentions large-scale enterprise applications as a main driver of service-oriented architecture, however, it does not mention other drivers and the reference model in [MacKenzie et al. 2006] abstracts away from enterprise computing.

Another perception of service-oriented architecture is that it is connected to the concrete web services stack technology stack. Actually web services technology has substantially contributed to the wide-spread adoption of service-oriented architecture. Nonetheless, it is important to point out that service-oriented architecture began as a technology-independent discussion, e.g., CORBA (Common Object Request Broker Architecture) technology can be very well used to build a concrete service-oriented architecture [McCoy and Natis 2003]. In general, service-oriented architecture is about system architectural concepts for enterprise computing and not about a particular technology.

As should now have become clear, service-oriented architecture is not a single paradigm. There are several service-oriented architecture visions and concepts that emerged over time as depicted in the overview of Figure 1. Service-oriented architecture emerged as a three-tier architecture for enterprise application integration as we will explain in Sect. 2. Then the term service-oriented architecture was hyped by the concrete web-service technology establishing the vision of creating a new lightweight, ubiquitous and open electronic data interchange platform based on this web-services technology. We will deal with this EDI (electronic data interchange) aspect of web-services technology and SOA in Sect. 3. As a result, web-services technology has become a synonym for service-oriented architecture for many stakeholders in the enterprise computing community. This is the reason why service-oriented architecture is now recognized as an enabler for flexible, adaptive business processes, i.e., simply by the wide-spread usage of web-services technology in state-of the art commercial business process management (BPM) technology products. We will deal with this BPM aspect of web-services technology and SOA in Sect. 4. Another strand of argumentation in service-oriented architecture is in the area of software development methodology and systematics. Here, inspired by the new tools for component maintenance, and in particular those that deal with service discover-

ability, innovators where thinking about new opportunities to accelerate the software development process. For example, the systematic integration of massive reuse of software components into software development projects was envisioned. Recently, the massive reuse of software components stemming from the multitude of projects in a large enterprise are considered, typically in the realm of SOA governance. We will deal with this software productizing aspect of SOA in Sect. 5. In Sect. 6 we provide an outlook onto current trends ongoing trends in service-oriented architecture and summarize our motivation and results in Sect. 7.

2. THREE-TIER SERVICE-ORIENTED ARCHITECTURE

Figure 2 shows Gartner's original tier terminology of service-oriented architecture from [Schulte and Natis 1996]. The architecture in Figure 2 should be understood as one out of many service-oriented architectures in the sense of [Schulte and Natis 1996]. According to the many possible different software clients, it is possible to build different service-oriented architectures. As always with the notion of "architecture", the term architecture can be used for the structure of a concrete system with all its considered components at the chosen appropriate level of abstraction, but can also be used for the common, i.e., generalized, structure of all the concrete systems that adhere to a certain architectural concept. In that sense service-oriented architecture as intended by [Schulte and Natis 1996] is rather characterized by the distinction between tiers *A* through B and their explanation and Figure 2 with all its concrete clients is an example architecture.

Tier *A* in Figure 2 is a tier of services that realize business logic and access to data that are common to several applications in the enterprise. Tiers *B* and *C* together consists of the enterprise applications that are software clients to the tier *A* of services. We do not delve into the distinction between *B* and *C* here, in [Schulte and Natis 1996] it is somehow used to explain the difference between batch and online users. We want to discuss the role of the service-tier and the options to build it instead.

In general we want to decide between the following kinds of service-oriented architecture:

- Fat service hub hub-and-spoke architecture, fat hub-and-spoke architecture for short,
- Thin service hub hub-and-spoke architecture, thin hub-and-spoke architecture for short.

In the fat hub-and-spoke architecture the service-tier actually implements some business logic and holds some data. With this architecture business logic that is necessary for several applications can be realized and maintained at a single spot, i.e., the service tier. In its extreme form, however, the fat hub-and-spoke architecture does not allow the service-tier to reuse business logic or data from the other software applications of tiers *B* and *C*. Here, all the logic and data that is provided by the service tier *A* must be implemented by the service tier. In its extreme form the fat hub-and-spoke architecture is therefore not about the mutual reuse of business logic and data that reside in the several applications of the enterprise. It is about this

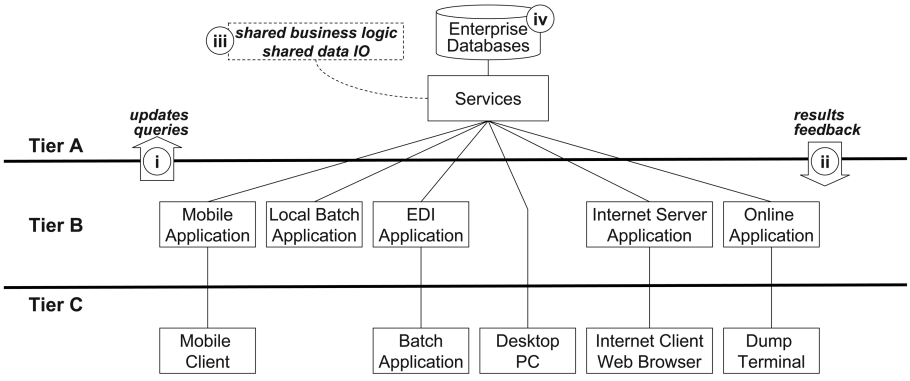

Figure 2. Gartner Group tier terminology for service-oriented architecture.

mutual reuse only in the sense that these logic and data can be taken from the several applications and be brought to the service tier *A* in a kind of overall system refactoring and system landscape refactoring step.

The obvious interpretation of the arrows labeled (i) and (ii) in Figure 2 leads us to the conclusion that the service-oriented architecture introduced in [Schulte and Natis 1996] is actually of the described extreme form of fat hub-and-spoke architecture. In Figure 2 updates and queries, which are represented by arrow (i), can be made by the software applications of tier *B* to the service tier *A* but not vice versa. Similarly, as depicted by arrow (ii), results and error messages can be delivered from the service tier to the applications in the other tiers but not vice versa. We therefore call this form of fat hub-and-spoke architecture also uni-directional hub-and-spoke architecture. Also in the uni-directional hub-and-spoke architecture there are message flows back and forth between the service tier and the other tiers. However, with respect to both of the two kinds of different message flows, i.e., update or query, on the one hand, and result or error feedback, on the other hand, the message flow is actually uni-directional. Or to put it the other way round: with respect to a complete cycle consisting of an update or query and a result or error message all message flows are uni-directional.

In a thin hub-and-spoke architecture, the service tier does not implement any business logic and does not hold any data itself. Here, the service tier is merely a broker for the reuse of logic and data between the several applications in an enterprise. This means, a thin hub-and-spoke architecture is about dropping box (iii) and database (iv) from Figure 2. Consequentially, both of the arrows (i) and (ii) have to be replaced by bidirectional message flows in Figure 2.

The discussed thin and fat hub-and-spoke architectures are extremes, however. In practice a mixture of both styles can be applied in an enterprise application integration project. Some of the business logic and data is then realized in the service tiers, other business logic and data access is just realized by appropriate wrappers in the service tier. The thin hub-and-spoke architecture leads to bidirectional message flow both for updates and queries and for results and messages. We therefore call it also a bidirectional hub-and-spoke architecture. A thin hub-and spoke architecture implies bi-directionality; however, the converse is not true. If the service-tier realizes some

business logic merely by being a broker, there can also be some direct realization of business logic in the service-tier. That distinguishes the bidirectional from the uni-directional hub-and-spoke architecture. A fat hub-and-spoke architecture does not imply uni-directionality, but vice versa.

A bidirectional hub-and-spoke architecture is the one that is found most often in practice. In concrete companies other names for the service tier are often used, for example, 'data wheel', 'business logic pool' or the like. Even names that contain a reference to EDI are often used, the term 'EDI pool' is a good example for this. If the service tier is only used for in-house software reuse purposes, a name containing EDI is strictly speaking incorrect, because electronic data interchange usually stands for inter-enterprise data interchange. However, in the true sense of the word electronic data interchange is a correct characterization of the functionality of a service tier. Business logic can be triggered by sending a data message via a so-called EDI pool and the answer of triggered business logic is also just data send back to the requester. Business logic can be understood very broad. Even functionality that deals with application workflow like a workflow system's task list can be revealed by a purely data interchanging message interface to the other applications in an enterprise. From that somehow low-level viewpoint every interaction among applications can be understood merely as data interchange. Furthermore, often one of the products that are in the B2B (business-to-business) gateway market segment [Lheureux and Malinverno 2008] like Microsoft BizTalk or TIBCO Software, just to name two, are used to build a service-tier, even, if it is only used for intra-enterprise data communication purposes.

A bidirectional hub-and-spoke architecture is also a common approach to the integration of legacy systems with emerging new systems, which is a common theme in enterprise application integration. Also the currently discussed enterprise service busses are bidirectional message-oriented middleware products.

The three-tier architecture discussed here must not be mixed up with the widespread known three-tier enterprise application architecture. The three tier architecture discussed here is an architecture for system landscapes, whereas the usual three-tier architecture is almost always an architecture of single enterprise applications. The advantage of a hub-and-spoke architecture for a system landscape is that business logic and data that is shared by several applications is controlled and maintained at a single spot. A hub-and-spoke architecture prevents an uncontrolled growth in the number of connections and interfaces between the various applications. Furthermore, the hub can be made subject to concrete policies; it can be supported by specialized technology like enterprise service bus products or as mentioned before by B2B gateway products.

Sticking to a legacy system to avoid unjustifiable effort is a common theme that in enterprise application architecture is often called investment protection. The usage of the term investment protection is a bit odd for this. Another and perhaps more obvious usage of the term investment protection is as a synonym for risk management in investment planning. The kind of investment protection that leads to the preservation of an existing system as an integrated legacy system in a new system landscape can be seen as the consideration of an originally planned amortization

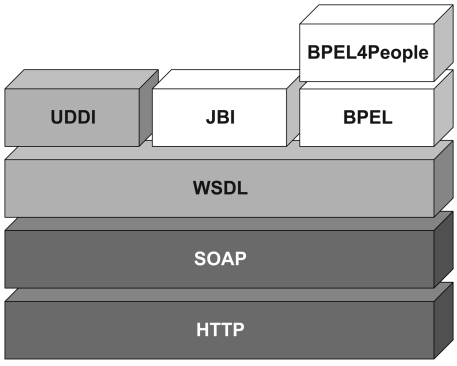

Figure 3. The stack of web services technologies.

period of an existing system in the planning of a new system. If the amortization period is not yet over, stakeholders might be biased against replacing an old system. The investment into the old system is then considered somehow as an asset that is worth saving in its own right. This is a bit odd, because the planning of a new system landscape should be done on the basis of systematic total cost of ownership and total economic impact analyses. In such analyses, the existing systems can be considered merely as a pool for software reuse like third-party products that are considered in build-or-buy decisions. A neutral viewpoint is also necessary particularly in presence of existing applications.

3. BUSINESS-TO-BUSINESS BASED ON SOA

Web services technology helped to make service-oriented architecture mainstream. After its huge success, Internet technology seemed to be the natural candidate for the service binding mechanism of service-oriented architecture. Concrete new products of major software vendors like IBM's Websphere and Microsoft's. NET were based on the web services technology stack. These products with their integrated development environments allowed for easy distributed programming via the web. In its original definition, service-oriented architecture is a set of technology independent concepts, in particular, it is independent from the web services stack. However, in practice service-oriented architecture is often discussed with web services in mind. Here, in concrete projects they are often very concrete technical aspects like the easier interplay with firewalls that influence the decision to use service-oriented architecture in favor of another technology.

The web services technology stack – see Figure 3 – has SOAP (Simple Object Access Protocol) [Box 2000] on top of HTTP as the basic remote method invocation mechanism. The Web Services Description Language (WSDL) is used for the specification of web services signatures, i.e., it provides the web services type system in terms of programming language technology.

The crucial point with the web services technology stack is that it goes beyond WDSL. It has UDDI (Universal Description, Discovery and Integration) on top. The UDDI standard was intended to build global yellow pages for subscription and

description of services by enterprises. With UDDI, service-oriented architecture became an electronic data interchange (EDI) initiative. The term business-to-business was coined with Internet technology and typically B2B is used for web services based electronic data interchange, whereas EDI is used for electronic data interchange based on established formats like the X12 transaction sets or UN/EDIFACT [International Organization for Standardization 1988] – EDIFACT (Electronic Data Interchange For Administration, Commerce, and Transport) for short. As an EDI initiative, the goal of the web services stack was to create a lightweight, open alternative to existing, established value-added networks. It is this new envisioned EDI the term B2B is used for. Established EDI scenarios are often asymmetric from a business-to-business viewpoint. A very typically example is provided by large manufacturers or large retailers that requests their smaller suppliers to be connected to them by EDI. At the same time established EDI is known to be quite heavyweight – technologically and organizationally [Emmelhainz 1993] – to adopt.

However, improved inter-organizational EDI is currently addressed more visibly by other standardization bodies like UN/CEFACT (United Nations Centre for Trade Facilitation and Electronic Business)and OASIS [Business Process Project Team 2001; UN/CEFACT 2006] (Organization for the Advancement of Structured Information Standards), which are experienced in the EDI domain. In the original strand of work on B2B, i.e., with UDDI as a world-wide yellow pages mechanism for service-enabled enterprises, B2B did not take off.

The problem with EDI is that parts of its heavyweightedness cannot be overcome simply by a new technology stack. This is the issue of pre-negotiations between EDI participants before an EDI connection is actually established. These negotiations are business related and cannot be automated, they encompass the negotiation of prices, quality of services, and contract penalties. Furthermore the results of these negotiations must be made justifiable. Paper-based trading has a long legal tradition but electronic trading had to become mature before a robust legal foundation for it could be provided. Furthermore, it must not be overlooked that the success of established EDI was only partly due to the creation of a technological infrastructure for electronic data interchange. Another success factor of established EDI initiatives was the standardization of business messages, which is a technology independent asset for the community. The original B2B initiative forgot about message standardization in the beginning.

Today more proprietary standards and technologies are based on WDSL representing two aspects of enterprise system architecture, i.e., enterprise application architecture and business process execution. JBI [Ten-Hove and Walker 2005] (Java Business Integration) is an attempt to standardize the notion of enterprise service bus (ESB), which is a currently widely discussed approach to enterprise application architecture. BPEL (Business Process Execution Language) is a currently discussed language for the execution of automatic workflow.

4. BUSINESS PROCESS TECHNOLOGY BASED ON SOA

SOA technology and business process technology are converging [Leymann et al. 2002]. The relationship between SOA and business processes has been somehow misrepresented, because despite its name, the SOA-based Web Services Business

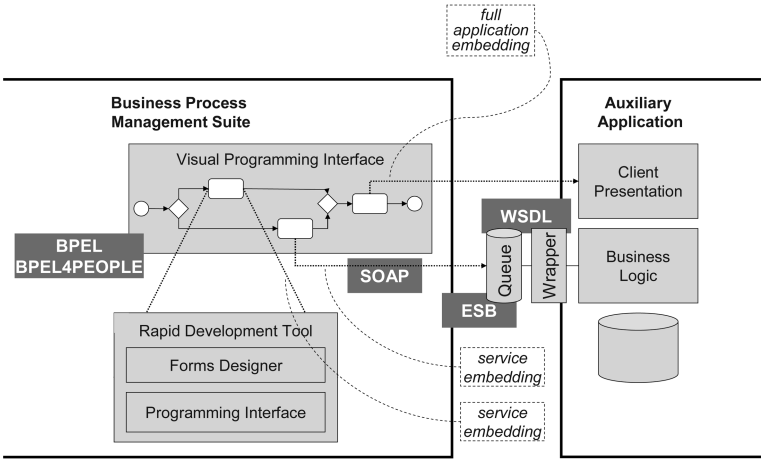

Figure 4. Typical business process management systems based on web-service technology.

Process Execution Language [Thatte 2003; Evdemon and Jordan 2007] (WS-BPEL) – BPEL for short – does not support crucial workflow concepts like user roles needed for business execution in today's business process management suites. BPEL is just a high-level programming language with web services as its primitives. Thus BPEL is a language for the execution of automatic workflow, however, it is not a language for the execution of human workflow. Therefore, concrete BPEL-based business process management products add there own, proprietary workflow concepts to BPEL. This need is also addressed by BPEL4People (WS-BPEL Extension for People) [Kloppmann et al. 2005; Agrawal 2007]. The current SOA-based business process management suites tackle the problem of how to exploit SOA technology in workflow-intensive applications; however, they do not raise the level of abstraction towards executable specification of business processes.

Figure 4 shows the typical exploitation of concrete web services technologies by contemporary business process management products. We distinguish several means to glue together software components with a business process management suite in Figure 4, i.e., full application embedding, service embedding and dialogue embedding. In the scenario in Figure 4, the business process management suite is used to combine several existing applications to a new one and to enrich them with new system dialogues that are developed with technology provided by the business process management suite itself. The new application logic is developed in the business process management suite with a visual programming language and tool. Here, a visual form of the web-service technology BPEL and a proprietary extension for human workflow is typically used allowing new system dialogues to be hooked into the visual program. For the development of these new system dialogues the business process management suite offers support via a rapid development tool that typically ships with a visual programming tool for form-based computer screens. The new dialogues can also be implemented without the rapid development tool that ships with the business process management suites. They are then new auxiliary applications in

the sense of the drawing in Figure 4.

Application can be integrated into the new developed application by actually embedding their dialogues. We call this style of application integration 'full application embedding' in order to distinguish it from reusing merely the business logic of an existing application. For dialogue embeddings of auxiliary applications no elaborate high-level standard mechanisms exist. They can be accomplished rather directly with program calls on the operating system level with its low-level program parameter passing mechanisms. Often, auxiliary applications are prepared with message-based application programming interfaces that then can be used to call the desired dialogues. For this, the web-service technology WSDL would be the current language of choice in building the necessary wrappers. We have not depicted this in Figure 4.

It is possible to integrate only the business application into the new developed application called service embedding in Figure 4. Here WSDL is typically used for building the wrappers and web-services technology SOAP is the transmission protocol of choice. One important issue that leads to a business process management project is often that users are not satisfied any more with the HCI (human-computer interaction) layer of the existing applications. Existing applications are often built with one of the many proprietary rapid development tools of the 1980s. So the goal is then to replace the HCI layer by new web-based dialogues or new state-of-the-art rich client interfaces with a modern look-and-feel. On the other hand, the target of replacing the existing system dialogues is usually not the only reason for a business process management project. Usually, there is actually the need for new application logic. At the same time the programs that make up the dialogues can make up a significant code base that cannot be replaced in one go. Therefore a typical migration path would start with building new application logic around the existing applications on the basis of full application embedding and would then proceed with the step-by-step replacement of the existing dialogues eventually resulting in complete homogenous service embedding.

5. SOA AS AN ENABLER FOR SOFTWARE PRODUCTIZING

There is yet another facet of service-oriented architecture that is currently widely considered. It is the software development and operations aspect. What many IT stakeholders in enterprises expect from SOA is the transformation of the many software applications into a code base that is open for reuse in their future enterprise application development projects. This expectation is about breaking silo applications into services. And it is about getting large, even distributed software development teams under control, i.e., it is about software productizing, which is a term used by Frederick Brooks in [Brooks 1975].

It is common sense among developers that programming large system is fundamentally different from programming small systems. The larger a system becomes the more complex it becomes and you need special mechanisms to deal with the complexity. All the abstraction mechanisms in programming languages have the purpose to get complexity under control. The usual abstraction mechanisms found in programming languages are sufficient to build arbitrary layers of abstraction, so, basically, they are sufficient to deal with programs of any size. On the other hand, also small programs

in the sense of programming in the large can be large enough for requiring the usage of programming language abstraction mechanisms in order to get into control of their complexity. So, the question arises: why do we need to discuss mechanisms that go beyond the usual programming language abstraction mechanisms? Or to pose the question differently: when is a program large, i.e., large in the sense of programming in the large [DeRemer and Kron 1975]? One possible answer could be: programs are not large, projects [Brooks 1975] are. Projects that are conducted by large teams differ fundamentally form projects that are conducted by small teams. There is substantially more overhead for communication, need for system documentation, need for system integration, and need for project management.

5.1 Designing Services as a Basis for Massive Reuse

Software productizing refers to various extra initiatives that can be performed to make software a product, e.g., well-documented, well-tested and therefore deliverable to buyers, and more general in the sense of being prepared for adaptation to different contexts and platforms. We therefore use the term software productizing here for the extra effort needed to make a piece of software more generally usable, i.e., to make it reusable for other software products than the one it is currently developed for. If, at the time development, a developer already has some other concrete applications in mind, for example, some applications that are also currently under development or applications that are planned, the extra efforts in generalizing a piece of software can lead to a quick win. Otherwise, software productizing is about anticipating potential future software development. Then, software productizing must involve analyzing which kind of future applications the several pieces of currently developed software could be useful for. This involves an analysis of how the design of these applications will probably look or it involves efforts for this family of future potential applications. There is an obvious tradeoff between these software productizing efforts and the costs saved by reusing the resulting generalized pieces of software. Software productizing in the discussed sense is somehow conceptually opposite to the current trend of agile software development, with its courage-driven design [Beck 2000; Draheim 2006; 2003; Guta et al. 2009] and continuous refactoring.

SOA governance [Holley et al. 2006; Ziebermayr 2010] is the entirety of efforts that contribute towards making the promises of service-oriented architecture a reality [Holley et al. 2006]. Definitions of SOA governance contain high-level descriptions of the problems encountered in projects that try to make service-oriented concepts a reality in an enterprise. Typical SOA governance definitions put a focus onto the operations of software services, i.e., monitoring, controlling and measuring services [Holley et al. 2006]. However, in SOA governance projects it is often expected that a SOA governance expert gives some advise on how to organize the IT development to enable better reuse [Ziebermayr et al. 2007] of existing software across all project boundaries. This heavily affects software process and development team organization issues. For example, initiatives like the Smart SOA [IBM Corporation 2007] best practices catalogue – which by the way must not be mixed up with SEI's SMART (Service-Oriented Migration and Reuse Technique) approach – try to address these issues. In practice, there is often a very straightforward understanding of SOA

governance in terms of features of tools that support service-oriented architecture [Kenney and Plummer 2008]. The most typical tools are service registries and service repositories.

In concrete SOA governance projects there is the need for a diversity of specialized tools, technologies and techniques like an appropriate service development and operations infrastructure, a service versioning concept, a service development and operations process, tools for service analysis, development, profiling and monitoring. A good example for the complex needs in a SOA governance project is the concrete enriched meta-model for web services elaborated in [Derler and Weinreich 2006], which contains information about the origin of a service, the developer in charge of a web service, the staging state a of service, versions and releases of services, coupling of services, the usage structure of clients and services, auxiliary text documentation, UML representations of WSDL specifications, UML statecharts for service behavior descriptions, the service deployment structure and documentation of generated Java stubs.

5.2 Silo Projects

Often software projects are stand-alone projects, i.e., they encompass a whole software management lifecycle that is isolated completely separate from other software projects. We call such software development a silo software development. Let us assume that each single project involves the several distinguishable stages of requirement elicitation, design, implementation and operation without loss of generality. In practice, concrete projects usually have more sophisticated software process model. However, for the purpose of explaining the issue of massive reuse across project boundaries a coarse-grained conceptual stagewise model in the following is appropriate. We want to emphasize that we follow a descriptive, informative approach in our argumentation. This means that we do not want to be prescriptive or normative one, in particular, when we are talking about software engineering processes. A crucial point here is that there is typically a long period of operation and maintenance in the whole software lifecycle compared to the relatively short stages of analysis, design and implementation. The concept of silo projects serves the purpose of a starting point to conceptually understand the other scenarios of mega projects and intertwined projects in the sequel. However, the notion of silo project does not serve conceptual purposes only. Parallel silo projects actually exist in organizations.

In reality the operation phase of a software lifecycle is actually also a software maintenance phase. And maintenance in general covers much more than bug fixing. It involves handling change requests and implementing functionality that fulfils new requirements that emerge during operation. So, there is some fully-fledged software engineering with requirement elicitation, design, and implementation in the operations phase. If there is a critical amount of software engineering during operations the situation is modeled more appropriately by an iterative software development process. The difference is that the overall project in a iterative software development consists of several iteration projects. The overall project starts with some requirement elicitation, design and implementation work until a first stable version can be launched. In parallel to the operation of the first version a new project for the next iteration is

started. In that project new requirements are gathered, the design may be refactored, and then existing functionality is changed and new functionality is added according to the new requirements as an implementation step. All this can be seen as a reuse scenario. The new version of the system is actually a new system that reuses parts of the former version. A significant amount of the former system is usually reused. If only a few lines of new code are actually added to a very large software system you might say that this is not really an example of software reuse but just a small change to the system. But for us it is important to insist that we deal with reuse here. We do so because of the setting of our discussion which is iterative software development.

Therefore, all arguments concerning reuse in the sense of software engineering and the software system lifecycle in particular apply here, and they do so independently from the question of how large the reused part of software is compared to the new software added. Furthermore, the described scenario is a reuse on the several levels of artifacts. Code is reused as well as existing requirement descriptions and existing design. The crucial point is that once the new version of the system is ready to be launched, the former version of the system is shut down.

5.3 Mega Projects

We have said that an iterative project consists of several iteration projects. During the overall lifecycle of the software product more and more iteration projects may be conducted. It would be possible to say that an iteration project is a sub project of the iterative project it stems from. We deliberately do not use the word sub project for an iteration project, because we want to reserve the word sub project for another phenomenon in project management, which is worth considering with respect to the complexity of enterprise-wide software development. Large projects easily become so large that it is necessary to create a number of sub projects. The usual reason is that the project is too large to be handled by the available human resources in terms of number or skills of the company that conducts the project [Gillette 1996]. Then, sub projects have to be defined that can be given away to sub-contractors. Now, a systematic division of labor across several, usually distributed teams must be managed. The lack of resources may not be the only reason for creating a sub project structure. Another reason can be that the project is simply too large to be managed without a leveled management approach. The crucial point is not the distribution of labor over sub teams, but the organization of these sub teams. Sub projects have the characteristics of full-fledged projects. This means, each sub projects has its own project manager and its own project management infrastructure.

After the overall large project has been started a requirement elicitation takes place as a first phase. Some of the design is done. At least a coarse-grained design which allows for the division of the whole project into sub projects has to be done. Then the sub projects are distributed to sub teams. This distribution means extra efforts. The initial design must be particularly consistent and robust, because once the sub projects are running it is hard to change things. At the same time the sub product description must be sufficiently accurate. In each sub project further design is done and then implementation takes place. Eventually the results of the single sub projects have to be integrated. Again, this integration means extra efforts.

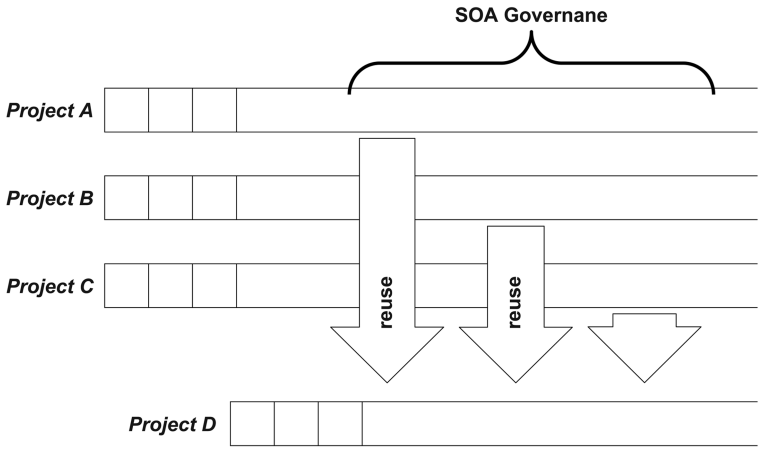

Figure 5. Reuse of software across projects.

5.4 SOA Governance for Ubiquitous Reuse

In an iterative project each iteration project reuses artifacts from the former iteration project. The projects in an enterprise are usually iterative projects, however, the single iterative projects can still be silo projects so that the reuse is limited to all the single projects. Things change fundamentally if we allow for reuse across project boundaries, which is depicted in Figure 5. The reuse across project boundaries must be managed. It is exactly that reuse, which gives rise to SOA governance.

Why is the reuse across project boundaries fundamentally different from the reuse inside a single project? Each of the projects has its own project management, its own software tools and repositories, its own development process and so on. If there should be reuse across project boundaries appropriate projects unifications must be conducted and, furthermore, extra organizational structure must be created. The more there is reuse across project boundaries the more the boundaries actually vanish and eventually the set of different projects must be handled as one huge project. All this is true from the static viewpoint of the mere code base, or more generally, from the viewpoint of the artifact base. Nonetheless it is also true for the dynamic viewpoint of running and evolving systems, which is depicted in Figure 6.

Figure 6 depicts a situation where a project reuses software from a product that evolves iteratively in another project. Assume that the new project B reuses some software while the first iteration of the iterative project A is still underneath. Now, assume that a next iteration of project B has been started and has just finished its implementation phase. The code is now to be deployed and made operative. Usually, the deployment of the new version in an iterative project would mean that the old version is replaced, i.e., that the old version is shut down. However, in the described scenario here this cannot be done without care.

Assume that in the new iteration the application has been changed at a place that effects software that is reused by project B. As a first problem, the stakeholders in project B must become aware that changes to the software in project A have happened

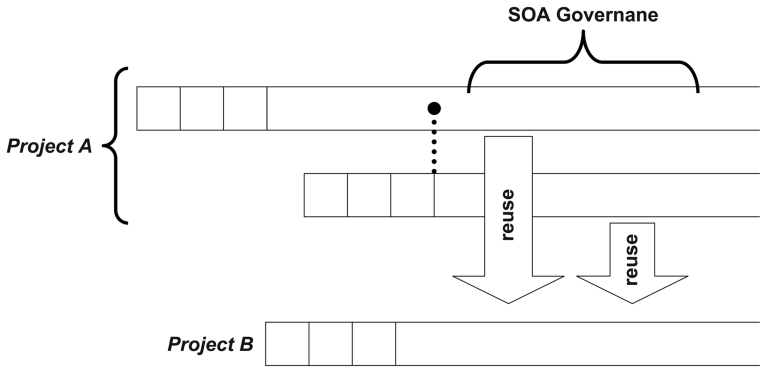

Figure 6. Reuse of software from a maintained product.

and then they must be able to determine whether the changes are relevant to the software of their project. This determination consists of two steps. First, they need to understand whether the changes are in any of the parts that have been reused. Then they have to decide, whether the changes should be adopted or not. For example, if the changes are rather bug fixes, project *B* very likely wants to adopt them. On the other hand, if the changes have been made on behalf of users of software product A that want the software to behave differently, it can also be, that these changes are not necessary and not appropriate for software product *B*.

Assume that the stakeholders of project *B* want to adopt a software change in project *A*. Then they need to distinguish between the two kinds of ways of realizing the reuse. One option is that software of project *A* was copied to project *B* where its copy has become a new part of the code base. Then this copy must be renewed by a new copy of the changed piece of software. But that is not all. In general, the product in project *B* needs also to be refactored. This is obviously so when the interface of the replaced software components has been changed. Beyond merely structural changes product *B* must also be refactored carefully with respect to the behavioral changes.

The other option for the realization of reuse in Figure 6 is that product *B* calls the application parts it needs as services. In the case the stakeholders of project *B* want to adopt a change to the software of project *A* they do not have to copy code, because it is there and running, but the real part of adoption, i.e., the refactoring of their product so that it fits the new version has still to be done.

We now turn to the really interesting case which makes things completely different from the silo project development. It is the case that stakeholders in project *B* do not want to adopt a certain change. In that case a version of the old software must be kept running in parallel to the new version. In Figure 6 we have depicted a solution where project *A* is responsible for the operation of the old version of the software, so that project *B* can keep using it as service by calling it. With respect to complexity this differs a little from a solution that copies the old version to project *B*, because the result remains the same, i.e., two versions of a software part are run and maintained henceforth in parallel.

Similarly, things become more complicated if a piece of software copied from

product *A* to *B* is changed sometime after its deployment by project *B*. Then a careful co-evaluation of both the changes in project *A* and project *B* is necessary and both kinds of changes have to be brought together by the developers in project *A*. This leads us to yet another observation with respect to reuse. So far, we have only considered the impact of changes to a software component on software that reuses this software. In general, however, also the other direction is relevant. A software piece can also be subject to change because of its usage in another project than the one it originally stems from. To realize the potential of all the software in an enterprise as a common code base for reuse the borders between the projects must be removed. At the next level all the projects with their sub projects and iteration projects can be seen as one huge project. It is the complexity of this general software development problem that must be properly managed by initiatives like SOA governance that promise to contribute to enterprise-wide software engineering.

All the described problems are exactly the problems of software variants. Getting a variety of instances of a software product under control is a concrete and severe problem in lot of software engineering and maintenance projects. Often, the problem is pervasive in projects and finding a smart solution to this problem is a central question. In real-world projects we observe that people that are under pressure to deal with the complexity of system variants introduce ad-hoc concepts to deal with the problem.

The need for a systematic approach to deal with variants of a system typically arises in what we call software service support scenarios. This means that there is a software vendor that sells some software product to a customer. However, though the product is a standard product in the product portfolio of the vendor, selling the product to the customer is not enough. The vendor cares also for the deployment and maintenance of the software product and, most importantly, for the adaptation of the product to concrete customer needs. This adaptation involves requirement elicitation activities with the customer, in particular, an analysis of existing and future business processes. If the necessary customer processes are not fully supported, new functionality has to be implemented. Often, existing functionality must be changed, so that the necessary processes are supported. Eventually, a new version of the product, i.e., a customer version, results. The choice to name the described scenario a software service support scenario is a particular good one, because it is very close to the problems addressed by the IT Infrastructure Library (ITIL) [Computer and Agency 2000; of Government Commerce 2002] service support process.

For the described scenario, it is not important whether the resulting customer version is actually deployed at the customer site or is run by the data centre of the vendor, i.e., it is not important whether the vendor is a software service provider. Similarly, it is not important whether the vendor is an independent software vendor (ISV) or a full-service commercial-off-the-shelf (COTS) software house. An independent software vendor is a software house that offers development of individual software solutions on an individual project basis. Actually, successful ISVs are often specialized in a certain sector. Then they usually have a proven code base for the solutions they develop. Often, there is no exact means to distinguish such a code base from a COTS product and sometimes it may be only a question of the vendor's marketing strategy

whether to mention this code base as an asset or not, and if so, whether to sell it as a framework solution or as a COTS product.

The variant problem is indeed also an issue for classical COTS vendors. This might puzzle the reader, because it is common sense that one of the disadvantages of COTS software is the assumption that it completely rules the business processes of the customer that deploys that software – as if there is no room for an adjustment of the COTS product. Actually, the converse is true. Part of the business model of a COTS software house is to sell the COTS product, but for some COTS software houses the services offered on basis of the COTS product has become the more important part of the business model, and so is the adjustment of the COTS product to the customer's needs.

A variant of a product is a version of a product. However, it is not a version in the usual sense of versioning, i.e., a product state in a software product lifecycle. Versioning is about the maintenance of deprecated versus actual versions of a software product, but variants in the described scenario are versions that fulfill different customer needs.

Because every product variant is the result of a new independent project with a customer, the need for mutual maintenance of the differences might not seem to be a key success factor for the single project and this might be the reason that it is not done in a lot of projects. It is simply not regarded as an urgent problem from the single project manager's point of view. Furthermore it is overhead. However, maintenance of variants pays off for two reasons, i.e., avoiding redundant efforts in product adoption projects and keeping variants consistent with versioning.

5.5 Software Use versus Software Reuse

We have discussed software reuse in development projects. Actually, IT stakeholders usually want SOA experts to say something about software reuse in their concrete projects. However, the earlier visions of service-oriented architecture concerning the fields of enterprise application integration, electronic data interchange, and flexible business processes were not originally visions about software reuse. In a strict sense the field of reuse is always also about design for reuse, i.e., the discipline of anticipating the reuse of a piece of software. For example, we believe that this viewpoint is widespread in the object-oriented software pattern community – note, that according to the subtitle of [Gamma 1995] design patterns are about reusable object-oriented software. SOA governance in the sense of best practices and tools is relevant for all the discussed SOA visions and not only the software productizing vision.

There is a huge potential of software reuse in large companies. On a global scale, there is even more potential for reuse through emerging service-orientation. One of the widely recognized visions of service-oriented architecture was the retrieval of services in global repositories as described in Sect. 3. This vision was tightly connected to the concrete UDDI standard of the web-services technology stack. This kind of service retrieval and use had the focus on electronic data interchange, i.e., business-to-business computing. This means, it was not in the first place about multi-tenant software use in the sense of SaaS (Software as a Service). It is a promising idea to see the global code base as a repository for reuse – see [Atkinson and Hummel 2007;

Atkinson et al. 2007] for a discussion and concrete supporting technology.

These observations also hold for cloud computing, which is next generation of SaaS. Cloud computing advances SaaS with respect to technical aspects and business model aspects. With respect to technical aspects it is about the application of advanced technology concepts like grid computing and virtualization. With respect to business model aspects it is about offering new kinds of services beyond the provision of standard application. Candidates for services in a cloud are all kinds of IT infrastructure service on the levels of operating system instances, file services, compute services, application programming interfaces, software tools and so on. Whereas SaaS has been a major issue in industry only, cloud computing attracts also the interest of academia.

6. SOA FUTURE TRENDS AND RESEARCH POTENTIAL

There have always been two strands of research to improve development and maintenance of software – both in practice and in academia: executable specification and reusable components. In the domain of enterprise applications the issue of executable specification is currently addressed by business process execution initiatives, the issue of reusable components is currently addressed by service-oriented architecture. And indeed, motivated by the promises of each of these current mainstream approaches we see a lot of projects where people try to bring them together, see e.g. [Chow et al. 2007; McNamara and Chishti 2006; Draheim and Koptezky 2007]. But then people encounter tensions between the two paradigms, because they were not designed for each other. How to exploit service-oriented architecture in a workflow-intensive information system scenario? How to implement workflow logic in a service-oriented manner? Then – though in theory service-oriented architecture is a technology-independent paradigm – in concrete projects these questions are sometimes approached very directly by asking where to establish web service layers in a workflow-based enterprise application. Basically, there are two alternatives. You can use web services to wrap units of logic that are controlled by the workflow engine. In current business process management suites web services are the usual invocation mechanism for business logic. Similarly, you can use web services to integrate system dialogues into the workflow logic. For example, with business process management suites it is possible to hook together system dialogues – typically developed with a rapid report and forms development tool – that bridge the workflow states. The other alternative is to reveal a whole workflow enactment service as a web services layer. Then, a fat client implements both the worklist as well as the system dialogues that bridge the workflow states. As you can see, such discussions easily lead to quite detailed technical questions. At the same time the question arises, whether wrapping of some code units in the one or other way really brings the speed ups and cost savings that one might expect from establishing a new paradigm – service-orientation in this case.

It is a commonplace now that service-oriented architecture is the natural candidate to bring the flexibility, agility and adaptivity to business process management suites and eventually to the enterprises that are supported by it. How come? Service-oriented architecture emerged as a kind of meta architecture for enterprise computing foreseeing a new more generalized, flexible role of single applications in an enterprise.

On the technical side, service-oriented architecture is exposed as clusters of design rationales and architectural principles. The problem is that the single design rationales and architectural principles are motivated from several different strands of thinking, for example, an overall system architecture viewpoint, a system management and maintenance viewpoint, a rather step-wise enterprise application integration viewpoint, and, very importantly, an electronic data interchange viewpoint.

The original vision of service-oriented architecture with its three tiers of services, applications and clients – see Sect. 2 – is rather a hub-and spoke architecture. The original service-oriented architecture is not a component metaphor, i.e., it is not about building arbitrary composition hierarchies or to, say it differently; it is not about assembling – wiring – services. This is where the Service-Component Architecture (SCA) [Weaver 2005; Beisiegel 2005] steps in. The Service Component Architecture addresses heterogeneous distributed computing. Heterogeneity is twofold in the case of the Service Component Architecture, it accounts for both several different implementation technologies and several different service binding technologies. Supported implementation technologies encompass languages of quite different styles. Supported service binding technologies are, for example, web services, JMS (Java Messaging Service), and CORBA IIOP. Furthermore, the Service Component Architecture systematically cares for the specification of quality of service. In its collection of standard profiles [Beisiegel 2007] are defined for security, reliability, personnel responsibilities and ACID transactionality [Robinson 2007].

7. CONCLUSION

This article aimed at contributing a clarification of the different strands of SOA concepts and technologies and their mutual dependencies and identified particular SOA concepts as instances of more general software engineering principles. Some key insights that have been presented are:

- We identified four well-distinguishable visions for service-oriented architecture, i.e., the enterprise application integration vision, the business-to-business-vision, the flexible processes vision and eventually the software productizing vision.
- We have identified two different styles of service-oriented architecture for enterprise application architecture which are basically distinguished from each other by whether the service tier implements business logic and holds persistent data and coin the terms fat hub resp. thin hub hub-and-spoke architecture for these architectural styles.
- We characterized SOA governance as an approach to massive software reuse.
- We have thoroughly motivated SOA governance as an approach to the maintenance of software product variants.
- We elaborated that software reuse can be distinguished from software use, i.e., that software reuse is the either a static use of arbitrary software or a dynamic use of multi-tenant software like SaaS and cloud computing.

The systematization and insights of the article can be used in projects to identify and prioritize more quickly the actual targets of the different IT stakeholders who are advocating a service-oriented architecture strategy, e.g., in an IT infrastructure project,

a software development project, in a reorganization of a software development department and so on.

REFERENCES

AGRAWAL, A. 2007. WS-BPEL Extension for People (BPEL4People), version 1.0. Tech. rep., Active Endpoints, Adobe Systems, BEA Systems, IBM, Oracle, SAP. June.

ATKINSON, C., BOSTAN, P., HUMMEL, O., AND STOLL, D. 2007. A Practical Approach to Web Service Discovery and Retrieval. In *Proceedings of ICWS 2007 – the 5th IEEE International Conference on Web Services*. IEEE Press.

ATKINSON, C. AND HUMMEL, O. 2007. Supporting Agile Reuse Through Extreme Harvesting. In *Proceedings of XP 2007 – the 8th International Conference on Agile Processes in Software Engineering and Extreme Programming, Lecture Notes in Computer Science 4536*. Springer.

BECK, K. 2000. *Extreme Programming Explained – Embrace Chang*e. Addison-Wesley.

BEISIEGEL, M. 2005. Service Component Architecture – Building Systems using a Service Oriented Architecture. Tech. Rep. Joint Whitepaper, version 0.9, BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, Sybase. November.

BEISIEGEL, M. 2007. ASCA Policy Framework, SCA Version 1.00. Tech. rep., BEA, Cape Clear, IBM, Interface21, IONA, Oracle, Primeton, Progress, Red Hat, Rogue Wave, SAP, Siemens, Software AG, Sun, Sybase, TIBCO. March.

BERNSTEIN, P. 1996. Middleware: a Model for Distributed System Services. *Communications of the ACM 39*, 2 (February), 86–98.

BOX, D. 2000. Simple Object Access Protocol (SOAP) 1.1 – W3C Note. Tech. rep. May.

BROOKS, F. 1975. *The Mythical Man-month – Essays on Software Engineering*. Addison-Wesley.

BROOKS, F. P. 1987. No Silver Bullet – Essence and Accidents of Software Engineering. *IEEE Computer 20*, 4 (April).

BROWN, A. 1997. CASE in the 21st Century – Challenges Facing Existing Case Vendors. In *Proceedings of STEP'97 – the 8th International Workshop on Software Technology and Enginering Practice*. IEEE Press.

BROWN, A., JOHNSTON, S., AND KELLY, K. 2002. Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications. Tech. rep., Santa Clara, CA: Rational Software Corporation.

BUSINESS PROCESS PROJECT TEAM. 2001. ebXML Business Process Specification Schema, Version 1.01. Tech. rep., UN/CEFACT, OASIS.

CHOW, L., MEDLEY, C., AND RICHARDSON, C. 2007. BPM and Service-Oriented Archtiecture Teamed Togehter: A Pathway to Success for an Agile Government. In *2007 BPM and Workflow Handbook*, L. Fischer, Ed. Workflow Management Coalition, 33–54.

COLAN, M. 2004. Service-Oriented Architecture expands the Vision of Web Services – Characteristics of Service-Oriented Architecture. Tech. rep., IBM Corporation. April.

COMPUTER, C. AND AGENCY, T. 2000. IT Infrastructure Library – Service Support. Tech. rep., Renouf.

DEREMER, F. AND KRON, H. 1975. Programming-in-the-Large Versus Programming-in-the-Small. In *Proceedings of the International Conference on Reliable Softwar*e. ACM Press, 114–121.

DERLER, P. AND WEINREICH, R. 2006. Models and Tools for SOA Governance. In *Proceedings of TEAA 2006 – International Conference on Trends in Enterprise Application Architecture, Lecture Notes in Computer Science 4473*, D. Draheim and G. Weber, Ed. Springer.

DRAHEIM, D. 2003. A CSCW and Project Management Tool for Learning Software Engineering. In *Proceedings of FIE 2003 – Frontiers in Education: Engineering as a Human Endeavor*. IEEE Press.

DRAHEIM, D. 2006. Learning Software Engineering with EASE. In *Informatics and the Digital Society*, Tom J. van Weert and Robert K. Munro, Ed. Kluwer Academic Publishers. 2003.

DRAHEIM, D. AND KOPTEZKY, T. 2007. Workflow Management and Service-Oriented Architecture.

In *Proceedings of SEKE 2007 – The 19th International Conference on Software Engineering and Knowledge Engineering*.

EMMELHAINZ, M. 1993. *EDI: A Total Management Guide*. Van Nostrand Reinhold.

Erl, T. 2007. *SOA: Principles of Service Design*. Prentice Hall.

EVDEMON, J. AND JORDAN, D. 2007. Web Services Business Process Execution Language Version 2.0. Tech. Rep. OASIS standard wsbpel-v2.0-OS, OASIS. April.

GAMMA, E. 1995. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GARTNER GROUP. 2004. The Gartner Glossary of Information Technology Acronyms and Terms. Tech. rep., Gartner Group.

GILLETTE, W. 1996. Managing Megaprojects: a Focused Approach. *Software 13*, 4.

GUTA, G., SCHREINER, W., AND DRAHEIM, D. 2009. A Lightweight MDSD Process Applied in Small Projects. In *Proceedings of SEAA 2009 – the 35th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE Computer Society.

HOLLEY, K., PALISTRANT, J., AND GRAHAM, S. 2006. Effective SOA Governance. Tech. Rep. IBM White Paper, IBM Corporation. March.

IBM CORPORATION. 2007. Smart SOA: Best Practices for Agile Innovation and Optimization. Tech. Rep. IBM White Paper, IBM Corporation. November.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. 1988. International Standard ISO 9735. Electronic Data Interchange for Administration, Commerce and Transport (EDIFACT) – Application Level Syntax Rules. Tech. rep., ISO.

KENNEY, L. AND PLUMMER, D. 2008. Magic Quadrant for Integrated SOA Governance Sets. Tech. Rep. Gartner RAS Core Research Note G00153858, Gartner Group. June.

KLOPPMANN, M., KOENIG, D., LEYMANN, F., PFAU, G., RICKAYZEN, A., RIEGEN, C., SCHMIDT, P., AND TRICKOVIC, I. 2005. WS-BPEL Extension for People – BPEL4People. Tech. rep., IBM, SAP.

LEWIS, G., MORRIS, E., O'BRIEN, L., SMITH, D., AND WRAGE, L. 2005. SMART: The Service-Oriented Migration and Reuse Technique. Tech. Rep. Technical Note CMU/SEI-2005-TN-029, SEI – Software Engineering Institute, Carnegie Mellon University. September.

LEYMANN, F., ROLLER, D., AND SCHMIDT, M. 2002. Web Services and Business Process Management. *IBM Systems Journal 41*.

LHEUREUX, B. AND MALINVERNO, P. 2008. Magic Quadrant for B2B Gateway Providers. Tech. Rep. Gartner RAS Core Research Note G00157460, Gartner Group. June.

MACKENZIE, C., LASKEY, K., MCCABE, F., BROWN, P., METZ, R., AND HAMILTON, B. 2006. Reference Model for Service Oriented Architecture 1.0, Committee Specification 1. Tech. Rep. document identifier soa-rm-cs, OASIS Open. August.

MAHMOUD, Q. 2005. Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI). Tech. rep., Sun Microsystems. April.

MCCOY, D. AND NATIS, Y. 2003. Service-Oriented Architecture: Mainstream Straight Ahaed. Tech. Rep. Gartner Research ID Number LE-19-7652, Gartner Group. April.

MCNAMARA AND CHISHTI, M. 2006. Business Integration Using State-Based Asynchronous Services. In *2006 BPM and Workflow Handbook*, L. Fischer, Ed. Future Strategies.

NATIS, Y. 2003. Service-Oriented Architecture Scenario. Tech. Rep. Gartner Research ID Number AV-19-6751, Gartner Group. April.

OF GOVERNMENT COMMERCE, O. 2002. ICT Infrastructure Management. Tech. rep., Bernan.

ROBINSON, I. 2007. ACID Transaction Policy in SCA, SCA Version 1.00. Tech. rep., BEA, Cape Clear, IBM, Interface21, IONA, Oracle, Primeton, Progress, Red Hat, Rogue Wave, SAP, Siemens, Software AG, Sun, Sybase, TIBCO. December.

SCHULTE, R. 1996. Service Oriented Architectures, Part 2. Tech. Rep. Gartner Research ID Number SPA-401-069, Gartner Group.

SCHULTE, R. AND NATIS, Y. 1996. Service Oriented Architectures, Part 1. Tech. Rep. Gartner Research ID Number SPA-401-068, Gartner Group.

TEN-HOVE, R. AND WALKER, P. 2005. Java Business Integration 1.0 Final Release. Tech. Rep. Specification JSR 208, Sun Microsystems. August.

THATTE, S. 2003. Specification: Business Process Execution Language for Web Services Version 1.1. Tech. rep. May.

UN/CEFACT. 2006. UN/CEFACT°Øs Modeling Methodology (UMM): UMM Meta Model – Foundation Module Version 1.0, Technical Specification. Tech. rep., UN/CEFACT.

WALDO, J. 1999. The Jini Architecture for Network-Centric Computing. *Communications of the ACM 42*, 7.

WEAVER, R. 2005. The Business Value of the Service Component Architecture (SCA) and Service Data Objects (SDO). Tech. Rep. Business Value White Paper, version 0.9., International Business Machines. November.

ZIEBERMAYR, T. 2010. *A Framework for Enhanced Service Reuse in an industrial SOA-Context – Dissertation*. Institute for Application Oriented Knowledge Processing, Johannes-Kepler-University Linz.

ZIEBERMAYR, T., WEINREICH, R., AND DRAHEIM, D. 2007. A Versioning Model for Enterprise Services. In *Proceedings of WAMIS 2007 - 3rd International Workshop on Web and Mobile Information Services*. IEEE Press.

Dirk Draheim holds a Diploma in Computer Science from the Technische University Berlin since 1994 and a PhD in Computer Science from the Freie University Berlin since 2002. Since 2002, he worked as a lecturer at the Freie University Berlin, University of Auckland, University of Mannheim, and Johannes-Kepler-University Linz. From 2006 to 2008 he was area manager for database technology at the Software Competence Center Hagenberg. Since 2008, he is head of the IT center of the University of Innsbruck.