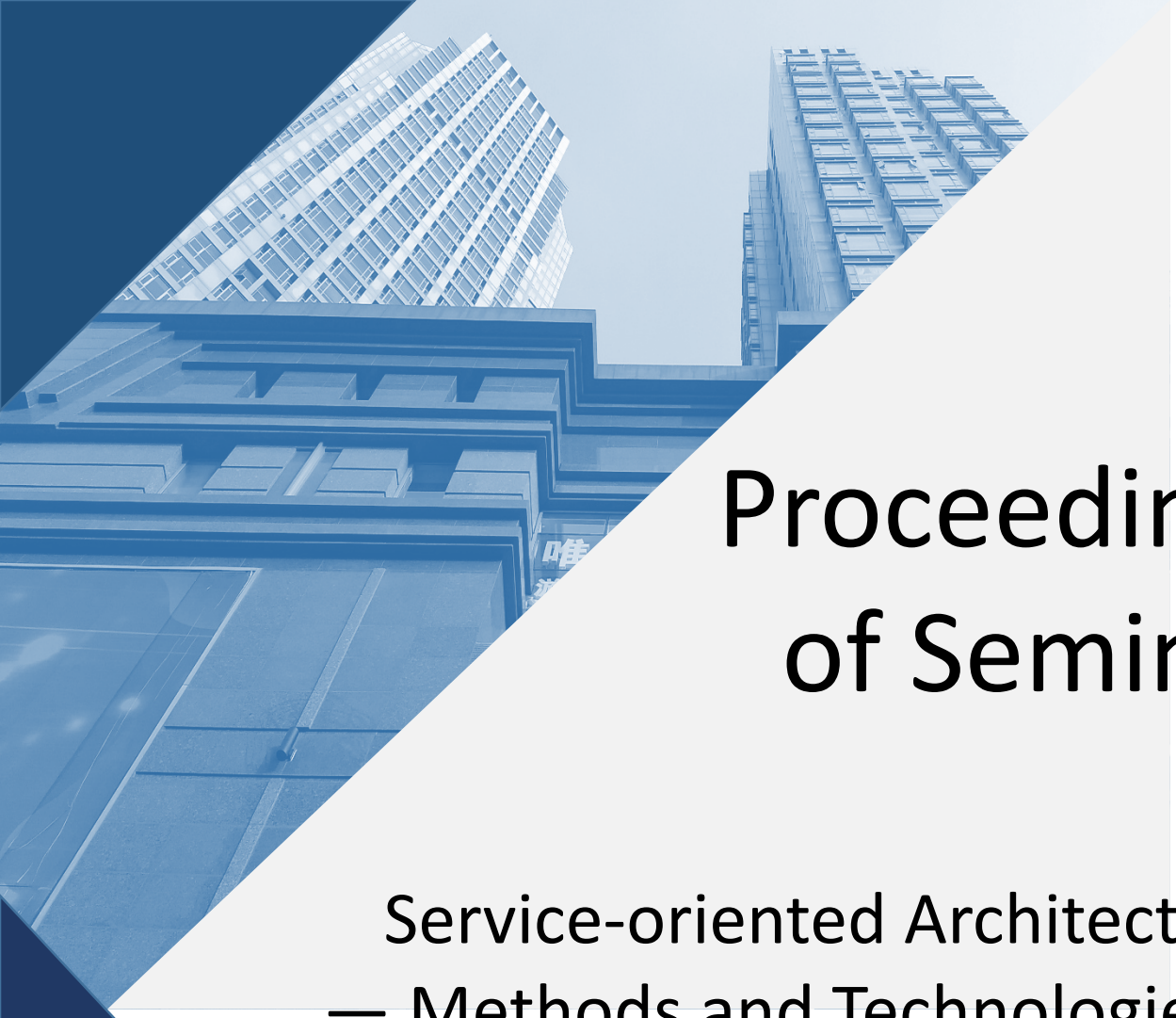

Proceedings of Seminar

Service-oriented Architectures — Methods and Technologies —

2009

Editors: Horst Lichter
Andreas Ganser

SWC Software Construction | RWTH AACHEN UNIVERSITY

Contents

Chapter 1

Service Oriented Architecture

Wasim Bari

Contents

Abstract: Service Oriented Architecture is immensely applied architecture in industry as well as research nowadays. Various technologies like Web Services and XML are used to implement SOA due to their extensible nature. Technology independent nature of SOA makes it an ideal choice for Enterprise to integrate their various systems. A huge list of echo standards and fields has already emerged with SOA realization like SOA Governance, Business Process Execution Language etc. Despite SOA popularity and usage, there are still many challenges to tackle. This paper presents the overall SOA state, benefits, applications, challenges and future in some extent.

1.1 Introduction

"SOAs are like snowflakes - no two are alike."
David Linthicum

Service Oriented Architecture (SOA) has been around now for a quite long time. Originally it was not a revolution rather an evolution of different paradigms and technologies influenced by business needs. Organizations are always looking for ways to integrate their existing systems with newly developed ones and for ideal situations in which all processes can speak with desired security. SOA provides architecture to achieve this goal. Service Oriented Architecture is a way of extracting, modeling, developing and deploying services which can be accessed and reused. Reusability and standardized access is the core of this architecture. The "heart" of this architecture, as its name depicts, is a "Service". A service can be anything ranging from logical items to physical entities. For example a service can be data, software or a printer. Common interface of all components ensures the integration of different systems with ease. All components in SOA are loosely coupled which was a major shift from Object Oriented Programming model where data and processor are bound together. The loose coupling promotes independent development and maintenance of components without dealing other parts of system. Sometimes Web Services, being the most common way of implementing SOA's, are referred to SOA as whole which is a misconception. SOA is more than Web Services; itś a way of thinking, planning, practicing and architectural patterns. Fig. 3.2 shows the some prominent features of SOA in different business domains.

1.2 Technologies and SOA

Close look at the evolution of SOA reveals major technologies which extraordinarily influenced SOA adoption. XML and Web Services are on the top of list. Platform independent nature of XML makes it a perfect candidate for communication in many SOA implementations, whereas, ease of Web Service creation,

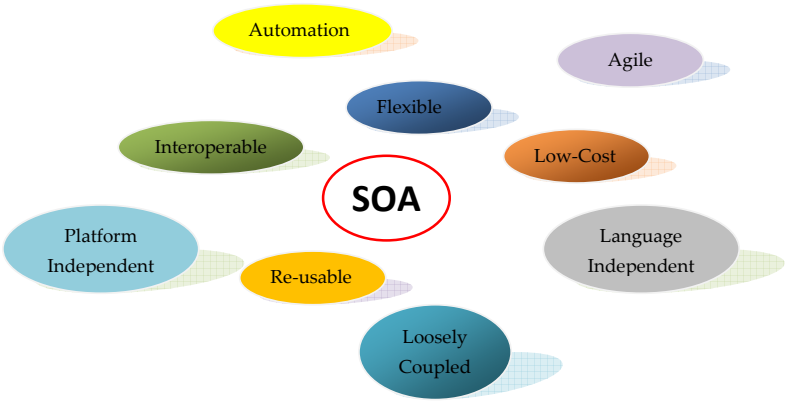

Figure 1.1: Service Oriented Architecture Characteristics

publishing, discovery and usage, promoted the use of 'Web Services' for creating SOA 'Services'. The influence came in two ways, these technologies also evolved with some changes to support SOA [XMLa].

1.2.1 XML

XML is a self defining way of presenting information. It is now the most popular data representation and protocol implementation language on Web [XMLb]. All the Web Services protocol like WSDL (Web Service Description Language), SOAP (Simple Object Access Protocol), UDDI (Universal Description, Discovery and Integration) etc. are based on XML [Web]. SOA heavily relies on XML. Efficient and secure SOA requires efficient and secure XML message exchange. XML is also used for ontology creation which makes the semantic interoperability possible [Ont]. SOA usually integrates applications which may have different data formations, XML is used as semantic interoperability for data exchange in these applications. New techniques are also evolving for XML performance booster which directly influences the SOA performance [XMLa].

1.2.2 Web Services

A Web Service is a piece of code that provides interoperable interaction among machines over a network. A service is published before use and must have a standard interface in order to be accessed. A generic Web Services interaction is shown in Fig: 1.2. A service provider publishes its service using some global registry(UDDI). A service consumer looking for a service, request to the global registry for desired service. Global registry replies, if such a service exists, with the WSDL file of the service. Now a client can invoke service using

this WSDL file. The interoperability is achieved by using different XML-based standards like WSDL for Web Service description, SOAP for message passing and UDDI for service discovery [Web]. According to a survey from Gartner, in 2008, Web Services and SOA will be implemented together in more than 75 % of new projects [Garb]. Gartner also published its top 10 technologies of 2008, many come from SOA [Gara]. Here a question may arise that Why Web Ser-

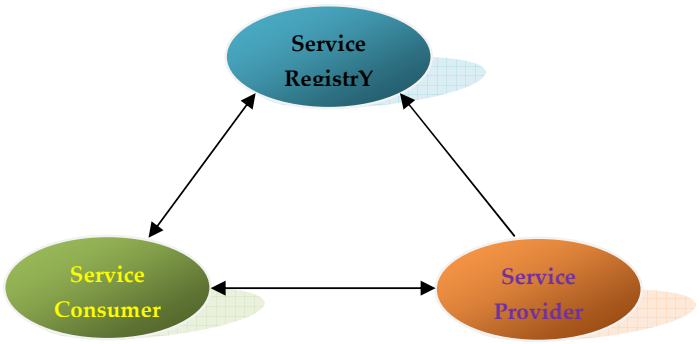

Figure 1.2: Web Service Triangle

vices are mainly chosen for SOA implementation? The answer lies in the Web Services' characteristics which go neck to neck with SOA requirements. Web Services are platform independent. No matter whatever platform a Web Service uses, it can still collaborate with other business applications. Web Service mainly uses industry open standards and protocols which are very popular. A large number of software and frameworks are available for creation and management of these protocols. These services are 'loosely coupled'. The main medium of communication for Web Services is Internet using SOAP, ResT or some other protocol over HTTP. Internet is a cheap medium as compared to other propriety solutions. At the Same time Web Services are Language independent. Organizations can select any technology of their interest, keeping in view their infrastructure, for Web Services implementation and these services can collaborate with services developed with other technologies too. This really helps in creating fast, problem specific and efficient services. Furthermore, Web Services are self describing which reduces development and integration costs. Web services framework also support automatic discovery mechanism which can not only reduce execution time but can also help to generate revenue by exposing services for consumers. Finally Web Services development has become easy with IDEs, frameworks and templates. Simple Web Services are just a click-next mechanism and most of the components like WSDL etc are created automatically from tools [WSD].

SOA, Web Services and XML have influenced each other and went through some extensions for performance optimization. A number of new XML technologies/languages are in market now. Extended XQuery, ebXML and XML Networking are few examples of these. Extended XQuery helps to create the orchestrator Web Services for SOA [Din07]. ebXMLis Electronic Business Ex-

tensible Markup Language (ebXML) by United Nations (UN) and OASIS for standard business processes and trading agreement among different organizations. XML Networking creates a new layer over transformation protocol which boosts up the message conversation for SOA applications [Sil].Web Services were originally stateless. In many business processes we need stateful objects like grid computing. This influenced extension in Web Services and in results WSRF and WS-* emerged which helps to create, monitor and destroy stateful services instances.

1.3 SOA Development Life Cycle

Softwares are developed with a structured engineering approach. A lots of model have been in use for this but no one completely fits with SOA, as its more than an ordinary software involving multiple domain factors. A recent paper outlined a strategy for SOA development. The development life cycle of SOA is shown in Fig: 1.3 [KLS08].

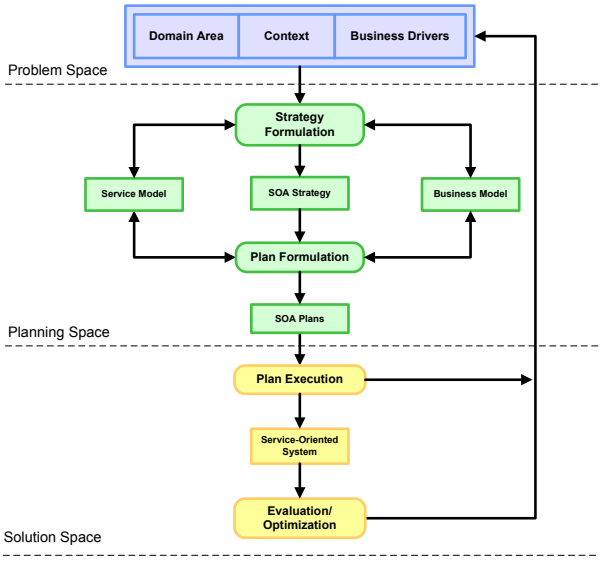

Figure 1.3: SOA Development Life Cycle [KLS08]

The Paper divides the development cycle in three spaces:

- Problem Space
- Planning Space
- Solution Space

Problem Space: Solution of the organization directly depends upon the domain area and business context. It enables or constrains the SOA strategy.From experience It's Observed, often SOA's strategy success is directly proportional to the business strategy of organization.

Planning Space: From the problem phase we come with SOA strategy and then plan our SOA's plans. These are dependent upon the input from the Problem Space, Business Model and Service Model. These plans work as input for the solution space as shown in Fig: 1.3.

Solution Space: SOA plans are executed to build a Service Oriented System. Some plans may fail because of certain changes or wrong assumptions hence we have to revise our SOA strategy as well as plans again. After we have Service Oriented System, we may again look at the problem space for evaluation and optimization purposes.

According to this SOA development life cycle, there may be many iterations. Any changes in business will be accommodated through the same procedure.

1.4 Benefits of SOA

Enterprises solely care for their business processes improvements and adopts Information Technology for this purpose. They want to reduce redundant infrastructures to cut down the expenses, provide access to systems within and outside organizations, automate the processes as much as possible, and reach out to every client so that they can increase their revenue. SOA promises for all such features. SOA adoption can bring advantages in business and IT; some of them are listed below:

1.4.1 Business Benefits

- Agility: Change is a norm in business not an exception. SOA's loosely coupling and openness to extensibility nature copes with the agile nature of business.
- Effective Integration with business partners
- Automation: BPM, BPEL and many other technologies built over SOA helps for business process automation. Complex Workflows with event driven can be easily managed.
- Efficiency of business processes with improved communication.

1.4.2 IT Benefits

- Reusability
- Low Cost
- Easy Governance
- Technology independence
- Cross Platform communication

1.5 SOA Standardization

Standardization plays an important role in information technology. Simple and easy standards attracts users. There is not a single organization currently working on SOA standards. Lately all efforts are driven by business needs with some short term plans and problems. The World Wide Web Consortium (W3C), Advance Open Standards for the Information Society (OASIS), The Open Group, The Institute of Electrical and Electronics Engineers Standards Association (IEEE-SA) and few more are currently developing standards for SOA [Kon07] [IEE]. Different vendors like IBM, Oracle, Tibco etc are also coming up with their own solutions and standards. There has been progress but with so many organizations working on the same stuff may harm the actual goal. It's also very unwise to work on same stuff by almost everyone. Too many standards also make it tough to choose one. For example there are currently more than 250 web services standards. We do need some sort of centralized way of doing stuff in future to be more effective [Sta].

1.6 Business Process Execution Language (BPEL)

Business processes are the core part of the Business. Normally a business process is not a single 'Service' rather collection of many services collaborating with each other in a specific order. Technically, business process is a collection of some operations carried out by either human or machine with available information and protocols to achieve a business objective. Some processes overlap, some are independent and can be executed in parallel and some are dependent. In SOA, services are the unit of work which make business processes an aggregation of Services/Web services. So we need some language for Business process modeling, work-flow management, execution of processes, control and monitor the processes, defined as services. Business Process Execution Language (BPEL) promises for these features. BPEL supports business process creation in two ways:

- Executable Processes
- Abstract Business Protocol

The First one follows the Orchestration style of Web Services and second supports Choreograph [Jur]. A BPEL process consists of steps, called 'activity'. Each activity can invoke other services in sequential or parallel order; activities may define pre-conditions to fire. These activities can be grouped for complex structures using different structures like flow, sequence, while etc. BPEL is also built on XML and supports Web Services technologies, WS-Reliable Messaging, WS-Addressing, WS-Coordination, and WS-Transaction. Faults are no exception in business processes so BPEL supports fault handling by a special mechanism of signaling and events [Jur].

1.7 SOA Governance

SOA has changed the whole nature of the systems. Now systems have no centralized control, may be contributed by inside or outside of organization. We need a way to manage and control these services, infrastructure and people for 'always-on' with desired quality and low cost. This is done by SOA Governance. "SOA governance provides a set of policies, rules, and enforcement mechanisms for developing, using, and evolving SOA assets and for analysis of their business value" [SEI]. Some people confused SOA governance as Governance of SOA which is not a correct perception. It's just an extension of IT and corporate Governance with respect to SOA. A recent survey from InfoWorld showed that lack of Governance is one of the key obstacle in effectively getting the benefits of SOA and adoption [Inf]. According to SEI, SOA governance is one of four pillars of service oriented systems development. Without SOA Governance, SOA infrastructure becomes unmanageable, the

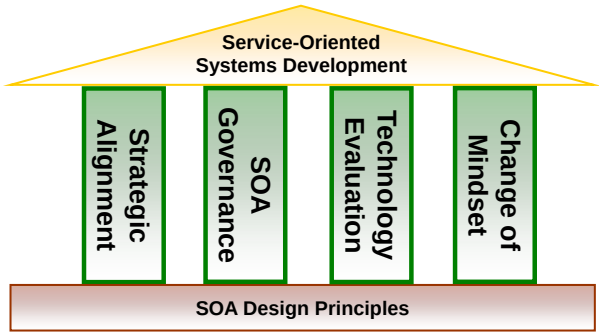

Figure 1.4: SOA Governance

reusability feature is not fully comprehended. Modifications to services can be

made without any prior notice, this way resulting business processes fail down. Generally SOA Governance must answer the following questions:

- Right Decisions for SOA assets
- Who is Decision maker?
- Realization of new high value Services
- Effectiveness of Services

1.8 SOA in Action

Werner Vogels, vice president, worldwide architecture and CTO at Amazon.com said during Gartner Enterprise Architecture Summit: ”Service orientation works, we never could have built [Amazon’s Linux blade server] platform without service orientation”. SOA is in action in many domains like health, information systems, grid computing (UNICORE), internet computing (Amazon S3), cultural heritage, WWW (Monster Job search) and a long list. We will present one SOA example in Grid Computing using UNICORE 6, an Open Grid Service Architecture (OGSA) implementation.

Grid computing is the application of multiple computing resources scattered at different places to solve some scientific or business problem which cannot be efficiently solved by normal computing systems. Grid may consist of different domains, different administration rules, spread across the organizations with multiple standards. Grid Middleware is used to make a Grid. A Service Oriented Architecture based, Open Grid Service Architecture (OGSA) was proposed ”that addresses this need for standardization by defining a set of core capabilities and behaviors that address key concerns in Grid systems” [I. 05]. A modern grid middleware must implement these services. OGSA is shown in fig. 1.5.

Execution Management Service: is responsible to discover available free resources and their locations. It selects the ”best” location, does necessary preparation for execution of actual job. After this it initializes the execution and finally manages it till it finishes. Execution can be an OGSA application or non-OGSA application.

Data Service: as name suggests, its responsibility is to deal with Data. It can access remote data, Stage-in and Stage-out data as required. It also replicates data for speedup and availability; Federation of data and Meta data management is done with Data Service.

Security Service: is responsible for authentication, authorization, ID mapping and credentials conversion for different security domains which helps in least

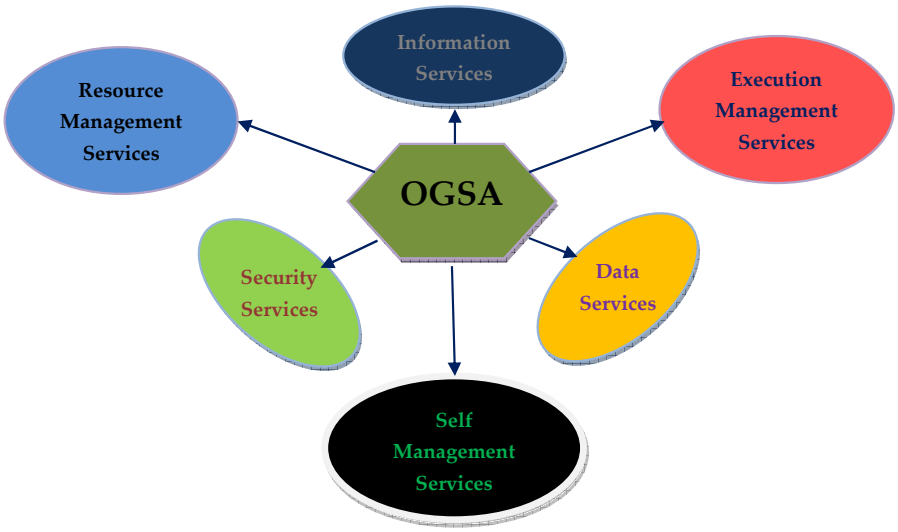

Figure 1.5: Open Grid Service Architecture

user interaction with system and makes it automatic. Security Service also takes care of audit, privacy and secure conversion.

Self Management Service: self configuration with changing environment and self healing of resources is carried out with Self-MS. It also performs monitoring, necessary analysis and is then able to provide fault tolerance. Stuff related to resources optimization can also be a part of Self Management Service.

Resource Management Service: carries three major tasks. Firstly it is responsible for the OGSA infrastructure, secondly resources management on Grid like reservation etc and thirdly resources management themselves like rebooting.

Information Service: naming capability both logical and physical one, resources discovery using registry, message passing mechanism and logging with monitoring are all done through information service[I. 05].

UNICORE 6 (**UN**iform **I**nterface to **CO**mputing **RE**sources) implemented OGSA using WSRFLite [UNI]. Fig. 1.6 shows the abstract three tier architecture of UNICORE 6. Clients authenticated by the Gateway to acces the services. These services interact with other services and XNJS to execute the jobs on desired target systems.

Target System Registry Service (TSR): Any client wishing to utilize UNICORE 6 must have information regarding available services and their description in a specific Grid. This information is published in TSR. It works as a single point of entry for clients. A TSR is shareable between sites.

Storage Management Service (SMS): serves as a logical storage unit. It is

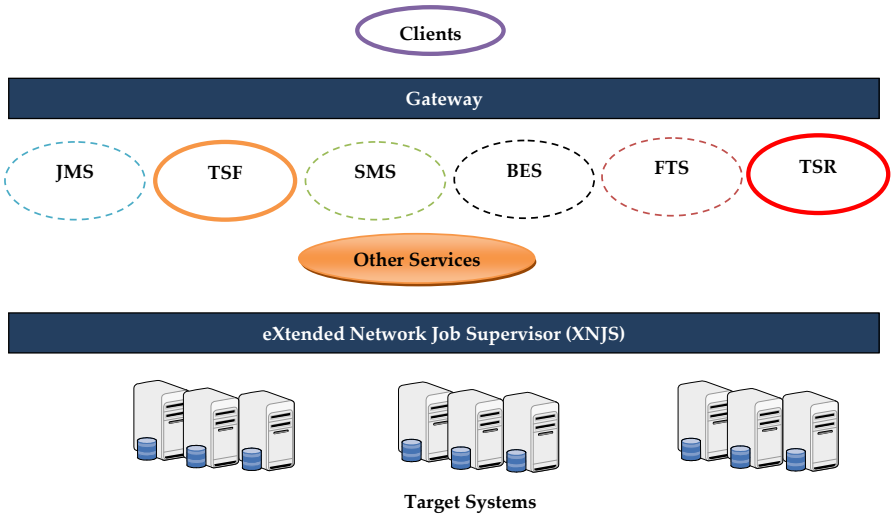

Figure 1.6: Abstract UNICORE 6 Architecture

used for listing storage contents and file transfer initiation. One UNICORE site may have zero to N SMS instances. A single storage can be manipulated with many SMS. SMS supports many File transfer protocols like RByteIO, SByteIo etc.

File Transfer Service (FTS): Actual File transfer is performed and controlled by FTS. A FTS instance is always created by SMS by invoking the appropriate method which checks supported protocols.

OGSA-Basic Execution Services: OGSA-BES is an emerging standard of job submission and management by Open Grid Forum (OGF). These services works almost same as UNICORE Atomic Services but advantage to adopt OGSA-BES is a standardized syntax of web services for job submission and management. With this adoption UNICORE can also speak with other Grid middlewares which implemented the same standard, like gLite, thus providing user a lot of freedom. With BES, some UAS services like TSF, TSS and JMS will be of no further use in future [OGS08].

1.9 Need for A Research Roadmap

SOA is immensely applied in industry but still lacks a clear research roadmap. SOA is a very complex field which overlaps a lot of other fields like software engineering, information systems, WWW, distributed computing, security, networking, middlewares, programming languages and many more. It's a dynamic area constantly reshaped by business demands. Adding more to this complexity, a number of organizations are working on SOA standardization and

solutions. There is not a global guideline, effort or vision available for SOA. With the adoption of SOA, complexity of architecture is increasing. Right now either wheel is reinvented or wastage of effort on similar issues is done. A clear research roadmap, as other engineering fields have, will not only boost the adoption of SOA but also organization will be able to get the true potential of architecture. Recently a paper was published to address this specific issue [KLS08]. The authors interviewed a number of researchers and related people and came up with a Services Research Roadmap as shown in fig. 1.3

1.10 SOA Challenges

The diverse nature of resources, multiple security models of organizations, services span over a number of organizations, different business natures and demands, various developers with or without any interaction and a vast list of technologies bring a number of challenges in SOA realization. Research is going on by multiple institutes to tackle these challenges.

Language independent standard interfaces for the 'Services' have achieved syntactic interoperability quite well. Publishing and discovering interfaces make it easy to look for specific services. But SOA still lacks semantic interoperability. Semantic Interoperability can be defined as the ability to understand the meaning of information exchanged among machines automatically and produce the required results. It mainly depends upon the interfaces describing the meanings of services and information shared with clients and services. It's difficult because the users of the service are almost unknown. They may be human, services or other systems. Developers need to provide information for services, user and SOA infrastructure to convey exact meaning at all levels.

Security is the most important factor in business applications. Across organization structure, the nature of SOA and multiple layers presence make it difficult to enable end-to-end security. We need some way of mapping the users to their credentials to make the flow uninterrupted. Looking at SOA layers, we need security from the lowest message level, to the topest Governance level. Service messages are mainly in SOAP or some other xml based protocol. OASIS provided a specification, WS-Security for message level security by defining how to integrate signatures and encryptions information to the SOAP messages. Security Assertion Markup Language (SAML) is used for virtual organization or across organization security.

Dynamic service composition is also another challenge. Composing services at run time may affect the quality of service; security is more problematic and may result in services incompatibility without proper semantics.

1.11 Future of SOA

SOA evolves as organization move ahead with SOA. Any technology or architecture's life is directly dependent upon its users and industry backing. All the big players of Information industry are either providing SOA solutions or using SOA. Oracle SOA Suit, the Smart SOA from IBM, Microsoft SOA and Business Process Management (BPM), SAP Business Suit, Tibco Solutions, Amazon Web Services and many more are contributing to the evolvement and realization of SOA in many industries. In fact, SOA has got unanimous support from all the giants. Many success stories from all around the world are also a major push towards SOA adoption. According to GARTNER "The Main Benefit of SOA is the opportunity for incremental development, deployment, maintenance and extension of business applications" which is also the true nature of business itself. The lifespan and reliability of service oriented-based system is much more than usual systems.

Many new SOA related terms are coined in market like Advanced SOA, SOA 2.0, WOA (Web Oriented Architecture). Companies are realizing the emergence of Web 2.0 and SOA as SOA 2.0. "SOA 2.0 is the term that we're using to talk about the combination of service-oriented architecture and event-driven architecture," said Steve Harris, vice president of Oracle Fusion middleware. New advancement in Web Services like REST services is also helping to gap the performance issues of SOA.

Bibliography

[Din07] Dino Fancellu, Edmund Gimzewski. Extended XQuery for SOA. Article: http://www.xml.com/pub/a/2007/09/12/extended-xquery-for-soa.html#References, September 2007.

[Gara] Gartner Identifies the Top 10 Strategic Technologies for 2008. Web Site: http://www.gartner.com/it/page.jsp?id=530109. Accessed on 10th Januray 2009.

[Garb] Letter From The Editor, Gartner. Web Site: http://www.gartner.com/pages/story.php.id.3586.s.8.jsp. Accessed on 12th December 2008.

[I. 05] I. Foster, Argonne and U.Chicago... Open Grid Services Architecture. Specification: http://www.gridforum.org/documents/GWD-I-E/GFD-I.030.pdf, January 2005.

[IEE] Service Oriented Architecture Standards. Web Site: http://www.soa-standards.org/. Accessed on 10th Januray 2009.

[Inf] InfoWorld. InfoWorld Research Report:SOA. Report: http://www.s2.com.br/s2arquivos/403/multimidia/197Multi.pdf. Accessed on 10th Januray 2009.

[Jur] Matjaz B. Juri. A Hands-on Introduction to BPEL. Article: http://www.oracle.com/technology/pub/articles/matjaz_bpel1.html. Accessed on 10th Januray 2009.

[KLS08] Kostas Kontogiannis, Grace A. Lewis, and Dennis B. Smith. A research agenda for service-oriented architecture. In *SDSOA '08: Proceedings of the 2nd international workshop on Systems development in SOA environments*, pages 1–6, New York, NY, USA, 2008. ACM.

[Kon07] Kostas Kontogiannis. Panel: A research agenda for service-oriented architecture. In *Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'07)*. IEEE, 2007.

[OGS08] Architectural Overview of OGSA-BES Adoption in UNICORE 6. Published: http://www.unicore.eu/community/development/OGSA-BES/UNICORE_OGSA_BES_Architecture_Edition_1_0.pdf, February 2008.

[Ont] OWL Web Ontology Language. Web Site: http://www.w3.org/TR/owl-features/. Accessed on 12th December 2008.

[SEI] Workshop on SOA Governance. Article: http://www.sei.cmu.edu/isis/pdfs/SOA_GovernanceWorkshop.pdf.

[Sil] Silvano Da Ros. Boosting the SOA with XML Networking. volume 9.

[Sta] Panel: A Research Agenda for Service-Oriented Architecture.

[UNI] UNICORE Project. Web Site: `http://www.unicore.eu`. Accessed on 10th Januray 2009.

[Web] Introduction to Web Services by W3C . Web Site: `http://www.w3schools.com/webservices/ws_platform.asp`. Accessed on 12th December 2008.

[WSD] Introduction to the WSDL Editor. Wiki: `http://wiki.eclipse.org/index.php/Introduction_to_the_WSDL_Editor`. Accessed on 10th Januray 2009.

[XMLa] Intel® XML Software Suite . Article: `http://software.intel.com/en-us/articles/intel-xml-software-suite/`. Accessed on 10th Januray 2009.

[XMLb] Introduction to XML by W3C . Web Site: `http://www.w3schools.com/xml/`. Accessed on 12th December 2008.

Chapter 2

Analyzing and designing methods for SOA

Tobias Ickler

Contents

Abstract: To exploit the idea of service-oriented architectures we have use new methods for analyzing and designing software. In this article I want to point out some requirements and challenges coming up by using service-oriented architecture and show two different approaches to solve some of these problems. The first approach is the service-oriented analysis and design method, which combines different analysis methods. The second approach is the generic view, which visualizes requirements and needs of different stakeholders to get an overview of all participants.

2.1 Introduction

Increasing complexity of business applications brings the need of new modeling methods. Service-oriented architectures demands of new modeling methods for complex enterprise software. But SOA is no universal buzz word that can solve all software engineering problems. Some problems will be solved, but some new challenges have arisen.

In 2.1.1 I want to describe what service-oriented architectures are and why the need has come up. It describes the important key words of those architectures and gives an overview of its components. Requirements and problems occurring in modeling such architecture are described in 2.1.2.

In the second part 2.2 of this article I focus on service-oriented analysis and design. After a short introduction the three components business process modeling in 2.2.1, enterprise architecture in 2.2.2 and object-oriented analysis and design in 2.2.3. To explain the idea of service-oriented analysis and design, in 2.2.4 I will point out why these methods on their own are not enough for a whole designing process, following the targets of the new methods, described in 2.2.5. Finally the interaction of the three components is described in 2.2.6 and a conclusion sums up the ideas of this analysis method in 2.2.7. Related work follows in 2.2.8.

The third part 2.3 focuses on the generic view. This concept is based on the concept of meta-models, which is described in 2.3.1. An example illustrates the meta-models idea is given in 2.3.2 with the reference model. Next step is a detailed description of the generic views concept in 2.3.3 and its areas 2.3.4. To give an example for the use of the generic view the model driven security architecture is presented in 2.3.5 and the benefits of a generic view in this approach is talked in 2.3.6. Once again I will sum up the ideas of this concept and explain in which part of the design process this method may help in 2.3.7 and the related work of this part is described in 2.3.8.

2.1.1 What is SOA about?

The complexity of software systems has risen to a high level. Companies and organizations use IT technologies for many different jobs. The most difficult part for software developers in this matter is not to build a single software solution, but to connect all these software applications to a whole system.

Today, connectivity is an elemental function. Distributed systems work simultaneously on their own and together with a lot of other systems. Data exchange, workflows and delegations of jobs are typical elements of connected systems.

Software developers have to find a solution to provide the exchange of information between applications. If a whole new system has to be designed, it is easy to create a public interface of each system, which provides other systems access to information and functionality. But if a system has to be extended or two or more independent systems have to be connected, no interfaces will be available and the designer has to modify the systems. How can this problem be solved universally?

Modern object-oriented software construction entailed a lot of paradigms, which are approaches for standard problems of software developing. The problem of modeling service-oriented architectures is just such a standard problem too, so analog to design patterns for instance, a software paradigm is needed. [GHJV94]

Service oriented architecture is such a paradigm to connect distributed software systems. Like object oriented design patterns it is no fixed solution that will solve every problem perfectly. It is just a paradigm that does not prescribe every detail.

The main idea of service oriented architecture is that different stakeholders have capabilities (called service-providers) and have needs for other capabilities (called service-consumers). Providing a capability is called a service, these services have to be registered in a central sand can there be claimed by another application or stakeholder. So service oriented architecture organizes and utilizes distributed capabilities. [MLM+06]

But service-oriented architectures are more than an IT design paradigm. It combines business logic with the design of application logic. All the important ideas can be followed by anybody without any knowledge about software design. The architecture does not dictate anything about the correlations between any stakeholders nor does it determine anything about the system itself. Using service in the design process will only affect the public interfaces of each system: the exchange of information is standardized and therefore these systems can easily be connected with each other.

The most important concepts of the SOA paradigm are visibility, interaction and effects. All these elements are representations from the business logic. They are no technical items.

Visibility on the one hand describes the fact, that service-providers can publish their capabilities and on the other hand, that service-consumers can find the service they need. A service must be described objectively to give consumers the possibility to decide which service is the one they need. Functional requirements, constrains and policies are basically items to identify suitable services.

Interaction is the process of using (consuming) those capabilities. Typically this interaction is presented by an exchange of messages contending the relevant information. How the messages are transferred from one system to another is irrelevant for the general paradigm and depends on the used system or network. [MLM+06]

Effects are the result of the interaction. An effect is what happens, if somebody consumes a capacity and of course it depends on the used service. For instance the effect may be the return of some information, a service-consumer has requested; another effect may be a physical action as a result of an electronic request - all types of requests are imaginable.

Once again it should be emphasized, that a service-oriented architecture does not provide anything that a developer could not do without SOA. The advantage of SOA is that a system or network can easily be extended and independent systems can communicate to each other without any difficulty. But such architecture has higher costs than other systems. It depends on the system and its size, if SOA is a valid option for the designers.

2.1.2 Modeling of SOA

There are a lot of important differences between designing traditional software architecture and a service-oriented architecture. The view of the system is another one; processes and functionality have to be grasped in another way. While a standard application has to meet the requirements of known users, here we have to identify services as public interfaces, which are probably used by unknown external systems.

The first problem in modeling SOA is therefore to use a very high level of abstraction. The first level of abstraction is to identify the actions that cause the needed real word effects. Next step is to group them and identify services. And the last but most important step is to merge these ideas with the business models and processes. It is irrelevant where to start; it can be done by bottom-up - beginning at the technical view - or by top-down development - beginning at the business view. [ZKG]

Here you can find the first problem: Merging business logic and application logic may be a difficult step. Responsibilities and costs are examples for terms that have totally different meanings in these two points of view. Often some actions are technically very difficult to realize, but a non-IT employee lacks the knowledge to see such problems. A SOA designer has to bring both parties together.

Next problem, that should be mentioned, is the view dilemma. Let us say we have found out which business logic is needed and we have identified our services. We do not know anything about the stakeholder who will use our service. For instance if a stakeholder requests information, we do not know which part of these information is relevant to the stakeholder. But of course it is important for designing the informations representation. What can a designer guess about the stakeholders? [EJSS08]

For this reason we have to identify possible stakeholders in a service-oriented architecture. And we still cannot look for technically aspects, because the types of stakeholders depend obviously on the business components. After identifying the existing stakeholders, we have to think about possible new stakeholders in the future. Finally if we have all possible stakeholders identified, we still have the problem of how to represent our information. What is the best data exchange type for all stakeholders?

Generally all known problems of software developing and all known problems of modeling business concepts apply in this matter too. Of course all solutions for those problems must be used by modeling a service-oriented architecture too. Combining these aspects with the other named problems creates a lot of extra costs. This is the price for modeling such a system and this is the reason why a service-oriented architecture may be no solution for every system.

2.2 Service-oriented analysis and design

In the following parts I want to describe two completly different approaches for designing a service-oriented architecture. The first part - service-oriented analyze and design - describes a model, that combines different existing modeling methods that focus on different aspects. The second part concentrates on the eye of the beholder. It takes special care of the point of view of different stakeholders, which will use the system. These approaches are no absolute solutions; they take care of a single part of the development processes focusing only on single aspects. [ZKG]

Service-oriented analysis and design (SOAD) was developed by IBM and has never been finished. It is a theoretical idea with practical backgrounds. But

so far it shows a way for some problems that occur in the context of service-oriented architecture. It is a combination of object-oriented analysis and design (OOAD), enterprise architectures (EA) and business process modeling (BPM). The main idea is to combine these methods to create a design method fitting ideally for service-oriented architectures. The reason for this new technique is that the three design methods have their right to exist on their own domains, but for creating a service-oriented architecture these methods are alone not enough.

2.2.1 Buseness Process Modeling

The domain of business process modeling in software engineering is the analyzing and illustration of an enterprises processes. It is used to improve and optimize the efficiency of enterprise processes. The modeling process should be independent of the used technology level. There exist a lot of different notations and styles to visualize processes. The UML or event-driven process chains are the common technologies. [Wei] Another method is the representing of executable flow models. [LRS02]

But there are still some problems using this modeling type. Common questions are not answered yet, for example how to evaluate non-functional requirement and quality-of-service characteristics. There is a lack of an universal standard for business process modeling and additionally it is not clear, which jobs belong to analysts and which jobs belong to the designer. At this point business process is reaching its limits and by using it in our design method, these problems have to be solved.

2.2.2 Enterprise Architecture

"'Enterprise Architecture is the organizing logic for business processes and IT infrastructure reflecting the integration and standardization requirements of the firms operating model.'" [Wika] It can be seen as a view between the business logic and the application logic and may be used as a connection between these elements.

The areas of enterprise architecture are business processes, capabilities and organizational models on the business side; interfaces and protocols on the application side; additionally meta data for information handling and specification of technologies.

2.2.3 Object-oriented analysis and design

Object-oriented analysis and design provides a large arsenal of approaches for general problems in software engineering. Design patterns, design paradigms and other standards are common today. There exist a lot of tools and facilities to transfer real problems into an object-oriented programming environment.

The main aspects of object-orientation are encapsulating, information hiding, polymorphism and inheritances. Encapsulating describes the logical separation and organization in packages. Information hiding means separating public interfaces and inner activities - a user of a class does not have to know how the object fulfills its requirements. Polymorphism and inheritances are technical aspects; existing classes can be extended or specialized and polymorphism describes a dynamic use of those inherited classes.

All these elements can be used in service-oriented architectures too. Actually these elements are part of the core elements, for example the packaging to services is just the use of information hiding and encapsulating. Here you can see the influence of object-oriented design on service-oriented architecture directly, but there are some other examples.

2.2.4 Why are these methods not enough?

Figure 1.1 shows the domains where the modeling techniques belong to, and in which part of the life cycle a method may be used. Now a service-oriented architecture must combine these three domains. Therefore we need a hybrid modeling technique that allows us to fulfill all requirements of each domain. This is what service-oriented analysis and design should be.

The external view of the services is a functional view of business. It is followed by a view of business processes. Next step is the identifying of the services still in the logic of business. These services must be transferred into software services and finally they can be assigned to special software components. On the other hand, if you start at the point of view of the technical side an object-oriented analysis can identify classes, but level of abstraction of classes is for the whole system much too low. This top-down or bottom-up process can only be handled by combination of these design methods. [ZKG]

2.2.5 What does SOAD have to provide?

We want to transfer some requirements of service-oriented architectures to our designing method:

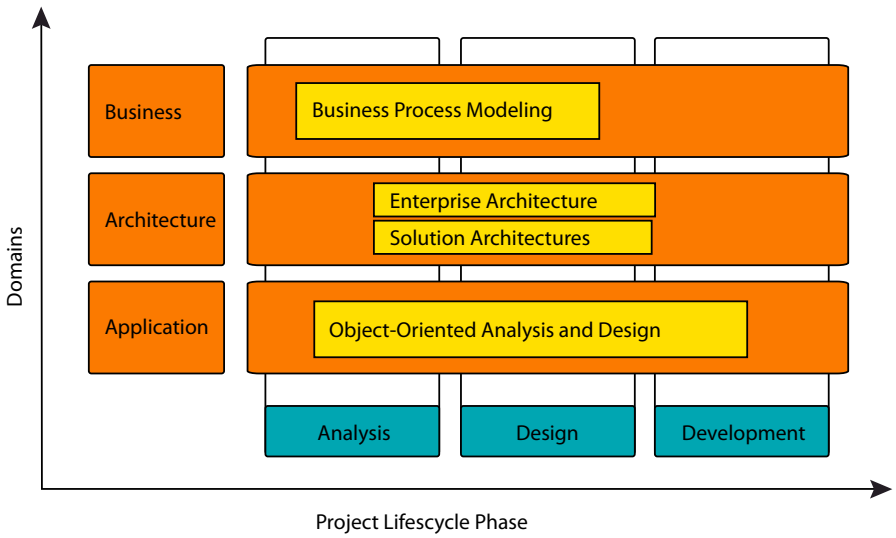

Figure 2.1: Lifecycle Phase

First we have a formal defined notation. Therefore we combine the elements of BMP (BPEL), EA, OOAD and some additional elements to close the connection between these technologies. The next important thing is a generalized way to identify services: from the view of business we must select the services on a technical level. This includes the question "'what are good services and what are not good services"', so there must be some criteria of quality. Finally of course there must be tools to use this technology flexibly and the tool has to be available for business and IT workers too.

2.2.6 How should SOAD work?

As can be seen in our requirements the most important thing is to identify the services. At first we are using a top-down business-level model to organize business processes. With these processes we try to find the services.

But as already mentioned before, in most cases using service-oriented architecture no complete system has to be designed. Instead of this existing systems have to be integrated in the design process and must be analyzed by the business view too. As those systems often only provide technical documentations and interfaces, business processes including existing systems must be abstracted too and here a bottom-up method is needed.

After or while modeling those business processes the object-oriented analysis and design techniques help us to identify concrete services. At this point we do not immerge into the domain of application view; we just have to group und

combine abstract concepts like modeling classes. The difference is the grade of granularity - the services are quite more abstract than classes or packages.

Next step is the merge of services. On the one hand we have an event-driven modeling and on the other hand we have services as instances. For instance one service has to use other services and these dependencies have to be modeled by a state model. This step will also help to identify the stakeholder as consumers of specific services.

Now we should be able to go one step closer to the technique view. Enterprise architecture helps us to find standards and conventions. For instance naming conventions or used protocols should be assessed here. There are a lot of question, which do not belong to the business view, but of course they are depending on the business view - so we have to use the previous results in this matter. The result of the enterprise architecture is the public interface for accessing services.

Finally we have to identify components, packages and at least classes. This process is a known process of software engineering and we can use object-oriented methods to solve the last step of the designing phase. At this part we can model independent applications, we know their responsibilities and we have given a public interface that must be implemented to make the service available.

2.2.7 Conclusion

Service-oriented analysis and design is a combination of three different modeling types. It has never become a common standard, but it shows the problems merging business processes, creating a global and reusable architecture and classic software engineering. The key element is to identify good services.

New systems can be created by top-down modeling; existing systems must be integrated by a bottom-up strategy. First step is analyzing the business logic, then we have to identify the services, next step is finding global assignments and finally design each system with it associated services itself.

With regards to requirements we have face up to service-oriented architecture we have get an overview of designing such architecture. We have come to know that we need a notation and analysis method to combine different aspects, because service-oriented architectures are interdisciplinary architectures.

2.2.8 Related Work

The concept of service-oriented analysis and design has been established as a usable tool by using UML notifications. [ZSWP] Because of the fact that the development is an IBM internal one, focus lies on the development of tools and techniques. This is the current state of development.

2.3 The generic view

The approach of the generic view focuses on creation and visualization of different stakeholders that may occur in a service-oriented architecture. It tries to find an answer to the view dilemma. The technique bases on an abstract meta-model, so first we want to present what meta-models are for. Then we want to describe the generic view. The result of the generic view model is a 3D visualization, but we focus on the analysis elements and will handle visualization in short. [EJSS08]

2.3.1 Meta-models

A meta-model in software engineering describes rules, constrains, models and agreements used for modeling a predefined problem. It may contents notations and methods to model a model. A typically meta-model is the UML for instance. ISO has published the standard for meta-models ISO/IEC 24744. [Wikb]

In our case we need a meta-model to describe the designing elements in a service-oriented architecture. The model has to show significant relationships between entities of the architecture. With such a model we are able to use standards and unifying concepts within our domains to create a useful vocabulary. Also such a model helps to abstract the existing entities and domains.

A concrete architecture is built by combining elements of our meta-model, patterns and the requirements of our system. So meta-models are powerful items, which can be used to design a concrete architecture or design another analysis tool.

2.3.2 The reference model in short

We want to present in short an approach for a meta-model for service-oriented architecture: the reference model. This model describes the relationships of important key items of service-oriented architectures. Full description can be

found in the reference, here we want to point out the idea of a meta-model, not the meta-model itself. [MLM+06]

The core concepts of the reference model are visibility, service, interaction, real world effects, contracts, execution-context and service description. These elements are connected with each other and connected with other elements - for instance visibility is connected with the concept of awareness as a precondition - and also other aspects can connect those key concepts - for instance a shared state is the connection of interaction and the real world effects.

In addition to that there exist several sub models in the reference model. These entities characterize specific situations. For instance the information model of a service describes the information that can be exchanged with the service. This sub model has relations to the service description entity and to the interaction entity: the service description must make this information available for consumers and the interaction is the process where a consumer will get this information.

The whole model has a lot of abstract entities, which could describe a service-oriented architecture - these entities must just be worked out. Creating architecture a designer may use this model to create the list of requirements. To do so he has to create concrete entities with specific attributes and the meaning and especially the relationships of the entities is determined by the meta-model.

2.3.3 Generic view concept

The generic view should be a solution for the view dilemma. It contains the problem that different stakeholders as capability consumers may have completely different views of the system. Probably two stakeholders may use the same service to get information, but they are doing completely different things with this information - so what is the correct representation of the information? Often some details are needed by one stakeholder, which are irrelevant to all others.

To find a solution it is not enough to identify all stakeholders, furthermore we have to identify the goals of the stakeholders. But the most important problem is to find a way, if we have all stakeholders and their goals identified - it may be pretty much - to cover the variety of different goals and views.

The idea of the generic view is to create - like the name says - a general view that allows it to compare goals and relevant views from the different stakeholders. Therefore it uses an on-demand and context-specific creation of views.

2.3.4 Areas

The core elements of the generic view are the information area, the profile area, the context creation area and the visual representation area.

The information area includes a meta-model to collect information about the environment and the requirements. The concrete scenario is described by entities and attributes and the core elements are process, activity, service, organizational unit and component. An example entity may be the existence of a specific service or business process.

To describe a stakeholders concerns and goals, in the profile area are created profiles, which are associated to the meta-models elements and are grouped and organized by dimensions, which explain the type of the information the profile is about. There are three structural dimensions - time, abstraction and domain - which describe the type of view, which will be created by the profile. On the other hand there are two non-structural dimensions - technical and functional - which describe the entities itself. One profile is a mapping of different information from these dimensions.

The context creation area takes the part of mapping this information. The result is the context and depends on the chosen profiles and of course the whole environment. It is a role-based task-oriented item incorporating all relevant information. But as raw data it has low use and next step must be the presentation in a readable way.

This is the part of the visual representation area. By creating a graphical instance of the context, the information is represented by 3D model in a handy way. Now the designer has an overview about all relevant stakeholders and their goals. The big advantage of this method is automating mechanism by creating a visual overview by choosing some profiles.

2.3.5 Model driven security views

To bring the view dilemma into an applied context, I will present the model driven security architecture. This is an approach to realize security-critical workflow tasks by providing a web-service-based framework. Interesting for us are the three views in the system and the special focus to the security requirements. [BHWN05]

The background for this approach is the lack of integrated security management in web services. There are different types of approaches for single security problems, but there exists no systematic design and realization method to use these approaches. Also the standards like BPEL or WSCI provides no opportunities to model individual security requirements given by the business

logic. [BEA03]

To describe the concept short as first there is an analysis phase where the security requirements are discovered, the knowledge of the business logic is supposed. Using UML notation processes with related security aspects are described formally. The next step contents the development of three views: global workflow model, local workflow model and the interface view.

The global view of the system describes an abstracted view of the workflows including all partners and their interaction of each other. What services exists, who consumes the service, who will provide it and which documents are exchanged are the main questions answered by this view. It is modeled as an UML activity diagram and the security requirements are embedded to the services as constraints. All in all it is a non-technical view of the system.

Each system itself is seen as a black box in the global view; the local workflow model takes care of this aspect. It describes internal actions done by calling a service or actions leading to external calls to other services. The local view is completely independent from the system and may be modeled anyway, but should also be a non-technical view.

The technical aspects are illustrated by the interface model. It describes the public interfaces of a component, which provides one or several services, but independent from the platform. On this level security requirements involve role-models and access rights for operations using a sub language of UML.

Finally these models are used to transform the architecture to a specific platform. The framework provides special elements, which handle the security policies. Related to our concern these processes are irrelevant and can be read in the referred paper.

2.3.6 Model driven security views and generic view

This architecture is a good example to show the problem of creating different views with a lot of different stakeholders. We have different requirements to each view, we have different requirements by each stakeholder and especially we have to include security aspects as a general key element. It is obviously that this job may have an extremely increasing complexity by an increase of participating stakeholders and requirements.

The method of the generic view may help to solve this problem and can be used as an intermediate step designing the different views. As an additional step this method may be included into the model driven security views by interpreting the given UML models of the first step and creating automatically meta-models entities and maybe even needed profiles. Using such automa-

tism a designer may easily create the interface model using given visualizations of different concerns.

2.3.7 Conclusion

The concept of a generic view focuses on the view dilemma. Its aim is not to identify any services like the service-oriented analysis and design do. It emanates from the point that services have been identified, but the details of the services interface are not clear yet.

An advantage of the generic view is the automating mechanism and the visualization of the view. It can easily used as a comfortable tool by a SOA designer at a specific point of developments life cycle, but it is important to say, that it only support other design methods and helps out for a small part of the architecture like we have seen in the example of model driven security architecture.

2.3.8 Related Work

The view dilemma can still not completely be solved by the generic view. Also this approach takes just care of a small part of the service-oriented development and there exist a lot of projects in this matter. The concept is used for example in the research of security driven models and another projects deal with the visualization part. [Kos02]

Bibliography

[BEA03] BEA AND IBM AND Microsoft AND SAP AG AND Siebel Systems. *Business Process Execution Language for Web Services Version*, May 2003. `http://www.ibm.com/developerworks/library/ws-bpel`.

[BHWN05] Ruth Breu, Michael Hafner, Barbara Weber, and Andrea Novak. Model driven security for inter-organizational workflows in e-government, 2005.

[EJSS08] Stefan Eicker, Reinhard Jung, Widura Schwittek, and Thorsten Spies. Soa generic views - in the eye of the beholder, 2008.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[Kos02] R. Koschke. *Software Visualization for Reverse Engineering*. Springer, 2002.

[LRS02] F. Leymann, D. Roller, and M.-T. Schmidt. Web services and business process management, 2002.

[MLM+06] C.M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, and R. Metz. Reference model for service oriented architecture 1.0, 2006.

[Wei] Peter Weill. Innovating with information systems: What do the most agile firms in the world do? `http://www.iese.edu/en/files/6_29338.pdf`.

[Wika] Wikipedia. Enterprise architecture. `http://en.wikipedia.org/w/index.php?title=Enterprise_architecture&oldid=267828014`.

[Wikb] Wikipedia. Metamodeling. `http://en.wikipedia.org/w/index.php?title=Metamodeling&oldid=263804018`.

[ZKG] Olaf Zimmermann, Pal Krogdahl, and Clive Gee. Elements of service-oriented analysis and design. `http://www.ibm.com/developerworks/webservices/library/ws-soad1/`.

[ZSWP] Olaf Zimmermann, Niklas Schlimm, Günter Waller, and Marc Pestel. Analysis and design techniques for service-oriented development and integration. `http://www.perspectivesonwebservices.de/download/INF05-ServiceModelingv11.pdf`.

Chapter 3

Aspect Orientation in Web Service Composition

Waqas Noor

Contents

Abstract: Web services have become the de facto standard for communication between machine to machine over the network. It allow Interoperability between heterogeneous and distributed systems. Recent research and development in the area of Web services provide the platform for composing, extending the web service. Business Process Modeling Language (BPML) and Business Process Execution Language (BPEL) are the output of the ongoing research. However, these languages (specifications) have several limitations regarding segregation of functionality and accepting dynamic changes at runtime. These web service composition languages compose the multiple concerns like logging, authentication, exception handling etc with the business logic. Such kind of compositions lack in modularity and hence required separation of concerns. Aspect Orientation in web services composition tries to separate the crosscutting concerns from business logic.

3.1 Introduction

In the past, Web was considered as static platform for sharing the resources over the Internet. With the invention of Service Oriented Architecture (SOA) and technology enhancements in Web services allow not only alow sharing of static resources but distributed services over the Internet. Web services specification conform the interoperability of services.

Web services are the way to realize Service Oriented Architecture (SOA) which enable the other distributed services to communicate over the internet in standardized and machine-to-machine interoperable way. These services can be published, consumed and discovered over the internet by using the open standards technologies such as WSDL (Web Services Description Language), SOAP (Simple Object Access Protocol) and UDDI (Universal Description Discovery and Integration).

Due to time-to-market pressure, sometime services need to be combined together for providing new functionality by reusing the other existing services. These kind of web service composition lead to many new specifications and languages such as BPML (Business Process Modeling Language), BPEL (Business Process Execution Language) [ACKM03]. These process-based languages get inspiration from the work flow based technologies. Workflows basically consist of tasks, grouped together and providing the functionality as unit work. These workflows are used to optimize and organize the business processes.

The process-based web service composition has limitations. In this report, I elaborate few shortcomings of these languages in subsequent paragraphs.

The first problem of web service composition is related to *modularization*. The specification of web service composition allows composition of different services and each service may specify a business process. The process may also consist of other services as well. Each service may consist of different

activities (tasks which composition designers write while making the composition of service) like *invocation of other services*, *exception handling*, *performance*, *logging*, etc. The resulting code of these activities in composition often does not fit into concept of modularization. To obtain the functionality mentioned above in web service composition, these activities cut across the process boundaries. In the absence of modularization in the specification of web service composition, the crosscutting activities code are tangled with other processes (specified in web service composition) [CM04a]. This cross cutting within process makes it difficult for the process designer of service to maintain and see the clear view of the web service composition. Change required in the composition of service needs to effect the several areas of composition. This changing process is time consuming, error prone and might be a threat for the business (In Business to Business (B2B) has usually long running process [AV06]).

The solution to above mentioned problems are discussed in Aspect Oriented Web Service composition with AO4BPEL [CM04b]. The authors believe that their new extension to process-based web service composition can tackle modularity and dynamic changes at runtime problems in composition of service. By applying the Aspect Orientation concepts, they argued that the modularity and flexibility of services is increased. Crosscutting concerns such as authorization, authentication, business rules, profiling and protocols are handled in different way as they handle in past by tangling the code. Code tangling example is shown below in figure 3.1.

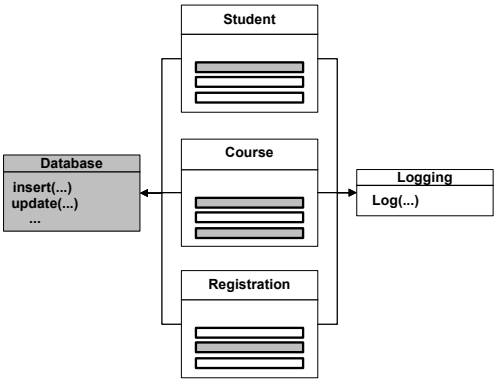

Figure 3.1: Code tangling example [cod]

Aspects are the crosscutting concerns which are written separate as standalone module to the core functionality. Combining together these aspects into core functionality is called *weaving*.

3.2 Service Oriented Architecture (SOA)

Over the decades, IT is enabling the businesses integration but increasing levels of complexities in architecture, requirements, integration issues, distributed nature of applications and time-to-market pressure leads problems such as integration, high cost, time consuming and even more increasing complexity. Previously, business integration solutions were more tightly coupled due to no clear standards in terms of programming, operating system, communication interfaces, etc. The change cost in such integrated, tightly coupled systems is expensive and required time and resources. In such business scenarios, these kinds of integration problems trend towards the need of new architecture.

Service Oriented Architecture (SOA) is architecture which promised to solve the above stated problems. According to SOA, systems are loosely coupled, consist of service provider or/and service consumer which interact with each other according to contract which is defined for them. Service Providers are functionality which encapsulated as services and service consumers use the services [Man05].

It is worth mentioning that SOA is not a technology rather an architecture. It models the applications as services. Business functionality is organized as a set of modular, reusable shared services. The services are loosely coupled and accessed/consumed in standardized way.

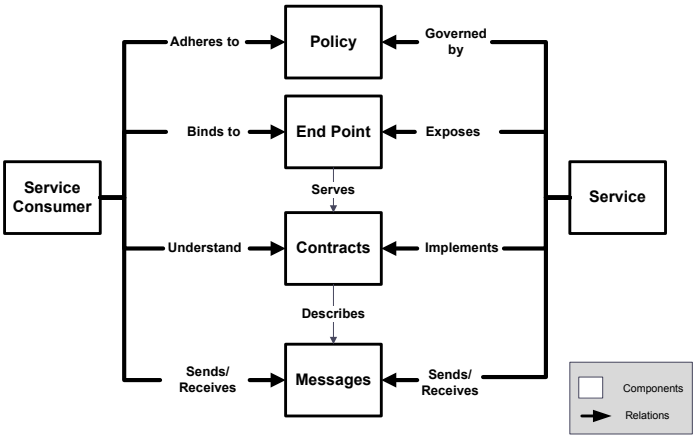

Figure 3.2: Components of Service Oriented Architecture [RGO06]

The core central part of SOA are Services, which are exposed as endpoints (End Point is URI, a specific address where the service can be located). The service consumers bind to that end points. The communication between services and service consumers are in the form of messages. The relationship between services and service consumers are defined as contracts. The policies in SOA define the security, authentication, auditing etc. By engaging the SOA into software, applications turn the services reusable, easy to integrate

and standard way to publish, consume and deploy. The services can share information more securely than ever with other business components. The configuration and change is really made easy in SOA. Efficiency is improved by reusability of services. The software cost is reduced by using standardized technologies [Man05].

3.3 Web Services

Web services are platform-independent distributed software components which enable the software to connect each other in loosely coupled manner. They use open standards, technologies like HTML, XML and SOAP. They also introduce and utilize new concepts, technologies like SOA, WSDL and UDDI. In Web services, application are composed of services which can be published, discovered and invoked. The components of web services are shown in figure 3.3

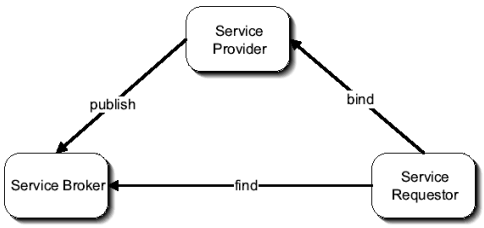

Figure 3.3: Roles and Interact in Services [Gis01]

Service provider encapsulates the implementation of functionality and only provides the interface to use the service. The service requester discovers the service which is published by service provider and binds itself to service. After binding to service provider, it can invoke the services. Service requester get the reference of service from the service broker. Service Broker is a repository of service providers.

Web services publish their services as WSDL which describe the interfaces, messages types, end points and some other properties of that service. All communication between web service provider and requester are as SOAP messages which are XML elements. SOAP messages adhere the message specification which is describe in WSDL [Man05].

3.4 Need for Web Services Composition

Currently, in the world of globalization, businesses need to adapt new changes quickly. Businesses need more flexible applications which require less or no change and accept changes at runtime. Growing and merging businesses require combining of 2 or more services without changing the implementation, communication interfaces etc. Along with such kind of requirements, business world is quite mature and developed applications based on open standards technologies like XML, SOAP and WSDL. Web Service composition is an open, standards technique for connecting existing web services together to create new services. The basic requirements for composition are to invoke other services synchronously and asynchronously as well. This should be reliable, scalable and should be adaptable by the IT environment.
Business Process Modeling Language (BPML) is used to model the business processes. It has capabilities to model generic business processes as well. Unfortunately, BPML did not take much part in businesses but it motivated industry giants to think of similar technology which should enable web service composition and execution of business processes.
Business Process Execution Language (BPEL) is subset of BPML but consider more stable, useful and de facto standard for a web service composition [BPM02]. BPML targets the composition at design level but BPEL is for execution of processes. BPEL process consists of activities, partner link and flow. The BPEL engine executes the BPEL process.
The Web Service Choreography Interface (WSCI) is an XML-based interface description language that describes the flow of messages exchanged by a Web Service participating in choreographed interactions with other services [ea02]. WSCI manage dynamically messages (messages which exchange) of different web services. It is more towards managing the messages rather than processes.

3.5 Web Services Composition

In this section, I will introduce the de facto standard for the web service composition named as Business Process Execution Language for Web Services (BPEL) and their short comings.

3.5.1 Introduction to BPEL4WS

BPEL (Business Process Execution Language) is XML based language which has deep roots from workflow based languages. A workflow model in workflow management systems depicts the work of person or group. It consists of basic set of activities (tasks) and the order of execution between these tasks. On

the other hand, BPEL, like other programming languages introduced the control structures. The control structure like if-else, loops, communication (synchronous, asynchronous) make the positive edge to BPEL over the workflow based systems. The business processes in BPEL consist of activities such as <*receive*>, <*invoke*>, <*reply*> and some other structured activities that control the flow of actions and data between these primitive activities. Along with these activities variables and *partners* are important elements in any BPEL processes. Variables hold the data either from the requester or from reply of some partner link. BPEL processes can execute on any BPEL compliant engine such as Oracle BPEL and ActiveBPEL [Vas07]. BPEL engine takes the definition of business process and executes it accordingly. The normal execution processes invoke the partner links, copy data from different services, update the database entries and response to other services, etc.

For example, listing 3.1 shows a BPEL process from [RK03]. The BPEL process consist of process named *echoString*. This simple process takes string as input and returns the string which is passed to this process as parameter. The variable name *request* is holding the string which is passed as input parameter. The sequence activity consist of other 2 different BPEL activities like <*receive*> and <*reply*>. The receive activity waits for the invocation of the operation *echo*. Once the operation is invoked by the client then BPEL process replies with input string in <*reply*> activity. The variable *request* is also used to hold the result string which contains the reply message. The complete listing is shown below in 3.1.

Listing 3.1: Simple BPEL Process

```
<process name = "echoString"/>
<variables>
        <variable name="request" messageType="StringMessageType"/>
</variables>
<partners>
        <partner name="caller" serviceLinkType="tns:echoSLT"/>
</partners>
<sequence name="EchoSequence">
        <receive        partner="caller" portType="tns:echoPT"
                        operation="echo" variable="request"
                        createInstance="yes"/>
        <reply          partner="caller" portType="tns:echoPT"
                        operation="echo" variable="request"
                        name="EchoReply"/>
</sequence>
</process>
```

3.6 BPEL4WS shortcomings

BPEL as like other programming languages has lack of modularity in modeling the business processes. Object Oriented Programming (OOP) languages consider being more flexible but they are also lacking in managing the crosscutting

concerns and lose the modularity. BPEL processes are also insufficient to accept the changes related to their composition at runtime. In the subsequent section, I discuss these issues in detail with examples.

3.6.1 Modularity issues in Modeling Crosscutting Concerns

As I discussed in above section that the current web service composition technologies like BPEL are lacking in modularity when the code for aggregating the different web services cut across the concern like logging, performance, authentication, business rules, etc. To achieve the business requirement for web service composition along with the concerns, the related pertaining code cut across the modularity of software. Let me elaborate this scenario with example of simple travel service shown in figure 3.4. The example shown in figure 3.4 is consists of BPEL process. This service exposes two operations getFlight and getHotel. These operations are specified in BPEL process. To keep the example simple, this travel service is only consist of two operations and some other features like logging, authentication, business rules. The horizontal bar shows the service and vertical bars shows the processes. The gray area depicts the cross cutting concerns.

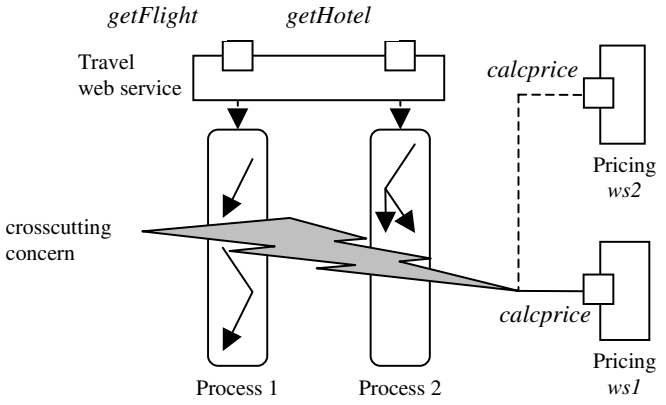

Figure 3.4: Example of crosscutting in process based composition [CM04b]

Due to business requirement, let suppose that BPEL service provider wants to incorporate the auditing functionality in their existing code (changes in BPEL process). The service providers capture the different state of values like response time, logging of request/response, failure handling, etc. To add the auditing functionality into existing service, the related code for these functionality scattered across many places in process. This is because of lack of modularity in BPEL. BPEL does not provide any specification to modularize such crosscutting concerns. Such kind of web service composition leads towards complex code, difficult to manage and very hard to design such kind of requirements as modules.

In real environment, such kind of deficiencies can be seen in many places in processes. For example, if we want to incorporate the price calculation functionality to our existing example, let say we introduced new service called *calculatePrice*. This service calculates the price, in this case it calculates the price for getFlight operation as well as getHotel operation. The example of this service is shown in figure 3.4. We expose our new calculation service as common service for other processes. Other processes like getFlight and getHotel can use this service. The pertaining code which enables the access of calculation service for this process is not modularizing in this process. The client (the term client is not similar to network client-server model. This is piece of code which access/call the service/object) of this service is crosscut the modular approach of process-based web service composition. This crosscutting scenario is shown in figure as grayed area.
Again, the major issue, crosscutting concerns, the pertaining code to these crosscutting concerns are placed in different places in BPEL process. Due to lack of modularity in Web service composition, specifically BPEL, the tangled code are scattered at different places in process definition. Industry practice shows that processes deal with many concerns at the same time. This kind of process definition leads implementation of single concern at many places. If these kinds of concerns are implemented as module then process designers can concentrate on writing logic of core concern [CM04b]. This kind of practice definitely makes the process simple for development, maintains and reusable as well.

3.7 Aspect Oriented Programming (AOP)

Aspect Orientation is a new but rapidly growing research area in software engineering. Till now, couple of powerful programming languages have been developed and solutions are provided for modularity. The example of such Aspect Oriented Programming (AOP) language is *AspectJ*. The concept of separating the crosscutting concern is not only limited to implementation. Aspect Oriented Modeling (AOM) deals with the crosscutting concerns at design level [SSK+06]. When we start decomposing the requirements into entities at design level, at that time we only deal with abstract form of concerns. There is the need for managing the crosscutting concerns at design level .When this design transforms into implementation; gap is created with the semantics of design and implementation. This problem leads the need for aspect oriented modeling [SHU03].

Aspect Oriented Programming (AOP) is a programming technique to address the problems of modularization of crosscutting concerns at implementation level. AOP is considered solution to limitations stated in above sections. Aspect can be applied to any language including Object Oriented Programming and Process Oriented Style. The use of aspects in web service composition

can solve the problems like modularization etc [CM04b]. The reference implementation of this concept shows that the resulting process is modularized.

3.7.1 Introduction to AOP

AOP provides the paradigm to modularize the concern, called aspect, in a complex systems. AspectJ is considered most powerful programming language in current AO programming. *Join Points*, *Joincuts* and *advice* are the most import terminologies are in AspectJ [KHH+01]. Join Points are the points in the code base which in turns part of execution of the program. The Join Points are like invocation of method, calling of constructor etc.

Join points are identified and added in the execution of program where modularity is required for the crosscutting concern. The pointcut is introduced for hooking the application context to AspectJ. The pointcuts can be selected, marked with the type of return, parameter, name of method etc. The AspectJ compiler translates the resulting join point into standard java byte code as method call. Advice is third most important terminology in AspectJ. This can be add/modify/override the functionality which is pointcut. Whenever execution of program meets with pointcut, the related advice code is executed. Currently AspectJ provide execution of advice in 3 ways. This can be *before*, *after* and *around*. As it is cleared in advice name, *before*, is executed before the method which meets the join point. Similarly *after* and *around* advice are executed after the method and around advice can take the control of original method execution respectively. Around advice can add more code which may have more join points and pointcuts.

As it is mentioned earlier than AspectJ compiler translates the aspects and produces the standard java byte code. AspectJ provides 2 types of possibilities to add the point cut. In *compile-time*, pointcuts are translated and placed in original code where join points meet the condition to match the name. In *Dynamic* approach, aspects can be (un)deployed at runtime so the behavior of application can be changed due to this dynamic aspect orientation [CM04b].

3.7.2 Overview of AO4BPEL

Aspect Orientation can fix the limitations in web service composition as described in early sections. The extension of BPEL4WS with AspectJ has ability to inject the aspect into web service composition at runtime. BPEL process consists of activities, so for Aspect, join points are well defined points in execution of BPEL process. Any BPEL activity possibly can be join points and properties of activity can be used as predicates. As BPEL process is XML based so XPATH is preferable language for querying and processing of XML documents. In extension of BPEL, like AspectJ, 3 types of advices are avail-

able which are *before*, *after* and *around*. The before advice in BPEL extension is an activity in BPEL process which should execute before the original activity as mentioned in pointcut executed. Similarly after advice execute after the activity. Sometime it is required to replaced the existing activity, the around advice do this job in extension of BPEL. BPEL has possibility to write the code segment to fulfill the task in any programming language. But this kind of addition of code segment makes the BPEL process less portable. For avoiding the portability issues, *infrastructural web service* are used which can access the orchestration engine at runtime [CM04b]. The java code execution server executes the java method like reflection in java. These methods are written as static methods in java class which are required by the extension of BPEL.

Figure 3.5 depicts the architecture of extended BPEL orchestration with Ascpect. The system is sub divided into 5 major components. The process definition and deployment tool, infrastructure web services, BPEL runtime, Aspect definition and deployment tool and Aspect Manager. The BPEL runtime is extended version which incorporates the aspects in it. Infrastructural web services are used to write and invoke the code segments other than the normal BPEL activity. Aspect definition and deployment tool is used manage the aspect's registration, deployments, activation and deactivation. Aspect Manager is responsible for controlling the aspects execution in the system

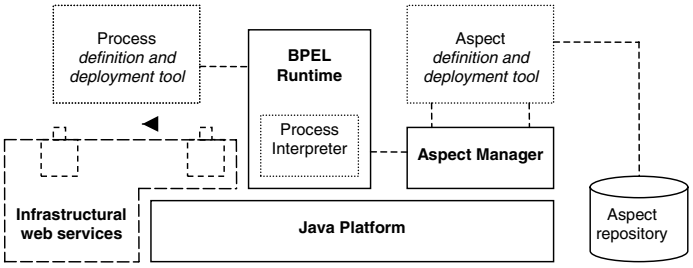

Figure 3.5: Architecture view of extended BPEL runtime - AO4BPEL [CM04b]

3.7.3 Modularity in AOBPEL

In this section, I will elaborate the solution to BPEL limitations regarding modularity.

As I mentioned in above section that modularity is one of the limitation in BPEL process and we can modularize the crosscutting concerns. Here, in this section I will elaborate the solution to modularization problem in the extension of BPEL. The aspect is shown in listing 3.2. For this process, we need to count the number of the times service is invoked. We will save the counter in a file such that we will access the file first, then read the old number, increase the number 1 time and then save it back to file system. BPEL does not have ca-

pability for file handling operation. The extended version of BPEL has one subcomponent ***infrastructural web services*** which can do this job for us. It has method which takes method name, class name and parameters. When the process execution reach at that point, it invokes the aspect after the activity, read the old count value, increase it and then save it back with modified value. Currently it only support primitive and string as parameter [CM04b].

Listing 3.2: Aspect aware BPEL process

```
<aspect name="Counting">
<partnerLinks>
        <partnerLink name="JavaExecWSLink" .../>
</partnerLinks>
        <variables>
                <variable name="invokeMethodRequest" .../>
        </variables>
        <pointcutandadvice type="after">
                <pointcut name="Lufthansa Invocations">
                        //process//invoke[@portType ="LufthansaPT" and
                        @operation ="searchFlight"]
                </pointcut>
                <advice><sequence><assign>
                        <copy>
                                <from>increaseCounter</from>
                                <to variable="invokeMethodRequest"
                                        part="methodName"/>
                        </copy></assign>
                <invoke partnerLink="JavaExecWSLink"
                                portType="JavaExecPT"
                                operation="invokeMethod"
                                inputVariable="invokeMethodRequest"/>
                </sequence></advice>
        </pointcutandadvice>
</aspect>
```

The Aspect Definition and deployment component is responsible for maintain the WSDL and configuration related to newly added aspect. Once the aspect becomes active, the Aspect Manager injects the aspect into extended BPEL runtime. Author argued and show with example that the AOBPEL extension of BPEL can solve the problems of modularity

3.8 Related

Dynamic changes in web service composition and flexibility to adapt the changes at runtime are new but growing research area. eFlow, a system that supports the specification, enactment, and management of composite e-services, modeled as processes that are enacted by a service process engine [CCI+00]. The Process is modeled in the form of graph which defines the execution flow of nodes. It models the business process as composite services. Like the extended BPEL runtime engine, it also supports the dynamic modification of

services. it provides 2 types of changes, Ad hoc (Changes to single running process) and Bulk changes (Changes many instances of same process). But unlike the AO4BPEL, it migrates the whole schema to destination schema of changes. This process follows the consistency rules for avoid any runtime-errors. AO4BPEL only weave the advice at given pointcut.

Dynamic AOP technique is used for adaptation of changes at runtime. But mostly it applies for hot fixes in services at runtime but not used for real purpose as modularization technique for the crosscutting concerns.

3.9 Conclusion

In this report, I discussed the limitations of modularity and accepting dynamic changes at runtime in web service composition languages. Web Service composition languages which inherit from the workflow based languages are incapable for modularization of crosscutting concerns and inadequate for handling the dynamic changes at runtime. Authors argued that the aspect orientation is solution for modularization of crosscutting concerns and adaptability of runtime changes [CM04b]. For the proof of concept, they extended the BPEL runtime engine. The new extension is called AO4BPEL. This BPEL runtime extension is Aspect aware. It means that aspect can be (un)pluged into BPEL process at runtime. Composition of service can also be altered at runtime with aspect. The resulting BPEL process is now modularize against crosscutting concerns and can adapt the changes at runtime.

3.10 Acknowledgments

I would like to thank Andreas Ganser for comments on earlier drafts of this report. I also thank the anonymous reviewers for their suggestions for improving this report.

Bibliography

[ACKM03] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer, November 2003.

[AV06] D.L. Amoroso and S. Vannoy. Translating the adoption of b2b e-business into measurable value for organizations. *System Sciences, 2006. HICSS '06. Proceedings of the 39th Annual Hawaii International Conference on*, 6:104b–104b, Jan. 2006.

[BPM02] BPMI. Bpml—bpel4ws a convergence path toward a standard bpm stack. 2002.

[CCI+00] Fabio Casati, Fabio Casati, Ski Ilnicki, Ski Ilnicki, Lijie Jin, Lijie Jin, Vasudev Krishnamoorthy, Vasudev Krishnamoorthy, Ming chien Shan, and Ming chien Shan. © copyright hewlett-packard company 2000 adaptive and dynamic service composition in eflow. Web Site: http://www.hpl.hp.com/techreports/2000/HPL-2000-39.pdf, 2000.

[CM04a] Anis Charfi and Mira Mezini. Aspect-oriented web service composition with AO4BPEL. In *European Conference on Web Services*, pages 168–182, 2004.

[CM04b] Anis Charfi and Mira Mezini. Aspect-oriented web service composition with AO4BPEL. In *European Conference on Web Services*, pages 168–182, 2004.

[cod] Aspect Oriented Programming. Web Site: http://phpaspect.org/documentation/aop.html. Accessed on 15th January 2008.

[ea02] A. Arkin et al. Web service choreography interface (wsci) 1.0. Web Site: http://www.w3.org/TR/wsci/, 2002. Accessed on 13th January 2008.

[Gis01] Dan Gisolfi. Web services architect: Part 1 an introduction to dynamic e-business. Web Site: http://www.ibm.com/developerworks/webservices/library/ws-arc1/, 2001. Accessed on 15th January 2008.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. pages 327–353. Springer-Verlag, 2001.

[Man05] Manoj Mansukhani. Service oriented architecture white paper. June 2005.

[RGO06] Arnon Rotem-Gal-Oz. What is soa anyway? 2006.

[RK03] S. Weerawarana R. Khalaf, N. Mukhi. Service-oriented composition in bpel4ws. 2003.

[SHU03] Dominik Stein, Stefan Hanenberg, and Rainer Unland. Position paper on aspect-oriented modeling: Issues on representing crosscutting features. 2003.

[SSK+06] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, and M. Wimmer. Towards a common reference architecture for aspect-oriented modeling. 2006.

[Vas07] Yuli Vasiliev. *SOA and WS-BPEL*. Packt Publishing, August 2007.

Chapter 4

Service Inventory Design Patterns

Roland Hildebrandt

Contents

Abstract: Service inventories are collections of services that all adhere to the same design standards the service inventory imposes on them.
Though greater in scope, less concrete, and resulting in greater consequences than classical design patterns, service inventory patterns provide useful strategic design advice to establish, structure, and govern a service inventory.

4.1 Introduction

With service-orientation a new paradigm in software construction originated in the recent years; it promises two ultimate goals: increased business agility and reduced IT expenses. Central to service-orientation is the realization of all software components throughout an enterprise as business-aligned, independent services. A key characteristic of these services is their interoperability. It allows flexible composition of existing services into new ones, to maximize reuse and agility within an enterprise.
When introducing service-orientation to your company, a special emphasis has to be placed on the establishment of the supporting architecture. Important parts of this architecture are built by so-called service inventories.

Service Inventory

“A service inventory is [defined as] an independently standardized and governed collection of complementary services within a boundary that represents an enterprise or a meaningful segment of an enterprise.” [Erl09]
In addition to this characterization of a service inventory to be independently standardized and governed, the services *themselves* may be individually developed and governed. But to keep them well aligned to each other and therefore interoperable, all adhere to the same design standards the service inventory imposes on them.
By introducing additional architectural elements like runtime platforms, a service inventory can—but not necessarily—represent the boundary for a concrete architecture implementation.

Design Patterns

Design patterns are schematic solutions of certain real-world problems that consistently reappear. Therefore they are worth being named and documented, to provide these best practices to others. Using design patterns in software construction has been very successful and popular up to now.
Service inventory design patterns are patterns especially centered on shaping a service inventory in desired ways.

As a pioneer in service-orientation and its documentation, Thomas Erl published a first mature pattern catalogue on service-oriented architecture design patterns, including several focusing on service inventory design.
Comparing these service inventory design patterns with the well-known classical design patterns in software construction may help to further comprehend these novel patterns.
Therefore section 4.2 starts with a summarizing discussion of classical design patterns in software construction. Section 4.3 accordingly introduces the service inventory design patterns documented by Thomas Erl. On this basis, section 4.4 finally compares the two pattern types.

4.2 Classical Design Patterns

Inspired by the architect Christopher Alexander and his development of architectural design patterns, the so-called “Gang of Four” applied his ideas to software construction. They came up with a seminal book, presenting twenty-three design patterns accumulated around the building of “elements of reusable object-oriented software.” [GHJV95]
Each of their patterns is documented using the same profile; consisting of intent, synonyms, motivation, applicability, structure, participants, collaborations, consequences, implementation, sample code, known uses, and related patterns. The pattern structures are expressed via UML class diagrams, and examples consist of concrete class diagrams and source code.
These design patterns especially provide practical and proven design knowledge. The deliberate naming of the patterns greatly assists communication across developers and documentation of software systems. Through their problem-oriented documentation, they build a helpful catalogue of repeatable design solutions; therefore this book established as a valuable reference for many software developers.
In order to further comprehend these classical design patterns, the following section will introduce a concrete example of such a pattern.

4.2.1 Example: Template Method

Software developers often need to specify an algorithm in several concrete occurrences that slightly differ in the details. But as those variants share a common structure, there is the risk of a redundant definition of this base structure. An example is the opening of a document. While the main steps (e.g., checking authentication and reading the file) are the same on an abstract level, the concrete realization of these steps differs for certain document types.
The possible redundancy when implementing these document types is prevented by the *Template Method* pattern; it suggests defining the basic algorithm structure in an abstract base class. Concrete realizations of this algorithm are

build via subclasses that define the details of the abstract steps.
Figure 4.1 displays the UML class diagram of a possible solution to the redundancy problem, using the *Template Method* design pattern: the abstract document base class defines the open document method. This method executes several base steps, including the abstract steps authentication-check and file reading. Concrete realizations of these steps are implemented in the spreadsheet and text document classes. These classes inherit the same base structure from the open document base function; therefore redundancy is avoided.

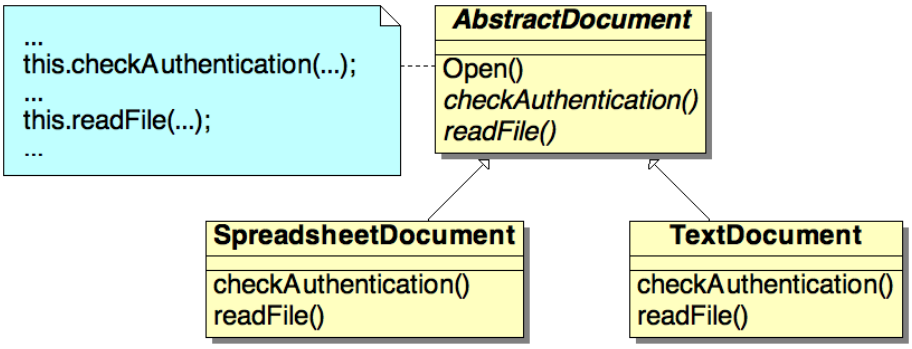

Figure 4.1: Template Method UML Class Diagram.

As seen with this example, these design patterns on the one hand are—besides requiring an object-oriented environment—independent from concrete technology. But on the other hand they provide fine-granular design advice on a level close to implementation.
The practical benefit of this pattern catalogue inspired many other authors in software. They either built on these seminal patterns by studying their application concerning specific technologies, or published new patterns assembled on other, more specific areas. With *Patterns of Enterprise Application Architecture*, Martin Fowler followed the second approach. [Fow03]

4.2.2 Patterns of Enterprise Application Architecture

In his pattern book Fowler restricts to what he calls “enterprise applications”—business centric data management systems, which are often confined by complex rules. He presents over fifty different patterns.
Although this new patterns focus on a more specialized area, the documentation and the level of application of these patterns remind of the Gang of Four patterns. Fowler describes each pattern by an introduction, details on how it works, when to apply it, and a source code example. Again UML class diagrams express the structure of the patterns.

4.2.3 Pattern Languages

Design Patterns are no mavericks, meaning they are not only documented together, but they are also intended for combined application. Therefore pattern catalogues usually contain sections on the relationships between patterns. These relationships can be specified differently in style and extend.
Subsection 4.4.4 compares the classical pattern language with the one for service inventory design patterns.

4.3 Service Inventory Design Patterns

In order to establish the context and intended end-results of the service inventory design patterns, this section will start with a quick revision of service-orientation principles and goals, while focusing on the relevance to service inventories. For the official definitions of these goals and principles, see *SOA Principles of Service Design* [Erl07].

Principles of Service-Orientation

- *Service Reusability*: A mature service inventory is essential to provide services and their capabilities as "reusable enterprise resources." [Erl07]
- *Service Discoverability*: Being able to find and understand existing services is a fundamental requirement that can be highly supported by a well-designed service inventory.
- *Service Composability*: Essential to service-orientation is the composition of single services. Standards applied throughout an inventory are necessary to provide combination possibilities on a high level.
- *Service Autonomy*: A service inventory has to find a balance between standardization throughout all its services and allowing each service to be individually developed and implemented.
- *Standardized Service Contract*: A service inventory is intended to build the boundary for certain service design standards; i.e., services within the inventory are forced to adhere to those standards, whereas external services are not.
- *Service Loose Coupling* and *Service Abstraction*: Service consumers are only depended on the service contracts, and should be independent from specific implementations. A service inventory can provide means to establish such a loose coupling, and additionally may enforce it to provide the desired service abstraction.

- *Service Statelessness*: As the minimization of individual resource usage is desired for every single service, it is beneficial to provide the means to do so in an inventory-wide standardized way.

Goals of Service-Orientation

- *Increased Intrinsic Interoperability*: All services within an inventory are supposed to be well aligned to each other. The goal is to make them not only interoperable in their initial configuration, but in a way that allows flexible rearrangements in new and maybe yet unforeseen combinations.

- *Increased Federation* and *Increased Vendor Diversification Options*: The services within an inventory are individually developed and maintained; yet the inventory is supposed to represent an uniting layer that increases the IT federation throughout an enterprise, but leaves possibilities to vary specific implementation platforms.

- *Increased Business and Technology Domain Alignment*: The establishment of inventory boundaries based on business domains helps to keep the technology and business domains aligned during further evolution of an enterprise.

- *Increased Return Of Interest*, *Increased Organizational Agility*, and *Reduced IT Burden*: Supporters of service-orientation expect to achieve these ultimate goals through the fulfillment of the four prior goals; therefore these three ultimate goals are also to be kept in mind when designing a service inventory.

In order to provide a complete and browse-able catalogue of service inventory design patterns, Erl divided them into foundational, logical layer, centralization, implementation, and governance patterns.
The profile for each pattern consists of a problem statement, an icon, a summary, a detailed problem and solution description, notes on the application and its impacts, the relations to other patterns, and a case study example. These examples are mainly short reports about the application of the patterns instead of concrete source code. This is due to the coarser-grained nature of the inventory patterns.
Because of this high level of abstraction from concrete technology, which is inherent to service-orientation, these patterns are only starting points for further architectural decisions. They are neither complete nor concrete in their application. Still they provide valuable support for fundamental decisions that set up the frame for further, more detailed design.
Erl describes the intention of his pattern catalogue accordingly as “focused solely on attaining the strategic goals associated with service-oriented computing.” [Erl09].

4.3.1 Example: Service Normalization

Whenever different teams develop services in the course of different projects, there is the risk that certain functionality covered by one service is at least partly covered by another. This redundancy in functional contexts might strongly jeopardize the results intended with service-orientation.This so-called “denormalization” of an inventory causes several problems, especially in reuse and governance.

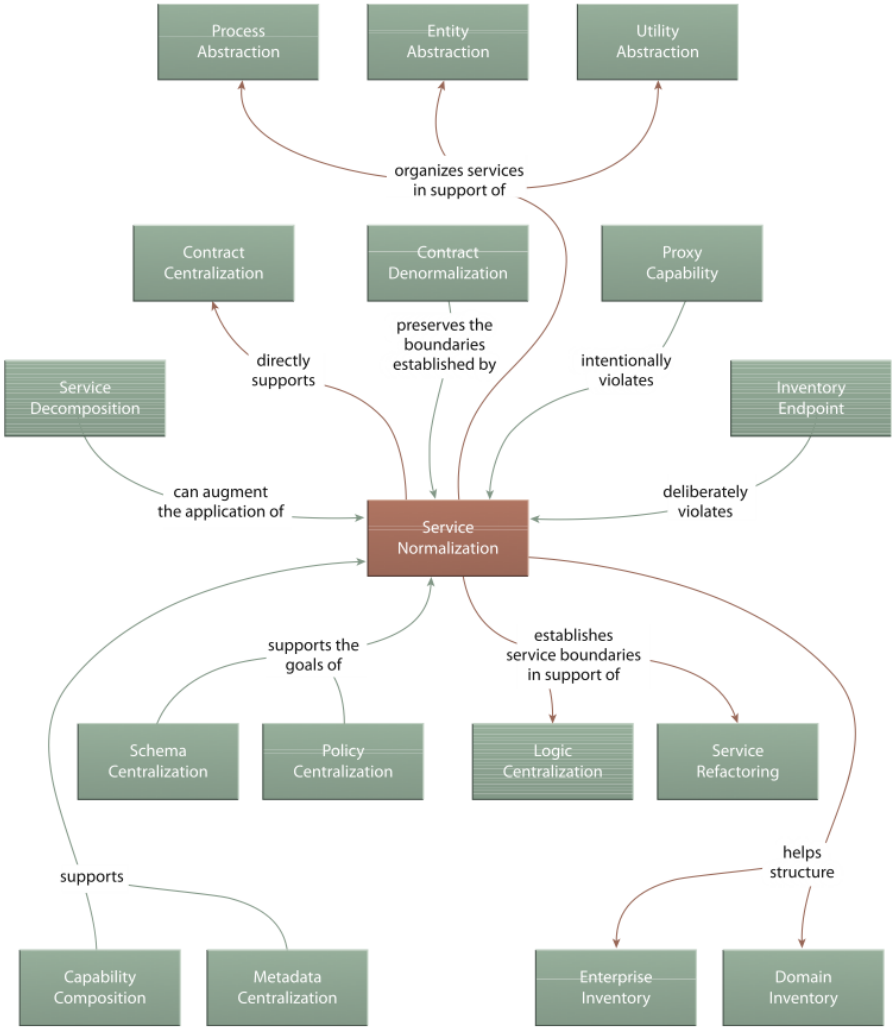

Figure 4.2: “Service Normalization . . . relies on the successful application of other patterns.” [Erl09]

The *Service Normalization* pattern addresses this problem of intersecting service capabilities’ contexts, by suggesting a collective modeling of all services, capabilities, and contexts in advance; i.e., before the service contracts are carved in stone, and with an emphasis on well-aligned service boundaries.

Although this is no guarantee for a completely normalized inventory, this pattern is supposed to increase at least the extend of normalization.
To support this goal, the pattern recommends basic application steps that are to be executed in several iterations. The first of these steps is to identify and disassemble so-called “business process definitions” related to the inventory. [Erl09] As a second step, the disassembled pieces are classified to be either integrated into existing service candidates, or to establish new ones. The final step is to validate the normalization; i.e., to make sure no two functional boundaries intersect.
This initial analysis effort as well as the later maintenance effort might be tremendous, especially for possibly wide-scoped inventories.
Other service inventory design patterns might also require the application of this pattern; e.g. *Contract Centralization* and layering approaches. These relationships to the other patterns are illustrated by figure 4.2. As demonstrated by the relation between *Service Normalization* and *Inventory Endpoint* depicted in this figure, a deliberate violation of certain patterns is sometimes recommended or even necessary to fulfill major goals.

4.3.2 Inventory Boundary Patterns

When establishing a service inventory, a first major step is to determine its desired scope, expressed by its boundary. The decision about the scope is supported by two fundamental inventory boundary patterns: *Enterprise* and *Domain Inventory*.
When software components are developed within an enterprise, they are probably developed by different teams in the context of different projects. This leads to the risk of incompatible architectures and implementations of these single components. Erl refers to theses as “silos”—aggregations of different technology and development styles that, beyond their initial technical environments, might not fit together well.
In order to avoid this possible danger to recomposition possibilities, the *Enterprise Inventory* pattern suggest to establish exactly one enterprise-wide inventory for all services. As all of an enterprise’s services lie within this inventory’s boundary, they all adhere to the same design standards. This leads to a better alignment of the services to support reusability and composition.
Whenever a company is too big, the IT structures are too complex, or resources are scarce, it is suggested to stick to several *Domain Inventories*, instead of one enterprise-wide. In this case the services within an enterprise are split up into several inventories that are highly aligned to distinct business domains. Although giving up enterprise-wide standardization, the alignment to the business is still preserved to a certain extend, and the otherwise overwhelming complexity is dealt with. Different domains could for instance be based on specific business areas or geographical locations. A trade-off of this approach is the possible loss of native interoperability of services across the inventories’ boundaries. As a consequence additional bridging effort might be

required, or redundant service capabilities might emerge.
Since the decision and its impact on the scope of the inventory are fundamental, these two inventory boundary patterns are essential when establishing a service inventory. Whereas too few inventories might be unmanageable, too many inventories might lead to incompatibilities between inventories, and restricted reuse possibilities. A plan for later union might also be unrealistic due to incalculable expenses: as these inventories are individually governed, they might develop in different and incompatible directions.

4.3.3 Layer Patterns

For many extensive systems it is reasonable to split them into layers, in order to manage their complexity. The same principle holds for structuring a single service inventory.
Basis for such a structuring approach is the *Service Layers* design pattern; it suggest the division of the services within a single inventory into distinct layers. A basic approach is to split the services into those representing single-purpose, and those representing multi-purpose functionality.
A more advanced approach is the separation according to abstract service types. This classification into so-called “service models” establishes a layer per type. Such service models that reoccur throughout different service-orientation projects in the industry are utilities, entities and processes.
The utility service model represents services that are not part of the business logic, but are considered as generic supporting tools to carry out business functionality (e.g., database access or error handling).
The business logic itself is commonly separated into entity and process services. Entities refer to the different objects a business uses; e.g., invoice documents or an accounting division. These entities—as well as the utilities—are termed “agnostic,” meaning they have no knowledge about the context they are used in. This makes them highly reusable in different instances of the third service model, the processes. These processes are accordingly named “non agnostic,” because they mainly represent a single-purpose functionality within a concrete context.
Based on these common service models are the interrelated patterns *Utility*, *Entity*, and *Process Abstraction*. These patterns further support the decision to use *Service Layers*, and are hence suggested for additional application. They provide advice and instructions on how to carry out certain divisions into layers. It is recommended to use them together, resulting in a *Three-Layer Inventory*.
Another advantage of this approach is the option to commit the administration of different layers to experts of certain fields; e.g., technology experts could be responsible for the utility services.
Figure 4.3 outlines the separation of a service inventory into these three service models, and depicts a service composition across all three layers: the *Revenue* capability of the *Annual Reports* process service is composed of the *Get Totals* and *Get History* capabilities of the *Accounts Payable (AP)* and

Commissions entity services. This composition further uses the *Report Exception* capability of the *Notifications* utility service.

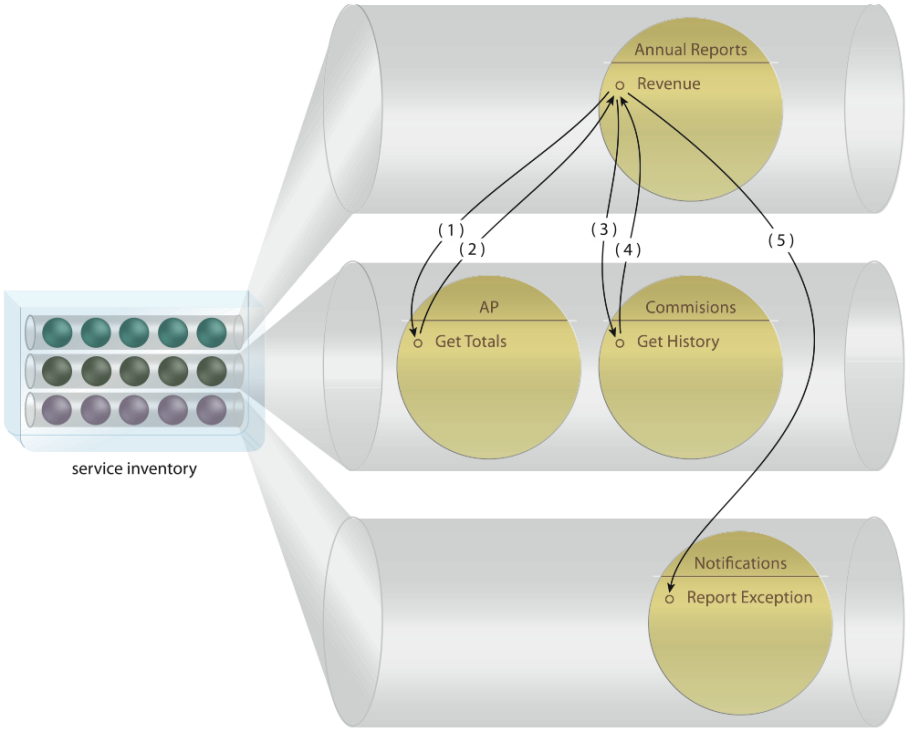

Figure 4.3: Service composition across a Three-Layer Inventory. Layers from top to bottom represent processes, entities, and utilities. [Erl09]

As explained in subsection 4.3.2, it is sometimes favorable to have several domain inventories within a single enterprise. But due to the agnostic and non-business nature of the utility services, there is a high likelihood that certain utility services are needed within each of the domain inventories. To avoid redundant implementations and increase reuse, the *Cross-Domain Utility Layer* pattern suggest the bridging of the individual inventory boundaries, to establish a shared utility layer for some or all of them. This approach may also reduce the number of overall services, and increase *Service Normalization*. Disadvantages might be hindered governance of too large-scaled cross-domain layers, or hindered administration across technically varying environments.
Closely related to the layering approach is the establishment of an official *Inventory Endpoint*; it provides access to certain service capabilities to external consumers via intermediate services, but the remaining service inventory is protected from undesired access. These official access methods can add supplementary extensions to the internal capabilities of an inventory, to fulfill certain needs in concerns of security, customization and compatibility (e.g., by additional protocols, or encapsulated compositions and transformations). To support their required flexibility, these special endpoint services are often excluded from at least some of the otherwise inventory-wide design standards.

As these layering approaches are of vital importance to an inventory's structure—and once-performed layering might be hard to change—the patterns discussed in this section need to be applied carefully and early in the analysis and design phases.

4.3.4 Centralization Patterns

A basis of keeping a service inventory synchronous and governable is the application of centralization. There are several design patterns that support the centralization of different components within an inventory; i.e., these components are made accessible via one distinct way. These patterns share the commonness of resulting in inventory-wide impacts, and therefore have to be applied carefully and maybe only to a limited extend.
When generally centralizing logic, the *Logic Centralization* design pattern is a first fundamental step; it proposes the enforcement of service reuse throughout an inventory to avoid redundant logic. Fundamental for the success of this approach is the discoverability, understandability, and practicability of the services intended for reuse. Once labeled as official means to access certain logic, these services are the exclusive access points for the encapsulated logic. In contrast to *Service Normalization*, this pattern therefore focuses on service utilization, instead of modeling issues.
Process, *Schema*, *Policy*, and *Rules Centralization* are more specialized design patterns, running for centralization in a physical, instead of just a logical location. This is supposed to benefit implementation, administration, and runtime performance. But application of these patterns might also imply the establishment of certain middleware that significantly influences the service inventory architecture.
Metadata Centralization is an additional centralization pattern, advocating the establishment of a central service registry to increase service discoverability. Fundamental to this registry is the required registration not only of already existing services, but also of those currently in development. Such a service registry might span multiple service inventories, to ensure interoperability and allow reuse across inventory boundaries. Therefore this pattern further supports *Logic Centralization* and *Service Normalization*, by avoiding redundant logic and overlapping functional contexts.

4.3.5 Canonical Patterns

Along with centralization comes the need to establish different design standards consistently throughout a service inventory. Canonical design patterns themselves are no design standards, but they advocate the establishment of enterprise-specific design standards within a service inventory.
A first step to ensure native interoperability of services is to use one main

Canonical Protocol for communication purposes of all services. This way the risk of incompatible protocols of services that may be created in the contexts of different projects—but now are combined—is limited. Such a restriction to a single protocol might impose certain limitations; therefore *Dual Protocols* provide further composition flexibility by establishing a secondary protocol for special communication needs.
In order to prevent conversion of different data representations between services, it is also recommended to use a *Canonical Schema* for common data models within an inventory.
Compatibility issues between services might also emerge, when various versioning approaches are followed within a single inventory. “Canonical Version” therefore suggests imposing design standards on contract version methods and version documentation. To further facilitate service discoverability and understandability, “Canonical Expression” refers to standardizing use of words and expressions across service contracts.
Besides standardizing these elements related to service contracts, it is also recommended to standardize common resource types within an inventory, to support their consistent use by various services. Such “Canonical Resources” could be databases, or special utility platforms and extensions.

4.3.6 State Patterns

A need reoccurring throughout many service inventories is the management of service state. There are several design patterns related to supporting state management in an inventory-wide manner.
To support the service statelessness design principle, the *Stateful Services* design pattern advocates the use of deliberately stateful utility services that manage everything concerning other services’ states. These utility services are specialized in managing state data effectively. This way, business related services are released from these presumably performance-affecting tasks.
A slightly different approach, either supporting *Stateful Services* or making them unnecessary, is the installation of a common *State Repository* for the data management purposes of all services.
The *Service Grid* design pattern further commends the application of grid computing to support *Stateful Services* or a *State Repository*, which otherwise might suffer limited scalability. This means—during runtime—not only a single, but several synchronized instances of each service are available. This way, high request peaks are addressed and the overall system stability is improved.

4.4 Service Inventory vs. Classical Design Patterns

With the classical design patterns published in 1995, and the service-oriented design patterns by Thomas Erl being officially published in 2009, there have

been fourteen years of pattern history in software construction so far. During this timeline, there has been evolution around the knowledge about design patterns, as well as the arising of new paradigms. On that account, the service inventory design patterns are different from the classical ones. The reminder of this section will discuss the most important of these differences.

4.4.1 Specialization

Whereas the initial patterns where build around the broad object-oriented paradigm and therefore span a wide area, further pattern publications are more and more specialized. For example Martin Fowler restricted to enterprise application as described in subsection 4.2.2. Thomas Erl focuses in another way by studying service-orientation. Within this paradigm the patterns are divided to address services inventories, services, and service composition.
This specialization is also represented is the increasing number of patterns presented. Whereas the Gang of Four initially presented twenty-three patterns, Martin Fowler already came up with fifty-one. Thomas Erl provides a whole of seventy-seven patterns for service-oriented architecture. Twenty-four of them are specially dealing with the design of service inventories.
The constriction to a field of study might imply a closer relation to technology, as happened with books about design patterns realization using concrete technology, like .Net [Bis07]. But the opposite is the case for service inventory design patterns, because service-orientation is further abstracted from concrete technology. Whereas the classical patterns are abstracted from concrete programming languages, the service inventory patterns are completely decoupled from any underlying technology. This abstraction from technology also impacts the concreteness and granularity of these patterns. As discussed in section 4.3, these patterns mainly deal with structuring, centralization, and standardization on an abstract level, focusing more on strategic decision than on technical implementations.
Martin Fowler's notion of patterns being "half baked," meaning the patterns have to be individually completed by the person(s) applying them, therefore applies to the inventory patterns on an even stronger level. This greater scope of individual completion especially influences the application and impacts of the patterns.

4.4.2 Application

For classical design patterns, in general only a small group of software designers and programmers is affected by the concrete implementation of a certain pattern. When shaping a service inventory by the means of design patterns, it is more likely to affect several IT divisions, the whole enterprise, or even partner companies.

Different to the classical design patterns, which are only directed to a technical audience, most service inventory patterns can easily be understood by and applied together with business experts. This further supports the business-driven characteristic of service-orientation.
Due to the high proximity of pattern and code of the classical patterns, these patterns are normally applied during the fine-granular design and implementation phases. Service inventory patterns like *Service Normalization* are applied starting in the initial analysis phase. When addressing special governance issues, as *Canonical Versioning* does, the patterns are carried out during the whole inventory lifetime. Therefore the overall scope of service inventory design patterns is significantly greater.
To follow the suggestions of some service inventory design patterns may imply significant decisions for a large part of a company's IT, and potentially high cost linked with it. The consequences of the application of such a pattern are also more durable and harder to change than with classical design patterns.

4.4.3 Documentation

The coarser-grained nature of the service inventory patterns is also reflected in their documentation. As the patterns provided by the Gang of Four and Martin Fowler are mainly about object and class relations, UML diagrams easily visualize them. Erl establishes his own symbols to express the different strategic situations that the service inventory patterns are applied in.
The same holds for the examples that accompany the patterns. Differently than the other authors, Erl illustrates the patterns not by source code examples, but through short reports; summarizing different strategic decisions that three example companies made when applying the service inventory design patterns.
With the naming of the patterns, there is also the introduction of new terminology. As discussed in section 4.2, this terminology can provide tremendous help in documentation and communication. Service inventory design patterns obtain their names from domain-specific terminology. Hence these names are more context-specific than those for general patterns; thereby they are especially helpful for domain experts.
An additional part of Erl's documentation on the service inventory patterns is an extended pattern language.

4.4.4 Pattern Languages

Owing to the far-reaching impacts of the service inventory patterns, Erl developed a pattern language in much more detail and extend than the Gang of Four did with the classical patterns. They restricted their development of a pattern language mainly to a single interrelationship diagram of all patterns.

Additionally the relationships sections on each pattern name the related patterns and describe each of the relations with one or two sentences.
The patterns Erl documented are interconnected in many more ways. As described in section 4.3, many patterns are complementary, require each other, or form alternatives. To further illustrate these interconnections, Erl additionally provides relationship diagrams for each pattern (e.g., depicted for *Service Normalization* in figure 4.2).
In addition, Erl also documented so-called "Compound Patterns" that describe combinations of certain patterns; e.g., the combination of *Utility*, *Entity*, and *Process Abstraction* is referred to as *Three-Layer Inventory*.
Studying this extended pattern language can add further understanding of the patterns' application and its consequences; therefore this pattern language adds great value to the overall pattern catalogue.

4.5 Conclusion

Service inventory patterns are greater in scope and less concrete than classical design patterns. Their application has greater impacts, possibly affecting whole enterprises and their business partners. But still they provide useful strategic design advice to establish, structure, and govern a service inventory. They are therefore a welcome addition to already established design patterns. Nevertheless the pattern language accompanying these patterns raises several questions that are worth further examination: Is a new pattern language really necessary? Is its extend sufficient or is the language maybe too extensive? In the end the crucial factor will be whether people are willing to learn and use these patterns or not.

Bibliography

[Bis07] Judith Bishop. *C# 3.0 Design Patterns*. O'Reilly Media, 2007.

[Erl07] Thomas Erl. *SOA Principles of Service Design*. Prentice Hall PTR, 2007.

[Erl09] Thomas Erl. *SOA Design Patterns*. Pearson Education, Inc., 2009.

[Fow03] Martin Fowler. *Patterns of Enterprise Application Architecture*. Pearson Education, Inc., 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Chapter 5

Service Design Patterns

Markus Arndt

Contents

Abstract: This paper deals with the application of Service Design Patterns in SOA environments targeting a cost efficient, flexible and reusable IT infrastructure.

5.1 Introduction

5.1.1 Introduction to SOA

Service Oriented Architecture is an IT architecture paradigm that structures business processes into independent, reusable, distributed software units (so called services) with loose coupling. These are composed into more abstract logic to deploy units that serve as business logic on a high level of abstraction. Though sharing ideas with Object Orientation, SOA resides on a much higher level, not considering portions of software programs, but whole enterprise resources such as databases or logical enterprise divisions.

5.1.2 Basic Definitions

Service

A service represents self-contained solution logic in the form of independent software.

Service Capability

A capability in SOA context is a function provided by a service. A hypothetical service 'Purchase Order' could have the capabilitys 'SubmitOrder', 'CheckOrderStatus' or 'CancelOrder'.

Service Contract

A service contract defines the public interface of a service. It expresses the mandatory conventions consumers have to stick to and further describes functional and non-functional requirements.

Consumer

Consumers are programs or services that make use of other services.

Inventory

Service inventories are groups of services that belong to the same segment of an enterprise or share commonalities and form some kind of logical unit.

Composition

Services that are being assorted and plugged together so they solve a certain business task are called compositions. Compositions consist of at least two services and a so called 'service initiator', a third service that controls the composition logic.

5.2 Design Pattern Basics

5.2.1 Definition

Design Patterns are field-tested, well documented solutions to well understood problems in software engineering and software architecture. Their application in software design contributes to the overall quality of the resulting system.

Ineffictive patterns are usually labeled as Anti-Patterns.

5.2.2 Design Pattern Origins

Design Patterns go back to the architect Christopher Alexander who published a collection of patterns in the 1970s, not relating to software, but building architecture.

In 1987 Kent Beck and Ward Cunningham created design patterns concerning graphical user interfaces in Smalltalk.

But the book that had the greatest impact on (object oriented) software design patterns is 'Design Patterns: Elements of Reusable Object-Oriented Software' [ERRJ95]. The authors, also known by the name 'Gang of Four', introduced 23 patterns and leveraged design patterns to their breakthrough.

The patterns we deal with in this paper target the service orientation paradigm and were introduced by Thomas Erl [Erl09].

5.2.3 Design Patterns in SOA

Design Patterns in SOA context are basically just design patterns that support the process of SOA environment creation and maintainance. A bright application of these patterns yields an improved overall quality of the software and contributes to the service oriented design principles we will now briefly cover.

SOA Design Principles

The main objectives or design principles of Service Oriented Architecture can be summarized as follows:

Related services (meaning being in the same service inventory) should follow the same contract design standards (Standardized Service Contract). The corresponding contracts are supposed to be built so that consumers and services have the most loose coupling possible (Service Loose Coupling). They should imply only the most necessary information (Service Abstraction).

Forthermore a main objective is that services can be reused and accessed by multiple consumers (Service Reusability), avoiding redundancy and fostering high-quality architectuire that is easy to maintain.

The presence of various consumers leads to a need for a high degree of control over a service's underlying resources (Service Autonomy) to produce predictable results and stable runtime behaviour at even difficult conditions (simultaneous access by a great number of consumers for example). Also it leads to the demand for cautious usage of those resources, so services are requested to minimize their usage by retaining as little state information as possible and the usage of state deferral (Service Statelessnes).

In order to actually use those services it is essential to ship these with certain meta information that descripe their capabilities and functionality (Service Discoverability). Services should be build so that they can be composed with ease and cooperate in an efficient way (Service Composability).

5.3 SOA Design Patterns

5.3.1 Foundational Service Patterns

These patterns are the most basic ones and are mandatory for transforming business processes into a service oriented driven environment.

In this chapter we find 'Service Identification Patterns', as well as 'Service Definition Patterns'.

Service Identification Patterns are meant to be applied to conventional solution logic, such as step-by-step descriptions of business processes, legacy software and other examples, that are all not yet service oriented. These patterns prepare the present solutions so they can be integrated into a service oriented environment.

Then again Service Definition Patterns actually define services, often out of what the Service Identification Patterns yield, but not necessarily.

Other related groups are 'Capability Composition Patterns' (covering issues arising when composing services) and 'Service Inventory Patterns' (management of service inventories), that we do not deal with here, but that are covered in dedicated papers.

Functional Decomposition

Motivation: Business problems are often solved by building one single piece of software. But a monolithic solution to complex business problems often brings a great governance burden - regarding monitoring and controling the software development - and handicaps reusability, as parts of the solution may be relevant to other business problems in the same enterprise.

Pattern: The approach of functional decomposition is to reduce the complexity of the original problem solution (for example given in textual step-by-step process description) by breaking it down into smaller concerns that are solvable independently (figure 5.1).
Dedicated solutions are then created to resolve these smaller concerns instead of building a single, monolithic solution for the overall problem. These solutions can later be composed to solve the original problem.

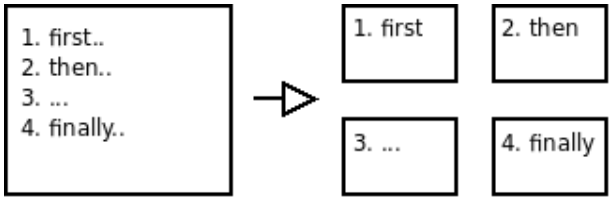

Figure 5.1: Functional Decomposition

As a Service Identification Pattern, this is one of the first patterns to be applied in the SOA process. It's results - small, self-contained portions of solution logic with well defined bounds - are the basis for further patterns (Service Definition Patterns).

It is important to understand that these dedicated solutions are not yet services, they are just parts extracted from the original problem solution that can be solved independently.

Correlations: This pattern is correlated to 'Service Encapsulation', as it isolates the fragments meant to be transformed into services from the rest of the logic. Also it is related to 'Agnostic Context', 'Non-Agnostic Context' being successors to 'Service Encapsulation'.

Agnostic Context

Motivation: Single-purpose logic reduces the potential of multi-purpose logic when coupled in implementations. Multi-purpose logic is reduced to only serve one special concern. For example on one hand an online store's administrative back-end may implement functionality to manage the store's stock database directly. This couples single-purpose logic (administrators can manage stock through back-end interface) with multi-purpose logic (access to the stock database). The multi-purpose functionality cannot be used by other programs, as it is integrated into single purpose software.

Pattern: First identify multi-purpose fragments in solution logic. Then isolate and outsource them into so called agnostic services. The solution logic becomes a potential resource to various consumers, that all share this specific implementation. In the recent example this pattern would imply to decouple the database functionality from the administrative back-end and place it into an own service, sharing the functionality, so a b2b application for retailers might want to use it to query the range of products.

This pattern makes the logic available to other parts of the enterprise, targets reusability and avoids redundancy. It is of the Service Definition Pattern type.

Correlations: The pattern being closest to 'Agnostic Context' is 'Agnostic Capability'. It is applied to services that have been declared agnostic. 'Agnostic Capability' redefines capabilities towards more general definitions, moving away from features that may have creeped in when the service has been abstracted from real business concerns.
Also there are specialized versions of the pattern 'Agnostic Context', namely 'Entity Abstraction' and 'Utility Abstraction'. They are dealt with in the paper 'Service Inventory Pattern'.

Other patterns in this category

Service Encapsulation, Non-Agnostic Context, Agnostic Capability.

5.3.2 Service Implementation Patterns

The patterns in this chapter actually affect the services implementation. They help dealing with ongoing architecture evolvement ('Service Facade', as a mediator between contract and service logic, decouples services from consumers with negative coupling possibilities, e.g. frequent contract variances) and increasing service requirements, such as runtime performance ('Partial State Deferral'), data and service breakdown safety ('Service Data Replication', 'Redundant Implementation').

Partial State Deferral

Motivation: It is essential for services to work in an efficient and predictable (referring to runtime here) manner, to guarantee a smooth behaviour of compositions they take part in. Sometimes services in compositions are waiting for other services to complete some kind of task and may be forced to hold large amounts of state data for that time. This, especially when multiple consumers come into play, can cause a decrease in system performance or bottleneck for that service and therefore the whole composition.

For example a service A may be some sort of meta online shopping application that offers procucts from several mail-order stores $B_1, .., B_n$. When finishing the order, program A queries all B_i to respond wheather all products are on stock (end-user may later choose where to buy). This is done holding the shopping list, the customer online profile details, etc. in memory, while waiting for the B_i services to reply.

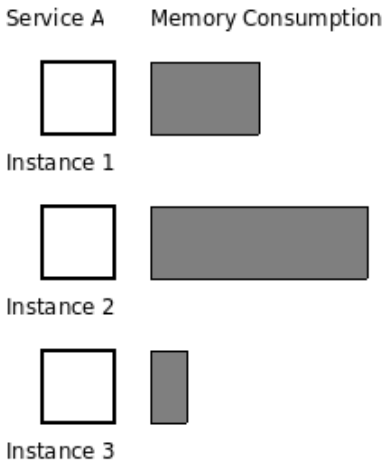

Figure 5.2: Resource hosting service A has critical memory load.

Pattern: It is possible to outsource the main part of such data into an external state deferral extension. This extension is usually a database encapsulated by

a service and holds the data for the time the service is on hold and returns it when needed. In the example the service could transfer the whole product list to an external service, only keeping the internal database key for the tuple the data has been saved in.

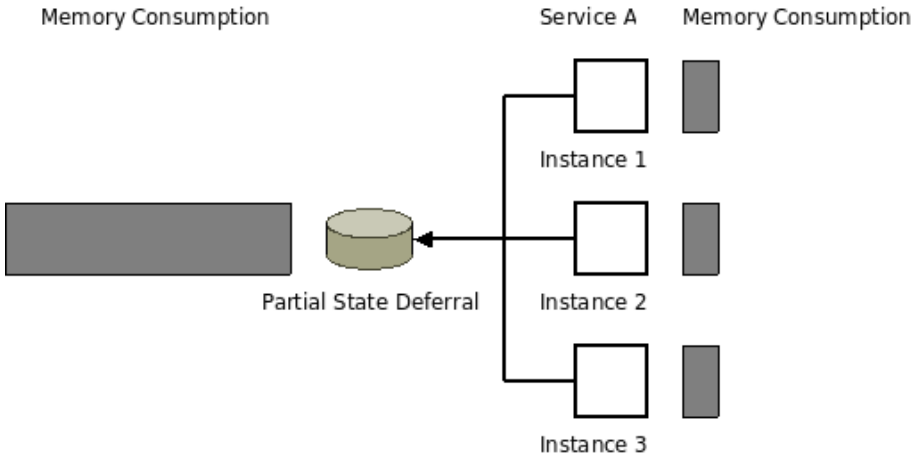

Figure 5.3: Memory load has been deferred to dedicated deferral service.

Correlations: This pattern is related to 'State Repository', 'Service Grid' (see the paper 'Service Inventory Patterns'), 'State Messaging' and 'Stateful Services' (see the paper 'Service Composition Design Patterns').

In application Partial State Deferral would use some sort of caching mechanism to avoid redundant data tuples. This would correspond to the flyweight pattern (mind this is an object orientation pattern) to some extend, which uses "sharing to support large numbers of fine-grained objects efficiently" [ERRJ95].

Other patterns in this category

Service Facade, UI Mediator, Redundant Implementation, Service Data Replication, Partial Validation.

5.3.3 Service Security Patterns

As services and service compositions have various consumers that are not yet all known at design time there is always the risk of security branches that were not considered back then. Therefore service security patterns deal with issues such as services that leak sensitive data ('Exception Shielding'), inbound messages with malicious intend ('Message Screening'), avoidance of direct access to essential resources for consumers ('Trusted Subsystem') and offer-

ing of internal enterprise resources to external consumers ('Service Perimeter Guard').

Exception Shielding

Motivation: Error handling routines can give away sensitive internal enterprise data, such as implementation, infrastructure or other kind of details, when a thrown exception is passed to a consumer. An attacker could intentionally trigger exceptions to gather application details, such as server name strings, environment variables, stack traces, etc.

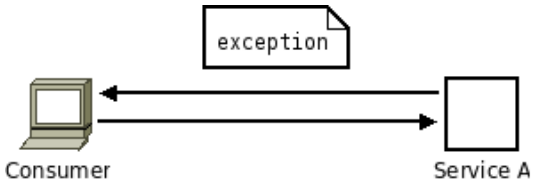

Figure 5.4: An exception is passed to the consumer containing sensitive data.

Pattern: To tackle this issue one can install additional logic that will monitor each outgoing service-response to the consumer. Such sanitization logic is usually put in between the contract and the service logic, that means it resides inside the service itself. If then a message containing sensitive data not meant for third parties to see is detected, the relevant details will be pruned or exchanged.

A simple approach is a system where sanitization routines write the original exception message into an error log (maybe into a dedicated error log database), and replace the whole exception text with a general error message. The entry in the error log will have a reference code that will be passed to the consumer in the modified error message. On triggering an error the third party may now contact the administration of that service and pass their error reference code for lookup purposes. The exception can be dealt with without exposing internal details to others.

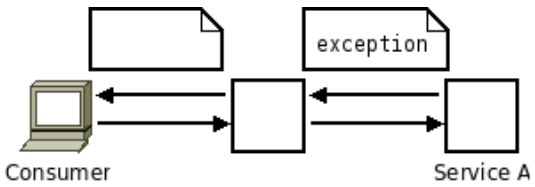

Figure 5.5: The sanitaztion logic strips all sensitive data.

Correlations: Shielding mechanisms are often implemented through 'Service

Agent' (see the paper 'Service Composition Design Patterns') or through Utility Abstraction (see the paper 'Service Inventory Patterns'). Service Perimeter Guard is often combined with this pattern, if it is not meant to be bound to one single service but should be used as a general shielding mechanism for several others.

Other patterns in this category

Message Screening, Trusted Subsystem, Service Perimeter Guard.

5.3.4 Legacy Encapsulation Patterns

When moving an enterprise towards an SOA driven environment established legacy systems have to be taken into account. They have to be altered to work together with the new service oriented surroundings. Therefore Legacy Encapsulation Patterns accomplish this task by equipping them with service interfaces corresponging to service contracts ('Legacy Wrapper') or introducing mediation logic for delivering data from legacy systems in a manner that matches the SOA idea ('Multi-Channel Endpoint').

Multi-Channel Endpoint

Motivation: Legacy systems that serve multiple 'delivery channels', such as personal computers, cellphones, other services, often had an own implementation each for every channel. For example let's assume some services 'A' and 'B' exist, service A offering some kind of service to only workstations, whereas service B can be accessed via workstations and mobile devices (see figure 5.6).
Usually legacy systems directly implement those delivery channels, meaning service A has some sort of directly implemented functionality to serve workstations and service B has implemented delivery channels for workstations and mobile devices on it's own. To make the situation even worse there might even be plans to deploy service A to mobile devices also in the future, which would mean to implement a new delivery channel into this legacy system from scratch. It's obvious to see that each delivery channel connection means a new implementation.

Running several of these legacy systems in parallel implies a lot of redundacy and governance burden.

Pattern: To tackle this problem it is suggested to install a mediator-service. On one hand each legacy system has to only implement a delivery channel

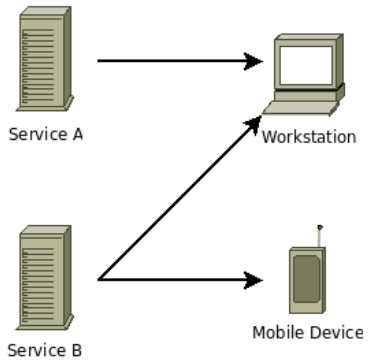

Figure 5.6: Multi-channel solution without Multi Channel Endpoint.

to this mediator (via legacy wrapper for example), on the other hand all delivery channel specific logic can be implemented in one single place, reducing redundancy.

The advantages are clear, delivery channel implementations are being reused and have to be implemented only once. If a delivery channel has to be modified, there is only one implementation that has to adopt the changes instead of one for each service, this eases service governance. Adding a new delivery channel will make it available to all services instantly.

Correlations: Multi-Channel Endpoint can be viewed as a specialization of 'Legacy Wrapper', as it offers a centralized service interface or contract for usage of legacy systems by certain delivers channels / consumers.

Also it can be seen as a form of the object oriented pattern 'Mediator' ("Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.", [ERRJ95]).

Because Multi-Channel Endpoint serves capabilities of multiple legacy systems in a unified way it is also related to data transformation patterns, such as 'Service Broker', 'Protocol Bridging', 'Data Format Transformation' and 'Data Model Transformation', that are not treated in this paper, but can be looked up in the book of Thomas Erl [Erl09].

To improve service performance and reliability 'Redundant Implementation' and/or 'Composition Autonomy' can be combined with this pattern. In 'Redundant Implementation' there are multiple implementations of a service residing in the environment to improve reliability, whereas in 'Composition Autonomy' agnostic services of a composition are implemented once more solely and dedicated for this composition, so it is not shared with other consumers, that may weight down the runtime performance (see the paper 'Service Composition Design Patterns').

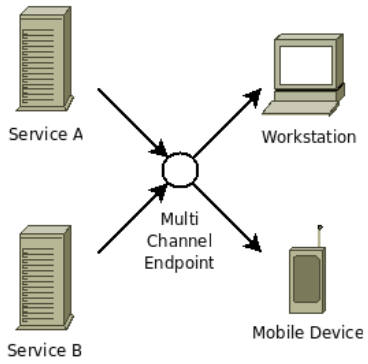

Figure 5.7: Multi Channel Endpoint

Other patterns in this category

Legacy Wrapper, File Gateway.

5.3.5 Service Governance Patterns

During the lifetime of an SOA environment there are changing requirements and new conditions that need to be dealt with. These changes may invalidate the original design approach, so the design will have to be updated to meet the new conditions. Service Governance Patterns are an approach to tackle issues as versioning of services, refactoring service logic in existing environments and rearranging compositions ('Service Decomposition', 'Decomposed Capability', 'Proxy Capability') while keeping a smooth architecture.

Service Decomposition

Motivation: Successive compositioning of services over time may lead to voluminous service compositions that divert from optimal performance because of their large footprint. Assuming we are working with a service 'PurchasingService' that represents purchasing departement logic, let's further assume suppliers and our own purchasing department work with this service, yielding capabilities such as AddOrder, AddInvoice, AddInvoicePolicy, GetOrder , GetOrderHistory, GetInvoice, GetInvoicePolicy, DeleteOrder, DeleteOrder, DeleteInvoicePolicy, ValidateInvoicePolicy.

It often occurs that services become a governance problem when they serve multiple purposes. It is difficult to manage a software that has no clear, dedicated purpose, but has a variety of not really related functionality implemented.

Also it's not unlikely that the greater a service's capabilities grow, the more it's runtime performance diminishes.

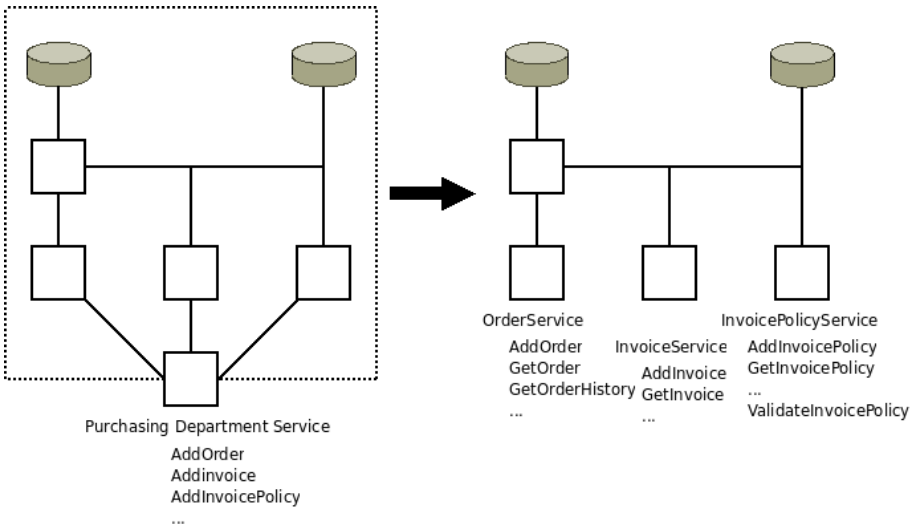

Figure 5.8: Complex service composition, serving various capabilities on the left and it's decomposition on the right.

Pattern: Service compositions can be split up into smaller, finer compositions. For our example this means we can split the service composition into the compositions OrderService (AddOrder, GetOrder, GetOrderHistory, DeleteOrder), InvoiceService (AddInvoice, GetInvoice, ..) and InvoicePolicyService (AddInvoicePolicy, GetInvoicePolicy, ..., ValidateInvoicePolicy). The service composition has been stripped down into a more accurate structure, complying with the business structure (e.g. a supplier should only access invoice records), thus easing administration and distributing the consumer load on multiple services.

Correlations: The most relevant pattern to Service Decomposition is Proxy Capability. If a service composition is split up via Service Decomposition, the consumers are left with no valid service contract to work with. This is where Proxy Capability comes into play, for further details please consider reading the corresponding chapter in this paper.

Minor related patterns are Entity Abstraction and Utility Abstraction, as Service Decomposition is frequently applied to agnostic services.

Proxy Capability

Motivation: A service decomposition invalidates the service contract, as the capabilities are no longer present in the same service, but distributed among different services. Consumers can no longer work with the contract. Let's

assume the example from the previous pattern (Service Decomposition) and further assume that some purchasing department software was capable of displaying invoices and orders. This program can no longer work as usual, as the original service has been split up and the service contract no longer exists.

Pattern: The service contract is therefore turned into a proxy service, preserving the contract and additionally implementing the old composition logic. This means that all cosumer software, like that purchasing department software mentioned previously, doesn't have to be altered instantly, but the proxy service can be used.

Comsumers using the proxy should still be urged to accomodate to the new architecture and use the decomposed services, to take advantage of the performance gain of the service decomposition and prevent the overhead of the proxy service.

Correlations: As stated before Proxy Capability is applied after Service Decomposition to preserve the functionality of compositions and consumers, therefore those two patterns have a strong relation.

Also it is clearly related to the object oriented pattern 'Proxy', which provides "a surrogate or placeholder for another object to control access to it." [ERRJ95], where in the context of service orientation the proxy replaces the former existing composition.

Other patterns in this category

Compatible Change, Version Identification, Termination Notification, Service Refactoring, Distributed Capatibility, Decomposition Capability.

5.4 Summary

In this paper we discussed several design patterns to achieve a solid, easy to maintain, high-quality (see 'Design Patterns in SOA' for SOA quality attributes) service oriented architecture. The patterns cover the issues concerning identification of service potential in business logic, reusability of solution logic, implementational details such as runtime performance enhancements, also security related patterns to keep up a reliable and secure environment, patterns that harmonize existing legacy systems with service orientation and patterns that help to deal with the further management and governance of the whole system during lifetime.

SOA patterns are a powerful tool to make the move towards a service driven IT environment in an elegant, yet simple way.

Bibliography

[Erl09] Thomas Erl. *SOA Design Patterns*. Prentice Hall, 2009.

[ERRJ95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman, Amsterdam, 1995.

Chapter 6

Service Composition Design Patterns

Christian Kuhl

Contents

Abstract: Code redundancy can decrease the efficiency of an enterprise's IT and it's applications. This paper presents the most important service composition design patterns from the book "SOA Design Patterns" [Erl08a] which help to overcome this problem by introducing and optimizing service orientation which implies a higher degree of reusability and flexibility.

6.1 A general problem when constructing business solutions

"Why do we need Service oriented architecture (SOA)?" One might ask, considering that traditional ways of developing software have proven to be quite successful in many different situations. But the approach of "identifying the business tasks to be automated, defining their business requirements, and then building the solution logic" [Erl07] might not be as good as it seems at first glance. Even though the requirements are very specific and each solution might be implemented in an efficient, straight forward manner, this approach demands that new applications are built from scratch over and over again or many different parts of them have to be changed in order to be suitable for new problems. First of all, this is highly wasteful because it "results in a significant amount of redundant functionality" and it would be much more efficient to avoid building and rebuilding solution logic that already exists in the system inventory [Erl07]. This also implies an inflation of the system inventory and much higher governance costs. Last but not least, "integration becomes a constant challenge" [Erl07] if applications have to share data at a later point but were not designed to be interoperable.

6.1.1 Applying SOA to this problem

Service orientation is an attempt to overcome these difficulties by dividing the solution logic into reusable, generic parts which then can be used in different contexts. As long as the task or function being provided is well defined and can be relatively isolated from the other tasks, it can be distinctly classified as a service. It can be viewed as a container of related capabilities [Erl07]. The main goal is to relieve the IT enterprise from unnecessary burdens and thereby "improving it's effectiveness" [Erl08b]. This paper first gives a brief explanation of the so called "goal states" and then discusses the most essential patterns of service composition as they are presented in "SOA Design Patterns" by Thomas Erl [Erl08a] to reach these states. These design patterns are followed by an examination whether and to which extend to realize them in the real world.

6.2 SOA Design Patterns

There are eight fundamental principles of service orientation which, when applied correctly, help to reach so called “goal states” or desirable situations [Erl08a] and [Erl07]. These eight principles are:

1. **Standardized Service Contract:** Service contracts have to be standardized. This helps to keep services within the same inventory more compatible to each other.
2. **Service Loose Coupling:** This principle regulates the coupling between services. Loose coupling implies a high degree of independence of the services from each other.
3. **Service Abstraction:** Only really necessary information about a service can be found in its contract. By hiding the other characteristics the integrity of future couplings with other services is protected.
4. **Service Reusability:** This is the most fundamental goal of service orientation. It simply imposes that services are reusable in different contexts and therefore try to avoid the redundancies mentioned above.
5. **Service Autonomy:** Services have to be able to run independently from their surroundings.
6. **Service Statelessness:** A service has to be stateless. Ideally, a service is used in many different contexts and compositions and numerous instances of it may exist. Maintaining state information would be counterproductive in such a situation.
7. **Service Discoverability:** For each Service there has to be information about its purpose and capabilities available.
8. **Service Composability:** Their design enables the services to be composed to more complex structures.

When introducing service orientation it is obvious that single services can’t do all the work that needs to be done and therefore it is essential compose more complex structures from services. But service composition has to be done in a certain way to take advantage of the structure of the single services, ensuring that it’s benefits materialize. The definition of a design pattern can be best described by the following quote by Christopher Alexander: “Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution of this problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” [AIS77]. A pattern following this definition has 4 basic elements [GHJV95]:

1. Pattern name
2. Problem
3. Solution
4. Consequences (trade offs, space and time complexity)

Erl describes a pattern as a “proven design solution for a common design problem that is formally documented in a consistent manner” [Erl08a]. This means that whenever a developer has to overcome one of these common design problems he or she can just apply this solution after reading it’s documentation without having to invent a new solution every time.

6.2.1 Capability Composition

When service compositions become more complex many different services interact with each other in different ways and a service may need to execute program logic outside of it’s boundary. However, this means a high risk of “compromising the integrity of service context” [Erl08a] in some cases. Instead of changing the service boundaries the service simply calls a capapility another service offers and all the executed program logic resides in capabilities of the involved services. This process is illustrated in figure 6.1. On the other hand

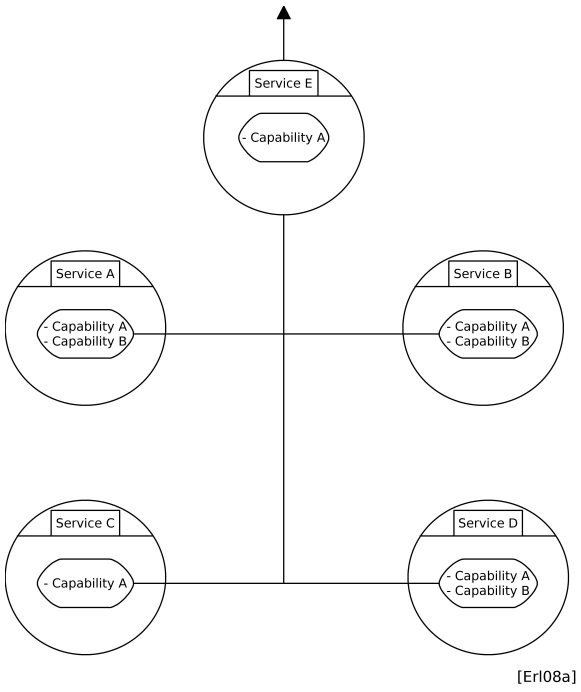

Figure 6.1: A service E invokes capabilities from other services.

this also means that the service gives up some of it's power because it lets other services direct certain tasks. Another disadvantage is the runtime overhead which can be caused by services invoking each other when they have a slow connection in between them. Capability composition is one of the most essential design patterns because it defines how services can take advantage of solution logic implemented in other services. Therefore it is a part of all eight design principles.

6.2.2 Capability Recomposition

The pattern "Capability Recomposition" is based on the results the previously discussed "Capability Composition" pattern generates. This time the focus lies on how a capability a service provides can be used in different contexts. As explained in the introduction one goal of service orientation is the avoidance of redundant solution logic. The application of this pattern means that it is possible to repeatedly reuse agnostic logic as part of different service aggregates [Erl08a]. Services become compatible and do not have to be integrated separately. In practice this means that it is now much easier to adapt the solution logic consisting of service compositions to changing business processes as well as an increased return on the initial investments made to create the services. As well as with "Capability Composition" this pattern's universal qualities let it support all eight design principles.

6.2.3 Service Messaging

Many protocols for remote data interchange are designed to establish a binary connection between two services which stays alive for as long as there is a need for communication. Especially in large service compositions this leads to massive overhead because these connections have to be maintained. One possible solution for this problem is the so called "Service Messaging" pattern. Whenever a service needs to communicate with another one it sends out a single and independent message without establishing a permanent connection. This pattern does not only effect the services themselves but also the underlying infrastructure of the enterprise. The technical infrastructure is responsible for routing messages to their destinations and makes an application of "Capability Recomposition" possible and services have to be designed to use it. However, the application of "Service Messaging" introduces security risks which will be addressed by "Data Confidentiality" 6.2.9, "Data Origin Authentication" 6.2.10 and "Brokered Authentication" 6.2.11.

6.2.4 Messaging Metadata

Because of the statelessness demanded by principle number 6 it is sometimes hard for a service to get state information about a running process. Instead of storing the state information within the service “Messaging Metadata” which is built upon “Service Messaging” proposes to enrich messages with metadata containing additional information about the process such as the mentioned state information as well as other instructions. But this pattern also helps realizing the 2nd principle “Service Loose Coupling”. Figure 6.2 visualizes this mechanism. The services don’t have to share as much process-specific logic as before because of the additional information contained in the messages and are therefore also more compatible with “Service Recomposition”. In practice this means that the messages have to contain a header to store the metadata. Usually there is a trade-off between reduced memory requirements on the one hand and higher performance overhead at runtime on the other one because services have to interpret the metadata and and react according to it.

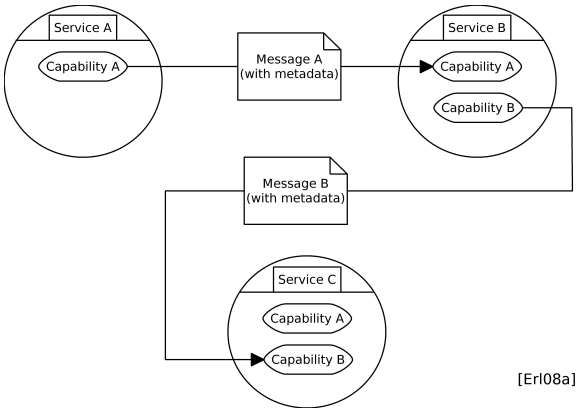

Figure 6.2: The requirements for services to contain embedded, activityspecific logic are reduced.

6.2.5 Service Callback

There are two types of situations when it makes little sense for services to communicate synchronously:

1. Service A sends a request message to service B and service B wants to answer with several messages
2. Service A sends a request message to service B and before service B answers it needs some processing time. During this time the resources which are in use by service A are blocked and it consumes memory.

Especially in large and complex service compositions the second point can lead to a massive decrease of performance at runtime. The solution is given by the pattern “Service Callback” which uses “Service Messaging” as well as “Messaging Metadata”. It proposes to attach a so called callback address to the message header. It contains a location the receiver has to send a callback message to as soon as it is done processing the input. This way the original sender can release it’s resources and commence doing other tasks in the meantime. This gives us the power to create more complex service compositions and therefore realizes the principles “Service Composability” as well as “Service Loose Coupling”. In practice this means that the underlying infrastructure has to be able to manage service addresses as well as offering a routing mechanism like a variant of a service agent. Also, contrary to synchronous messaging, here an acknowledgement mechanism has to be implemented if it is important to get a confirmation of arrival.

6.2.6 Asynchronous Queuing

When a service sends a message to another one, deallocates it’s resources and begins another task one problem arises: it needs a mechanism to store the incoming answer (as well as other messages) while it is still occupied.Therefore “Asynchronous Queuing” proposes to equip each service with an intermediary buffer to store incoming messages. This queue then polls an unavailable service repeatedly until it is available or the message transaction is considered a failure. “Asynchronous Queuing” enables the architect to construct more sophisticated and creative message exchange patterns as well as to optimize existing communication relationships between services. But also these advantages come with trade-offs: complex service compositions become even more difficult to control and a service does not get any information about the successful delivery of the messages it sent. As mentioned above, “Service Callback” is usually applied together with “Asynchronous Queuing” because of their common goal to eliminate synchronous communication relationships. Another pattern used to enhance “Asynchronous Queuing” is “Messaging Metadata” because it provides messages with instructions and information which is needed to direct the message flow. “Asynchronous Queuing” part of another important pattern called “The Enterprise Service Bus” which is discussed in section 6.2.14.

6.2.7 Event Driven Messaging

When a consumer waits for a service’s response, events which are important to this consumer may happen within the boundaries of the service provider. One way the consumer would be able to learn about these events would be to constantly poll the service. “Each polling cycle involves a synchronous, request-

response message exchange“ [Erl08a] as shown in figure 6.3. This way of dealing with the problem is highly inefficient though. A much more elegant

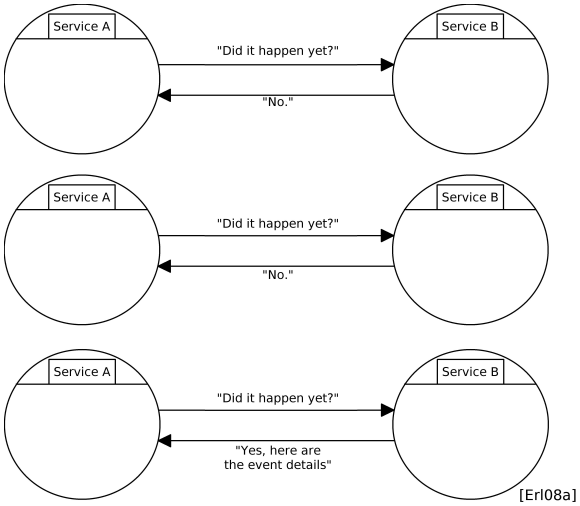

Figure 6.3: Service A constantly polls service B in order to learn about an event as soon as possible.

solution provided by the pattern ”Event Driven Messaging“ is the introduction of a so called ”event management program”. The service consumer registers as a subscriber whereas the service itself acts as the publisher. As soon as the event occurs, the publisher sends a message to the event manager which then distributes it to all subscribed consumers. Figure 6.4 shows this mechanism. Of course the necessary infrastructure for the event manager has to be provided. This is usually done through runtime platforms, messaging middleware or ESB products [Erl08a]. It is based on asynchronous message exchanges and therefore based on other patterns like “Asynchronous Queuing” and “Service Messaging”. Furthermore it realizes the 2nd principle “Service Loose Coupling” as well as principle number 5 “Service Autonomy” because services become more independent from each other since they don’t have to keep polling each other constantly. This pattern is closely related to the “Observer Pattern” [GHJV95]. In this case too one object acts as the observable subject which allows other observer-objects to register and unregister themselves. As soon as a change in the subject occurs it calls a “notify”-procedure which then notifies all registered observer-objects. The main difference is that we don’t find an additional layer called “event manager”. It’s functionality is taken over by the subject.

6.2.8 Agnostic Sub Controller

Service compositions are assembled to handle a specific task. Often we find task-specific logic as well as agnostic logic which could be used in other con-

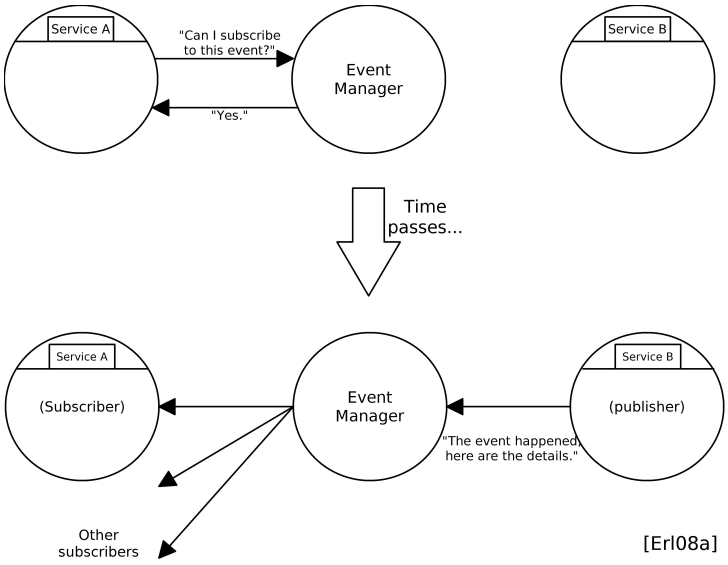

Figure 6.4: Now an event management program distributes messages about the event that occurred in Service B.

texts within a service. This lessens the reuse potential and leads to redundancy. The solution is to outsource reusable logic and direct it with a separate service called an “agnostic sub-controller” as it can be seen in figure 6.5. This logic may still be needed by the parent service but can now be addressed independently by other programs as well. As already mentioned above, the main goal of this pattern is to reduce redundancy and increase the possibilities to build complex structures. Therefore it supports the two basic principles number 4 “Service Reusability” and number 8 “Service Composability”. In practice it can be realized in two different ways [Erl08a]

- *New Agnostic Service:* The logic forms the basis of a new agnostic service
- *New Agnostic Capability:* The logic remains within the task service and is made accessible via a new capability in the contract.

6.2.9 Data Confidentiality

This pattern supports “Service Messaging” and “Messaging Metadata” by protecting the transmitted data against attackers. There exist techniques to secure message contents at transport layer against secret listeners between two stations. The problem with this method is that it only guarantees secure data transmission if only point-to-point connections are taken into consideration. As soon as we talk about large service compositions and the message

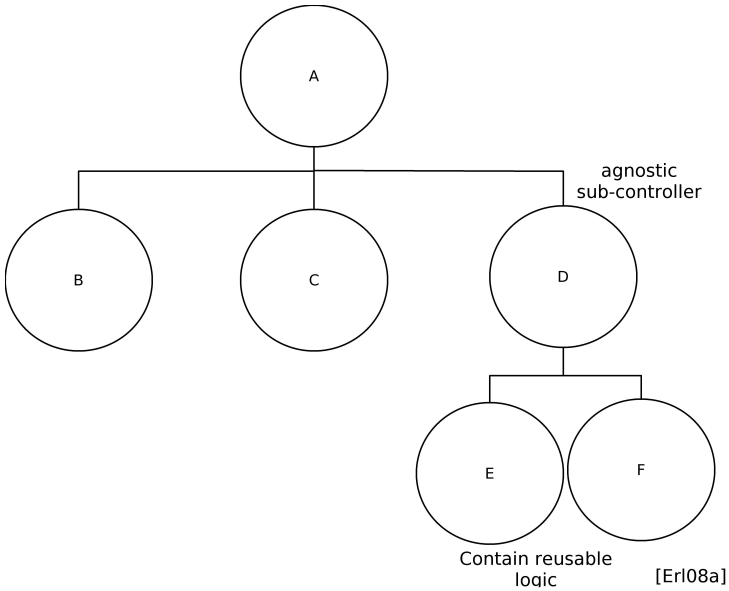

Figure 6.5: The reusable logic of D and E is now managed by it's own sub-controller E.

passes through several intermediaries a secure transfer cannot be guaranteed anymore because when a message passes through such a service it gains access to the data inside of the message. The solution is to encrypt messages independently from their transport through the composition. The downside is again more runtime overhead to successfully encrypt and decrypt the data. In the case of Web Services this is commonly done with XML Encryption which is very "efficient because information that isn't confidential can be sent unencrypted"[Gar03]. It even allows "encrypting different portions of the same document according to different encryption keys, and selectively distributing these keys to the various users according to the access control policies"[BE02]. In general, there are two ways to encrypt messages:

- *Symmetric encryption:* Sender and receiver both share a common key and use it to encrypt and also to decrypt messages.
- *Asymmetric (public key) encryption:* Every station has a public and a private key. The sender encrypts the message with the public key of the receiver who can then decrypt it with his private key. These keys can also be used for signing documents as explained in "Data Origin Authentication" 6.2.10.

Encryption does not only impose a performance overhead at runtime. There is also a governance overhead for managing the keys and ensuring that all keys are truly chosen randomly. This technique alone however does not prevent cryptographic attacks when many messages were collected which is why

usually a session based key is used [Erl08a]. Also, as mentioned above, another problem might be that it can't be validated if the sender is really who he declares to be. This will be taken care of in the next section.

6.2.10 Data Origin Authentication

As already stated in "Data Confidentiality" the sender's identity can't always be verified and someone else might inject false messages into the network under a different name. One way to solve this problem is presented by the pattern "Data Origin Authentication" in which the sender signs every message digitally. This way two things can be assured [Erl08a]:

- "The message has not been altered while in transit"
- "The message originated from the expected sender"

Signing a message can also be done in two different ways:

- *Symmetric:* The most common one is the "Message authentication code (MAC)". The MAC is calculated from the message checksum and a shared secret between the two services.
- *Asymmetric:* Here, the private key is used to create the signature while the public key serves as a tool to verify it.

"Data Origin Authentication" has the same impacts on runtime and governance overhead as "Data Confidentiality".

6.2.11 Direct Authentication

Often a service deals with data which is supposed to be only available for a certain group of consumers. So whenever a consumer program asks for the data it is required to provide some credentials to prove it's identity. This is usually done by asking for a username and a password or alternatively using a keyed hash message authentication function. The service then contacts an identity store which contains the rights and privileges of all users to verify whether to grant access to the data or not. This process is illustrated in figure 6.6. Note that "Direct Authentication" needs to be supplemented by other security patterns like "Data Confidentiality" and "Data Origin Authentication" to prevent misuse of the system. If every service is provided with it's own identity store this implies governance costs to keep this data consistent. On the other hand, if multiple services share identity stores they loose their autonomy. Another important aspect in practice is the password management. Both sides

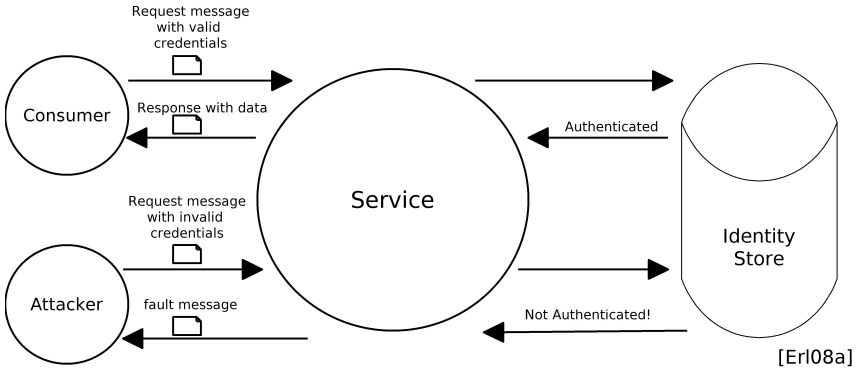

Figure 6.6: Visualization of “Direct Authentication”.

have to keep the passwords secure to prevent a misuse of the service or service composition. Also, because there is no sign-in functionality, a consumer may have to send identification data several times.

6.2.12 Brokered Authentication

In the following scenario we assume that per default service and consumer don’t trust each other. But in this case there was no previous communication negotiating who trusts whom. Also, consumers don’t want to use different login data for each service. The pattern we will now discuss is an alternative to “Direct Authentication”. What we need here is a “middle man” represented by a so called “authentication broker”. It validates credentials and is trusted by both sides. Whenever a consumer wants to use one or more services it contacts the authentication broker and is given a token which then can be used to access the services. “The broker can issue a token to an authorized consumer that is specifically scoped for that service” [Erl08a]. One well known protocol is called the “Kerberos Protocol”. The consumer “uses a series of encrypted messages to prove to a verifier that a client is running on behalf of a particular user” [NT94]. It is based on the DES-Algorithm, a symmetric encryption method. Initially, client and server do not share a key so it is the task of the authentication server to generate one and send it to both sides. In Kerberos even “subsequent authentication without re-entry of a principal’s password” [NT94] is possible and tokens are usually valid for 8 hours. However, the Key Distribution Center (KDC) must be constantly online and without it many parts of the service composition may stop working. It must also be extra secure because if hackers could get access to a KDC they could use the information to do serious harm to a service composition.

6.2.13 Transformation Patterns

When following a bottom-up approach one might encounter several interoperability problems. The following 3 patterns are meant to convert between different protocols, data formats and data models. “The required use of these patterns can indicate an inability to achieve the standardization required to realize service-orientation to its full extent within a given environment.” [Erl08a]

Two or more services may use different schemas to represent the same data. Sometimes schemas are standardized in the beginning but when services are reused to form other compositions incompatibilities are often introduced. In those cases “Data Model Transformation Logic” is used to convert between the two formats. It can either be part of the service architecture or of a separate middleware platform. This approach increases the overall complexity of the composition as well as it imposes a performance overhead for converting between the models.

In the case of mismatching data formats between services a layer of “Data Format Transformation Logic” is used for conversion. In practice it can either be a part of a service’s logic, a separate service or implemented as a service agent. Whenever this pattern is applied this also implies a transformation of the source data model and therefore an application of “Data Model Transformation”. And just like this pattern, “Data Format Transformation” adds to the runtime overhead and increases the design complexity.

It is also commonly applied together with the third transformation pattern called “Protocol Bridging”. In this case two services use different communication protocols which imposes the need of a bridging logic for dynamically converting between the protocols.

An important thing to keep in mind is that these three transformation patterns are only used out of necessity and are only applied when other patterns fail to reach the necessary level of standardization. “Their overuse is considered an anti-pattern” [Erl08a] because it can undermine goals of other patterns like “Capability Recomposition”.

6.2.14 The Enterprise Service Bus

The last pattern in this collection represents a pattern composed of several other patterns discussed before. It provides a foundation to transfer messages from service A to service B. First, the “Service Broker” pattern converts the message, then “Intermediate Routing” sends it to it’s destination “Asynchronous Queuing” takes care of storing it until service B is ready. It can of course be extended by several other patterns also presented in “SOA Design Patterns” [Erl08a].

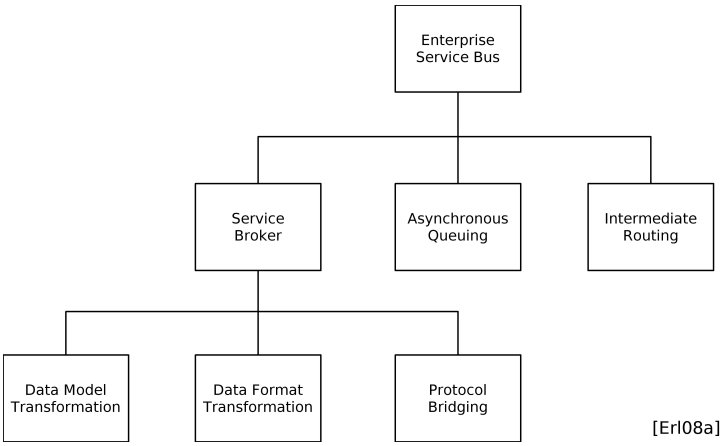

Figure 6.7: Structure of the Enterprise Service Bus.

6.3 Summary and Discussion

It does not only come down to the question of whether to migrate to service oriented concepts or not but it is also important to plan to what extend to realize them. Different degrees of service orientation may have different benefits for different companies. On the one hand the initial costs can be significant: The staff has to be trained accordingly, the company may need new equipment and/or a new technical infrastructure and of course it takes time and effort to convert some existing applications to a service oriented design. At least in theory as soon as most of the migration is done it starts to pay off - especially in large enterprises which suffered from the above mentioned problems such as redundant code and a big and inefficient IT.

Take for example a company that wants to create an inventory record for each placed order and cross reference it to associated back orders. So far each application implements this functionality but it would be much more elegant to create an own inventory-service with an add-capability which only adds a record. The back order query could be already a capability of another service and simply be evoked when needed. This correlates to the pattern “Service Composition”. Further, “Service Recomposition” could be realized by standardizing service contracts in this situation.

In a different scenario the company may already have introduced a service oriented concept but services are still maintaining persistent connections to exchange data. Here, realizing “Service Messaging” can reduce overhead. And if Service A needs some data only Service B can retrieve it has to send Service B a message. But instead of just waiting for the answer like in a synchronous connection it can do other tasks in the meantime if the company introduces “Asynchronous queuing”.

These examples mean to illustrate the fact that there are always benefits connected to introducing a new service oriented pattern but in some cases the costs to realize it may exceed them. So the bottom line is that the degree of service orientation which has to be realized is highly dependent on the situation and the structure of the company and ultimately on an analysis of costs vs. benefits.

Bibliography

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press, August 1977.

[BE02] Elisa Bertino and Ferrari Elena. Secure and selective dissemination of xml documents. *ACM Transactions on Information System Security*, 5:291–331, 2002.

[Erl07] Thomas Erl. *SOA Principles of Service Design*. Prentice Hall, 2007.

[Erl08a] Thomas Erl. *SOA Design Patterns*. Prentice Hall, 2008.

[Erl08b] Thomas Erl. What is service orientation, 2008.

[Gar03] Lee Garber. Taking steps to secure web services. *IEEE Computer*, 36:14–16, 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[NT94] B.Clifford Neumann and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communication Magazine*, 32(9):33–38, 1994.

Chapter 7

Metric-Based Clusters Enabling Component Re-Use in Service-Oriented Environment

Andreas Hahne

Contents

Abstract: Re-usability is a great concern in software development. SOA is supposed to improve this requirement. This paper describes an approach by He et al. to transform a component-based system into an SOA architecture by reusing some of the legacy components. Before, an explanation of how to properly analyse an existing system by dividing it into subsystems is given.

7.1 Introduction

Since the beginning of software development the resulting architectures became more and more complex. With growing systems the difficulty of maintaining the code grew accordingly. This resulted in the software crisis beginning in the 1960s [NR68].

A general idea to overcome this problem is to avoid a redevelopment of a completely new system in each project and instead to reuse existing components and configure them in the way that they collaborate correctly with each other in the new system. Such solutions have already been addressed at the beginning of the 1980s, e.g., by Belady and Evangelisti and used to promise the reduction of development costs since then [BE81].

An essential problem of code re-use is that a module is maintained by several developers throughout its lifetime which blurs the architecture. Documentation is likely to be out of date and cannot be used for re-engineering, either.

During decades different paradigms arose promising better support for understandability, maintainability, and re-usability. The small and thematically coherent pieces of object-oriented software systems gave rise to that hope, especially in connection with the introduction of the Unified Modeling Language (UML) which provides a standardized design approach. Software-Oriented Architectures (SOA) have come up in the last few years claiming that reuse can be improved further by that paradigm. This seems to be true, at least partially, since developers do only have to know the knowledge of a service and not the actual implementation. Indeed, the latter can be replaced without changing the rest of the system. But as with any new concept, it only works if it is applied in the right manner and best practices have to be invented.

Despite all these improvements the quality of system structure kept on suffering over time. Thus it became an important task for software engineers to redesign their system. A basic technique for that is to analyze inter-dependencies between modules (in procedural programming) or classes (in object-oriented programming). As traditional software systems usually are interconnected tightly detecting clusters is a difficult task. However, SOA claims that the interfaces between the components, called services, in this context, are loosely coupled and can thus be clustered more easily. The re-usability can then be realized by composing existing services to new ones.

Throughout the paper it will often be referred to an example of an insurance company. Such an enterprise has to fulfill a lot of different tasks in different departments which interact with each other. Hence, such a use case seems to be helpful to illustrate the general idea.

The remainder of the paper is structured in the following way. In Section 7.2 references to some clustering techniques are given as well as references to further re-usability approaches. Later in Section 7.3 a few clustering techniques, especially for analyzing component-based systems, and corresponding metrics are presented as well as a short description of the principle called *domain modelling* followed by a description of the system transformation approach presented by He et al. in Section 7.4. In Section 7.5 the lightweight approach is analyzed regarding applicability and some drawbacks are pointed out. Finally, a summary on the paper is provided in Section 7.6.

7.2 Related Work

While there exist many proposals on how to transform existing component-based and object-oriented systems into a well-structured architecture, ideas of how to transform existing systems into service-oriented architecture are just emerging in the last two years as presented int the sequel.

An important work in this field has been done by He et al. who transform a component-based system into an SOA environment [HTZZ08]. Doing so they make use of the Service Component Architecture framework developed 2007 by OSOA [OSO07]. A plug-in for designing such an SCA has been developed for the Eclipse IDE meanwhile [Fou09]. This approach of SCAlization, i.e. constructing components compliant with the SCA paradigm, assumes that the original system is designed in a reasonable manner. This paper will mainly refer to their work.

As the well-structured architecture is a prerequisite for the success of this approach, it is mandatory to consider different approaches how a software architecture can be analyzed regarding its structure and can even be improved. A lot of effort has been done in this field since the 1980s both for automatic and semi-automatic approaches [BE81] [HRY95]. Most of these works concentrate on the module level. The approaches aim at dividing, i.e., clustering, a system in reasonable subsystems, either flat or hierarchical ones. As the transformation of He et al. considers component-based systems also the presented clustering-approaches will mainly deal with this field of architectures. Clustering object-oriented systems however is also covered in literature[DM01].

7.3 Clustering Techniques

As described in the introduction, a lot of turnovers in module responsibility cause design information to blur. To overcome this problem it is necessary to restructure the system by analyzing the dependencies among modules or classes respectively. The term *dependencies* in this context mostly refers to generalizations, includes and the like. To this end, two main techniques have been developed. One aims at analyzing the dependencies by looking at the source code, the other at deriving dependencies by monitoring the system at run-time. The former is considered in this section whereas the approach of He et al. relies on black-box analysis mainly.

The general idea is to divide a system into subsystems, which can be eventually decomposed into subsystems again yielding a hierarchical structure. To evaluate the quality of a particular clustering several metrics have been developed. Of course, the modules of a system can generally be clustered in an arbitrary manner. However, it is common sense that a system is well-clustered if dependencies between clusters occur rarely whereas components of a single cluster are strongly interconnected. These two aspects refer to the two concepts of *cohesion* and *coupling*. The usual way of visualizing dependencies is via a *module dependency graph* which is explained in detail in section 7.3.1.

Regarding the employed algorithms for clustering there is one special class worth mentioned in this context, the genetic algorithms (GA) [Mit96]. As the name suggests already, it is oriented at the evolution theory and the principle of survival of the fittest. In this case, possible solutions of a problem deal as the population which may produce a new population in the next generation by selection, recombination, and mutation. In that way increasingly better clusterings are created from the original one. In general, this approach is usually employed for problems which are not computable efficiently as the case for clusterings. An actual clustering approach that refines the idea of genetic algorithms to *Evolution Strategy* is proposed by Khan, Sohail, and Javed [KSJ08]. Mancoridis et al. use the concept of GA, too [MMR+98].

Besides the mathematically founded metrics, another approach for structuring the system on a high level is described in section 7.3.3 and compared with the method of clustering. The Domain Model idea claims that companies of a particular domain, like insurance companies, are structured in a comparable way. Thus, the software systems can be designed in the same manner and many components can be re-used therefore. The Domain Model can be used to validate the constructed clustering of the software system.

7.3.1 Module Dependency Graph

A *Module Dependency Graph* (MDG) describes the dependencies (i.e., includes, invocations and the like) between modules in an intuitive way interpreting the modules as nodes and the relationships as (directed) edges.

A very small example is given in Figure 7.1. If we consider an insurance company as an example, there certainly is communication between the employees and the customers realized by the module `comm`. For information on the customers the database has to be accessed via the module `db` which logs information, just like `comm` does, via the `log` module. Moreover certain communication with customers affect existing or upcoming contracts processed in the corresponding module. Vice-versa, contracts may require communication from time to time, for instance if they are going to expire. Finally, to calculate contracts mathematical functions have to be taken into account using the `math` module. What can be derived from the figure especially is that cycles are well possible and also likely among modules (like between `comm` and `contract`).

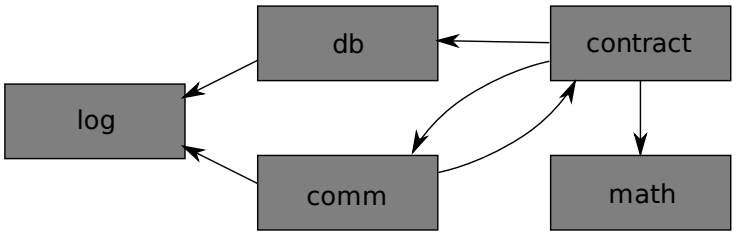

Figure 7.1: Module Dependency Graph

7.3.2 Clustering Algorithms And Metrics

An early work on clustering techniques has been provided by Belady and Evangelisti in 1981 [BE81]. They propose an automatic clustering approach and define a metric based on the assumption that the complexity of code understanding is proportional to the number of interconnections between the modules. They define a graph where the nodes do not only represent modules but code blocks, too. An edge between any two nodes means that a module refers to a control block. The aim of the clustering approach here is to avoid that a control block is referenced by too many modules as this will decrease the understanding of the system. The measurement Belady and Evangelisti define counts both the edges within and between clusters. They found that there exist local minima of the complexity based on the number of clusters in a system meaning that it is not desirable to have arbitrarily small clusters since the interconnections increase and have to be understood by the developers as well.

Mancoridis et al. propose a hierarchical clustering of the software architecture and introduce a measurement to evaluate the quality of a partitioning called *Modularization Quality* (MQ) [MMR+98]. They point out that the documentation of many systems is out of date and cannot be used to derive the original system architecture. Many changes on the architecture are most likely not compliant to the original design decision. So they concentrate on analyzing the source code automatically and cluster the system into a hierarchy of subsystems. The MQ calculates the quality based on the module dependency graph by awarding relationships within a cluster and penalizing connections among clusters. To this end, they define two measurements for *intra-connectivity* (A_i) of each cluster i and *inter-connectivity* ($E_{i,j}$) between clusters i and j in the following way:

$$A_i = \frac{\mu_i}{N_i^2} \qquad\qquad E_{i,j} = \begin{cases} 0 & i = j \\ \frac{\epsilon_{i,j}}{2N_iN_j} & i \neq j \end{cases}$$

where μ_i is the number of relationships in a cluster, $\epsilon_{i,j}$ the ones between clusters and N_i and N_j the number of components in the cluster i and j respectively. Both A_i and $E_{i,j}$ yield values between 0 and 1. A high value for the intra-connectivity indicates a good clustering as a change in one cluster rarely affects other ones whereas a high inter-connectivity is considered to be bad for maintenance since changes affect a lot of other clusters and changes have to be propagated to the developers of other subsystems. In order to derive MQ from these values it is defined as follows yielding a value between -1 and 1:

$$MQ = \begin{cases} \frac{1}{k}\sum_{i=1}^{k} A_i - \frac{1}{\frac{k(k-1)}{2}}\sum_{i,j=1}^{k} E_{i,j} & k > 1 \\ A_1 & k = 1 \end{cases}$$

Applied to the graph in Figure 7.1 a reasonable clustering (whereas probably not optimal) might be the partition:

$$\{\{\texttt{log}\}, \{\texttt{db}, \texttt{comm}, \texttt{contract}\}, \{\texttt{math}\}\}$$

Using MQ as a quality measure Mancoridis et al. developed several algorithms to find a partitioning which is optimal with regard to MQ and implemented a tool called *Bunch* realizing that. The term *partitioning* is used here in the usual sense, i.e. a set of subsets of a set S where the subsets contain the whole set S and where the distinct components are disjoint. The number of possible k-partitions, i.e., partitions of a set into k subsets, grows exponentially in the size of S which is reflected by the *Stirling numbers of the second kind*:

$$S_{n,k} = \begin{cases} 1 & k = 1 \vee k = n \\ S_{n-1,k-1} + kS_{n-1,k} & \text{otherwise} \end{cases}$$

An explanation for this formula is for instance given by `PlanetMath.org` [Pla07].

The **Optimal Algorithm** provided by Mancoridis et al. calculates MQs for all possible partitions and is thus applicable to systems with "up to 15 modules" only. This partitioning would yield a modularization quality of 0.0736.

Another important property used by the more sophisticated algorithms is the *Neighboring Partition* (NP) where partition P is considered to be a neighbor of partition P' if the partitions are the same except for one module located in a different cluster. Mancorids et al. state that the advantage of NP as they apply it is that a certain clustering is not fixed over time, but can be adjusted if necessary. Thus a bad decision at the beginning can be corrected in the later run.

Employing this idea, a **Sub-Optimal Algorithm** is based on selecting a random partition and select a better neighboring partition until no better one can be found with regard to MQ. This approach converges to a locally optimal solution, but it does not necessarily yield the partition with the overall best MQ. Another improvement is achieved by a **Genetic Algorithm** starting with a population of randomly selected partitions. The algorithm selects a set of partitions out of this population and calculates better neighboring partitions (with regard to MQ). In the next generation, the *fittest* partitions are selected, i.e. those partitions with the best MQ survive and can be recombined in the next generation. The algorithms can be combined to produce hierarchies on the system by applying them on the whole system and on particular clusters afterwards.

A general hypothesis of Mancoridis' et aliorum approach is that "well-designed software systems are organized into cohesive clusters that are loosely interconnected". Especially the last part gives rise to the assumption that such a "well-designed" architecture can be transformed into an SOA by considering the clusters as services which are loosely coupled among each other. As they propose a hierarchical clustering one could also think of combining several low-level services together in order to achieve more sophisticated ones.

In a later work they extended their framework in the way that two particular subsystems can be defined by the user or can be calculated automatically: one holding "client" modules which call many other modules and "suppliers" which are frequently called [MMCG99]. Moreover the user can use the system interactively and influence the process of clustering. Finally, it is preserved that small changes in the architecture do not affect the whole clustering.

A further analysis on MQ has been done by Chiricota and Jourdan [CJM03]. They propose an algorithm, which calculates kind of an edge density of a graph, and compare the results with the ones from Mancoridis et al. using the MQ metric. Furthermore, they explain which MQ values can be considered to be good, an issue that had not been addressed before. In contrast to Bunch they consider undirected edges. The idea of the algorithm is to consider an edge e with end points u and v as well as all possible 3- and 4-cycles

through e. To this end the neighborhood of u and v is partitioned in the way that one set contains all nodes neighbored with both u and v (denoted as W_{uv}), whereas the remaining ones contain those that are exclusively neighbors of u or v respectively (M_u, M_v). Using this partition 3- and 4-cycles through e can be identified in the following manner: (1) All 3-cycles contain one node of W_{uv} apart from u and v. (2) All 4-cycles contain either both remaining nodes in W_{uv} or one in W_{uv} and the last one in either M_u or M_v or one in M_u and the other in M_v.

Based on this idea, the edge density of 3- and 4-cycles through e can be defined. Two utility functions have to be introduced to this end:

$$s(U,V) = \frac{e(U,V)}{|U||V|} \qquad s(U) = \frac{e(U)}{\binom{|U|}{2}}$$

where $s(U,V)$ calculates the ratio of edges connecting nodes from U with nodes from V compared to the maximal possible number and $s(U)$ the ratio of edges in U with regard to the total possible number. The density for 3- and 4-cycles can thus be expressed in the following way:

$$\gamma_4(e) = s(M_u, W_{uv}) + s(M_v, W_{uv}) + s(M_u, M_v) + s(W_{uv})$$

$$\gamma_3(e) = \frac{|W_{uv}|}{|W_{uv}| + |M_u| + |M_v|}$$

The metric defined by Chiricota and Jourdan, called *Strength Metric*, is simply the sum of both the 3-cycles and 4-cycles density of a particular edge, called $\Sigma(e)$.

An argument for this metric is that a small value $\Sigma(e)$ for an edge corresponds to the fact that few small cycles lead through this edge e. Contrarily, if the edge was removed it is likely that the graph is split into separate components, e deals as an *isthmus*. The algorithm simply removes edges with a low $\Sigma(e)$ temporarily in order to identify possible clusters. By varying the threshold below which an edge is removed influences the number of resulting clusters.

In three case studies in which they clustered *ResynAssistant*, a software used in organic chemistry, as well as the *Mac OS9* and *MFC* API the resulting modularity quality was compared both for *Bunch* and their *Strength* algorithm. The resulting MQ values are comparable even if the values are slightly higher with Bunch. However the calculation speed using the strength metric is much higher. Interestingly, when clustering Mac OS9 and MFC respectively much more clusters have been computed by *Strength* than by *Bunch*.

If the algorithm is applied to the MDG from the example the problems comes up that some of the $s(U,V)$s are undefined as some endpoints of edges do

not have exclusive neighbors (like `log` for the edge $(\texttt{comm}, \texttt{log})$). Those edges are accounted with an infinite density meaning that there is no threshold for which the edge will be removed. There are only two edges which have a finite density: $(\texttt{db}, \texttt{contract}) = \frac{4}{3}$ and $(\texttt{comm}, \texttt{contract}) = \frac{4}{3}$.

So if the graph is clustered with an edge threshold of 1.4 or more the graph is decomposed into two components $\{\texttt{log}, \texttt{db}, \texttt{comm}\}$ and $\{\texttt{contract}, \texttt{math}\}$. Obviously this clustering is very different from the assumption made initially. Indeed, the modularization quality calculated for this clustering on the given graph is exactly 0 and is hence smaller than the one from the estimated clustering above. The main reason for this is probably that the edges are considered to be undirected by *Strength*. The original graph however suggested that there are two main components, `comm` and `contract` which require the others. This is no longer represented in the *Strength* approach. Moreover the system is very small and it is likely that a more reasonable clustering may come up if the graph is larger and if few edges have an infinite weight.

Additionally, Chiricota and Jourdan gave a theoretical foundation of what can considered to be a good MQ value. They analyzed the distribution of MQ values for arbitrary clusterings and found that this yields a Gaussian distribution with a peak around -0.2 meaning that most of the randomly clustered graphs have an MQ of that value. Everything above that value is a fairly good cluster, everything below is a poor one.

7.3.3 Enhancements Using Domain Model

As presented in the previous section finding the right clustering is not easy to achieve. A further heuristic to justify a clustering is the idea of *Domain Models* which represent an abstract view on a particular business domain. It identifies essential components of an enterprise, called *entities*, and connects them via *relationships*. It is claimed that the same model is applicable to all enterprises in a particular domain. Of course, this increases the possibility of reuse, too, since applications developed according to a domain model can be migrated to other enterprises with less effort than without such a model. Also in SOA environments a domain model can be very helpful if the entities are realized by services.

An essential problem however is to generate a domain model. One approach to that has been proposed by Oldfield [Old02]. Once a domain model has been created and proved to be reasonable it can be used to validate a clustering: If entities of the domain model can be mapped to the clusters of a module dependency graph in a coherent manner (e.g., by preserving relationships) it is likely that the clustering is reasonable, too. Furthermore, these clusters can be realized through services as the boundaries are justified. If a domain model exists for a particular domain the service architecture can be adopted by other projects in the same domain. Such a proposal has been made recently by

Wang et al. [WYF+08].

7.4 Component Re-Use In SOA Environments

Service Oriented Architecture promised to improve re-usability. However, repeatedly people declare the dead of SOA like Manes did [Man09], who claimed about the term *SOA*, which often is misused in her eyes, and not the general concept though. An approach to realize re-usability it proposed in this section by wrapping existing components with SOA elements and providing services accessing them [HTZZ08]. A mandatory requirement for reusing old code is, obviously, that existing components are known to the designer just like the semantics of them. To this end, services have to be described in a standardized manner in the way that the information can be retrieved and provided to a developer. Such an idea via XML-based description is presented by Sillitti and Succi [SS08].

In the sequel a standardized way of describing services and relationships is presented with the *Service Component Architecture* (SCA) [OSO07]. Afterwards a lightweight approach to constructing a service architecture respecting SCA from an existing component-based system by He et al. is introduced [HTZZ08].

7.4.1 Service Component Architecture

As stated by OSOA, SCA is a ”model for the assembly of services” and ”for applying infrastructure capabilities” to them [OSO07]. It has been developed by the *Open SOA collaboration* (`www.osoa.org`) in November 2005 and the Organization for the Advancement of Structured Information Standards (OASIS) has initialized the standardization process in April 2007 [Gey07].

The idea is to describe the composition of and interaction between services in both a graphical way and textual way. In SCA the provided interface is the actual *Service* whereas the implementation is called *Component*. The behaviour can be modified by a set of *Properties* and one service component can require the interface of another service component, which is called a *Wire* between the two components. A set of service components can be combined to a *Composite* which again can be combined together with others to higher-order composites. Besides that the composites, services etc. can be described with XML documents where the names of services, references or the classes of the actual implementations are connected with each other. As the implementation does not matter the model is applicable to any programming language. However, the examples described in the specification are limited to Java. Using this standardized way SOA can be described efficiently. The approach

to reuse components in a service architecture, which is described in the next section, is based on this specification.

7.4.2 SCAlization

An approach to partially reuse components from an existing system in a new service-oriented environment has been proposed by He et al. [HTZZ08]. It has been designed for scenarios in which the source code of the legacy system is not available for the migrant. Thus, the internal structure has to be retrieved indirectly. In the paper the two possibilities mentioned are the inspection of invocations at run-time by making use of reflection strategies and the analysis of design documents. Whereas the former has the disadvantage that not all dependencies can be detected in an insufficient analysis, the latter holds the drawback that design documents tend to be out of date compared with the system or even do not exist at all. Furthermore it is assumed that the received structure is sound with respect to the original idea of the designers and has not suffered from previous changes already.

As only some of the components of the legacy system should be re-used one has to identify them at first. Usually, those components reference others which can be derived from the module dependency graph and will be referred to as *depended components*. Finally there are components which will be reimplemented and are not referenced by components to be re-used. However these may reference the re-used components. These three types of components are transformed differently:

- **Components to be reused** are transformed into *SCA surrogates*, i.e. components according to the SCA paradigm. The original invocations are redirected to the components to be reused since the behaviour of those should not change. The existing components remain in the new system, but outside the SCA structure.
- **Depended components** remain in the new system unchanged (but outside the SCA part). If the components to be re-used reference them they propagate the request via the SCA surrogate to the new implementation of the corresponding component, which is an SCA component now. Since the invocation has to be redirected to the services a wrapper for the original component is necessary, called *Legacy surrogate*.
- **Remaining components** are, similar to the depended components, reimplemented as SCA components (possibly with new features). As they are not referenced by components to be reused, the old component implementation does not have to be migrated. If the old component referenced a particular component in the legacy system the new SCA component will reference the corresponding service which will redirect the request to the component to be re-used.

As mentioned before the legacy components partially remain in the emerging system and are called SCAlized components because the SCA structure is wrapped around the old components which can still be used in the new environment. This SCAlized components themselves are no SCA components of course and thus have to be packaged independently of the rest. In Java they could be packaged into an EJB container for instance. The SCA components themselves, i.e., the SCA wrapper of the components to be re-used as well as the re-implementations of the other components which are not re-used are packaged together in an SCA container.

Considering the example of an insurance company again it might become a strategic desire to transform the component-based system into an SOA as this represents the business structure in a more comprehensive way. In the analysis process it becomes clear that the functionalities of `comm` and `contract` should be extended whereas the behaviour of `db` can remain unchanged. Neglecting the modules `log` and `math` as well as the relationships between `comm` and `contract` for the sake of simplicity, an architecture as depicted in Figure 7.2 could come up.

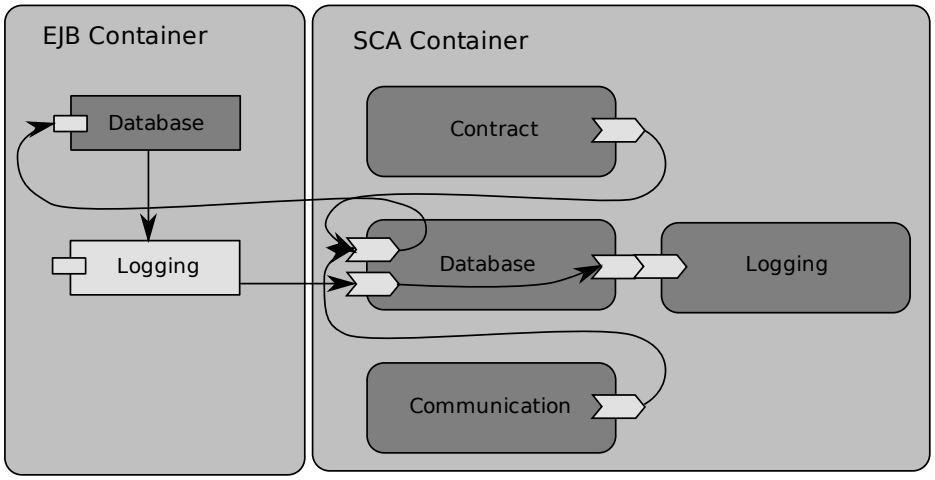

Figure 7.2: SCA version of insurance company

Since `db` is the component to be re-used it remains unchanged in the EJB container. Moreover an SCA component for the *Database* is created. The `db` component requires the `log` module and so a legacy surrogate for that is generated which can be called by `db` and processes the request via the *Database* SCA surrogate to the *Logging* SCA component. *Contract* and *Communication* both are supposed to be changed by assumption. As the legacy components called the `db` component the new SCA components call the SCA surrogate *Database* which propagates the request to the `db` component to be reused.

To SCAlize a whole system with a single pair of both EJB and SCA container is only applicable for very small-scale systems. In practice it is probably more

reasonable to cluster the system with one of the approaches mentioned in the previous section after the dependencies between modules have been analyzed. In that way, the resulting containers will remain relatively small and can be recombined to design further systems in advance.

To design SCA compliant environments an Eclipse plug-in has been developed as part of the SOA Tools Platform Project (STP) [Fou09]. It provides a graphical user interface to compose services and interconnect them as well as it generates the corresponding SCA meta model according to the specification. Although usability can be improved for this tool, it is another indicator that SCA is becoming more important in the SOA development.

7.5 Discussion And Comparison

The presented approach to SCAlize components is a very simple one to preserve legacy code in an emerging SOA environment. However it relies on the assumption that the legacy system itself is well designed. As the process is automated, once the components to be re-used have been identified by the developer, a bad structure of the legacy system would lead to the same bad architecture in the new environment, simply wrapped by SOA components. Thus, in order to use this approach effectively, the system being transformed has to be clustered before and possible drawbacks in the architecture have to be detected. This clustering is even more difficult as it is assumed that the source code is not available. So the strategies presented above can only partially be applied although the metrics are also valid for module dependency graphs derived from the run-time information, of course. Analyzing the dependencies based on the run-time behaviour and existing design documents has some disadvantages as mentioned before. After the old system is confirmed to be reasonably structured the described SCAlization technique is a powerful one. Small sets of components can be composed to SCA containers and can be combined with other containers derived in the same manner. Doing so, a hierarchical structure can be created which may improve the maintainability as a clearly structured new system emerged that can be managed better than the old one because the structure can now be explicitly identified.

7.6 Summary And Conclusion

In this work several approaches on how an existing system can be decomposed into subsystems, called clustering, were presented. Those approaches basically concentrate on analyzing source code. An important metric on how to evaluate the quality of a clustering was introduced with the Modularity Quality (MQ). Also related work analyzing and improving MQ has been provided.

These strategies are not bound to SOA environments, but are applicable to every component-based system. In the next section an approach to transform a component-based legacy system in a SOA architecture which complies to the description standard SCA has been presented. It wraps components that will be re-used into SCA containers and assumes that the original source code is not present. So the dependencies are analyzed based on run-time analysis as well as inspection of design documents. This lightweight approach makes only sense if the original system is considered to be well-structured as a bad structure would survive the transformation.

Bibliography

[BE81] L. A. Belady and C. J. Evangelisti. System Partitioning and its Measure. *The Journal of Systems and Software*, 2(1):23–29, February 1981.

[CJM03] Yves Chiricota, Fabien Jourdan, and Guy Melançon. Software Components Capture Using Graph Clustering. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 217, Washington, DC, USA, 2003. IEEE Computer Society.

[DM01] Lei Ding and Nenad Medvidovic. Focus: A Light-Weight, Incremental Approach to Software Architecture Recovery and Evolution. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, page 191, Washington, DC, USA, 2001. IEEE Computer Society.

[Fou09] Eclipse Foundation. SOA Tools Platform Project, SCA sub project, 2009. `http://www.eclipse.org/stp/sca/index.php`.

[Gey07] OASIS (C. Geyer). OASIS Advances Standards to Simplify SOA Application Development, April 2007. `http://www.oasis-open.org/news/oasis-news-2007-04-11.php`.

[HRY95] David R. Harris, Howard B. Reubenstein, and Alexander S. Yeh. Reverse engineering to the architectural level. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 186–195, New York, NY, USA, 1995. ACM.

[HTZZ08] He Yuan Huang, Hua Fang Tan, Jun Zhu, and Wei Zhao. A Lightweight Approach to Partially Reuse Existing Component-Based System in Service-Oriented Environment. In H. Mei, editor, *High Confidence Software Reuse in Large Systems*, volume 5030 of *Lecture Notes in Computer Science*, pages 245–256. Springer-Verlag Berlin Heidelberg, 2008.

[KSJ08] Bilal Khan, Shaleeza Shail, and M. Younus Javed. Evolution Strategy Based Automated Software Clustering Approach. In *Advanced Software Engineering and Its Applications*, pages 27–34, 2008.

[Man09] Anne Thomas Manes. SOA is Dead; Long Live Services, January 2009. `http://apsblog.burtongroup.com/2009/01/soa-is-dead-long-live-services.html`.

[Mit96] Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, Cambridge, MA, USA, 1996.

[MMCG99] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: a clustering tool for the recovery and maintenance of software system structures. In *In Proceedings of International Conference of Software Maintenance*, pages 50–59. IEEE Computer Society Press, 1999.

[MMR+98] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, page 45, Washington, DC, USA, 1998. IEEE Computer Society.

[NR68] Peter Naur and Brian Randell. Software Engineering: Report of a conference sponsored by the NATO Science Committee. October 1968.

[Old02] Paul Oldfield. Domain Modelling, 2002. `http://www.aptprocess.com`.

[OSO07] OSOA. Service Component Architecture – Assembly Model Specification, March 2007. `http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf?version=1`.

[Pla07] PlanetMath.org. Stirling numbers of the second kind, March 2007. `http://planetmath.org/encyclopedia/StirlingNumbersSecondKind.html`.

[SS08] Alberto Sillitti and Giancarlo Succi. Reuse: From Components to Services. In H. Mei, editor, *High Confidence Software Reuse in Large Systems*, volume 5030 of *Lecture Notes in Computer Science*, pages 266–269. Springer-Verlag Berlin Heidelberg, 2008.

[WYF+08] Jianwu Wang, Jian Yu, Paolo Falcarin, Yanbo Han, and Maurizio Morisio. An Approach to Domain-Specific Reuse in Service-Oriented Environments. In H. Mei, editor, *High Confidence Software Reuse in Large Systems*, volume 5030 of *Lecture Notes in Computer Science*, pages 221–232. Springer-Verlag Berlin Heidelberg, 2008.

Chapter 8

Security as a Service

Markus Vervier

Contents

Abstract: This seminar work presents the relevant technologies necessary to implement a *security service* in an SOA. In large scale enterprise environments securing services via traditional security measures does not work well. It is not feasible to implement security in each service seperately. This problem can be solved in a way natural to an SOA by implementing a central security service. The basic concepts and relevant technologies for a security service will be presented in this work. By using the SAML and WS-Security extensions to SOAP it is possible to secure large scale enterprise systems.

8.1 Introduction

The term Service Oriented Architecture (SOA) was first mentioned by the marketing company *Gartner* in 1996 by Natis in [Nat03]. Also mentioned in the name SOA, the *service* is the most important concept in a SOA.

In the lexicon [Mil06], a Service is defined as: *Work done by one person or group that benefits another*. This definition can also be extended to non-human agents. Basically a service can be used by a consumer, who benefits from it.

In technical terms, a service is defined in [MLM+06] as *a mechanism to enable access to a set of one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description*.

SOA enables the building of applications which use other applications via their services. Generally it shifts development from an application centric point of view to a service centric.

Services posses desirable attributes that come in handy in overcoming the drawbacks of organizing IT along application boundaries Kanneganti and Chodavarapu state in [KC08, page 7]. They define the following important attributes of services in [KC08, page 8]:

- The service definition is defined by the convenience of the consumer.
- A service is self-describing, so consumers are able to learn by themselves how to invoke the service. A service description should include service interface, wire format, transport, location, policies and the Service Level Agreement.
- A service is technology independent. It is interoperable with any possible consumer and idependent of the hardware or sofware platforms.
- A service is discoverable. Consumers looking for a service can discover its presence, usually via a service registry (like the yellow pages in a phone directory).

- A service can consist of other services building a high level service. For example several “technical services” can be composed into a “buisiness service”.
- A service is context-independent. That means it does not matter what the consumer did before invoking the service.
- A service is stateless.

A service is provided by the *service provider* for consumers to use. The consumers do not need to know anything about the service provider itself and vice versa the service provider does not need to know what possible uses the consumers are utilizing the service for. We can already imagine, that this may pose new security problems.

8.1.1 Definition of an SOA

There is no single accepted definition for an SOA. A well known definition can be found in the *Reference Model for Service Oriented Architecture* [MLM+06] by the Organization for the Advancement of Structured Information Standards (OASIS):

Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains.

A similiar definition is given by Wilkes and Harby in [WHB+04]:

A service-oriented architecture can be defined as a way of designing and implementing enterprise applications that deals with the intercommunication of loosely coupled, coarse grained (business level), reusable artifacts (services). Determining how to invoke these services should be through a platform independent service interface.

The term SOA is used heavily for marketing and there are a multitude of other definitions. Often *webservices* are mistaken with SOA, which is actually much more. Webservices may be used to implement an SOA, but technologies like CORBA or Remote Procedure Calls (RPC) may be used as well.

8.1.2 Enterprise SOA

For enterprise class SOA development, special requirements apply. In enterprise environments, a security solution is necessary to develop frameworks and architectures. Enterprise level SOA solutions model the whole business

of a company and are typically very large scale. Therefore standard enterprise concerns must also apply to an enterprise SOA security solution.

Kanneganti and Chodavarapu state three major concerns for an enterprise SOA security solution in [KC08]: **Ease of development**, **Manageability** and **Interoperability**.

In many services, the implemented security measures are overlapping and similiar. In classic software development, common elements of security to different services would be exported to a library to meet the three mentioned enterprise concerns.

To meet these goals in SOA enterprise development, security can also be implemented as a service. Since the service is a core concept of SOA it is natural to implement security as service.

8.2 Secure SOA

Due to the distributed nature of an SOA, security is especially necessary and hard to achieve at the same time. In distributed systems the intermediary devices and agents are beyond control of the endpoints. Often they must also be considered as being malicious. Messages in real world SOAs have to be transferred via the internet in many cases. There may be different intermediaries in these environments. A direct connection from source to destination may not be possible because of firewalls or other networking obstacles. Therefore standard security measures like Transport Level Security (TLS) will not work. Also problems like authorization and authentification must be solved. In the following sections, a basic definition of security and an overview of relevant technologies for secure SOA will be given.

8.2.1 Security

According to the guideline *Leitfaden IT-Sicherheit* [fSidI06] of the *Bundesamt für Sicherheit in der Informationstechnik* (BSI) there are four main requirements for security:

- **Confidentiality:** Confidential data has to be protected against unwanted disclosure.
- **Availability:** Services, functions and information are available at the required moment.
- **Integrity:** Data cannot be changed illegitimately and is complete and unchanged. This also applies to metadata like the time of creation.

- **Authenticity:** Network nodes and services are authentified to their peers and clients.

These are the main functional aspects of security, which are of concern for a secure SOA. *Privacy* is only partially covered by *confidentiality* and important, but cannot be viewed as a functional requirement.

Directly related to these requirements are the problems of *Authentification* and *Authorisation*.

Authentification

The problem of verifying the identity of a user is called authentification. The ways of presenting authentification information to an applications are called *authentification factors*. Kanneganti and Chodavarapu mention three authentification factors for a user to prove his identity in [KC08, page 14]:

- *What he knows:* Presenting secrets like username/password to the application.
- *What he has:* Posession of a special item like an RSA token which displays the right authentification number at the current time.
- *What he is:* Presenting biometric evidence like fingerprints or retina scans.

A good authentification mechanism does not rely on just one factor, but enforces at least a *twofactor authentification*.

Authorisation

Following *authentification*, an application must know which ressources and functionalities a user may use. This is *authorisation*. Which functionalities a user may use may depend on his role, the groups he belongs to and other predefined parameters the application may know about. For example an administrative user may be able to modify data, while a guest user may only read data.

8.3 Security as a Service

Traditionally the task of securing services is implemented seperately for each service and service consumer. Having multiple implementations of security in

each service makes meeting the enterprise development concerns stated in 8.1.2 difficult, if not impossible.

It is difficult to maintain and secure a multitude of services where each service has its own implementation. Additionally there is a huge overhead, because security measures have often the same properties and differ only in details. When a service needs to use other services, it has to transfer the security context to them. This can be a problem for *interoperability* and may cause a large overhead.

To mitigate these problems, security can be implemented as a service called the *security service*. This means outsourcing most of the security logic from consumers and services to a specialised service. Some security logic may remain at the service consumers and services, for example communication with the security service.

A security service can be invoked in different ways. *Out-of-Band* invocation means the security service is explicitely invoked by the source or destination endpoint. *Intermediary Invocation* means that an intermediary or some other device implicitely invokes the security service. This may happen transparently, without knowledge of the endpoints. For both types of invocation use case examples will be given now.

8.3.1 Out-of-Band Invocation

Explicit invocation of the security service by the endpoints may happen in three different ways. Basically the security service is invoked when needed by the source or destination endpoints or by both of them. The actual communication between the endpoints is done according to the assertions made by the security service. Of course in this scenario the security service has little control about the actual actions of the endpoints.

8.3.2 Intermediary Invocation

Intermediary invocation may happen transparently or explicit. Usually a router or similiar intermediary device invokes the security service when necessary. The security service can act as an explicit intermediary. The source and destination endpoint only talk via the security service, which is able to enforce security assertions. Of course the security service is a single point of failure and suffers huge load as central communication platform.

Another possibility is transparent intermediary invocation. The intermdiary invocating the security service must then be able to inspect and understand messages transferred from the source to the destination endpoint. By making

network devices such as routers capable of understanding and intercepting application level communication, they can be used for implementing a security service and transparently do securiy tasks. This is called Application Oriented Networking (AON).

8.3.3 Relevant Technologies

We will now have a look at the building blocks of a secure SOA. Since SOAP (formerly known as Simple Object Access Protocol (SOAP)) based web services are the most common type of service for an SOA. The relevant technologies for secure SOA presented below will focus on this type of service.

SOAP

SOAP was formerly known as the *Simple Object Access Protocol*, but this acronym was dropped with version 1.2 of the SOAP standard. It is a message exchange protocol and the core of most web services. According to Kanneganti and Chodavarapu in [KC08, page 55] the key elements of the SOAP model are:

- *A SOAP message is a transmission from one endpoint to another. Two transmissions can be combined synchronously or asynchronously to make a request/response. Multiple transmissions can be combined to make a conversation.*
- *A SOAP message is created by wrapping any application message with a standard XML-based envelope structure. The envelope structure enables applications to express semantics such as what is in the message and how it is encoded.*
- *Error handling is carried out using a standard SOAP fault mechanism.*
- *For **RPC!** (**RPC!**) calls and responses, application messages are structured using SOAP conventions.*
- *Data serialisation may be done using a standard SOAP defined encoding. Other encodings may be declared and used as well.*

SOAP messages are higher level than application transport messages and can be transferred via a multitude of other protocols and services, as the Java Messaging Service (JMS) or even File Transfer Protocol (FTP). SOAP messages are eXtensible Markup Language (XML) documents. An example SOAP message is shown in 8.1. Data is wrapped in a so called *SOAP Envelope*, which contains a header and a body. The header may contain additional information

for processing by the receiver of the message, as for example security token and authentification.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Header>
   ...
   </soapenv:Header>
   <soapenv:Body>
   ...
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 8.1: *Example of a SOAP message.*

SOAP has no build in security measures, but is extensible, thus working security extensions can be built around it.

Layers

According to Kanneganti and Chodavarapu in [KC08, page 89] there are three logical layers in which security in SOAP based services is commonly implemented. The lowest layer is the **Transport Layer**. It transmits arbitrary byte payloads, for example the HyperText Transfer Protocol (HTTP). The mid level layer, in which application layer data is transferred as a SOAP message, is the **SOAP Layer**. The highest level layer. The actual application data e.g. serialised application objects is transmitted on the **Application Layer**, which is the highest layer.

Working security measures need to work at least on the *Transport Layer* and the *SOAP Layer*. Only securing the transport layer is not sufficient. Sometimes direct communication is not possible and additionally it does not help to talk to a malicious attacker via a secure channel.

8.4 Protocols For Security Services

The security service needs to express its findings about security assertions and requests. For the communication of security information there are two protocols. **The SAML Protocol** is a protocol for security services based on the Security Assertion Markup Language (SAML).

WS-Security is a protocol to enforce *integrity* and *confidentiality* for web services. Both protocols will be described more detailed in the next sections.

8.4.1 The SAML Protocol

When the security logic is implemented at the service endpoint, security information cannot be held in-memory. It is available from a seperate security service. A security service needs to have interfaces to express its findings. The protocol commonly used to express security information is the SAML protocol. The SAML Protocol is based on the Security Assertion Markup Language (SAML). In the SAML it is possible to express authentification and authorisation information. SAML was developed by the OASIS and is available as a major revision of 2.0 since 2005. It was developed with focus on *Single Sign-On*, *Federated Identity* and *Web Services* according to Hughes and Maler in [HM05]. It consists of four basic components: *Profiles, Bindings, Protocols* and *Assertions* as shown in figure 8.2. These components will be described in detail below.

For example the problem of *Single Sign-On* is a common problem in web based SOA solutions. Since browser restrictions do not allow multi-domain cookies for security reasons, different services have problems passing authentication state to each other. This is solved by a security service speaking SAML which may act as a broker, issuing assertions to the different services.

According to Kanneganti and Chodavarapu in [KC08, page 327], a standard *SAML assertion* always consists of an *identifier* which makes it distinguishable, an *issuer* which says who is making the assertion and a *timestamp* which says when the assertion was issued.

Normally three standard statements are made within an assertion:

- **Authentication statement**: Asserting authentication results to indicate that the identity of a caller has been verified by the security service. For example after the security services has authenticated a user it can express who, how and when the authentication has been done.
- **Attribute statement**: Asserting user attributes. Additional security information can be passed, for example which roles a user has or which groups he belongs to.
- **Authorisation decision statement**: Asserting authorization decisions, as permissions on ressources. For example permission to modify data at a given ressource.

SAML can be used for a number of request/response protocols. For example the *Authentication Request Protocol* defines a `<AuthnRequest>` message which causes a `<Responose>` with assertions returned.

When SAML protocol messages are embedded into other protocols this is called a *Binding*. They can be the payload of a HTTP *POST* message. The

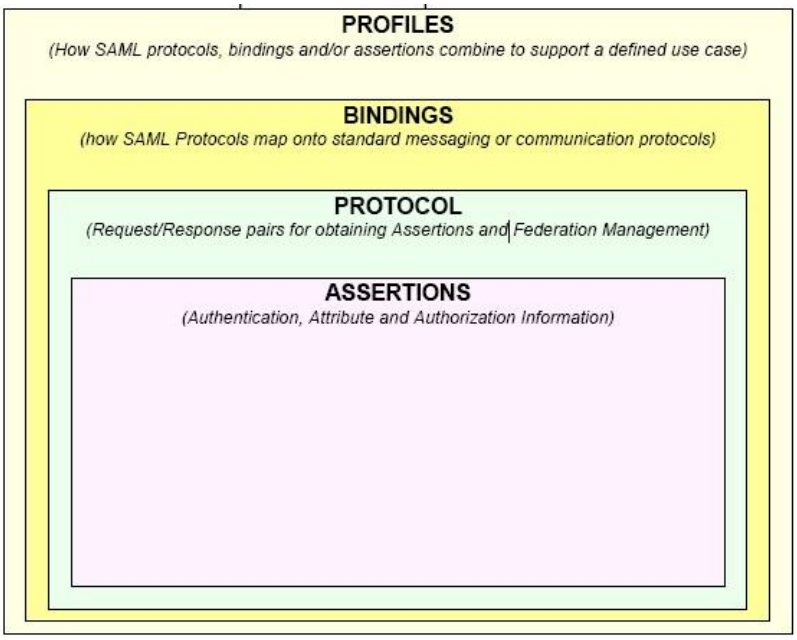

Figure 8.2: Basic components of SAML (picture taken from [HM05])

most interesting binding for a security service is the *SOAP-Binding*. A complete list of all SAML bindings is given by Hughes and Maler in [HM05]-

A *SAML Profile* describes how assertions, protocols and bindings are combined. For example it defines how the request/response query protocol uses a SOAP binding to obtain assertions.

Commonly the SAML is used in a binding over SOAP and are external. SAML messages do not secure any data in the body of a SOAP message. It is normally contained in the body itself.

8.4.2 WS-Security

The problem of *end to end security* arises in environments where there are intermediaries and no direct communication may be possible. Standard transport level security measures are insufficient because there may no direct connection from source to destination. Traditionally security was often implemented at the application layer. This causes difficulties with interoperability and management. Therefore it is better to implement security a the SOAP layer.

Web Services Security (WS-Security) is an enhancement of SOAP to provide confidentiality and integrity and fulfill the requirements stated in 8.2.1. Additionally WS-Security may be used to encode binary tokens which may act

as authorisation tokens.

WS-Security ensures message-level security, which means ensuring *confidentiality* and *integrity*. It also allows to make security claims, as for example *"Message signed by X."* or *"X is allowed to access ressource Y."*.

Cryptography is commonly used in systems where endpoints need to communicate in a secure way. However their intermediaries cannot be trusted. To provide integrity and confidentiality the common choice is to use a Public Key Infrastructure (PKI). *Digital signatures* and encryption can provide both requirements. WS-Security extends SOAP to provide these security features. According to the *WS-Security Specification* in [TC06, page 8] it was built with the following key driving requirements:

- *Multiple security token formats*
- *Multiple trust domains*
- *Multiple signature formats*
- *Multiple encryption technologies*
- *End-to-end message content security and not just transport-level security*

WS-Security does not require a special encryption technology or formats, it is a framework that enables the use of such technologies. With help of WS-Security it is possible to implement arbitrary security protocols.

WS-Security Structure

As seen in section 8.3.3, the SOAP header may contain additional information about the message enveloped. In WS-Security an additional header block named *wsse:Security* is added. A typical WS-Security header is depicted in figure 8.3. As we can see in the *wsse:UsernameToken* an encrypted username and password is transmitted.

UsernameToken is not the only token-type which may used in WS-Security. There are three other standard security tokens. *Binary Security Tokens* may have arbitrary non-XML formats as payload, which must be encoded. The encoding is specified in the *EncodingType* attribute, for example *Base64*.

Encryption

As mentioned above, WS-Security can transmit security critical data in encrypted form. They are represented by the *xenc:EncryptedData* element and

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Header>
      <wsse:Security ...>
         <wsse:UsernameToken wsu:Id="666">
            <wsse:Username>
               <xenc:EncryptedData>...</xenc:EncryptedData>
            </wsse:Username>
            <wsse:Password>
               <xenc:EncryptedData>...</xenc:EncryptedData>
            </wsse:Password>
         </wsse:UsernameToken>
   </soapenv:Header>
   <soapenv:Body>
      ...
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 8.3: *Example of a WS-Security header. Source: [KC08, page 97]*

should have an embedded encryption key or be referenced by an external encryption key, according to *OASIS Web Services Security TC* in [TC06]. The description of an PKI and basic encryption knowledge is beyond the scope of this work and assumed as known. *EncryptedData Tokens* hold encrypted data and are replaced by their cleartext message while processing. In general, fragments of the WS-Security header can be encrypted as well as fragments of the body.

```
<xenc:EncryptedData
   xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
   Type="http://www.w3.org/2001/04/xmlenc#Element"
   Id="EncryptedData-26882784-0" >

   <xenc:EncryptionMethod Algorithm="..." />

   <xenc:CipherData>
      <xenc:CipherValue> ... </xenc:CipherValue>
   </xenc:CipherData>
</xenc:EncryptedData>
```

Figure 8.4: *Example of an encrypted element. Source: [KC08, page 97]*

An example of an encrypted XML fragment is shown in figure 8.4 and the interesting parts will be pointed out below.

The attribute *Type* specifies which XML fragment the encrypted element substitutes. Here it is a full XML element. *xenc:EncryptionMethod* specifies which type of encryption is used here. The encrypted payload can be found in *CipherData*. There are two possibilities to provide the encrypted bytes according to Kanneganti and Chodavarapu in [KC08, page 240]: *CipherValue* provides the encrypted bytes directly base64 encoded and *CipherReference* may provide a reference to an external location to fetch the encrypted bytes from.

Digital Signatures

```
<ds:Signature>
   <ds:SignedInfo>
     <ds:CanonicalizationMethod Algorithm="..." />
      <ds:SignatureMethod Algorithm="..." />
      <ds:Reference URI="...">
         <ds:Transforms>
            <ds:Transform Algorithm="..." />
         </ds:Transform>
         <ds:DigestMethod Algorithm="..." />

         <ds:DigestValue>...</ds:DigestValue>
      </ds:Reference>
   </ds:SignedInfo>

   <ds:SignatureValue>
...
   </ds:SignatureValue>

   <ds:KeyInfo>
...
   </ds:KeyInfo>
```

Figure 8.5: *Example of an encrypted element. Source: [KC08, page 280]*

Encryption alone is not sufficient for message level security, because even if an attacker is not able to read the encrypted data, he may still try to tamper it. Another attack might be to replay the encrypted message, for example if the attacker guesses that it might be an authentication statement. Even if he cannot decrypt the message, he may still use it to authentify himself. This is called *replay attack*. To mitigate such attacks the receiver should be able to know if the message was changed during transmission. For this purpose *digital certificates* are used. The holder of a digital certificate can sign a message with his private key and the receiver can verify it via the public key.

XML documents like SOAP messages cannot be signed as a whole easily. Bytewise different messages may be syntactically the same, because an XML parser ignores whitespaces between the different elements. Therefore the XML message has to be cannonicalised before signing or the signature needs to be embedded in the XML message itself. WS-Security has a mechanism for signing individual elements of an XML element through a *Signature element*. The XML signature element consists of three child elements and is shown in figure 8.5. The *ds:SignedInfo* element contains the digest of the referenced signed XML element and tells about the used algorithms and necessary cannonicalisation (in case the format of referred element was changed during transmission). It contains the actual signing information, basically a receipe about what computations and transformations are necessary to verify the protected data.

The *ds:KeyInfo* element contains information about the key which can verify the signature. For example a reference to the actual key.

The *ds:SignatureValue* element contains the encrypted digest of what is signed, i.e. of the *ds:SignedInfo* element.

8.5 Conclusion

In an SOA environment traditional security measures do not work well. This comes naturally because often an SOA is designed for enterprise sized applications and operates in a distributed way with many services. Therefore security measures like TLS do not work, because there may be several intermediaries which are untrusted. Also concerns like authorisation and authentification need to be met.

An approach which comes natural to an SOA is the creation of a *security service*. It manages the security operations, makes security assertions, issues tokens and does authentification. In SOAP based services the most relevant technologies which can be used to implement such a service are the *SAML Protocol* and *WS-Security*. The security service can express security assertions and other findings in the SAML. Additionally the SAML messages need to be transferred to the consumer in a secure way. This can be achieved by the use of WS-Security, which provides message level security by encryption.

Acronyms

SOA Service Oriented Architecture

OSI Model Open Systems Interconnection Basic Reference Model

PKI Public Key Infrastructure

OASIS Organization for the Advancement of Structured Information Standards

LDAP Lightweight Directory Access Protocol

SOAP formerly known as Simple Object Access Protocol

FTP File Transfer Protocol

HTTP HyperText Transfer Protocol

XML eXtensible Markup Language

SAML Security Assertion Markup Language

TLS Transport Level Security

WS-Security Web Services Security

SSL Secure Sockets Layer

PKI Public Key Infrastructure

Bibliography

[fSidI06] Bundesamt für Sicherheit in der Informationstechnik. *Leitfaden IT-Sicherheit*. Bundesamt für Sicherheit in der Informationstechnik, 2006.

[HM05] J. Hughes and E. Maler. Security Assertion Markup Language (SAML) V2. 0 Technical Overview. *OASIS SSTC Working Draft sstc-saml-tech-overview-2.0-draft-08, September*, 2005.

[KC08] R. Kanneganti and P. Chodavarapu. Soa Security. 2008.

[Mil06] George A. Miller. Wordnet - a lexical database for the english language. 2006.

[MLM+06] M. MacKenzie, K. Laskey, F. McCabe, P. Brown, and R. Metz. Reference Model for Service Oriented Architecture. *OASIS Committee Draft*, 1, 2006.

[Nat03] Y. Natis. Service-Oriented Architecture Scenario. *Gartner Research Note AV-19-6751*, 2003.

[TC06] OASIS Web Services Security TC. Ws-security specification. URL: www.oasis-open.org/specs/index.php#wssv1.1, 2006.

[WHB+04] S. Wilkes, J. Harby, W.P. BEA, P.P. BEA, S. Jones, H. Grant, C. Peltz, S. Metzger, B. Rotibi, A. O'Toole, et al. SOA Blueprints Concepts. *Draft v0*, 5, 2004.

Chapter 9

A Research Agenda for Testing SOA-Based Systems

Robert Morys

Contents

Abstract: Oriented Architecture is a paradigm that organizes distributed IT resources, which create a solution to a business problem. The most common misconception is that they can be tested the same way traditional systems are tested. That is why many project managers spend too little time in the SOA testing process, which often ends in an unstable and insecure architecture. This paper will describe the arising challenges connected to testing SOA systems and work out a possible research agenda for testing them. It will focus on important areas like SOA governance, web services, traditional testing techniques and security testing.

9.1 Introduction

Service-oriented architectures (SOAs) become increasingly popular these days. They are architectural styles for enterprise systems that promise many desirable advantages like cost-efficiency, flexibility, adaptability and many more. An efficient SOA system can be one of the key-factors for the business success of an organization. While IT-systems always tend to get bigger and more complex, the trend goes towards splitting them into smaller parts so that they result in one big distributed system. That is why maintenance and testing of those systems gets more and more difficult and challenging. Especially the area of SOA testing has many left research potential and open questions.
This paper will at first describe service-oriented architectures and discuss the main challenges connected to testing such a system. Then it will give an overview about how to apply traditional testing techniques to an SOA system. After that it will describe important areas of SOAs like SOA governance and web services and give an idea on how to test them. In the end the main mistakes, when testing SOA security and user acceptance, will be outlined and the risk based testing approach will be introduced.

9.2 Service-Oriented Architecture

9.2.1 The most important facts about SOAs

Just about the mid 1990s the term Service-Oriented Architecture arose for the first time. It is not easy to find out who actually invented the term SOA, but Roy Schulte at Gartner says that Alexander Pasik, a former analyst at Gartner, coined the term SOA for a class on middleware that he was teaching in 1994 [Jos07]. SOA is a strategy, which designs and organizes IT-systems in a way that they get more adaptable, cost-efficient and agile. One of the main purposes of these architectures is that the organisation which uses them does not have to make their business decisions depending on the given technology, but gets those supported by them.
The SOA strategy is based on three main elements namely services, the

Enterprise-Service-Bus and policies [Jos07].
SOA services are single independently usable business tasks, like opening an account in a Bank or checking the product availability in a mail-order company. These services can be put together to achieve a more extensive business process like an order by a customer in a mail-order company, which includes the former named availability check as a part-service. All services are independent from a certain platform and technology and have a public interface which is accessible through a network. Besides they have to be registered in a catalogue which describes what the service does and how to interact with it. So there are service provider which execute business logic, service consumer which use the provided services, and service repositories where all services are registered [PT08]. The different types of services can interact with each other in a special way. As figure 1.1 shows, the service provider first publishes its interface and access information in the service registry so that the consumer can easily check out the available services and then send the desired request to the found service provider.

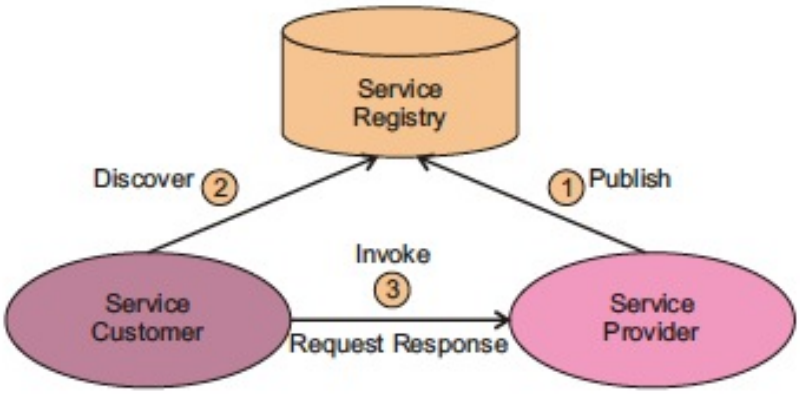

Figure 9.1: Service types

The Enterprise-Service-Bus (ESB) is the infrastructure of the system which allows easy combination and reassembling of services in order to accomplish changing requirements without any interruptions. It can connect many different technologies like web services, batch files or package applications so that the organization which uses the SOA does not have to worry about any technology limitations. The ESB does not only have IT benefits like easier implementation of new services and simple reuse of existing resources, but also business benefits like better customer service, cost reduction and a better response to changing business conditions.
Because of its highly distributed and dynamic architecture it is important to have policies which regulate and also restrict the services. More precisely SOA policies are rules, which a defined set of services has to follow, like ”all externally-accessible services must use HTTP”. There are different types

of policies like schema, communication and behaviour policies, which affect the services during design and run time. The schema policies for example are rules that affect the interaction schema between service consumer and provider, like ”platform specific data structures cannot be used in the schema”. Behavior policies directly influence the behavior of the service independent from the consumers message, like ”requests from gold customers must be given priority over other requests”[Foo].

9.2.2 SOA in comparison to Distributed Computing

Some properties of an SOA, like the hardware or the basic set-up, are similar to the the technology of distributed computing, which was invented nearly 30 years ago. Distributed computing means splitting a program into smaller parts and distributing it to many different computers which communicate over a network. The different computers calculate their part-solution and then put them together in order to achieve a solution for the whole problem. This strategy is mostly used for programs which need very much computing power and so cannot be solved by a single computer.
SOA adapted from the distributed computing strategy but there are also some important differences between these two techniques. The main difference is that SOA systems are used to support the business tasks whereas the business component does not matter in distributed computing. Also the main SOA technologies like web services, XML, SOAP and others are not used in distributed systems.
Besides in distributed computing the interaction between components is specified during design time and then implemented in the programming code whereas in SOA the services are much more flexible because of its structure and underlaying idea. Another difference is the way the information are exchanged. Distributed systems send information by calling certain methods, which then will send the desired data to its destination. In the SOA system services exchange data between themselves by sending self-sufficient messages, which include all necessary information like meta-information, policies and processing instructions.[PT08]
So all in all the SOA promises many desirable and efficient properties, but because of the highly distributed nature, the many different used technologies and some other properties, it is very difficult to test them.

9.3 Testing a Service-Oriented Architecture

9.3.1 SOA testing challenges

The area of testing software systems has always been and still is a very challenging topic in computer science. Especially bigger projects still pose many problems for scientists and engineers whereas the special features and properties of an SOA make it even harder to test such a system. There are many challenges like for example the lack of a graphical user interface or the fact that the services are built and tested without any information about where or by whom they will be used. So the testers cannot expect a single project which runs on a single server in a standardized browser interface. This is why they have to think of many different scenarios and interaction possibilities between technologies and services.
Besides the SOA services can be for example encoded as XML messages and therefore only interact with other applications but not with the end user. That is why testers cannot use the traditional approach of functional validation with a graphical user interface. They rather have to imagine the end users environment and recreate it in order to make testing possible. To accomplish all these challenges the testers have to use already existing techniques and modify them or even create new ones.

9.3.2 Traditional software testing techniques

Traditional testing strategies are mostly applied at the integration level which is the phase where the individual software modules are put together and tested as a group. In SOA systems the main components are the individual services and their quality defines the quality of the whole SOA, because if they have bugs and cannot be reused then the main concepts of reusability, flexibility and agility of the SOA cannot be accomplished. That is why SOA systems should much rather focus on testing at the service level than on the system level.
Besides SOA testing should also pay attention to the business logic which takes care of the information exchange between the user interface and the database. Its main tasks are coordinating the application, making calculations, processing commands and moving data between the user interface and database. Some traditional testing techniques like black-box or white-box testing could be modified to be applicable to SOA systems too.
Black box testing is a method which tests the functionality of the system without any information about the internal structure. Therefore the testers apply valid and invalid inputs and then determine the correct output. On the other hand white box testing is the exact opposite to black box testing and designs the test cases based on knowledge about the internal structure of the system. Therefore the tester chooses test case inputs to exercise paths through the

code and then determines the appropriate outputs. Both methods are mainly performed at the integration level. So in order to apply these techniques to SOA-based systems black box testing must contain XML support and use a web service language like WSDL. White box testing needs the source code of the system which would be the code of the services in an SOA system. But because of the fact that it is not available for the testers they have to analyze the XML documents which contain the information about the services.

9.4 Testing SOA governance

9.4.1 SOA governance

Another important area of testing is the SOA governance. SOA governance is defined as ensuring and validating that assets and artifacts within the architecture are acting as expected and maintaining a certain level of quality [KP07]. It is a special form of IT-governance and belongs to one of the responsibilities of the management department. IT-governance is an important part of the business management with the task to ensure that the business strategies and goals get supported by the IT department. Most organizations have a certain level of IT-governance because it is necessary to create the connection between business and IT processes by creating policies for people, infrastructures and processes.
SOA-governance is very important as without it the whole system would end in chaos. SOA-governance has mostly the same goals as IT-governance but extends it with service-oriented aspects and some minor changes. SOA-governance for example sets its focus on effective IT implementations in order to achieve adaptability and agility and does not care about who is allowed to monitor, define and authorize changes to the SOA. It not only creates a connection between the IT and business departments but also requires a much closer collaboration between these two.
If an organization chooses to use SOA they want to be sure to achieve continuity of already existing business processes, more efficiency and cost reduction. But it is also possible that third party services are invoked so that the system has to be changed in order to achieve the named goals. That is why the governance has to manage those changes and support all SOA areas like its development, policies or processes.

9.4.2 Types of SOA governance

There are different types of SOA governance which all need to be tested in a different way. The most important can be described as follows [PT08]:

- The design time governance sets policies for the architectural style of the SOA and ensures that there is not just a bunch of unsorted services. It establishes policies for the design, development and deployment of services and also their supporting artifacts. This could be for example how services are created, their reusability or strategic design elements.

- The organizational governance deals with how the teams, projects and hardware is managed and structured like for example funding of services or communication and coordination between the different departments.

- The change time governance takes care of changes in customization, composition and configuration of services, which is very important as one of the main reasons why SOA systems are used is their flexibility. That is why change time of services has to be governed in order to create a system which can easily and quickly adapt to changing business requirements.

- Another important SOA governance is the run time governance which defines rules for deploying services and also operations after the service has been deployed, like trust establishment and Quality of Service.

9.4.3 Testing run time governance

Unfortunately there are very few governance testing tools out there so that no established and standard method is known. However there are some approaches like analyzing the Service Level Agreements (SLAs) in order to test the run time governance. SLAs are contracts between the service consumer and provider which specify what exactly the service should be able to do, its quality and costs. SLAs are benefits for the service provider as well as the service consumer. On the one hand the consumer gets a guarantee of an appropriate service quality whereas the provider on the other hand has satisfied customers and makes his Quality of Service comparable to services offered by other providers.
The SOA testers or testing tools can analyze these agreements and compare them to the actual performance of a service. They have to place many different inputs on the service to see if it holds its quality promises or if scenarios exist which the consumer and provider did not think of. So the SLAs are an important part of an SOA and offer a good basement for testing it.

9.5 Testing web services

9.5.1 Web services

The web service technology is one of the most important implementation standard for SOAs. One common and abstract definition for a web service is the following:

A software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols [ABFG04].

One big advantage of web services can already be seen in this definition. They are not committed to a specific size so that they can reach from a small web script to a large server-spanning software system. Besides they are easy to understand and discover because of the use of XML. The XML documents are an accepted standard. They are text only files and if well designed, written in a manner that humans can easily interpret. Web services can interact with many other software systems and offer cross-enterprise integration which also makes them very valuable for SOA systems. All in all one can also describe web services shortly as interfaces that describe operations that are network accessible through standardized XML messaging [PT08].

In order to understand how web services are defined, located, implemented and communicating one has to take a look at the web services protocol stack, which is shown in figure 1.2. Each level has different functionalities which can be described as follows:

- The lowest level contains the transport protocol which is responsible for transporting messages between the network applications. It includes the wide spread HTTP, SMTP and FTP protocols.
- The messaging layer encodes the messages in a common XML format so that both ends of the connection are able to understand them whereas for this purpose the most used protocol these days is SOAP.
- The description layer is the part that describes the functionalities of a web service. It is also the interface which the programmer can use to access the service and communicate with it. The mainly used interface for this purpose is WSDL.
- The quality of service layer is responsible for defining quality requirements of the service using for example the previous discussed SLAs.

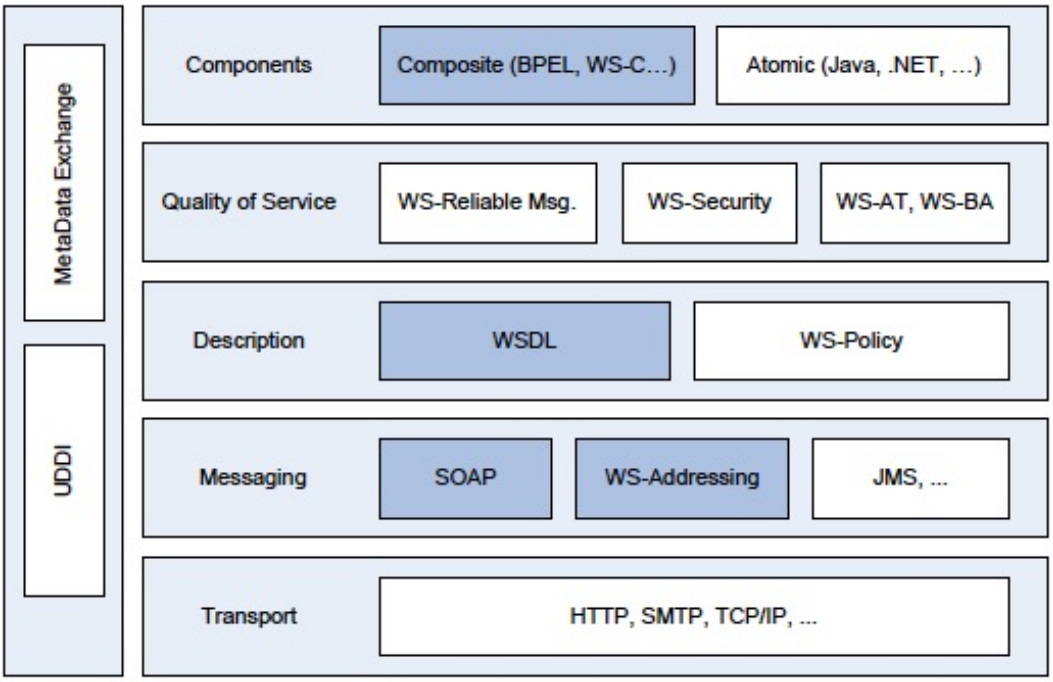

Figure 9.2: The web services architecture stack (inherited from [May06])

- Last but not least the top level has the task to compose the services into business processes. It creates processes by defining the order in which services will be invoked. One language which accomplishes this task very easily is BPEL, which was invented by IBM, BEA and Microsoft by merging their different already existing standards.

The next section describes the most important standards in more detail and illustrates how they should be tested.

9.5.2 Web service standards

The three core elements of the web service technology that have prevailed are WSDL, UDDI, and SOAP. WSDL and SOAP are as already mentioned located on the description resp. messaging layer. UDDI on the other side is a vertical layer of the web service protocol stack which means that it includes multiple components which are description, messaging and transport.
UDDI stands for ”universal description, discovery and integration” and its main purpose is to list the available services and to define how they interact with each other. Business organizations can sign in the UDDI registry and publish their services. It is based on standard technologies such as DNS, HTTP, and XML and can be interrogated by SOAP messages and provides access

to WSDL documents. Because of the fact that technologies like DNS or HTTP were not used in traditional software systems, many testers do not have the necessary knowledge about testing them. But in order to test an SOA system properly, it is inevitable to be informed about these technologies and how to test them.
WSDL stands for ”web service description language” and represents a XML file which contains all necessary information about a web service and its interaction possibilities. Its advantage is the standardized format which allows a more consistent communication and interaction between services. The problems considering testing WSDL documents are that different testers can interpret the document in a different way and the input and output are not specified in detail. That is why many scenarios and conditions have to be considered when testing them.
The services listed in the WSDL document can be called by using the SOAP messaging standard. SOAP enables data transfer between different systems using XML to present the data and internet protocols. It easily constructs requests and responses between services using FTP, HTTP or SMTP as the transport protocol. Several properties of SOAP messages have to be tested including the format or the header, which contains information about intermediaries, other actors and recipients. Besides also the functionality and security of the intermediaries specified for example in the SLA contract have to be tested in a proper way [PT08].

All in all testing web services is a topic that still has many open research issues and will need more attention in order to create effective and standardized testing tools or procedures. But there are also some other important SOA properties that need to be tested, which will be discussed in the next chapter.

9.6 Testing other SOA properties

9.6.1 Load testing

Some properties of a system, like load, performance and security, are important testing areas for traditional systems, but require even more testing emphasis in an SOA structure.
The goal of load testing is to get the response time of the system and to see how it works under extreme conditions like many users accessing a web service or very large documents being processed by a word processor. The most important parameters when load testing web services are for example the response time, the number of passed or failed transactions and the memory or CPU utilization. In SOA systems the components that could be critical in relation to load are the services, because they can be used by different applications at the same time. The problem is again the missing graphical user

interface of the services so that traditional load testing tools cannot be used in this case. When testing services the testers have to emulate the behavior of service consumers who will exchange messages with the service provider.

9.6.2 User acceptance testing

Many software development organizations just include their users respectively key business stakeholders in the beginning of the project to define the business requirements and in the end in order to test the implementation for bugs or missing requirements [Har07]. But nowadays it is already known that this is not the best way to go. With this approach the testing phase can take very long and users could find very important missing features or fatal defects, which would mean a bunch of additional work for the development team.
A better approach is to include the users throughout the whole project life cycle and to be in continual agreement with them. This saves a lot of work and will in the end produce a much better project, because the developer team does not have to make big changes to their project with every new requirement and the software perfectly fits to the users wishes. Of course this not only holds for traditional software systems but also for SOAs and has to be adhered to when testing them.

9.6.3 Security testing

Security is about making software behave in the presence of a malicious attack, even though in the real world, software failures usually happen spontaneously [PM04]. Besides functionality, the security is one of the most important properties of every system.
In SOA the necessity for security testing is even bigger than in traditional systems. That is mainly because of its distributed nature. With every new service can also come a new security gap, so that after including new services there also has to be a security test. Besides sensitive data is transferred in a big internal and external network, databases are updated by several sources and services are located in many different parts of the network. So SOAs offer many potential security gaps which have to be closed and tested. Like in user acceptance testing, many organizations perform security testing only in the end of the projects life cycle. This again is not the best solution because in this way heavy security bugs are found very late and the security design of the whole project suffers.
That is why the developers have to include security analysis and testing into the entire life cycle [Har07]. This starts with defining security requirements in the beginning and performing security risk assessment during the design phase. However the most important tests have to be executed already on a

service level and not only once the whole system is completed. There are many different security testing tools out there which can find potential security weaknesses by performing for example a penetration test which is a technique that tries to break the security of the system by attacking it with worms or as a hacker.

Tools that direct such a security, load or user acceptance test are very useful and desirable because they save a lot of time and work. There are many tools for traditional software systems which quickly and repeatedly test them based on a record and playback strategy. But because of the special structure and technologies used in SOA systems, most of the traditional tools can not be used for those. However there are already some approaches and rules that can mostly direct such tests successfully.

9.6.4 Regression testing

A software regression is a software bug which causes a properly working feature to stop functioning after a change to the software system has been done. These changes can be for example a system upgrade or patch. There are different types of software regressions which all have different effects. A change can for example create a bug in the changed component or module so that there just exists a local regression. But changes can also create bugs in other remote components or unmask already existing bugs that did not have an effect before.
Software regression also plays an important role in SOA systems. If they are well designed they promise flexibility and easy adjustability to changing business needs. That is why changes to services should not pose any problems and are often made. These changes can be for example fixes to a defect or service adjustments due to changing business requirements. Already little changes to a service can have a significant impact on the entire SOA system. Typical software systems with N modules for example have N2 dependencies [Yun]. So every time changes are made to a service the testers have to examine its dependencies and previous tasks. They have to retest the modified service in order to check if it still accomplishes all its tasks without any errors. Besides they have to check if all services, which are dependent on the modified one, still function properly.
This type of testing is called regression testing and can be shortly described as identifying unintentional errors or bugs that may have been introduced as a result of changing a program module [Yun].
Organizations should spend time and money in this type of testing in order to ensure that all services work properly and no unexpected SOA system defects arise. There are already some automated regression testing tools on the market which have proven to work effectively and can save much work and time.

9.6.5 Risk based testing

Because of the fact that the SOA systems always gets bigger during their life time and services are reused by more business processes, there exists an infinite number of test cases. That is why the organization has to set priorities for the test scenarios.
A common approach to do that is risk based testing. It means that testing is prioritized based on the probability that some feature of the system fails, whereas the likelihood of failure itself is dependent on the frequency of use [Mil]. There could be for example seldom used functions that are highly critical or on the other hand safe features which are used every day. Good examples are the eject pilot operation in a fighter aircraft which is essential for the survival of the pilot but only used very rarely. A simple deposit check in a bank is used by millions every day but poses hardly any risks.
Organizations have to spend much emphasis in testing features that are used very often, because they are important and fundamental for their business, but they also have to set a focus on rarely used functions that could have fatal or dangerous consequences for the organization or customer. The challenge is to find the right balance of time and money that is spent in testing those different types of features.
Risk based testing can help to find this balance and has important benefits like reduction of costumer reported problems, maintenance costs and test phase length by a factor of approximately two.

9.7 Conclusion

The paper has shown that an SOA system cannot just be tested. At first the testers have to divide the whole structure into single domains like governance, security, services and so on, and then test each one of them separately. After that they also have to check if all the domains work together properly. Therefore the testers do not only have to understand the whole SOA system itself but also the underlying technologies of every domain. That is one reason why SOA testing is very challenging and comprehensive.
The paper has also worked out some approaches on how to successfully test an SOA system including the most important properties and technologies of SOAs and testing techniques. The most important approaches can be summarized like this:

- A test team has to be involved right from the beginning and the test design has to be specified with the business requirements and technical design.
- Traditional testing techniques like black box and white box testing can be used to test an SOA system, but they need modification and a different

mindset.

- The testers have to know much about SOA systems and their underlying technologies like for example web services.
- Strong governance and policies are needed to create an efficient and successful SOA. SLAs for example can be used to test runtime governance, but all in all this area of SOAs still needs much research to be done.
- Sometimes the structure or functionality of SOA services gets changed. Regression testing is a good approach to ensure that modified services still work properly and do not create new defects in other components of the SOA system.
- Security and user acceptance testing need to be done throughout the whole projects life cycle. The business key stakeholders or users have to be involved not only at the beginning and the end but also throughout the whole time.
- A risk based testing approach prioritizes test cases and saves much testing time and maintenance costs.
- There are some helpful and good testing tools on the market and organizations should not hesitate to invest in them.

So all in all SOA testing is important to guarantee an effectively working software system which supports all business decisions and processes. Unexpected defects can be avoided if the organization spends enough time and money into a testing team and testing tools.

Bibliography

[ABFG04] Daniel Austin, Abbie Barbir, Christopher Ferris, and Sharad Garg. Web services architecture requirements. http://www.w3.org/TR/wsa-reqs/, February 2004.

[Foo] Dan Foody. Clearing up the confusion of the term ”policy” - soa zone. http://soa-zone.com/index.php?/archives/18-Clearing-up-the-confusion-of-the-term-policy.html.

[Har07] Harris. Soa test methodology. *T. Business Solutions*, 2007.

[Jos07] Nicolai M. Josuttis. *SOA in Practice: The Art of Distributed System Design (In Practice)*. O'Reilly Media Inc., 2007.

[KP07] L Frank Kennedy and Derryl C Plummer. Magic quadrant for integrated soa governance. *Gartner Core Research Note G00153858*, December 2007.

[May06] P Mayer. Design and implementation of a framework for testing bpel compositions. Master's thesis, University of Hannover, 2006.

[Mil] James Miller. Risk-based testing. www.ucalgary.ca/ ageras/wshop/pres/f2003/jm-tw-001.pdf.

[PM04] B. Potter and G. McGraw. Software security testing. *Security and Privacy, IEEE*, 2(5):81–85, Sept.-Oct. 2004.

[PT08] T. Parveen and S. Tilley. A research agenda for testing soa-based systems. *Systems Conference, 2008 2nd Annual IEEE*, pages 1–6, April 2008.

[Yun] Mamoon Yunus. Codeproject: Intro to soa regression testing: A hands-on approach. free source code and programming help. http://www.softwaremag.com/trk.cfm?uid=33.

Chapter 10

Quality attributes for SOA

Eduard Renz

Contents

Abstract: In the past years the service orientated architecture (SOA) approach got more and more used and accepted. During the design and creation of such systems many things have to be considered. Different system users have different requirements to system attributes like security and performance. It is important to know and understand the different quality attributes for every party involved in such an SOA system. As these quality attributes can have impact on each other, it is also necessary to examine their interactions. Service level agreements (SLA) are used to ensure compliance with regulations that the service user and the service provider set up to guarantee a certain quality of service. A detailed understanding of all these attributes, their interaction and monitoring, as well as the understanding of the SLA is important for an SOA systems usage and success.

10.1 Introduction

A service orientated architecture (SOA) is a paradigm, which allows grouping of functionality around business processes and packaging these into services. These services communicate with each other by sending messages, can be combined to create new services and are business and not technology centric. Service orientation can be seen as another form of distributed computing and modular programming. Now that SOA systems got more mature and are used widely, it is important to understand and know the quality attributes like security, performance, dependability, testability, interoperability and extensibility of such systems. Many of these quality attributes are defined by specifications that were created by special organizations like the W3C and OASIS. This work gives an overview over the different quality attributes and their underlying specifications. An important aspect is the interaction between these quality attributes which will be presented in the second part of this work. This interaction often requires tradeoffs, so that the whole system can still operate and offer a certain level of quality. This quality can be often measured and is used to verify, if the service has met the required quality attributes. Service level agreements are used to summarize these quality requirements and will be presented in the third part of this work.

10.2 Quality Attributes

Since SOA system differ considerable from other architecture styles the different quality attributes are also altered. This section will describe the different quality attributes for SOA systems and give an insight on the underlying technology.

10.2.1 Interoperability

One of the biggest benefits of SOA systems is their ability to share specific information and to operate on it based on specific rules. To achieve this, the service implementation must be clearly abstracted from the interface, the service exposes to its environment. The interface definition focuses on how the service may fit into a larger business process. The underlying implementation is hidden from the service user which makes the service business process centric and not technology centric. This independence of technology is achieved by the usage of a call-return mechanism on which the interface is based. With this interface centric construction the language and platform used by the service user is not restricted and makes the integration between different platforms and languages possible.

To standardize the message based communication in SOA, two standards have been recommended by the W3C: the web services description language (WSDL) and SOAP.
WSDL is a XML-based language that provides a model for describing SOAs. SOAP is a communication protocol specification to exchange XML-based information and provides a basic framework to create web services. SOAP is often used together with WSDL and provides access to its functions. The combination of these two technologies is used to create web services. SOAP and WSDL are the base for SOA communication and can be extended by special purpose WS-* specifications to be able to adapt to situations that are not covered by SOAP and WSDL. The term WS-* refers to a set of web service specifications that are maintained by various organizations and are named with WS as their prefix.

However problems arise if features beyond WSDL and SOAP are needed or if distinct versions of this specifications are used. This decreases the interoperability and makes integration not seamless [Bas07]. To deal with such problems and to increase interoperability the Web Services Interoperability Organization (WS-I) was created. Its aim is to create and promote standards and to provide tools to verify standard conformance. For that purpose the WS-I publishes profiles which are descriptions of conventions and practices for the combination of web service standards. If service providers adhere to these standards then they can offer some level of assurance to their customer that a certain level of interoperability is achieved. The WS-I is an important mechanism to achieve and simplify interoperability.

The benefits that result from the interoperability of the SOA also bring some problems and concerns with them. A major concern is the dependability of the systems which will be elaborated next.

10.2.2 Dependability - Reliability and Availability

SOA platforms can only prevail if they keep operating over time without failure and are operational and accessible. Otherwise the service user may not be able to meet the functional requirements. This also means that the service provider will not get financial compensation and a negative interference on his reputation. To avoid such problems it is important that the SOA platform can provide continuity of correct service delivery on the following different levels:

- **Services provider level**

  To ensure that the service is operating correctly and is accessible, the service provider may want to add redundancy of computation and data. This approach can be combined with load sharing to gain performance and clustering to avoid a single point of failure [EM05]. For the sake of easier replication services should be stateless.

- **Service level**

  During failure and concurrent access it is important to preserve data integrity. In normal service implementations the transaction begins and ends with the service [Bas07]. Because SOA platforms can consist of the composition of different services the transaction management becomes more difficult and may require nested transactions. Taking into account the distributed structure of SOA creates the need for a distributed transaction model. A possible solution for these problems is the two phase commit which is used in the SOAP/WSDL extension module WS-Transaction. The WS-Transaction specification provides the well known transaction principle from databases for web services and offers different transaction types either for individual operations or for long running transactions [IBM04].

- **Transport layer level**

  This level offers some reliable transport protocols like HTTPR (Reliable HTTP), which is however not widely accepted. Another option is the usage of enterprise messaging infrastructure middleware like IBM WebSphereMQ or JMS. The problem with these solutions is that they are all proprietary and all points on the transport path must use the same technology.

- **SOAP message layer level**

  To solve the problem of unreliable communication channels or the wrong delivery of messages, the two specifications WS-Reliability and WS-ReliableMessaging have been approved as standards by the Organization for the Advancement of Structured Information Standards (OASIS). These specifications assure reliability through the following four assurances which can be combined [Bas07]: In-order delivery, at-least-once delivery, at-most-once delivery and exactly once delivery. This approach

makes the SOA platform responsible for providing reliability but holds the problem that the underlying messaging infrastructure must be able to understand the reliability information transported by the SOAP message header [EM05].

- **Business process layer level**

  At this level reliability is achieved through compensation-based transaction protocols and dependable composition of services through failure handling [EM05]. This can be accomplished by the usage of the lightweight reliable messaging framework wsBus. Additionally, emergency plans can be developed which will find alternative service providers if a service should become unavailable.

Looking at the problem more generally the following principles help to obtain dependability as proposed in [EM05]:

- Fault prevention through rigorous design and implementation.
- Fault removal through verification, validation and diagnosis.
- Fault tolerance based on correct service delivery even if faults should occur.
- Fault forecasting which predicts the presence and the consequences of faults.

With these principles and the technologies presented for the different layers, some level of dependability can be achieved. However new techniques are required that also take the dynamic binding and discovery of web services more into account.

After comparing reliable and unreliable web services messaging, the following disadvantages of dependability have been found:

- To achieve reliability, additional communication is needed which results in not only an increased size of the SOAP header but also in an increased message size.
- Also considerable performance issues have been observed making reliable systems slower by the factor 200-400 [Ant05].

These results show that dependability does have a quite big impact on the performance of SOA based system. More details on these issues are provided in the following section.

10.2.3 Performance and Scalability

The performance of a service is basically defined through response time, throughput and timeliness. This means that the service should respond quickly, deliver a specified amount of data and meet time deadlines which are especially important for real-time systems.

In SOA systems or services the performance is often negatively affected because of the following problems which are outlined in [Bas07]:

- SOA systems often include distributed computing whereas the service user and provider are often communicating over a network which increases response time and cannot guarantee deterministic latency. This makes SOA a bad choice for real-time applications that require timeliness.
- The dynamic binding ability of SOA systems has also a negative effect on performance because dynamic binding requires an extra call to locate the service which increases the total response time. One possible solution for this is to use caching which however has to be reestablished if a failure occurs.
- SOA platforms may also use stubs, skeletons, SOAP engines or proxies to perform data marshalling to be able to handle the communication between the service user and provider. These communication infrastructures create a performance overhead.
- Additionally, the usage of XML as the primary message language in SOA increases the message size by the factor of 10-20. The processing time, needed to process the XML message, is also increasing because it requires parsing, validation and transformation. These operations are extreamly CPU and memory intensive.

However the location transparency principle used in SOA also brings advantages concerning performance. It allows changing the service location without affecting the service user and can be used to create load balancing strategies to improve performance. This scalability allows SOA systems to adapt to volume or size changes without performance degeneration. This scaling can be horizontal through load balancing or vertical through increased server capacity. To simplify the scaling services should be stateless. Different service scope strategies have also influence on the performance. A new instance of the provider could be created for each request, for each new service user or once for all requests from all service users [Bas07].

Other possibilities to improve performance are caching in content distributed networks (CDN) or geographical distribution as described in [ILKR03]. The request type is also performance relevant and can create an overhead depending upon if static or dynamic data is requested.

Performance tests in SOA are often connected with high costs and are time consuming. An alternative to the native approach is to analyze the performance aspects with the help of mathematical models, which can be used to predict selected performance aspects. One approach is based on stochastic Petri nets which includes four basic constructs (sequence, concurrent, choice and loop) and allows calculating the expected delay time of complex service compositions. This is described more in detail in [ZYL08]. Another alternative is the simulation based analysis which can provide accurate performance properties without high costs. However the results may become totally wrong if the underlying model is wrong. The combination of all performance test practices is recommended to achieve accurate test results [Sea05].

As performance is a central quality attribute of SOA systems many factors have influence on it. One of these factors is the security of an SOA system. Security concerns and solutions are described in the next part.

10.2.4 Security and Auditability

The creation of secure SOA systems is not only important for its success but also very challenging. Security for SOA focuses especially on the following aspects as described in [GEP04] and [Bas07]:

- Confidentiality can be achieved by the usage of encryption and communication over secure channels. However encryption is not only required while sending but also while storing data. This can increase the data size.
- Authenticity includes access restriction based on the provided identity. On the other hand the identity of the external service provider has also to be confirmed. Because SOA systems may rely on data from a public directory the identity of the publisher and directory provider has also to be authenticated.
- Integrity can for example come through checksum tests.
- Availability from the point of view of security includes the prevention of Denial of Service attacks through redundant systems.
- The service must keep record about its usage to support financial or legal audits which is especially important in the financial and health sector and security relevant applications.

Security can also be examined from the perspective of different communication layers [PK04]: Transport layer security, message level security and message content security.

Methods to ensure security at these different layers are the usage of SSL, encryption, digital certificates or ticket systems like Kerberos. An additional approach is the usage of one of the different security models like WS-Authentificaton and WS-Privacy. These SOAP extensions provide message content integrity and confidentiality. Other security models like SAML or XACML allow services to exchange security information without the need to change existing security solutions or provide a language to specify role-based access control rules [Bas07]. These models are currently developing into standards. However these approaches are not able to protect the end-to-end communication. The problem is that the data is received and forwarded by intermediaries beyond the transport layer and data integrity or security information may be lost or may have been manipulated [GEP04].

Another concern is the flexibility of SOA systems which makes audits difficult. Dynamic binding and discovery make it difficult to track which services are actually used. Services that are compositions of other services have even more problems concerning audits. One way to achieve end-to-end auditability is to include meta data in each SOAP header so that it can be captured in audit logs. This however will only work if the service provider and the service user will use the same standards that allow audit [OBM05].

To verify the level of security special tests have to be run. SOA testing is described in the following section and will give an insight into the problems that come together with the SOA.

10.2.5 Testability

Software testing is an essential development phase for any kind of architecture including SOA. The degree to which a service facilitates the establishment of test criteria and the performance of tests to determinate if the test criteria have been met is referred as testability. Unlike as in traditional software development, testing in SOA systems is more complex and challenging. Because SOA software is used and not owned, the system elements are normally located on different machines which brings many problems as described in [Bas07]:

- The source code may not be available which makes it impossible to use the following testing strategies [CP06]:
  - White-box testing which is based on the code structure and data flow knowledge.
  - Mutation based testing which modifies the code and data to locate errors.
- There may be no access to log files or other output that is important for testing and debugging.

- The test cases can only be defined based on the published interface and documentation.
- If services are discovered at runtime it is almost impossible to predict which service will actually be used.
- The identification of the problem location is difficult because it could be everywhere: the provider, the user, the network, the discovery agent, the XML file and so on.
- Since SOA relies on a changing and dynamic environment the replication of errors is very difficult.

Looking at testability from different perspectives raises different problems and concerns as outlined in [CP06]:

- The service developer and provider try to handle exceptions properly. However they are not able to do nonfunctional testing because it is impossible for them to take the customers infrastructure or network configuration into account. One key issue for the service provider is to build confidence that the service will deliver a function with the expected Quality of Service (QoS). To increase testability the service provider needs to build special services that support testing and debugging.
- The service user tests to gain confidence that the service meets the assumptions that were made at design time. If dynamic binding is used the testing task gets more complex. Testing creates not only costs for the service user but also wastes the service provider resources.
- The 3rd party certifier assesses a services fault-proneness. This assess reduces the testing activities from the provider and so saves him resources. The certifier has, just like the service provider, the same problem that he cannot take the service users infrastructure and network configuration into account which makes the certified confidence level untrustworthy.

Even though testing possibilities are restricted it is definitely not impossible to test SOA systems as illustrated in [CP06].

- Test cases can be created that are based on black-box tests.
- The input can be mutated based on some testing strategies.
- Preconditions and postconditions based tests can be used to create test oracles.
- Stress testing with search-based techniques like generic algorithms that will likely generate errors and lower the QoS.

- Regression testing retests a system after changes have been made to ensure that the change doesn't affect the system.

Even though this methods offer some testing possibilities they still need to be improved. Especially the number of possible end-points for testing should be reduced. Massive testing can create DoS like conditions for the service provider and costs him resources and bandwidth. The QoS of a service composition depends on the combination of the actual bindings which makes test case generation for this complex structure more difficult and expensive. Additionally, the QoS is unpredictable and may vary over time because of different circumstances which doesn't necessarily reflect the real QoS.

One approach to reduce testing costs is to use service monitoring which helps to reports failures, verifies that the service meets pre- and postconditions and is able to trigger recovery if necessary.

10.2.6 Extensibility, Adaptability and Modifiability

Business requirements often change and evolve to adapt to the market which requires changes in the SOA service as well. The ease with which the service can be changed and adapted to new requirements is denoted as adaptability and the ease with which the service capabilities can be extended without affecting other parts of the system is denoted as extensibility [OBM05].

SOA systems have some benefits concerning adaptability and extensibility which include the principle of location and transport independence. This allows not only changing the system location more easily but also changing the used implementation language without major adjustments. Changes in the service architecture should not have influence on the service usage, the service discovery and its dynamic binding. This can be achieved by extending the service architecture to add additional services or extending the service functionality without changing the service interface. A major problem here is the extensibility of the used messages. The messages need to be restricted to allow easier parsing. On the other hand if the messages are too restricted then they limit the growing and changing capabilities of the interface which leads to reduced extensibility. Tradeoffs between restrictions and extensibility are needed to achieve the right balance.

Another aspect is that it should be possible to add or swap services without making major changes to the underlying application. This can often be achieved quickly and cost-effectively because SOA services are loosely coupled, self-contained, modular and accessed via interfaces [OBM05]. Since services are used in different environments and platforms it is essential that the service is configurable to the environment in which it resides. This requires spiral development with incremental deliveries to different platforms and back-

wards compatibility to previous releases [OBM05].

A key success factor to achieve adaptability is the management and monitoring of services and their underlying infrastructure which includes measurements of capacity, performance and availability.

10.3 Interaction

The different quality attributes that were described in the previous section do have influence on each other. These influence and the interaction between the different quality attributes is examined next.

Reliability

To achieve reliability often the usage of special messaging systems is needed as described in 10.2.2. These systems however decrease interoperability because bridges are needed that are based on a fixed set of interactions that cannot be easily changed. But not only flexibility suffers from increased reliability but also the performance. Systems with reliability are about 200-400 times slower than systems without reliable messaging.

Performance

SOA possibilities like dynamic binding and service discovery at runtime does not only have benefits. One major disadvantage connected with dynamic binding is a performance loss. It is possible to hard code the addresses for the service location to increase performance but this would decrease flexibility. A partial solution to this problem is the usage of caching and binding renewal on failure. Another aspect that concerns flexibility, interoperability and performance is the needed data marshalling to handle the communication between the service user and provider. If the data structure is too limited this will decrease the systems flexibility. However flexible data structures make parsing and processing difficult and performance intensive. Tradeoffs have to be made to gain a flexible high performance system. However interoperability can also have a positive effect on performance. It is possible to create load balancing strategies to improve performance which is generally also coupled with availability.

Security

An important security aspect is confidentiality which can be achieved through encryption. However this will result in increased message size which will have a negative impact on performance. Additionally, the usage of a security model will have a negative influence on interoperability and modifiability. This also holds for the usage of transport layer security methods like SSL. Availability is not only generally important but also to create secure systems. The needed availability and security can be achieved through redundant systems.

Testing

As testing possibilities for SOA systems are limited, the service user may want to use stress testing and other voluminous testing methods. This however can affect the performance and the availability of the SOA system and therefore has to be done very carefully.

Extensibility

A key aspect to achieve higher extensibility is the flexibility of the message structure. However the more flexible the message structure the more complex is it to parse and process this message. This complex parsing has a negative influence on performance.

Auditability

As already partly described in 10.2.4, are audits difficult to accomplish for SOA systems. The more complex a systems gets, for example through system composition, the more complex the audition gets. Hence, flexibility makes audits difficult.

Summary

Most of the quality attributes do have a considerable influce on the systems performance which makes it a key aspect to consider when using an SOA. Also many quality attributes do have a negative impact on interoperability which makes it not only a big benefit but also a problem.

10.4 Service Level Agreements

The nature of SOA allows dynamic binding to a service. This means to enter and dismiss a business relation with a service provider on a case-by-case basis and on-demand. To ensure that the used service meets a certain QoS, guarantees and obligations of both parts are needed. These are expressed in a service level agreement (SLA). Nowadays SLAs are used in a many areas including hosting, communication services, help desks, problem resolution and so on. Many SLAs are formulated in plain natural language which makes monitoring them slow and expensive. The usage of SLA templates, which include several automatic processed fields did decreases costs but was limited concerning flexibility. Therefore a flexible SLA specification and monitoring framework was created: the Web Service Level Agreement (WSLA) which will be described below based on [Lud03].

To ensure that the needs of the service provider can be modeled and monitored properly the WSLA allows the definition of resource metrics and composite metrics. On the other hand SLA parameters and business metrics can be defined which are important for the service user and meet his needs. These resource metrics use the concept of measurement directives, which contain the information needed to retrieve the metric. Some metrics have been presented in the previous chapter. Composite metrics are created by combining several resources and can be specified through functions. In order to put the metrics into the context of a specific customer, SLA parameters are used, which provide high/low watermarks to allow checking if the metrics meet/exceed/fall below the defined values. Finally the business metrics relate the SLA parameters to business terms, like finances, specific to the service user. Other important properties of the WSLA framework are its flexibility so that it can be applied to a wide range of SLA, its possibility to be integrated into electronic commerce systems and the possibility to delegate monitoring tasks to third parties.

The WSLA life cycle consists of 5 stages:

- Stage 1 - SLA negotiation and establishment: This stage includes the negotiations and signing of the SLA by both the service user and the service provider. Values like price and metrics are defined and combined into one single SLA.

- Stage 2 - SLA deployment and submission: The deployment service is responsible for checking the validity of the SLA and its submission to the involved parties.

- Stage 3 - Service level measurement and reporting: Information based on metrics is collected and evaluated through a special service. If the measured parameters should violate the SLA the according services are informed.

- Stage 4 - Corrective management actions: If SLA violations are reported, the management service tries to act in order to solve the problem. Every action of the management service needs to be verified by the business entity which can be an ERP system. This process is very company specific.

- Stage 5 - SLA termination: The conditions under which the SLA is terminated are specified which may include penalties for breaking SLA clauses. Also an expiration date can be specified.

The WSLA language itself is based on XML schema which allows modeling the earlier defined properties. By using the WSLA framework SLAs are described in a way that allows automatic monitoring which saves costs and reduces error-prone manual interventions. It allows defining exactly how metrics are measured and gives the opportunity to include third parties into the process of measuring and monitoring. Other attributes like extensibility, flexibility and the independence from specific interfaces makes the WSLA language applicable to a wide range of scenarios.

However this specification does also have disadvantages like its need for a procedural interpreter for its execution. Also the rules are based on a simple Boolean logic which makes modeling complex SLA rules difficult. If the specification needs to be extended it will require extending the interpreter as well. Even with these disadvantages the WSLA language is an important instrument for SLAs and their monitoring.

An alternative to WSLA is the WS-Agreement specification which was created by the Global Grid Forum in 2005.

10.5 Conclusion

This work introduced some different quality attributes and their specifications. Problems and benefits have been pointed out. This was however a superficial introduction to the topic which is very widespread. The second part of this work showed the different interactions between the quality attributes and gave some leads about what has to be considered concerning quality attribute interaction. Especially performance is a quality attribute which is negatively influenced by many other quality attributes. The last part of this paper showed how SLAs are structured and introduced the WSLA framework. Its usage automates many tasks and helps to monitor the quality of service that was defined in a SLA. Further research is needed especially in the area of quality attribute interaction to be able to understand the influence each attribute has on other attributes.

Bibliography

[Ant05] John Anthony. The cost of reliability: A comparison of reliable and unreliable web service messaging. *21st CS Seminar*, 2005.

[Bas07] Liam O'Brien & Paulo Merson & Len Bass. Quality attributes for service-oriented architectures. In *Systems Development in SOA Environments*, page 3. IEEE Computer Society, 2007.

[CP06] G. Canfora and M. Di Penta. Testing services and service-centric systems: challenges and opportunities. *IT Pro.*, 8(2):10–17, 2006.

[EM05] A. Erradi and P. Maheshwari. A broker-based approach for improving web services reliability. *IEEE*, pages 355–362, July 2005.

[GEP04] Carlos Gutiérrez, Fernández-Medina, Eduardo, and Mario Piattini. Web services security: Is the problem solved? *Information Security Journal: A Global Perspective*, (3):22–31, 2004.

[IBM04] Microsoft et al. IBM. Web services transactions specifications, 2004.

[ILKR03] Iyengar, Ludwig, King, and Rouvellou. Performance and service level considerations for distributed web applications. *In Proc. of the 7th World Multiconference on Systems*, 2003.

[Lud03] Alexander Keller & Heiko Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11:57–81, 2003.

[OBM05] Liam O'Brien, Len Bass, and Paulo Merson. Quality attributes and Service-Oriented architectures, 2005.

[PK04] J Chapman et al. P Kearney. An overview of web services security. *BT Technology Journal*, 22(1):27–42, 2004.

[Sea05] Hyung Gi Song and Yeonseung Ryu et al. *Metrics, Methodology, and Tool for Performance-Considered Web Service Composition*, pages 392–401. Lecture Notes in CS. 2005.

[ZYL08] Zhaoli Zhang, Zongkai Yang, and Qingtang Liu. Performance analysis of composite web service. *IEEE International Conference on Granular Computing*, pages 817–821, Aug. 2008.

Chapter 11

SOA Maturity Models

Christian Kalla

Contents

Abstract: Maturity models are one kind of measurement that enable enterprises to improve their processes, structures and services. This paper gives an overview of maturity models in general and models especially designed for service oriented architectures. Here the independent SOA maturity model (iSOAMM) is a quite new maturity model and will be the central aspect of this work. Furthermore the important aspects that are crucial for the design of maturity models are examined and a concluding view of the discussed models is given.

11.1 Introduction

The evaluation of product development processes as the development processes for software can be regarded as a very important issue for an enterprise. By evaluating the structure of the IT landscape by taking several key aspects into account, it is possible to get an impression of how the current processes look like and what they should look like in future. But how to judge the quality of development processes and management issues? This is where maturity models come into play.
In general they define scales and metrics in the form of maturity levels that serve as indicators for the quality of processes. Each of those levels can be reached by fulfilling a number of criteria that are well defined by the model. Those criteria normally cover technical issues as well as management issues and describe how those issues look like on every maturity level. Thus enterprises are given a roadmap how structures should look like and the management knows the requirements that are needed to reach the next level.
This paper will first cover maturity models in general and then focus on maturity models especially designed for SOAs. As SOA has to be understood as a completely new paradigm, maturity models have to consider every area that is affected by the changes of an SOA adoption. This means that the presented models have a more general view compared to the technical view of common software maturity models. The aim of this paper is to give an overview and a comparison of SOA maturity models and also describe the difficulties that arise for companies during SOA adoption.

11.2 Maturity models in general

Measurement is an aspect you always come across in everyday life. However this aspect is not widely spread in software companies, because it is not always easy to define metrics for software, development processes or management issues. But in order to improve processes within a company, it is important, especially for the management, to know how good processes currently work according to a well defined measurement. Without this information it would be more difficult to make progress and to maintain or improve the market position. Furthermore a measurement used by several companies gives them the opportunity to compare their ranking and to get an impression of how good their competitors are.
Maturity models can serve as such a measurement, although it has to be called into question if the key indicators of the used model really cover all important aspects and how those aspects are weighted in detail. Because of this it is important to design the model in such a way that all major aspects of the domain it has been designed for are stressed and that the model is kept up to date. The next section deals with the design of maturity models and the aspects you have to pay attention to.

11.2.1 Evaluation criteria for maturity models

There are a number of evaluation criteria that form the basis for a maturity model. The most important criteria that have to be considered are discussed in the next paragraphs.

The domain of interest
Every domain a model has been designed for has its special characteristics and it is a difficult task to figure out the most important ones. The CMMI which will be discussed in the next section takes four categories as key indicators (KIs) for each maturity level. Perhaps a different choice of the criteria would have been suitable as well, but the authors considered those criteria as most important. Of course it makes sense to connect IT and management issues and for example take the criterion ”project management” into account, because IT and business are strongly related in today’s companies and business processes have a direct influence on IT. As we will see later on, there are again some other criteria to be considered for SOA maturity models.

Product independence
In order to construct a generic maturity model it does not seem reasonable to make the maturity levels product dependent, because this links the companies using the model to certain vendors which could be a problem in the future when trying to be independent from their products again. In order to give the enterprises the flexibility they need, a maturity model should solely describe certain key criteria that have to be fulfilled, regardless of the vendors of the technologies.

Technological independence
Moreover maturity models should not dictate the technologies to be used. There should be a description of general goals instead, because this enables enterprises to reach the defined goals with the technology they like and they are also given the possibility to keep their old technology and upgrade it if the old one is no longer sufficient. Otherwise it could be very expensive to give up the historically grown old IT landscape, because the maturity model demands the use of new technologies.

11.3 CMMI - An SOA independent maturity model

One maturity model used by many companies is called Capability Maturity Model Integration (CMMI). This model was developed by the software engineering institute of the Carnegie Mellon University in Pittsburgh and was released in 1991. The current version is 1.2.
The model supports the improvement of processes of different kinds of companies: companies that develop software systems or hardware (CMMI-DEV), companies that acquire software or hardware, but do not develop themselves (CMMI-ACQ) and companies that provide services (CMMI-SRV). The model

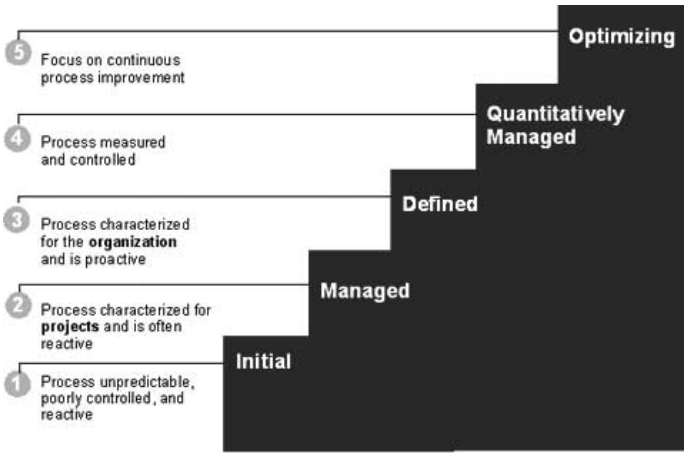

Figure 11.1: Maturity levels of the CMMI (taken from [cmmb])

defines several process areas that are grouped into four main categories. Those categories depend on the kind of company, but are very similar in general. For the CMMI-DEV those categories are project management, engineering, support and process management.
The main model consists of five capability levels and five maturity levels. The capability levels describe the things the management has to do to be ranked on a certain level. The maturity levels are reached if the capabilities for each process area correspond to this level. Each level is associated with a number of so called ”best practices” for each process area that have to be fulfilled to be ranked on the level. The principle is to divide the process areas into clearly defined tasks. For example the process area project planning is divided into the tasks ”Make estimates”, ”Develop a project plan” and ”cause duties for the project plan”. ”Make estimates” can again be subdivided into the best practices ”estimate the complexity of the project”,”estimate the costs” and so on. By considering the best practices of the subcategories a ranking of a company can be computed. It has to be stressed that the CMMI only defines goals which means that it is always stated what to do and not how to do it. This underlines the technology independence and enables an enterprise to reach the maturity with its preferred technologies. If a company uses all the defined best practices that are necessary for the level, it will be ranked on the level. Figure 11.1 gives a brief overview of the maturity levels without mentioning all the subcategories and the best practices in detail.

Level 1: Initial There are no requirements for a company to reach this level. It is reached automatically.

Level 2: Managed There is already project management established, but it is it not as well organized as it should be. There is no clearly defined process that describes how project management steps should be done in detail.

Level 3: Defined There is a standard process that defines how project de-

velopment should look like and this process is improved continuously. A systematic and well organized way of process management has been introduced.

Level 4: Quantitatively managed There are defined metrics that allow the quantitative analysis of the development process (e.g. software metrics) and help the management to organize the process by analyzing the measured data.

Level 5: Optimizing The measurements that have been established on the previous level are used to improve the process and everything that is connected to it.

CMMI appraisals
In order to obtain an official CMMI ranking, enterprises have to accomplish so called "SCAMPI-appraisals" (Standard CMMI appraisal method for process improvement) that can only be lead by lead appraisers. In Germany those appraisers are organized in the German CMMI Lead Appraiser and Instructor Board (CLIB). Due to appraisal results from 2004 most of the companies are ranked on level 2 or 3, but there are also several companies (mostly located in India) that have reached level 5. The latest appraisal results can be found on [cmma].

11.4 SOA maturity models

This section deals with maturity models especially designed for SOAs. There will be a general overview of the models currently available with a focus on an independent model for SOAs, the so called iSOAMM. The KIs and levels of this model will be discussed in detail. Afterwards the models are compared to each other.
In contrast to general maturity models as the CMMI, SOA maturity models have to stress some additional aspects. Those aspects concern the connection between business and IT and the resulting consequences for software development and organization. In general SOA maturity models take a closer look at services and service integration and do not consider the design of standalone applications any more. In order to improve the connection to business processes the services have to be orchestrated and also casual activities like monitoring and security of services have to be taken into consideration. The technologies used in SOA models are often based on web services and the business process execution language (BPEL), although there should be a high level of technology independence. The organizational demands concentrate on the establishment of SOA teams and the separation of the departments into subdivisions according to the service categories in most cases.

11.4.1 The Architecture Capability Maturity Model (ACMM)

The ACMM is described in [oC] and mainly focuses on criteria concerning the system architecture. Nevertheless it also covers organizational aspects. The

Table 11.1: Maturity levels of the ACMM

1.	None
2.	Initial
3.	Under Development
4.	Defined
5.	Managed
6.	Measured

Table 11.2: IT Architecture Characteristics of the ACMM

1.	Architecture Process
2.	Architecture Development
3.	Business Linkage
4.	Senior Management Involvement
5.	Operating Unit Participation
6.	Architecture Communication
7.	IT Security
8.	Governance
9.	IT Investment and Acquisition Strategy

model was developed by the US Department of Commerce (DoC) and consists of two components:

- The description of the maturity levels and the key criteria on each level (Department of Commerce IT Architecture Characteristics of Operating Units' Processes at Different Maturity Levels)
- A method to derive the maturity level from the data gained in the step before (DoC IT Architecture Capability Maturity Model Scorecard)

The existence of the scorecard underlines the intention of the ACMM to conduct an assessment of the IT architecture for a CIO. Table 11.1 and 11.2 show the 5 maturity levels and the 9 IT Architecture Characteristics.
Due to the high number of KIs the model allows for a very detailed analysis of the architecture and therefore enables the management to figure out which area to improve to get a higher overall maturity. However, the specific details of an SOA are not covered by this model. A detailed description of the subcharacteristics will be omitted at this point as it can be found in [oC].
The scorecard to determine the maturity ranking offers two possibilities for presenting the evaluation results to the management. The first one computes the maturity level for each of the nine subcharacteristics and calculates the arithmetic mean afterwards. The second one takes a look at each maturity level and counts how many of the subcharacteristics were ranked on this level. Afterwards the relative frequency of those occurrences is computed. This methods allows the CIO to determine at once which of the KIs related to the corresponding maturity level have to be improved.

11.4.2 The New SOA Maturity Model (NSOAMM)

The companies SonicSoftware, AmberPoint, BearingPoint and Systinet developed the NSOAMM which currently is the most well-known SOA maturity model. The model consists of five maturity levels (see Figure11.2), whereas level 3 is divided into the two sublevels "Business Services" and "Collaborative Services". An overview of the maturity levels and the KIs can be found

in [BNKH06]. The levels can only be reached by the establishment of the demanded technology and software standards that are also clearly stated by the model.
Level one is a basic level that is characterized by SOA pilot projects. Typical technologies for this level are XML, XSLT, WSDL, SOAP, J2EE and .NET. The next level mostly deals with the standardization of development processes and SOA integration. There are again concrete standards like UDDI or XQuery given. This makes obvious that the NSOAMM mostly depends on the use of web services as technological basis. Level 3 is separated into two parts and turns its attention to the orchestration of services to be able to support business processes. As a technology to realize this WSBPEL is given which also plays a role in the iSOAMM discussed later on. There are two possibilities for a company to reach level 4, either by method 3a or method 3b. Method 3a tries to reach the realization of complete business processes, whereas method 3b puts the focus on collaboration with other companies. A main problem here is the connection from internal to external services and the exchange of data between enterprises. One important standard for this is RosettaNet.
The key goals of the next two levels are improving the measurement of services by introducing metrics and advancing from reactive to real-time business processes.

11.4.3 The (Open Group) Service Integration Maturity Model ((O)SIMM)

The SIMM was developed by IBM in 2005 and consists of seven maturity levels. IBM did not publish the KIs required for each level at once, but sourced the project out to the Open Group in 2007 which took the model as basis for the OSIMM. Due to the similarity of the models they will be discussed here together.
An overview of the levels can be found in figure 11.2. The first 3 levels deal with applications and their integrations, whereas the last 4 levels focus on the use of services. There are 7 KIs that help to evaluate the maturity, namely Business, Organization, Methods, Applications, Architecture, Information and Infrastructure. These KIs are subdivided into smaller parts that can be characterized by certain questions. An extract of questions can be found in table 11.3. The whole catalogue of questions is defined in [wg07]. The first four levels describe the development from standalone application to coupled services that can be regarded as a first SOA approach. On Level 5 (Componentized Services) the services can be used to support business processes, e.g. by using composition languages such as BPEL that is also part of many other maturity models. Level 6 considers services no longer as programs with a clearly defined physical infrastructure, but as virtual services that map from an abstract virtual description to a concrete service. So the services are no longer called directly, e.g. by defining server and port, but there is a system that gets a more abstract service description as input and looks for the right service that is invoked later on. Level 7 requires a better adaption of processes to the dynamic behaviour of business processes, e.g. by a modelling language that is even more business-oriented than before.

Table 11.3: Questions related to the key indicators of the OSIMM ([wg07])

Business	How agile are your current business processes?
Organization	How does IT governance relate to your SOA?
Methods	What design methodologies and best practices are you currently adopting?
Applications	What is your current application development style?
Architecture	How mature are your web services implementations?
Information	Are the data models defined by a language that includes taxonomies, ontologies or other high-level logical representations?
Infrastructure	What tools are used for configuration management?

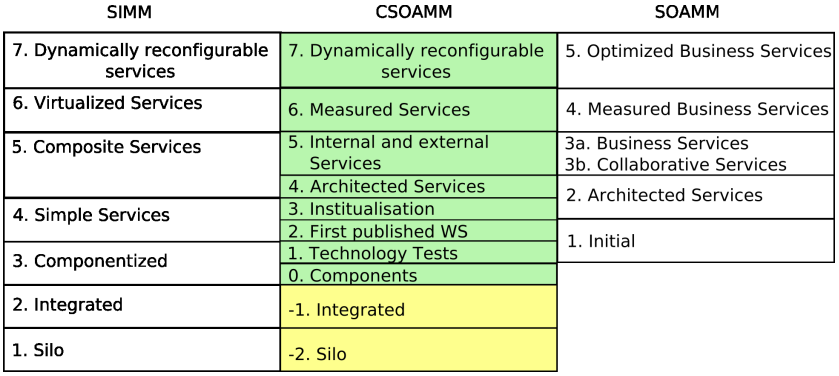

Figure 11.2: Mapping of the maturity levels: SIMM-CSOAMM-SOAMM (taken from [Mei06])

11.4.4 The Combined SOA Maturity Model (CSOAMM)

This model is a combination of the SIMM and the NSOAMM and was introduced in Meier's master thesis at the university of Skövde ([Mei06]). It is a scientific approach that was supported by the authors of the SIMM and the NSOAMM which tries to form a better model that contains characteristics extracted from both models for the maturity levels. The combination of the level characteristics contained in the models helps enterprises to better understand their current standing of SOA adaption, because there is a more detailed description of the key criteria now in comparison to the use of one single model. As the author stresses in his thesis the CSOAMM was not designed to be regarded as an independent model, but to serve as a "translator" between companies who use the SIMM and others who use the NSOAMM.
Figure 11.2 shows how the maturity levels are combined and which new levels are introduced.
The maturity model consists of 10 levels (-2 to 7). Some of those levels are identical to levels of the SIMM or the NSOAMM, others are sublevels that are

needed to create a suitable mapping between the models. An important point to realize is that the levels of the SOAMM are mapped to higher levels of the other models and that the mapping does not start at the bottom, as this is done for the SIMM. This is because the SOAMM is a model especially designed for SOAs and does not consider service maturity in general like the SIMM. The SOA maturity of the CSOAMM is described within the levels 0 to 7. As the NSOAMM is focused on web services, its first two levels are mapped to COAMM level 1 that deals with web services and not to level 0 that does not consider web services in detail. More exact information about this model can be found in the previously mentioned master thesis ([Mei06]).

11.4.5 The Oracle Maturity Model (OMM)

The OMM consists of the five maturity levels Opportunistic, Systematic, Enterprise, Measured and Industrialized. Those levels are grouped into the eight KIs Infrastructure, Architecture, Information & Analytics, Operations, Project Execution, Finance and Portfolios, People & Organization and Governance. This granularity assures that every area that is affected by an SOA is considered as KI. The maturity levels cover first SOA approaches (Opportunistic) up to automated business processes that are monitored and measured and support the reaction to business events (Industrialized). The technology related KIs (Infrastructure and Architecture) are realized by web services. The aspect ”Finance and Portfolios” is not covered in other models in such a specific way and deals with the funding of the SOA on each level.

11.4.6 The independent SOA Maturity Model (iSOAMM)

This section deals with the main aspect of this paper, an independent maturity model for SOAs. It has been developed at the FZI Research Center for Information Technology in Karlsruhe and is described by Rathfelder and Groenda in [RG08]. The motivation was to create a model that is independent from any vendor and also technology independent. Furthermore an important aspect was the development of a model that both considers technical and organizational aspects. The model is based on the existing SOA maturity models introduced before, own experiences of the authors gained in a project called ”Karlsruher Integriertes Informationsmanagement”(KIM) ([FLM+06]) and published case studies of companies. Those case studies take a closer look at the SOA landscape of the companies Deutsche Post ([LH07],[KBS],[Hel06]),Credit Suisse([LH07],[KBS]) and ABB ([GHS05]). Another important aspect when constructing the model was to study articles about SOA implementations to figure out the key indicators for the maturity levels (KIs) that will be discussed next.

Service Architecture This point considers the setup of the software architecture. This means to take a look at the layers the architecture consists of, the application landscape and the services. Furthermore it has to be examined how the business processes of the enterprise are represented using the

software architecture and if there is user and customer interaction.

Infrastructure As the change of business processes influences IT to a very high extent, a stable infrastructure is needed. This is an important aspect that is also discussed in the SOA series of the CIO magazine taking a focus on ”Deutsche Post” ([Hel06]). The task of the infrastructure is to enable the communication between the services which is realized by network components. Furthermore this infrastructure can be extended by additional components as monitoring or security tools. In order to bring the running applications together to enable a service landscape, a stable ”communication bus” is needed. This emphasizes the connection between the infrastructure and the service architecture, although they still have to be regarded as different indicators for a good SOA structure.

Enterprise Structure The realization of an SOA also affects the business department of an enterprise and that is why the organization and duties of the different divisions often have to be changed when introducing SOA. SOA concerns everybody in the company and historically grown old organizational structures often hinder the introduction of an enterprise-wide SOA.

Service Development As pointed out in [CK05] from IBM research SOA changes the development mostly concerning management tasks. There is no longer a development process for a distributed system landscape as before, but there are management interfaces to map business processes directly to the service infrastructure. So the development processes are no longer the same and have to be adapted to the needs of SOAs.

Governance Besides the aspects ”Enterprise structure” and ”Service Development”, the whole company is affected by migrating to an SOA. There have to be governance policies that communicate new guidelines and rules to bring the employees away from keeping their old workflows and habits and make them adapt their working behaviour to the changes SOA introduces.

The maturity levels

This section introduces the five maturity levels of the iSOAMM and describes how the previously discussed KIs have to look like on each level. The model is very similar to general maturity models like the CMMI concerning the setup of the model and the decrease of granularity up to ”best practices” that give a very clear impression what an enterprise has to do to reach the corresponding level. Figure 11.3 is taken from [RG08] and gives a short description of the key indicators for each maturity level.

Level 1: Trial SOA
This level is characterized by independent SOA projects which can be regarded as a first try to gain experience with services. There are no standards concerning development processes or technologies.

Maturity Level \ Viewpoint	Service Architecture	Infrastructure	Enterprise Structure	Service Development	Governance
5 On Demand SOA	dynamic services	service marketplace	service as business	service on demand	automated
4 Cooperative SOA	processes	management, event-driven	service alligned	model-driven	fair competition control
3 Administered SOA	orchestrated services	monitoring, security	centrally managed	documented, tool support	rules
2 Integrative SOA	integrated applications	communication	IT-oriented	hands-on experiences	guidelines
1 Trial SOA	islands	inhomogeneous	separated	unstructured	none

Figure 11.3: Maturity levels of the iSOAMM (taken from [RG08])

The legacy applications are connected to services that can be used by other applications. Due to the lack of standardization it can be the case that there are services using different technologies and so there is no communication possible. Additionally the infrastructure does not consist of a standardized communication system, but of many inhomogeneous systems. This leads to incompatible ”Service islands”.
The business is divided into several subunits which all have their own IT departments. Because there is no central IT department the communication only takes place within the departments and not enterprise-wide.
The development of services does not follow guidelines and differs for each SOA project. This can be regarded as a test phase to figure out best practices to be able to construct a standardized development process later on.
SOA projects are done within the IT departments and do not affect the management. There is no communication across the borders.

Level 2: Integrative SOA
The experience from the previous level is used to improve the infrastructure and the development process. The main goal is the realization of Enterprise Application Integration (EAI).
A common service bus (SB) is introduced and an API for the (standardized) services is developed which can be used by frontend applications. The service are now connected to each other and there are no longer service islands. Because there is a lot of communication between the applications to form a service the requirements for the structures that enable the interaction between applications and services are very high. The previously mentioned ”Service Bus” can be built up using many different middleware technologies offering a high abstraction level. There are companies using CORBA (like ”Credit Suisse”), Web Services (like ”Sparkassen Informatik”) or J2EE technology (like ”Deutsche Post”). One important aspect is that the service bus allows logical addressing of services in order to have the option to change its physical location.
The enterprise structure is adapted to SOA by setting up an own SOA team

that is responsible for all SOA aspects within the company. This team develops the service bus and trains employees for developing services. The importance of an SOA team becomes obvious looking at the fact that even well-known companies like ”T-Com”,”Credit Suisse” and ”Deutsche Post” introduced such an SOA team as pointed out in the case studies of [LH07].
The development of services is now better organized due to a knowledge base for developers that contains all the best practices learned on the previous level. Furthermore the development is supported by better tools focused on the design of services.
Because the developed services are often changed and there are a lot of dependencies between services, a versioning system becomes necessary. This is not only important for the developers but also for the management, because change requests must be handled within the whole company and the rollout of new services has to be communicated. Another important point is that the developers do not have much experience with the integration of new services into the service bus. Because this rising complexity higher costs occur which have to be balanced by an enterprise wide compensation payment system.

Level 3: Administered SOA
The main difference between level 3 and level 2 is the orchestration of services. The services developed on layer 2 communicating via the service bus can now be organized in such a way that it is possible to adapt them to business processes. It is now much easier for the management to connect business needs to IT by using a so called ”Orchestration layer”.
Figure 11.4 contains elements of the IT landscape of an enterprise on iSOAMM level 3 (the top level elements are not yet included). The legacy application are connected to services which are orchestrated by the orchestration layer. This layer provides its interface to the frontend which is used by employees or customers. One example for constructing an orchestration layer is to use the ”Webservice Business Process Execution Language”(WSBPEL) as described in [OAS07]. The services are framed by a communication layer which enables the communication between them, a security layer that is responsible for a safe data transfer (this could be important if sensible services as they are needed in the banking domain are considered) and managing access rights for the services and a monitoring layer which logs how often and by whom services are used. In order to stress the relation between management and IT it also makes sense to use business relevant data types to have a direct representation of business ”vocabulary” in software and facilitate the exchange of data between the services.
Of course those abstract things like an orchestration layer have to be realized in the infrastructure. WSBPEL for example produces process description and not executable code at once. That is why an orchestration engine is needed that translates the abstract descriptions into code that can be used by the sublayer. The monitoring and security infrastructure is also needed and can be realized with a so called ”Web Services Architecture Stack” that is presented in [BHM+04].
As the services on IT level can be divided into several categories like customer

data, customer relationship management or human resources, it also makes sense to structure the business departments in the same way. So every department is responsible for the services within its service domain. The SOA team introduced on level 2, is still needed and it is enriched with business experts on this level to define data standards and the distribution of services to service domains.
The knowledge base used by the service developers is enlarged and more automated development steps are introduced. The orchestration of services allows for a Model-Driven Software Development, as the services are standardized and the same implementation language can be used.
As before, it is important for the governance to stress the importance of the service orientation paradigm within the company.

Level 4: Cooperative SOA
The main issue of this level are so called Service Level Agreements (SLAs). These agreements are arranged between the service provider and the customer and guarantee the use of a service with a defined quality if a clearly specified usage profile is given. SLAs are strongly related to the IT Infrastructure Library (ITIL) which describes best practices for the management of services. The next picture shows the introduction of a new process layer which offers an amount of business processes and is connected to a choreography layer and a portal. There are two kinds of business processes: Automated processes within the business (B2B processes) and processes that require human interaction.
Processes with human interaction can not be implemented as orchestrations and need to be implemented as choreography between processes. This is why a user interface is required which is the portal that is also represented in the figure. The portal integration to interact with the user can be realized with ”WSBPEL Extension for People” presented in [AAD+07]. Because events play a central role in daily business life, it is not sufficient to have communication between the single services, but a possibility to react on events also has to be established.
In order to support the new process layer and the user interaction the infrastructure has to be extended. One example for an infrastructure that makes it possible for services to react on events is the ”Event Bus Infrastructure” developed by ”Credit Suisse”([KBS]). Similar to the last layer, where a runtime environment for the orchestration layer was needed, an additional environment for the processes is required now. This environment supports the choreography of processes and the integration of user interaction. Another important thing is that business people must be able to construct processes without big technical knowledge. This is why the service repository has to contain semantic descriptions. An example for this could be a UDDI-based repository.
The granularity of the enterprise structure has to be changed again. There is a proposal from Bieberstein([BBWL05]) which suggests to form teams that are responsible for a single service, but it has to be called into question if this makes sense for every enterprise.
Models and processes are designed using graphical representations that makes

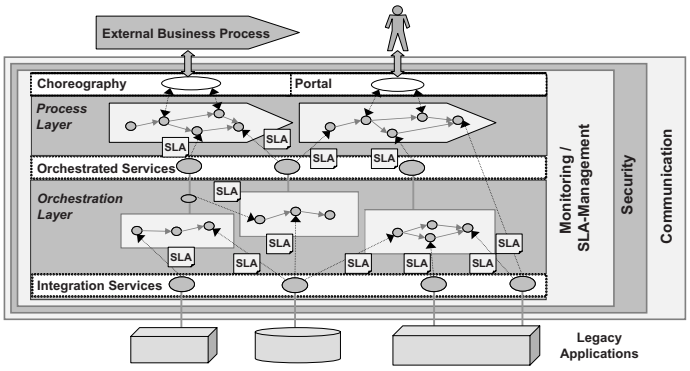

Figure 11.4: Architecture characteristics of iSOAMM level 4 (taken from [RG08])

it easier for business people with few technical knowledge. Another important task at this level is adapting the process development to the SLAs, because the process has to fulfill specified quality requirements that have to be guaranteed whenever the customer uses the process. To cope with this it is important to pay attention to the quality of service to satisfy the customer.
On this level it is absolutely necessary that every legacy application provides an interface to a service and is integrated into the SOA. Furthermore the definition of metrics is essential to control the maturity of the SOA and the business processes.

Level 5: On Demand SOA
In everyday life the service needs of customers frequently change and so there are often no long term SLAs, but the SLAs have to be adapted to the customer's needs very fast. So there must be a possibility for the customer to change the SLAs without high effort.
The service architecture has to be changed in such a way that services can be chosen at runtime according to the wishes of the customer. To determine the service that offers the right functionality with a demanded quality, a data ontology is needed to convert between data formats of the process and the services.
A trading platform for services, a so called marketplace has to be offered to the customer to make it possible to give him the choice between available services he has to pay for. This sale of services via the marketplace means the runtime negotiation of SLAs which has to be supported by the infrastructure. In order to reach this the monitoring of SLAs, orchestrations and processes is needed or has to be improved.
The service development has to change, because services are now chosen at runtime whereas they were selected at design time before. Orchestrations and processes must have repositories containing the semantic specification of services that can be used during runtime.

11.4.7 Comparison of the discussed models

The main setup of the models is always similar: There is a basic level that is reached when companies make their first experience with SOA. The levels afterwards describe the advance from loosely coupled services to orchestrated services up to automated business processes with customer interaction. The CSOAMM serves as an example for the fact that the granularity of the maturity levels differs from model to model. Although the setup of the levels is nearly identical in most cases and oriented at the CMMI, the key criteria are always different. The ACMM focuses architecture details as well as organizational aspects, but does not consider criteria as coupling or reuse of services that are important for an SOA. It is the only model that provides a concrete mechanism for calculating the maturity level by providing a scorecard. The KIs of the OMM are technology-dependent, because the realization of the SOA requires the use of web services in this model. Additionally aspects like automated business processes and user interaction as they are part of level 5 of the iSOAMM are not taken into account. The KIs of the NSOAMM are especially designed for SOAs and also take web services as basic part of an SOA realization. Important areas like security and monitoring of services or the governance of an enterprise are not regarded and some KIs show a high level of product dependency. IBM's model SIMM and its successor OSIMM only describe SOA maturity on the levels 4-7, but the granularity of the KIs is well-thought and important SOA aspects are covered. The iSOAMM is a very new model that tries to be as most technology independent as possible and stresses the connection between business and IT aspects.
It is not easy to say which of the described maturity models is better than another one, because there is no proof that certain KIs assure a good SOA adoption. There is no "universal definition" of SOA and therefore it might be too early to publish those models. This criticism is also mentioned in Meier's thesis ([Mei06]) and it calls the value of those kind of models into question. Nevertheless a well designed SOA model that covers all important aspects should be a good guideline for enterprises.

11.4.8 Examples for companies and their rankings

Unfortunately there are no big companies who publish their rankings they achieved in a specific SOA maturity model. As stated by the authors in [RG08] the KIM project started on level two and reached level three in 2007. The SOA of "Deutsche Post" is also ranked on level 3 whereas the SOA of "Sparkassen Informatik" is an example for rank 2 according to the iSOAMM. Level 4 and level 5 of the iSOAMM have not been reached by any company up to now and it has to be called into question if this ever happens. In general the higher levels of every SOA maturity models described above are difficult to reach, because they require high technical and organizational demands that cannot be reached that fast with respect to the available IT and business structure of today's companies. An article about the main problems of an SOA adoption can be found in [Her]. The described problems are related to the business-IT

linkage in most cases. There are only few employees with good SOA knowledge, but those people in general have a technical background and are not familiar with business topics. This is why companies like ”Deutsche Post” try to improve their business knowledge by further education. Furthermore SOA projects are often initiated by IT departments and the CIO and not supported by the overall management and the CEO. Besides those major problems, there are also technical difficulties. There is an excess of standards (e.g. in the domain of web services) and the testing of services is not carried out with the needed accuracy.
Regarding general maturity models like the CMMI things look a bit different. There are several Indian companies that reached level 5 which is a good advertisement to gain new customers. Because CMMI is widely spread the rankings can be compared quite easily and so it is a good indicator for the market position of an enterprise. Of course, this is not the case for SOA maturity models at the moment.

11.5 Summary/Conclusion

This paper tried to motivate why it is important to use maturity models and gave an overview of the SOA independent model CMMI as well as special models designed for SOAs. At this point the emphasis was placed on the iSOAMM developed by the Research Center for Information Technology in Karlsruhe. It became obvious that the choice of the KIs plays an important role during the construction of a maturity model and that it is important to pay attention to stress the connection between business and IT and not to dictate which technologies to use. It remains to be seen how those models are used in practice and if some of them will get the status the CMMI has today. Currently people are no longer that enthusiastic about SOA that was a hype topic during the last few years. ”SOA is dead. Long live services”, a cite from Anne Thomas Manes from the Burton Group ([Man]) at the beginning of this year, describes the situation of many IT companies today. Some of the SOA approaches did not pay off and brought just costs instead of benefits.

Bibliography

[AAD+07] A. Agrawal, M. Amend, M. Das, M. Ford, C. Keller, M. Kloppmann, D. König, F. Leymann, R. Müller, G. Pfau, K. Plösser, R. Ragaswamy, A. Rickayzen, M. Rowley, P. Schmidt, I. Trickovic, A. Yiu, and M. Zeller. WS-BPEL Extension for People (BPEL4People). `https://sdn.sap.com/irj/sdn/bpel4people`, 2007.

[BBWL05] N. Bieberstein, S. Bose, L. Walker, and A. Lynch. Impact of service-oriented architecture on enterprise systems, organizational structures and individuals. *IBM Systems Journal*, 44(4):691–708, 2005.

[BHM+04] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferric, and D. Orchard. Web Services Architecture, 2004. W3C Working Group Note.

[BNKH06] J. Bachmann, D. Ng, S. Kline, and E. Horst. SOA Maturity Modell, oder der Weg zu einer Service Orientierten Architektur. whitepaper ,`http://sonicsoftware.com/soamm`, 2006.

[CK05] D.E. Cox and H. Kreger. Management of the service-oriented-architecture life cycle. *IBM Systems Journal*, 44(4):709–726, 2005.

[cmma] `http://sas.sei.cmu.edu/pars/pars.aspx`. CMMI Appraisal Results.

[cmmb] SEI CMMI Tutorial. `http://www.tutorialspoint.com/cmmi/index.htm`.

[FLM+06] P. Freudenstein, L. Liu, F. Majer, F. Maurer, C. Momm, D. Ried, and W. Juling. Architektur für ein universitätsweit integriertes Informations- und Dienstmanagement. In *INFORMATIK 2006- Informatik für Menschen*, volume 1, pages 50–54. 2006.

[GHS05] D. Gizanis, R. Heutschi, and T. Solberg. Global Order Management Services Support Businesses at ABB. `http://www.alexandria.unisg.ch/Publikationen/23667`, 2005.

[Hel06] J. Helbig. Englewood Cliffs, reprint edn. prentice hall ptr edition, 2006.

[Her] Wolfgang Herrmann. Die zehn schwersten SOA Hürden. `http://www.computerwoche.de/knowledge_center/soa_bpm/597036/`.

[KBS] D. Krafzig, K. Banke, and D. Slama. SOA Serie: Teil 1-5. CIO. `http://www.cio.de/schwerpunkt/d/Deutsche-Post-Brief.html`.

[LH07] C. Legner and R. Heutschi. SOA Adoption in Practice- Findings from Early SOA implementations. In *Proc. of European Conference on Information Systems (ECIS 2007)*, 2007.

[Man] Anne Thomas Manes. SOA is dead. Long live services. `http://apsblog.burtongroup.com/2009/01/soa-is-dead-long-live-services.html`. Blog entry.

[Mei06] F. Meier. Service Oriented Architecture Maturity Models - A guide to SOA adoption, 2006.

[OAS07] OASIS. Web Services Business Process Execution Language (WSBPEL), 2007.

[oC] Department of Commerce. Introduction-IT Architecture Capability Maturity Model. `http://ocio.os.doc.gov/groups/public/@docs/@os/@ocio/@oitpp/documents/content/prod01_002340.pdf`.

[RG08] Christoph Rathfelder and Henning Groenda. iSOAMM: An Independent SOA Maturity Model. In *DAIS*, pages 1–15, 2008.

[wg07] The Open Group (OSIMM working group). Launch Presentation and WG Updates 1.0. `http://www.opengroup.org/projects/osimm/`, 2007.