A

Seminar report

On

Service Oriented Architecture

Submitted in partial fulfillment of the requirement for the award of degree
Of Computer Science

SUBMITTED TO:

www.studymafia.org

SUBMITTED BY:

www.studymafia.org

Preface

I have made this report file on the topic **Service Oriented Architecture (SOA)**, I have tried my best to elucidate all the relevant detail to the topic to be included in the report. While in the beginning I have tried to give a general view about this topic.

My efforts and wholehearted co-corporation of each and everyone has ended on a successful note. I express my sincere gratitude to ……..who assisting me throughout the preparation of this topic. I thank him for providing me the reinforcement, confidence and most importantly the track for the topic whenever I needed it.

Content

Introduction

SOA implementations rely on a mesh (**Mesh** consists of semi-permeable barrier made of connected strands of metal, fiber, or other flexible/ductile material. Mesh is similar to web or net in that it has many attached or woven strands.)Of software services. Services comprise unassociated, loosely coupled units of functionality that have no calls to each other embedded in them.

Each service implements one action, such as filling out an online application for an account, or viewing an online bank-statement, or placing an online booking or airline ticket order.
Instead of services embedding calls to each other in their source code they use defined protocols that describe how services pass and parse messages, using description metadata.

SOA developers associate individual SOA objects by using orchestration (**Orchestration** describes the automated arrangement, coordination, and management of complex computer systems, middleware, and services.). In the process of orchestration the developer associates software functionality (the services) in a non-hierarchical arrangement (in contrast to a class hierarchy) using a software tool that contains a complete list of all available services, their characteristics, and the means to build an application utilizing these sources.

Underlying and enabling all of this requires metadata in sufficient detail to describe not only the characteristics of these services, but also the data that drives them. Programmers have made extensive use of XML in SOA to structure data that they wrap in a nearly exhaustive description-container.

Analogously, the Web Services Description Language (WSDL) typically describes the services themselves, while theSOAP protocol describes the communications protocols. Whether these description languages are the best possible for the job, and whether they will become/remain the favorites in the future, remain open questions. As of 2008 SOA depends on data and services that are described by metadata that should meet the following two criteria:

The core components which make up an SOA implementation

1. The metadata should come in a form that software systems can use to configure dynamically by discovery and incorporation of defined services, and also to maintain coherence and integrity.

2. The metadata should come in a form that system designers can understand and manage with a reasonable expenditure of cost and effort.

SOA aims to allow users to string together fairly large chunks of functionality to form *ad hoc* applications that are built almost entirely from existing software services. The larger the chunks, the fewer the interface points required to implement any given set of functionality; however, very large chunks of functionality may not prove sufficiently granular for easy reuse. Each interface brings with it some amount of processing overhead, so there is a performance consideration in choosing the granularity of services.

The great promise of SOA suggests that the marginal cost of creating the n-th application is low, as all of the software required already exists to satisfy the requirements of other applications. Ideally, one requires only orchestration to produce a new application.

For this to operate, no interactions must exist between the chunks specified or within the chunks themselves. Instead, humans specify the interaction of services (all of them unassociated peers) in a relatively *ad hoc* way with the intent driven by newly emergent requirements.

Thus the need for services as much larger units of functionality than traditional functions or classes, lest the sheer complexity of thousands of such granular objects overwhelm the application designer. Programmers develop the services themselves using traditional languages like Java, C,C++, C# or COBOL.

SOA services feature loose coupling, in contrast to the functions that a linker binds together to form an executable, to a dynamically linked library or to an assembly. SOA services also run in "safe" wrappers (such as Java or .NET) and in other programming languages that manage memory allocation and reclamation, allow *ad hoc* and late binding, and provide some degree of indeterminate data typing.

As of 2008, increasing numbers of third-party software companies offer software services for a fee. In the future, SOA systems may consist of such third-party services combined with others created in-house.

This has the potential to spread costs over many customers and customer uses, and promotes standardization both in and across industries. In particular, the travel industry now has a well-defined and documented set of both services and data, sufficient to allow any reasonably competent software engineer to create travel-agency software using entirely off-the-shelf software services.

Other industries, such as the finance industry, have also started making significant progress in this direction.

SOA as an architecture relies on service-orientation as its fundamental design-principle. If a service presents a simple interface that abstracts away its underlying complexity,

users can access independent services without knowledge of the service's platform implementation.

Requirements

In order to efficiently use a SOA, one must meet the following requirements:

- Interoperability between different systems and programming languages that provides the basis for integration between applications on different platforms through a communication protocol. One example of such communication depends on the concept of messages. Using messages across defined message channels decreases the complexity of the end application, thereby allowing the developer of the application to focus on true application functionality instead of the intricate needs of a communication protocol.
- Desire to create a federation of resources. Establish and maintain data flow to a Federated database system(A **federated database system** is a type of meta-database management system (DBMS) which transparently integrates multiple autonomous database systems into a single **federated database**). This allows new functionality developed to reference a common business format for each data element.

Principles

The following **guiding principles** define the ground rules for development, maintenance, and usage of the SOA:

- Reuse, granularity, modularity, compensability, componentization and interoperability.
- Standards-compliance (both common and industry-specific).
- Services identification and categorization, provisioning and delivery, and monitoring and tracking.

The following **specific architectural principles** for design and service definition focus on specific themes that influence the intrinsic behavior of a system and the style of its design:

Attributes of a SOA

- Service **encapsulation** – Many web services are consolidated for use under the SOA. Often such services were not planned to be under SOA.
- Service **loose coupling** – Services maintain a relationship that minimizes dependencies and only requires that they maintain an awareness of each other.
- Service **contract** – Services adhere to a communications agreement, as defined collectively by one or more service-description documents.
- Service **abstraction** – Beyond descriptions in the service contract, services hide logic from the outside world.
- Service **reusability** – Logic is divided into services with the intention of promoting reuse.
- Service **compensability** – Collections of services can be coordinated and assembled to form composite services.
- Service **autonomy** – Services have control over the logic they encapsulate.
- Service **optimization** – All else equal, high-quality services are generally preferable to low-quality ones.
- Service **discoverability** – Services are designed to be outwardly descriptive so that they can be found and assessed via available discovery mechanisms.
- Service **relevance** – Functionality is presented at a granularity recognized by the user as a meaningful service.

The following references provide additional considerations for defining a SOA implementation:

- SOA Reference Architecture provides a working design of an enterprise-wide SOA implementation with detailed architecture diagrams, component descriptions, detailed requirements, design patterns, opinions about standards, patterns on regulation compliance, standards templates etc..

- Life cycle management SOA Practitioners Guide Part 3: Introduction to Services Lifecycle introduces the services lifecycle and provides a detailed process for services management through the service lifecycle, from inception to retirement or repurposing of the services.

  It also contains an appendix that includes organization and governance best-practices, templates, comments on key SOA standards, and recommended links for more information.

Conceptual model of a SOA architectural style

In addition, one might take the following factors into account when defining a SOA implementation:

- Efficient use of system resources
- service maturity and performance
- EAI (**Enterprise Application Integration (EAI)** is defined as the use of software and computer systems architectural principles to integrate a set of enterprise computer applications.)

Web services approach

Web services can implement a service-oriented architecture. Web services make functional building-blocks accessible over standard Internet protocols independent of platforms and programming languages. These services can represent either new applications or just wrappers around existing legacy systems to make them network-enabled.

Each SOA building block can play one or both of two roles:

1. Service Provider

The service provider creates a web service and possibly publishes its interface and access information to the service registry. Each provider must decide which services to expose, how to make trade-offs between security and easy availability, how to price the services, or (if no charges apply) how/whether to exploit them for other value. The provider also has to decide what category the service should be listed in for a given broker service and what sort of trading partner agreements are required to use the

service. It registers what services are available within it, and lists all the potential service recipients.

The implementer of the broker then decides the scope of the broker. Public brokers are available through the Internet, while private brokers are only accessible to a limited audience, for example, users of a company intranet. Furthermore, the amount of the offered information has to be decided. Some brokers specialize in many listings.

Others offer high levels of trust in the listed services. Some cover a broad landscape of services and others focus within an industry. Some brokers catalog other brokers. Depending on the business model, brokers can attempt to maximize look-up requests, number of listings or accuracy of the listings.

The Universal Description Discovery and Integration (UDDI) specification defines a way to publish and discover information about Web services. Other service broker technologies include (for example) ebXML (Electronic Business using extensible Markup Language) and those based on the ISO/IEC 11179 Metadata Registry (MDR) standard.

2. Service consumer

the service consumer or web service client locates entries in the broker registry using various find operations and then binds to the service provider in order to invoke one of its web services. Whichever service the service-consumers need, they have to take it into the brokers, then bind it with respective service and then use it. They can access multiple services if the service provides multiple services.

SOA and Web service protocols

Implementers commonly build SOAs using web services standards (for example, SOAP) that have gained broad industry acceptance. These standards (also referred to as Web Service specifications) also provide greater interoperability and some protection from lock-in to proprietary vendor software. One can, however, implement SOA using any service-based technology, such as Jini, CORBA or REST.

Other SOA concepts

Architectures can operate independently of specific technologies. Designers can implement SOA using a wide range of technologies, including:

- SOAP, RPC
- REST
- DCOM
- CORBA
- Web Services
- WCF (Microsoft's implementation of Web service forms a part of WCF)

Implementations can use one or more of these protocols and, for example, might use a file-system mechanism to communicate data conforming to a defined interface-specification between processes conforming to the SOA concept. The key is independent services with defined interfaces that can be called to perform their tasks in a standard way, without a service having foreknowledge of the calling application, and without the application having or needing knowledge of how the service actually performs its tasks.[*weasel words*]Many implementers of SOA have begun to adopt an evolution of SOA concepts into a more advanced architecture called SOA 2.0.

Service-Oriented Modeling Framework (SOMF) Version 2.0

SOA enables the development of applications that are built by combining loosely coupled and interoperable services.

These services inter-operate based on a formal definition (or contract, e. g., WSDL) that is independent of the underlying platform and programming language. The interface definition hides of the language-specific service. SOA-based systems can therefore function independently of development technologies and platforms (such as Java,.NET, etc). Services written in C# running on .NET platforms and services written in Java running on Java EE platforms, for example, can both be consumed by a common composite application (or client).

Applications running on either platform can also consume services running on the other as web services that facilitate reuse. Managed environments can also wrap COBOL legacy systems and present them as software services. This has extended the useful life of many core legacy systems indefinitely, no matter what language they originally used.

SOA can support integration and consolidation activities within complex enterprise systems, but SOA does not specify or provide a methodology or framework for documenting capabilities or services.

High-level languages such as BPEL and specifications such as WS-CDL and WS-Coordination extend the service concept by providing a method of defining and supporting orchestration of fine-grained services into more coarse-grained business services, which architects can in turn incorporate into workflows and business processes implemented in composite applications or portals

As of 2008 researchers have started investigating the use of Service Component Architecture (SCA) to implement SOA.

Service-oriented modeling is a SOA framework that identifies the various disciplines that guide SOA practitioners to conceptualize, analyze, design, and architect their service-oriented assets. The Service-oriented modeling framework (SOMF) offers a

modeling language and a work structure or "map" depicting the various components that contribute to a successful service-oriented modeling approach.

It illustrates the major elements that identify the “what to do” aspects of a service development scheme. The model enables practitioners to craft a project plan and to identify the milestones of a service-oriented initiative. SOMF also provides a common modeling notation to address alignment between business and IT organizations.

SOMF addresses the following principles:

- business traceability
- architectural best-practices traceability
- technological traceability
- SOA value proposition
- software assets reuse
- SOA integration strategies
- technological abstraction and generalization
- architectural components abstraction

SOA and network management architecture

As of 2008 the principles of SOA are being applied to the field of network management. Examples of service-oriented network management architectures include TS 188 001 *NGN Management OSS Architecture* from ETSI, and M.3060 *Principles for the Management Of Next Generation Networks* recommendation from the ITU-T.

Tools for managing SOA infrastructure include:

- HP Software & Solutions
- HyPerformix IPS Performance Optimizer
- IBM Tivoli Framework

Benefits

Some enterprise architects believe that SOA can help businesses respond more quickly and cost-effectively to changing market-conditions. This style of **architecture** promotes reuse at the macro (service) level rather than micro (classes) level. It can also simplify interconnection to – and usage of – existing IT (legacy) assets.

In some respects, one can regard SOA as an architectural evolution rather than as a revolution. It captures many of the best practices of previous software architectures. In communications systems, for example, little development has taken place of solutions that use truly static bindings to talk to other equipment in the network. By formally embracing a SOA approach, such systems can position themselves to stress the importance of well-defined, highly inter-operable interfaces.

Some have questioned whether SOA simply revives concepts like modular programming (1970s), event-oriented design (1980s) or interface/component-based design (1990s). SOA promotes the goal of separating users (consumers) from the service implementations. Services can therefore be run on various distributed platforms and be accessed across networks. This can also maximize reuse of services

SOA is an architectural and design discipline conceived to achieve the goals of increased interoperability (information exchange, reusability, and composability), increased federation (uniting resources and applications while maintaining their individual autonomy and self-governance), and increased business and technology domain alignment.

Service-Oriented Architecture (SOA) is an architectural approach (or style) for constructing complex software-intensive systems from a set of universally interconnected and interdependent building blocks, called services.

SOA realizes its business and IT benefits through utilizing an analysis and design methodology when creating services. This methodology ensures that services remain consistent with the architectural vision and roadmap, and that they adhere to principles

of service-orientation. Arguments supporting the business and management aspects from SOA are outlined in various publications.

A service comprises a stand-alone unit of functionality available only via a formally defined interface. Services can be some kind of "nano-enterprises" that are easy to produce and improve. Also services can be "mega-corporations" constructed as the coordinated work of sub-ordinate services.

Services generally adhere to the following principles of service-orientation:

- Abstraction
- Autonomy
- Compos ability
- Discoverability
- Formal contract
- loose coupling
- Reusability
- Statelessness

A mature rollout of SOA effectively defines the API of an organization.

Reasons for treating the implementation of services as separate projects from larger projects include:

1. Separation promotes the concept to the business that services can be delivered quickly and independently from the larger and slower-moving projects common in the organization. The business starts understanding systems and simplified user interfaces calling on services. This advocates agility.

2. Separation promotes the decoupling of services from consuming projects. This encourages good design insofar as the service is designed without knowing who its consumers are.

3. Documentation and test artifacts of the service are not embedded within the detail of the larger project. This is important when the service needs to be reused later.

An indirect benefit of SOA involves dramatically simplified testing. Services are autonomous, stateless, with fully documented interfaces, and separate from the cross-cutting concerns of the implementation. The industry has never been exposed to this circumstance before.

If an organization possesses appropriate defined test data, then when a service is being built, a corresponding stub is built that reacts to the test data. A full set of regression tests, scripts, data, and responses is also captured for the service. The service can be tested as a 'black box' using existing stubs corresponding to the services it calls. Test environments can be constructed where the primitive and out-of-scope services are stubs, while the remainders of the mesh are test deployments of full services.

As each interface is fully documented, with its own full set of regression test documentation, it becomes simple to identify problems in test services. Testing evolves to merely validating that the test service operates according to its documentation, and in finding gaps in documentation and test cases of all services within the environment. Managing the data state of idempotent services is the only complexity.

Examples may prove useful to aid in documenting a service to the level where it becomes useful. The documentation of some APIs within the Java Community Process provide good examples. As these are exhaustive, staff would typically use only important subsets. The 'ossjsa.pdf' file within JSR-89 exemplifies such a file.

References

www.studymafia.org
www.google.com
www.wikipedia.com