Service-Oriented Architecture:
Making the most of SOA – What, Why and How

Coenie Vermaak
Solutions Architect – Britehouse Automotive

15 October 2018
The benefit potential offered by SOA can only be truly realized when applied across multiple solution environments.

This does not mean that the entire enterprise must become Service-Oriented.

SOA belongs in those areas that have the most to gain from the features and characteristics it introduces.
What is Service-Oriented Architecture (SOA)

Thanks to the popularity of the topic, many promotes **Service-Oriented Architecture** as a technical architecture comprised of Web-Services.

- Whilst it is true that Web-Services can, and does, have an important part to play in a Service-Oriented Architecture, it is a very narrow and dangerous assumption

- Service-Orientation is an **implementation-agnostic** paradigm that can be realized through **any suitable technology platform**

- Services exist as physically independent software programs (or components) with distinct design characteristics

- Service-Oriented Architecture has three major objectives
  - Design software components to be **loosely coupled** to applications, so that they can be used only when needed, and **reused** by other software components
  - Provide a mechanism for publishing available services
  - Control the use of the services in a manner that **avoid security and governance problems**
Is the notion of a Service-Oriented Architecture really New and Revolutionary?

Consider for a moment an average cosmopolitan city, or ancient Rome, if you like
• It consists of individual companies that each provide distinct services that can be used by multiple consumers
• Collectively, these companies comprise a business community, that provides services based on their areas of expertise and governed by a service contract of some sorts
• By decomposing the community into specialized, individual outlets, we achieve an environment in which these outlets can be distributed

Experience gained over time provides us with the following wisdom:
• We want to allow industries or companies to interact and leverage each other’s services, however, we want to avoid tight connections or relationships that result in constrictive inter-dependencies
• We want to encourage independence, but must ensure that they adhere to certain baseline conventions, rules and contracts
"A Service-Oriented Architecture is a style of software design where services are provided to the other components, through a communication protocol over a network" - Wikipedia

Notwithstanding the fact, that this definition is from the most reliable source ever ...It does provide a different perspective

Service-oriented computing builds upon past distributed computing platforms and adds new design layers and governance considerations

From a technology architecture perspective, a SOA implementation can consist of a combination of technologies

What is required, is the understanding of how Service-Orientation shapes the technical architecture. It consists of many things:

• A Design Paradigm
• Design Principles
• Design Patterns
• Distinct Architectural Model
• Technologies
• Frameworks
**SOA Design Principles**

Service Contracts impose low consumer coupling and are themselves decoupled from their surrounding environment.

Service Contracts only contain essential information. Information about the services is limited to the published service contract.

Services are effective composition participants, regardless of the size and complexity of the composition.

Services are supplemented with communicative meta data by which they can be effectively discovered and interpreted.

Service

- **Loose Coupling**
  - Standardized Service Contract
  - Reusability
  - Autonomy
  - Composability
  - Statelessness
  - Discoverability

  The fundamental part of a service contract consists of the service description documents that express its technical interface.

  Services contain and express agnostic logic and can be positioned as reusable enterprise resources.

  Services exercise a high level of control over their underlying runtime execution environment.

  Services minimize resource consumption by deferring the management of state information when necessary.
Why and when should we use / implement Service-Oriented Architectures

#1: Monolithic Legacy application comprising of the following modules
• We’ve identified the need to add a **Service Bookings** module as part of the core application
• There is a third party company currently developing a **Workshop loading** application that is identified as a potential add-on
• Business indicated a requirement for Mobile and Web interfaces
Let’s consider the example for a moment...

- The core competency of our business is providing a Dealer Management System - Therefore adding the Service Booking module makes sense
- It has been decided that we need to closer align the business with market trends and emerging technologies
- The company that is developing the workshop loading application specialize in Vehicle Service Center Management and might develop a Service Booking module down the line

Build vs Buy

Building custom software can unlock a host of benefits, but companies should only pursue that strategy if:

- Better software can provide a competitive advantage relative to your competitors
- You are building a large business that can spread the cost of a proprietary system over a large number of clients

The best approach in this case will be to build the Service Booking module adhering to the principles of a Service-Oriented Architecture

- A Loosely Coupled module is required, as it might be replaced by the third-party application in the future
- Interaction points with the current system can be created by introducing a service layer, exposing current legacy functionality to a service interface, by means of a service façade

Does this mean that the additional complexity of web services must be introduced to enable communication between core components of the system?
Service-Oriented Architecture (SOA) has become the focal point of the IT Industry

- Partly because of creative Marketing teams understanding and exploiting the value of Buzz words
- Partly because of the attractive principles and ideals proposed by the paradigm
- Mostly because SOA is a collection of principles that is technology and vendor agnostic

Benefits:
- Services can be appended to traditional distributed applications or used as wrappers to expose Legacy System logic
- Service-Oriented is based on the notion that it is beneficial to break down large problems into a series of individual concerns – Separation of Concerns – we no from other development paradigms that this is a very successful strategy
- It provides a natural and simplistic way to integrate different technology platforms, thereby allowing utilization of the best aspects of the particular technologies

Risks:
- A Large number of implementations fail, here are some reasons and possible pitfalls to avoid
  - Impact of organizational change is underestimated – very often, people and processes are so entrenched in legacy that adhering to the SOA principles becomes impossible
  - The complexity of SOA is underestimated – Focus is placed on the Academic/Purist approach rather than common sense
  - Vendors drive the architecture – The vendor’s goal is to sell you as much stuff as possible, your goal is to implement SOA successfully

Most of the risks derive from false assumptions and what it takes to implement and operate Service-Orientation

Apply SOA where it makes sense - it is not a new hammer, and everything is most definitely not a nail! Don’t add complexity, or reinvent the wheel, for the sake of it. Base your architecture decisions on sound reasoning based on a comprehensive understanding of your business and technology requirements
A Service-Oriented Architecture is a style of software design where services are provided to the other components, through a communication protocol over a network. - Wikipedia

This definition would suggest that, ensuring the appropriate design of the services required for the new functionality, and exposing it by means of an AppServer, as opposed to traditional Web Services, we fully comply with the spirit of Service-Oriented Architectures.

For the initial development, following an incremental modernization approach, all that is required to set us up for a full blown SOA at a later stage is the following:

- Structure the new development (Modernized approach) architecture in a manner that adheres to the “Separation of Concerns” principle – Conveniently similar to OERA.
- Add a Business Service Layer that will serve as a service interface in the current implementation.
- Expose the current Legacy functionality to the Business Service layer by means of a Business Service Façade.

The reason for not considering any Web Service technology at this point, is the fact that no benefit is derived by using Web Services. At best an unwanted performance overhead and complexity will be created.
From a **Service Design Perspective**, four granular **Services** could be created

- ServiceBooking
- Communications
- Customer
- PartsInventory

From an Entity or Class design perspective, the below depicts a possible design for the new functionality – structured in a Service-Oriented Architecture
To ensure that we adhere to the SOA principles:

- **Loose Coupling** – will be achieved by the design rule, that a BusinessEntity may not call a method in a BusinessEntity from another domain directly – Calls must be made by means of the BusinessService (ServiceInterface)
- The **Service Contract** is defined by virtue of the method signatures
- **Autonomy** is guaranteed by the abstraction and encapsulation provided by the object implementation
- **Composability** is achieved by calling into an available method in the BusinessService
- **Discoverability** is provided by the method declarations in the BusinessServices
To expose the Business Services in our example to Web- or REST services (Which will be required when implementing the Web and Mobile Interfaces), two options exist

1. Expose the Methods contained in the Business Services by defining a service interface for the service end-points

2. By Adding a new **Service Layer** to the architecture
   - The service interface definition for the service end-points will then be done in this layer
   - This is my preferred method, as it provides the opportunity to do any (message) data normalization or invoke any utility services independent of any potential business layer calls

The **Services** can now be consumed Internally as **services** in the **Business Service Layer** and Externally as **Services** in the **Service Layer**
Considering the proposed additions to the core legacy application as stated before:

- If the final solution should comprise multiple technologies, the architecture as proposed in the previous slide will require no change.
- The service layer can expose the required services that will enable the disparate application technologies to consume the services (agnostic of any technology) in a standardized manner.
- No change will be required by the underlying business logic and legacy application.

It is very important to note that:

- It is not necessarily beneficial for an organization to have a vendor-diverse environment.
- However, it is beneficial to have the option to diversify when required.

When designing and architecting solutions, it is important to balance complexity with extendibility.
Utility Services

- There is not always a need to associate business logic with a model or process – it can be highly beneficial to deliberately establish a functional context that is non-business centric.
- Utility services are dedicated to providing reusable, cross-cutting functionality, such as:
  - Event logging
  - Notifications
  - Currency Conversions
  - Security
  - Etc
- Utility services are ideally application agnostic and makes its functionality available with a very specific processing context.
How Components in a SOA Interrelate

- A **Message** represents the data required to complete some or all parts of a unit of work
- An **operation** represents the logic required to process messages in order to complete a unit of work
- A **Service** represents a logically grouped set of **operations** capable of performing related units of work

A **Process** contains the business rules that determine which service operations are used to complete a unit of automation, that is, a process represents a large piece of work that requires the completion of smaller units of work

- A **process instance** can compose services
  - A Process instance is not necessarily defined by its services, because it may only require a subset of the functionality offered by the service
  - Every process instance invokes a unique series of operations to complete its automation
Goals and Benefits of SOA

Increased intrinsic interoperability

- The more interoperable software is, the easier it is for them to exchange information and the lower the risk of destabilizing the software when third-party communication is required

Increased Federation

- A Federated environment is one where resources and applications are united while maintaining their individual autonomy and self-governance

Increased vendor diversification

- This could be a game changing advantage for an organization. It provides the freedom to change, extend or replace solution implementations and technology resources, without disrupting the federated service architecture

Increased organizational agility

Reduced IT burden

- Reduction in waste and redundancy
- Reduction in overhead associated with governance and evolution
Comparison between SOA and Object-Oriented Principles

**Service Reusability**

Much of Object-Orientation is geared towards reusable objects

- Principles such as **abstraction** and **encapsulation** further support reuse by requiring a distinct separation of interface and implementation logic
- **Service Reusability** is therefore a continuation of this goal

**Service Contract**

The requirement for a **Service Contract** is very comparable to the use of **Interfaces** when building Object-Oriented applications

The **Interface first** is also considered an Object-Oriented best practice

**Service Loose Coupling**

Although the creation of interfaces somewhat decouples an object from its consumers, coupling in general is one of the primary qualities of Service-Orientation that deviates from Object-Orientation
Comparison between SOA and Object-Oriented Principles

**Service Abstraction**

- The Object-Oriented principle of abstraction requires that a class provide an interface to the external world and that it be accessible via this interface. Encapsulation supports this by establishing the concept of Information Hiding.
- Service Abstraction accomplishes much of the same as Object abstraction and encapsulation.

**Service Composability**

- Object-Oriented supports association concepts, such as aggregation and Composition. These, within a loosely coupled context, are also supported by Service-Oriented Orientation.
- That is, the same way a hierarchy of objects can be composed, a hierarchy of services can be assembled.

**Service Autonomy**

- The quality of Autonomy is more emphasized in Service-Oriented design than it has been with Object-Oriented approaches.
- Cross-Object references and inheritance-related dependencies within Object-Oriented design supports a lower degree of object-level autonomy.
Design Patterns

What is a design pattern?
- It’s a proven solution to a common problem
- Individually documented
- In a common format
- That is usually part of a larger collection

Design patterns are useful because they:
- Represent field tested solutions to common problems
- Can be used to ensure consistency in how systems are designed and built
- Can become the basis for design standards
- Because the solutions are proven, their consistent application tends to naturally improve the quality of system design

It is important to acknowledge that most patterns do not propose a black or white design option, they can often be applied at different levels. Also important to note, is the effectiveness of a given pattern will generally be equivalent to the extent to which it is realized.

There may be practical considerations that limit the degree to which a pattern can be applied in the real world.
It is important to document the use-case for a design pattern

### Legacy Wrapper:

**How can wrapper services with non-standard contracts be prevented from spreading indirect consumer-to-implementation coupling?**

<table>
<thead>
<tr>
<th><strong>Problem</strong></th>
<th>Wrapper services required to encapsulate legacy logic are often forced to introduce a non-standard service contract with high technology coupling requirements, resulting in a proliferation of implementation coupling throughout all service consumer programs.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solution</strong></td>
<td>The non-standard wrapper service can be replaced by or further wrapped with a standardized service contract that extracts, encapsulates, and possibly eliminates legacy technical details from the contract.</td>
</tr>
<tr>
<td><strong>Application</strong></td>
<td>A custom service contract and required service logic need to be developed to represent the proprietary legacy interface.</td>
</tr>
<tr>
<td><strong>Impacts</strong></td>
<td>The introduction of an additional service adds a layer of processing and associated performance overhead.</td>
</tr>
<tr>
<td><strong>Principles</strong></td>
<td>Standardized service contracts, service loose coupling, service abstraction.</td>
</tr>
</tbody>
</table>

The good news is that I won’t rumble through a thousand patterns, instead, I’ll adhere to the sound principle “**Knowledge belongs in the world**”. Therefore I’ll tell you where to find extensive documentation about Service-Oriented Design Principles.
In Summary

- As with any design and architecture implementation / change, consider:
  - The requirement and the use-case – Will it be beneficial to the software to apply the paradigm change?
  - Extensibility of the solution – Will it provide a competitive advantage, or are you merely falling into the Buzz word trap
  - The performance implication – Weigh up the performance overhead against the flexibility in the solution

- Adhere to the principles of Service-Orientation
  - Service reusability
  - Formal contracts
  - Service abstraction
  - Service Loose coupling
  - Service Autonomy
  - Service Statelessness
  - Service Discoverability
  - Service Composability

- Document design decisions
  - It might make sense to you at the time, but at some point, it will form part of a legacy system, and having information about why certain decisions was made will become invaluable

Design patterns provide proven solutions to common problems – Consider their use carefully and remember the trade of between adhering to the principles and the effectiveness of the pattern
Bibliography

Illuminating what's next™