Practical SOA for the Solution Architect
A quickstart guide to realising the benefits of Service-Oriented Architecture
# Table of Contents

Part I – Why Practical SOA? .................................................................................................................. 3
  Introduction ........................................................................................................................................... 3
  How to be a SOA-Savvy Solution Architect ....................................................................................... 4

Part II – Practical SOA at the Technology Layer .................................................................................. 5
  Essential Building Blocks .................................................................................................................... 6
    The Service Container ...................................................................................................................... 6
    The Broker ....................................................................................................................................... 7
    The Process Coordinator ............................................................................................................... 7
    Building with Blocks ..................................................................................................................... 9
  The Misunderstood Broker ............................................................................................................... 9
    “When all you have is a hammer, the whole world looks like a nail” ............................................ 9
    A Broker is not a singleton, centralised component .................................................................. 10
  Supporting Components at the Technology Layer ........................................................................... 10
    Rules ............................................................................................................................................... 11
    Data Access .................................................................................................................................... 11
    Registry/Repository ...................................................................................................................... 11
    Governance Support .................................................................................................................. 11
    Activity Monitoring ................................................................................................................... 11
    Complex Event Processing .......................................................................................................... 11
    Presentation Support .................................................................................................................... 11
    Identity and Access Management ............................................................................................... 11
    All Together Now! ........................................................................................................................ 12

Part III – Practical SOA at the Data Layer .......................................................................................... 13
  What is Wrong with this Picture? ....................................................................................................... 13
  Why Data Design is Important ........................................................................................................ 14
  Principles for Loose Coupling at the Data Layer .............................................................................. 17
    Identify Implicit Data and Make It Explicit .................................................................................. 18
    Eliminate Needless Dependencies Between Systems ................................................................... 18
    Keep Domain Data and Message Data Loosely Coupled ............................................................. 18
    Standardise Vocabulary Within a Logical Domain, Not Across the Entire Organisation ............... 19

Part IV – Industry Examples ............................................................................................................... 22
  Banking – Opening an Account ......................................................................................................... 22
    Sequence Diagram ....................................................................................................................... 23
    Analysis .......................................................................................................................................... 24
  Insurance – Providing a Quote ......................................................................................................... 25
    Sequence Diagram ....................................................................................................................... 27
    Analysis .......................................................................................................................................... 28

Part V – Conclusion and Next Steps .................................................................................................. 31
Part I – Why Practical SOA?

Introduction

There are a few problems with the practice of Service-Oriented Architecture (SOA) in the IT industry today.

- Some IT practitioners regard SOA as yet another buzzword that has come and gone, and therefore consider it largely irrelevant in today’s environment.
- Some view it as a useful theory, but are not always able to put its core principle of “loose coupling” to work in a practical sense.
- Some, of course, have never grasped this core principle at all, and are content to use SOA products from reputed vendors as an acceptable surrogate for doing SOA.

Not surprisingly, these attitudes and approaches often result in tightly-coupled “SOA” solutions, which create the unfair impression that SOA itself is ineffective.

At WSO2, we believe that SOA is a commonsense discipline that is as relevant today as it has been ever since the information age began. Done right, SOA reduces the artificial and unnecessary dependencies that, over time, creep into large-scale information systems and rigidify interactions that should ideally be flexible. The practice of SOA combats this “corporate arthritis” and brings suppleness back into the system. It improves agility, encourages functionality reuse, reduces the ongoing cost of development and operations, and reduces operational risk.

Not only do we believe in the value of SOA, we believe we have hit upon a way of explaining SOA to practitioners that makes it immediately and practically applicable, which is why we wrote this paper. SOA is too important to the bottom-line to be imperfectly understood and sub-optimally applied!

Our target audience is the Solution Architect, who is the person tasked with integrating existing and planned functional components to create an end-to-end solution that meets a business need. As that job description suggests, integration forms a major part of a Solution Architect’s job, and so it is very important to get this right. We recognise that busy Solution Architects have no time to waste on long-winded theory or even a “best practice” method that is too heavyweight. So this paper provides a practical approach to using SOA that is easily grasped and immediately applicable. It is also vendor-neutral since it deals with universally applicable concepts and also logical components that are likely to be found in any vendor’s product line-up. Think of this paper as empowering you to use your favourite vendor’s products more intelligently and effectively.

It is important to point out right at the outset that SOA’s core concept of loose coupling requires more than just technology building blocks. There is a critical data side to doing integration right, and so we will describe how to achieve loose coupling at the data level as well.

We also provide examples from a couple of vertical industry segments to drive home these concepts, so you can see how SOA could work for you. You will be a more effective Solution Architect after spending just half an hour reading this white paper.
How to be a SOA-Savvy Solution Architect

In the IT division of an organisation, it is the Solution Architect who plays a pivotal role in determining the shape of technology solutions to business problems. Business analysts talk to business representatives to understand and formalise their requirements. Technical designers and developers create the low-level technology components of the solution. But it is the Solution Architect in the middle who understands the business requirement as defined by business analysts and puts together a high-level architecture showing how existing and new components fit together to meet this requirement. The development teams then flesh out this architecture and build the physical components that deliver the solution to the satisfaction of the business.

The Solution Architect is therefore key to the quality of the technology solution. If the Solution Architect is not competent enough to architect the solution well, the organisation suffers. Many Solution Architects can put together an adequate design for a given project, but the design is often so rigidly tailored to the needs of that particular project that it creates impediments for future programs of work.

A practical knowledge of SOA principles can raise the quality of a Solution Architect such that they are able to create flexible, reusable and future-friendly designs. It is a common misconception that designs that cater to future requirements and the requirements of multiple business units must necessarily be more expensive or take longer to implement. The value of SOA lies exactly in proving that misconception wrong.

There are two specific aspects of SOA that need attention:

- At the technology level, Solution Architects need to understand the right tools for the job.
- At the data level, they need know how to reduce or eliminate unnecessary dependencies between systems.

In our experience, many solution designs that are ostensibly SOA-based suffer from the inappropriate use of technology, analogous to using a screwdriver instead of a hammer to drive in a nail. Using the wrong tool for the job is a fairly common mistake, as the popular term “ESB-itis” suggests. This paper provides a simple vendor-neutral thumb rule to determine the right SOA technology component to use in a given situation.

Many solution designs also turn out to suffer from rigidity caused by tight coupling at the data level, negating all the benefits that derive from investing in SOA technology. Inadequate attention to data is expensive, but so is over-design, and both are common mistakes. This paper also describes a simple and cost-effective way to design data for effective and flexible integration.

The following diagram shows how effective SOA requires correct decisions to be made at two different layers – technology and data. SOA products provide support for loose coupling at both layers. However, they cannot prevent poor design at either layer, and so the mere use of SOA products, however reputed, is no guarantee of success. The SOA architect needs some special skills to exploit these tools to maximum benefit.

Fortunately, these skills and the concepts behind them are not hard to master. This paper will teach you to think “Practical SOA”.
The Two Layers of a Solution Design based on SOA Principles

Part II – Practical SOA at the Technology Layer

Essential Building Blocks

Let’s start with the technology layer, since it is more tangible and the concepts here are easier to grasp. From practical experience, there are three core SOA technology components that are most frequently used, and these are:

- The Service Container
- The Broker
- The Process Coordinator

We have deliberately used terms that are generic and vendor-neutral. You can map these terms to specific brand names in your favourite vendor’s product line. There are, of course, many other components that refine and enrich a SOA ecosystem, but it is important to recognise that these are the three core building blocks, some or all of which you will need in every situation that requires integration. The other components can be viewed as the supporting cast for these main actors, and we will look at them a bit later.

Let’s talk about when you would use each of the three core SOA components.
The Service Container

Sometimes, the business requirement you have to meet through your solution design demands a functional component that does not yet exist, is not available off the shelf, and must therefore be developed in-house. You may find it worthwhile to expose this new component’s functionality in a way that can be consumed by other components, perhaps even beyond this particular project. In other words, this business logic is to be exposed as a service, and a **Service Container** is what you need to host this logic.

The diagram above shows that custom functionality, when hosted within a Service Container, is exposed as a reusable service to other components in the system.

We’ll look at how such services are used after we cover the remaining two core components, but what we’re essentially saying is that you should try and build new functionality in the form of readily integrable Lego™ blocks, and the Service Container lets you do just that.

The Broker

The Service Container is a useful component when you need to host a particular piece of functionality that you had to develop yourself. In practice, you do not generally develop complete, end-to-end solutions from the ground up. The solution you put together will usually be able to exploit pieces of logic or data that already exist somewhere in the organisation, within some legacy system or application. However, you will invariably come up against some complications that prevent you from readily accessing and using such components within your solution.

- Perhaps the legacy component that hosts your required logic speaks a specialised or proprietary protocol. To use an electrical analogy, you need an adapter to let you plug into this differently-shaped socket. An **adapter** can translate proprietary protocols into more open ones.

- Perhaps the data that this legacy component provides is not in exactly the right format that you can readily use in your solution. Again, to use an electrical analogy, you need a transformer to step its voltage up or down. **Transformers** can enrich data and massage it into a form that is more readily understood and usable.

- And perhaps there are reasons why you do not want your solution to directly access this legacy functional component. You may want to hide, proxy or throttle access to it. An extension block or power board lets electrical components tap into power without directly accessing a wall socket. In the SOA context, we call such a component a **mediator**.
There is something in common between the functions of an adapter, a transformer and a mediator, and a single component can usually perform all three functions at once. We call this combined component a **Broker**.

The Broker may be the most common component you use in your solutions, because it has the flexibility and power to turn your legacy components into Lego™ blocks, and most of the components in a solution design tend to be legacy (i.e., pre-existing) components.

Having said that, you must be careful not to use a Broker in situations where it is not appropriate. Hosting custom logic is one such case, where a Service Container is the more appropriate tool to use. Let’s look at another case.

**The Process Coordinator**

Lego™ blocks are great for building solutions quickly, and a Broker can even tie together services in a rudimentary way, but sometimes you don’t know what you need in advance. You’re building a house, but should it have 2 bedrooms or 3? Should the roof be tiled or metal? Perhaps the nature of the problem is such that you won’t know the answers to these questions until you actually start building!

When you need to pull components together dynamically, based on a status that can only be evaluated at run-time, you need a **Process Coordinator**.

**Building with Blocks**

To summarise, a Solution Architect needs to put together existing and planned functional components to create an end-to-end business solution. Quite often, the exact components and the order in which they are invoked can only be determined dynamically, at “run-time”.

Accordingly, we see the Process Coordinator as the highest-level component of the three. It ties the other two components into a process that delivers the required end-to-end business functionality. Service Containers
and Brokers provide Lego™ blocks that may be used by themselves, in static combinations, or in dynamic combinations under the control of a Process Coordinator. Service containers host pieces of logic that are designed from the start to be used as Lego™ blocks, while Brokers convert components that were not so farsightedly designed into Lego™ blocks too. Needless to say, the logic hosted within a Process Coordinator is itself exposed as a service and may therefore be used in turn as a building block in its own right.

Once you have Lego™ blocks and the means to put them together, either in advance or on-the-fly, you can build business solutions quickly and cheaply, and maintain them more effectively as well.

The following cheat sheet summarises the reasons for using each of these components, and the diagram that follows it shows one possible way for the three core components to interact.

**When to Use What – A Cheat Sheet**

<table>
<thead>
<tr>
<th>Situation</th>
<th>Component to Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Business logic does not exist before and is not available off the shelf. Logic has to be developed in-house and must be hosted in a way that makes it easy for other components and systems to invoke its functionality.</td>
<td>Service Container</td>
</tr>
<tr>
<td>2  Business logic or data exists on a legacy system, but is not readily usable because a specialised or proprietary protocol is required to access it. An “adapter” component seems to be required to translate from that protocol to a more open one.</td>
<td>Broker</td>
</tr>
<tr>
<td>3  Data exposed by a legacy system is not in a suitable format for consumers. A “transformer” component may be required to change the structure of the data or enrich it in some way. The legacy component’s functionality should not be accessed directly by consuming applications in a point-to-point manner. The access needs to be proxied, either for location-transparency, caching, throttling or some other reason. A “mediator” function seems to be required to decouple the service consumer from the service provider.</td>
<td>Process Coordinator</td>
</tr>
<tr>
<td>4  More than one activity needs to be performed to satisfy a business requirement. The exact sequence of steps is not always known in advance and may need to be determined dynamically, at run-time.</td>
<td>Process Coordinator</td>
</tr>
</tbody>
</table>
A sample way to pull the three core SOA components into a solution design

The Service Consumer sees a single interface with which to invoke business logic. The business logic is dynamically determined by a Process Coordinator that decides which services to invoke, as part of a business process. Some of these invoked services have been custom-built and hosted in a Service Container. Others are hosted in a raw form within a legacy system and are converted into usable services by a Broker.

The Misunderstood Broker

We find that two of the most common mistakes made by Solution Architects at the technology layer pertain to the use of the Broker component.

“When all you have is a hammer, the whole world looks like a nail”

Brokers are very powerful, and SOA vendors often provide extensions to make them even more capable, but those extensions are often misused in solution designs. Obvious examples are the use of a Broker in place of a Service Container or a Process Coordinator. Oftentimes, this is because developers are very familiar with the Broker’s features and can “hack it” to perform these functions as well, but such hacks often create tight coupling in addition to impacting the Broker’s performance. You maximise flexibility and optimise performance when you use the right tools for the job, so ensure that developers are equally proficient in the use of all three core SOA components.

A Broker is not a singleton, centralised component

Perhaps because of the prohibitive cost of many commercially available Brokers, there is a tendency to install just one instance of this component for the entire organisation and push all traffic through it. The obvious implications of such a centralised, hub-and-spokes architecture relate to performance (i.e., scalability of throughput) and to availability (because of its emergence as a single point of failure).
However, there is nothing in the Broker’s functionality that implies it must be used in this manner! On the contrary, the Broker’s mediation, transformation and adapter functions are best performed closest to the endpoints where functionality is either exposed or consumed. This decentralised or “federated” architecture as illustrated below also ties in very well with a pragmatic data model, as we explain later.

The SOA landscape is meant to be “flat” (i.e., all service endpoints appear the same regardless of where they are physically hosted or exposed), and a registry (discussed in the next section) can help you with the governance of a federated ecosystem such as the one shown above. If economics is the reason for centralising the Broker, talk to your vendor about their support for federation and their licensing model. Alternatively, try to procure some of your Broker instances from more affordable sources, because forced centralisation violates SOA tenets and impacts performance and availability.

**Supporting Components at the Technology Layer**

The Service Container, the Broker and the Process Coordinator are the three basic components in a SOA ecosystem, and if you understand when to use each, that will help you build the skeleton of your solution design, and in many cases, that is itself sufficient.

However, there are other components that help you flesh out your solution further and perform certain tasks more elegantly. These are not central in the way the three core components are, but they are also useful.

**Rules**

Logic can be specified in many ways. Procedural logic written in traditional programming languages like Java or C# is appropriate in many situations, and a Service Container can be used to host such logic. However, certain types of logic are best expressed as “rules”. Rules engines are specialised Service Containers that are optimised to store rules and to evaluate a hierarchy of sometimes conflicting rules to arrive at decisions. As with any Service Container, these decisions can form inputs to other services. Process Coordinators often consult rules engines to decide on next steps.
Data Access

Data storage is an important component to integrate when building end-to-end solutions. Brokers can certainly access data stores through specialised adapters. However, it is often a convenience to expose a data store as a set of “data services” that can be directly consumed without the need to understand internal data schemas or special access protocols, even open ones like SQL. Data Access can be provided by dedicated Brokers.

Registry/Repository

Services and other SOA artifacts can be stored as well as referenced through a centralised component that other components in a SOA ecosystem can look up. The registry component holds references to artifacts. The repository component holds the artifacts themselves. A registry helps in governance because the physical location of the artifact is no longer an issue. Managing services hosted on or exposed through multiple instances of Service Containers, Brokers and Process Coordinators is no more difficult than managing them on a single node. As the number of service artifacts and nodes grows, a registry/repository gradually turns from a luxury to a necessity.

Governance Support

While the principles of SOA are simple enough, they can be unwittingly violated even by the most conscientious development teams. Good discipline is essential, but it cannot be left to extraneous control systems. Incorporating good discipline into the very protocol of development and operations is what governance is all about. Many SOA tools offer governance support, often by leveraging rules and registries/repositories. Front-end development/operations tools are usually designed to work aligned to these processes.

Activity Monitoring

Building a SOA ecosystem is one thing, but ensuring its health under load is another. Good systems are often victims of their own success, and their popularity could increase the load placed on the supporting infrastructure. A SOA tool suite normally also provides the means to monitoring whatever is happening within the system, both at a technical level (e.g., the number of messages flowing through a Broker) and at a business level (e.g., the number of customer enquiries converted into sales).

Complex Event Processing

As organisations become more mature at managing their SOA ecosystem, they feel the need for greater sophistication in their ability to detect subtle or complex situations. These could be in the areas of fraud detection, system health monitoring, changes in customer behaviour, etc. Complex Event Processing refers to the ability to model combinations of events and detect the occurrence of those patterns during the regular use of the system. While Process Coordinators and Rules engines can often provide such capability, dedicated CEP engines are better specialised for the task.

Presentation Support

Most middleware is tucked away unseen to the end user, because it typically deals with the integration of back-end systems. However, there are opportunities to exploit this integration capability to provide support to user interfaces as well. The more seamless these tools are, the more powerful they can be in linking the user tier of an application to the heart of the business process.
Identity and Access Management

Security is an important part of integration, which we normally gloss over in our discussion of services that are universally consumable. Having made business logic functionality readily accessible, we also need to ensure that it can only be consumed by the right actors. Identity and Access Management (IAM) tools, when well-integrated with other SOA components, can help to ensure that only authorised users access certain functions and data.

All Together Now!

The following diagram provides an example of how the basic and supporting components of SOA can be put together in a solution design.

The Core and Supporting Components of the SOA Technology Layer
Part III – Practical SOA at the Data Layer

Now that we understand the core and supporting building blocks of a SOA solution at the technology layer, let’s look at the data layer. We need to ensure that the gains we have made through the use of appropriate SOA technology components are not negated by tightly-coupled data design.

**What is Wrong with this Picture?**

Consider this real-life example of a SOA implementation. An organisation purchased a business system in the form of a packaged product. The business process that this system was meant to support required that related documents (files) be stored alongside the application’s structured data. (To make it tangible, consider a bank storing income certificates as part of a loan application, or an insurance company storing photographs as part of a claim lodgement.)

Since the packaged product did not itself support the storage of documents but could hold references to their location, the organisation decided to use its existing Enterprise Content Management (ECM) system for document storage. This is typical of the way organisations work, because they seek to leverage the assets they already have. The business application was expected to hold references to related documents held in the ECM system, and as we know, this approach was supported by the product. The ECM system had a proprietary API to check in documents and to retrieve them. When a document was checked in, it returned a unique ID for it. To retrieve a document from the ECM system, its document ID needed to be provided.

The business application, being an independent product, did not natively support the ECM API.

From our previous discussion on core SOA components, we can readily see that a Broker with its built-in adapter functionality would probably be required to integrate these two systems. And indeed, this is the way the two systems were integrated by the organisation’s Solution Architects.

The business application now calls a service exposed by the Broker to check in a document. The Broker uses its adapter to invoke the ECM system’s corresponding proprietary API. The ECM system returns a unique ID for the checked-in document, which the Broker returns to the business application as the service response. The business application stores this document ID reference as part of its structured data.

---

1 The application probably passes in the location of the document within a local filesystem, and the check-in process copies this to a filesystem within the control of the ECM. However, for the purpose of this analysis, no harm is done if we consider the document itself to be streamed through the Broker to the ECM.
When the business application wants to retrieve a document, it makes a call to the appropriate service exposed by the Broker, passing in the document ID reference that it holds. The Broker makes the corresponding proprietary API call to the ECM system as before and returns the document to the business application.

Any data transformations required to translate the document’s metadata from the application’s own format to the format required by the ECM system are performed by the Broker.

The question is – Is this a good design from a SOA perspective?

**Why Data Design is Important**

Disappointingly, the organisation suffered from multiple operational issues on account of this design. Why was this the case when their Solution Architects had clearly implemented a loosely-coupled integration mechanism between the two systems?

First of all, let us drill a little deeper into the characteristics of the data relating to these simple services.

- **Data references are tightly coupled to physical instances**

  Remember that a physical instance of the ECM system generates a unique document ID when a document is checked in. Hence the document ID is only unique within that instance of ECM, not across the entire organisation. The organisation has several instances of ECM serving various other applications as well.

- **Implicit data dependencies exist**

  Since the document ID only makes sense within a particular ECM instance, it is obviously not enough for a business application to hold a reference to just the document ID. It also needs to hold an implicit reference to the ECM instance within which the document is stored. Each business application therefore also holds a global reference to an associated ECM instance as a configuration parameter. The Broker only knows which ECM instance to go to because the business application passes in that information with each service call.

- **Transformation logic is tightly coupled to physical instances**

  The Broker needs to apply different transformation logic for document metadata depending on the business application that calls it. But since the associated ECM instance is now a surrogate for the business application (there is an assumed one-to-one mapping between the two in the solution design), the Broker has been designed to use the ECM instance parameter to determine the transformation logic to be used. Now the transformation logic is tied to the instance of ECM, even though there is no logical connection between the two.

While this sequence of constraints and design decisions is perhaps understandable at a certain level, it leads to many problems:

- **Unnecessarily large number of ECM instances**

  The one-ECM-instance-per-application pattern became entrenched because of the customised transformation logic, and thereafter, every new application began to be provisioned with its own instance of ECM. This had an operational support cost as well as a software licensing impact.
• Inability to load-balance across ECM instances

   It would have been desirable to provide more than one ECM instance in a load-balanced configuration behind applications that had a high document throughput. Unfortunately, the implicit association of an application with a single ECM instance made that impossible.

• Inability to share an ECM instance between applications

   There were many applications that did not have high document volumes and could easily have shared an ECM instance from a performance perspective. But the use of the ECM instance as a surrogate for the business application in order to determine the right transformation logic to use, made this impossible.

• Inability to move a document to another ECM instance once checked in

   Some rationalisation of instances should have been possible, but since a fresh document ID is generated when a document is moved to another ECM instance, the corresponding business application had to be updated with the new document ID. The logistics of such an update and the possibility of disruption to the work of the business user made this unviable.

• Inability to implement innovative archiving solutions

   Closely related to the use case of moving documents from one instance to another was that of archiving documents off an ECM instance meant to hold a small number of “current”, frequently-accessed documents to one holding older and less frequently-accessed ones. This was also rendered impossible for the same reason – visibility to the business application and possible disruption to the business user.

Quite clearly, the failure to design the data side of this integration with care has played havoc with the flexibility and agility of the organisation, and has raised development as well as operational costs significantly.

In hindsight, the data elements in this interaction should have been designed to be decoupled from the physical components of the system. The business applications should have been shielded from a knowledge of ECM instances and instance-specific document IDs, and instead been presented with document IDs that were unique across the entire organisation. That would have neatly decoupled the ECM instances from the business applications, allowing all the optimisations and operational housekeeping tasks that proved impossible with the current design.

Such a design would have looked like the following diagram.
Possible Solution Design for Document Check-in

The simple document check-in service of the Broker is now front-ended by a process hosted on a Process Coordinator.

When invoked, the first step of this process is to consult a Rule to determine the appropriate ECM instance to use. The ECM instance is no longer an implicit reference within the business application. Another rule provides the transformation logic to be used. Once the document is checked into the right ECM instance using the right transformation logic and an instance specific document ID is returned, this is mapped to a newly-generated, globally unique ID that is passed back to the business application. The mapping is stored in a separate database exposed as a Data Service. Now the business application is decoupled from the ECM instance.

All of the optimisations discussed earlier are now possible. The total number of ECM instances can reduce. High-volume applications can use more than one ECM instance for load-balancing through appropriate rules, and low-volume ones can share an ECM instance. Documents can be moved between ECM instances at will. Only the mapping held in the mapping database will change and business applications will be unaffected. Similarly, various archiving strategies can be implemented without impact to applications. This is a much better SOA solution because it has achieved loose coupling between systems.
How to Think about Dependencies and Loose Coupling

With a SOA-based design, a “hinge” is implemented through a service contract that promises to shield other parties from internal changes, and appropriate mapping tools that make the required adjustments at the boundary when such changes occur. Thus, a constant interface is exposed to the outside, ensuring stability of the interaction in the face of change.

Principles for Loose Coupling at the Data Layer

It may be disheartening to realise that what looked like an open-and-shut approach to implementing loosely-coupled interaction at the technology layer should run so badly aground because of a relatively minor oversight in data design. The serious negative implications were in no way obvious up front. How do we guard against them?

Here is a checklist of four simple rules that can help to avoid such pitfalls.

- Identify implicit data and make it explicit
- Eliminate needless dependencies between systems
- Keep domain data and message data loosely coupled. Rely on data mapping rather than data generation or derivation to achieve this.
- Standardise vocabulary within a logical domain, not across the entire organisation.
Identify Implicit Data and Make It Explicit

You need to identify every piece of information that a system relies on for its operation. In the document check-in example, the ECM instance ID was a piece of information that turned out to be required, even though it was never obvious at the outset. Implicitly required data needs to be made an explicit part of some service interface in the process, as we did later using the rules engine.

Eliminate Needless Dependencies Between Systems

Very often, your investigation will uncover data items that systems depend upon, but the rationale for their existence is either weak or a relic of some previously valid reason that no longer applies. In the previous example, a onetime assumption that the ECM instance could be assumed to be a surrogate for a business application created a dependency that wasn’t justified. In such cases, it is best to break the dependency as expeditiously as possible, because it contributes to rigidity and operational cost. However, it’s not always easy to do this, for any of the following reasons:

- It takes effort to rationalise a system, and you may not have the time or money to do it now.
- The change may impact working production systems, and there could be resistance to making the change because of risk ("If it ain’t broken, don’t fix it"). Regression testing is required to mitigate this risk, and this again imposes cost and time constraints.
- The change may impact other groups, including business users, and there is usually strong resistance to such disruption unless there is an immediately visible benefit. “Cleaning up an interface” is generally not recognised to be a valid reason in most organisations.

Nevertheless, it is important to identify and document all instances of unnecessary dependencies in the ecosystem. These dependencies must be eliminated whenever the opportunity presents itself, such as when a related change is to be made.

Keep Domain Data and Message Data Loosely Coupled

Some pieces of information refer to data held inside a local domain. Exposing this data outside the domain (through a service interface) allows external systems to become dependent on it in some way. If this internal data changes (as it should freely be able to), those external systems will break. As we saw with the instance-specific document ID, this was something that should have been hidden within the domain and not made visible to client applications.

The key term here is “loosely coupled”. Domain data and message data (i.e., data exposed through the service interface) are neither identical nor unrelated. They are related, but not in such a way that one of them can be directly derived from the other. They are therefore neither tightly-coupled nor decoupled, but loosely-coupled.

Some SOA tools allow message data to be automatically generated from domain data or vice-versa, but this is bad practice and must be avoided. Mapping, on the other hand, maintains the association between two pieces of data, yet not in a way that allows one of them to be derived from the other. Mapping is a good way to achieve loose coupling in most situations. The data service we created to manage the mapping between globally unique IDs and instance-specific IDs is one such example. The two IDs are related, but neither can be derived directly from the other.
**Standardise Vocabulary Within a Logical Domain, Not Across the Entire Organisation**

Clearly, a system with no standard vocabulary across its services would be chaotic and incomprehensible. However, it is hugely expensive to try and standardise a vocabulary across an entire organisation. Both extremes lead to high cost, in development as well as in operations. The pragmatic alternative is to standardise smaller groups of vocabularies corresponding to logical business domains. For example, a bank has logical domains such as retail banking, institutional banking and wealth management, each with its own specialised technical terms. Rather than attempt to create a standard vocabulary across the entire bank, let each domain define its own terms. This works much better, logistically as well as politically!

All service-providing applications within any of these domains must speak in the vocabulary of that domain, so they appear consistent from outside the domain. Any application that consumes services from another domain is responsible for understanding the vocabulary of that other domain.

A “canonical data model” is therefore unnecessary and a waste of effort. The federated alternative is simpler and more practical. It also maps nicely to a federated broker model, with data transformations done close to the end points (see following diagrams). A Broker exposes local services to the outside world in a way that conforms to the local domain data model. On the service consumption side, the same Broker converts the data models of external services to the local domain data model before letting local service consumers access the data.

The above four data-related rules should help you achieve loose coupling at the data layer.
Impractical Approach – Boiling the Ocean in Search of a Canonical Data Model (and Suffering the Performance and Availability Drawbacks of a Centralised Broker)

* All interactions between systems go through a centralised Broker.

* All data is translated by the Broker to/from a single canonical data model and the various data models understood by individual systems.
Pragmatic Approach – A Federated Data Model Based on Logical Domains, and a Correspondingly Federated Broker Configuration

* A Domain Broker enforces a Domain Data Model for a group of related systems.

* External systems see a consistent Data Model (but different from their own) when interacting with any service hosted on systems within that domain and exposed through that domain's Broker.

* It is the responsibility of systems to understand other data models when interaction crosses domain boundaries (which is relatively rare). Local Domain Brokers perform any required translations.
Part IV – Industry Examples

Now that we know what the core SOA components are and have a rough idea how to use them, let’s look at specific business process examples from a couple of vertical industries to make this abstract learning more concrete.

Banking – Opening an Account

Here is a simplified description of a common process in retail banking.

When a potential customer walks into a bank branch to open an account, the first thing the bank does is establish the person’s identity. In these days of heightened security, laws in many countries require “strong” identification, which precludes opening accounts online and requires physical presence at a branch. It also often means two independent forms of photo ID with signatures on them, such as a passport and a driving licence. In the absence of one of these, the person will have to provide two or more additional pieces of identification that notionally add up to the same “strength”.

Once the person’s identity is established, the branch staff perform a query against a customer database to see if the person is an existing customer, based on their name, address and other attributes. Legacy banking systems are based on account numbers, but banks today are focused on getting a “single view of the customer”, which requires the generation of customer numbers. It’s therefore important to ensure that a customer is not assigned more than one customer number by mistake. The customer may not provide this information reliably, so this check has to be foolproof.

If the person is a prior customer, the customer number is retrieved from the customer database. If not, a fresh customer number is generated. This customer number is then associated with the new account number of the account that is to be opened. Accounts are still held on thirty year old mainframe systems, which alone can generate fresh account numbers.

A debit card is normally issued with the first cheque or savings account that is opened. However, not all branches have the equipment to generate and issue them on the spot. Hence cards are issued centrally and sent to the customer’s address by snail mail. If this is an existing customer, the new account is “linked” to their existing debit card on the spot. For new cards, the PIN also needs to be disclosed to the customer in a secure manner. Most branches are unable to provide secure PIN printing facilities, so this too is centralised. A special two-layered paper is used to print the PIN, such that the number is not visible on the outer layer and any attempt to peel this layer to see the PIN on the layer below is tamper-evident. The debit card and PIN are mailed separately to the customer. The card is activated when the PIN is first used, and this must be done within a certain period.

The mainframe can only send and receive messages over a message queue, and uses a COBOL copybook format to structure its data inputs and outputs. The card system is an off-the-shelf product that supports web services, but these conform to a vendor-provided schema that is different from the rest of the bank’s systems.

As a Solution Architect, how will you design a solution for opening an account?
Sequence Diagram

Customer
- Request to open account
  - Request for identification
    - Two forms of strong ID
      - Manually verify ID
        - Initiate account opening with person details
          - Check if existing customer
            - If new customer, request customer creation
              - Customer ID
                - Request account creation
                  - Account created
                    - If existing customer, link account to debit card
                      - If new customer, request debit card issue
                        - New card number
                          - Link account to debit card
                            - Mail card to customer
                              - Mail PIN to customer
                              - Issue debit card and PIN
                                - Request PIN mailer
                                  - Print PIN
                                  - Link account to card
                                    - Create new account
                                      - Create new customer record
                                        - Customer Master (mainframe)
                                          - Account System
                                            - Card System
                                              - PIN Mailer
                                              - Branch Staff (Acct Opening Process)
                                                - Customer
                                                  - Customer
                                                    - Account System
                                                      - Card System
                                                        - PIN Mailer
                                                          - Branch Staff (Acct Opening Process)
                                                            - Customer
                                                              - Customer
Analysis

The entire process of verifying identity, elaborate as it is, is a purely manual function and will not form part of the process we are designing. Our process starts only after the customer’s identity has been established. There is only one process (“Open Account”) that needs to be initiated, and this consists of a number of steps, which would be natural points at which services are invoked. The functions performed at these various steps are:

1. Check if existing customer: This is the first thing that needs to be done, because the subsequent processing for prior and new customers is different. This query is to be made against the customer database, which requires a Data Service in front of it to turn it into a true service.

2. Create customer: This service is only invoked if the customer is new. This is once again a function that needs to be implemented against the customer database, so another Data Service is required to turn it into a true service.

3. Create account: This needs to be done regardless of whether this is an existing or new customer. Accounts are maintained on the mainframe, and account creation can only be invoked by passing in a special message formatted as a COBOL copybook sent over MQ. A Broker is required to act as an adapter to the MQ interface as well as a transformer to present the COBOL copybook syntax as a more open XML document.

4. Link card to account: This is done for both existing and new customers, but invoked at different points in the process. Again, this is performed on the mainframe, and a Broker is required to mediate the interaction.

5. Issue card: This is for new customers who don’t already possess a debit card. The actual processing occurs on a card system that exposes a web service. However, the data elements of this service don’t conform to the data model of the retail bank. A Broker is again required to transform the XML documents supported by the card system to equivalent ones that conform to the bank’s data model.

As a matter of detail, the five functional steps above are not called services in Web Services terminology but operations. Services are logical groups of operations, and so the actual design of services could look like this:

<table>
<thead>
<tr>
<th>Process</th>
<th>Service</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Open Account</td>
<td>Customer</td>
<td>1. Check if existing customer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Create customer</td>
</tr>
<tr>
<td></td>
<td>Account</td>
<td>3. Create account</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. Link card to account</td>
</tr>
<tr>
<td></td>
<td>Card</td>
<td>5. Issue card</td>
</tr>
</tbody>
</table>

The following diagram shows how these components hang together.
Bank Account Opening – High-Level Solution Design Showing SOA Components

1. Check if existing customer,
2. Create customer,
3. Create account,
4. Link card to account,
5. Issue card.
Insurance – Providing a Quote

In the General Insurance industry, there are many types of products such as Home and Contents Insurance, (Vehicle) Compulsory Third Party Insurance, Workers Compensation Insurance, etc.

Many of these products allow potential policyholders to ask for a quote online and then purchase the policy if they find the quote acceptable.

We will look at a slightly more complex case where the product is sold through a broker. The broker is approached by a client who asks for quotes from various insurance companies and selects the most suitable one. Looking at the process from the perspective of one of the insurance companies providing quotes, these are the functions that need to be supported:

The broker enters their own account details into an insurance front-end application, so they can be paid their commission later. They also enter the details of the client and the details of the insurance product to be purchased. These details are sent to the insurance company’s underwriting system. The underwriting system consults a rules engine and provides a quote, which the broker’s front-end application then displays. Sometimes the rules engine refuses to quote, perhaps because the client represents too high a risk. Sometimes, the client turns out to be a borderline case, in which case the underwriting system, on the advice of the rules engine, refers the case to a human underwriter. The broker’s system is told that a response will be provided within a certain time.

The human underwriter has an inbox with borderline cases like this. When the underwriter has finished deciding whether to quote or not, and how much, this response is provided to the broker’s system through the broker’s inbox. The broker then informs the client through a phone call or email.

If the company provides a quote and the client finds it acceptable, they may then purchase the policy (using a payment gateway). The underwriting system then gets a back-end policy system to convert the quote into a policy.

This process consists of a couple of human tasks in the workflow. The underwriter as well as the broker have an inbox of tasks which drives their work. There is an escalation from an automated process to a manual process that needs to be managed seamlessly.

The underwriting system could be a relatively recent and modern system supporting web services, but the back-end policy system is probably a decades-old mainframe that only supports a COBOL copybook-over-MQ interface.

What would a SOA-based solution look like for the insurance company?
Analysis

This is a more complex case than the account opening process of a bank. For a start, there are two distinct processes - “Get quote” and “Purchase policy”. Only some quotes provided in response to the first process result in a purchase request by a customer, which then kicks off the second process.

The quote process is also complex because of the presence of manual processing as a key part of it. However, a simple way to visualise human tasks is to see the interaction as occurring through store-and-forward mechanisms, because it is unlikely that human beings can respond in the timeframes associated with automated systems. The standard store-and-forward mechanism used for human actors is the “inbox”, which holds task items for them to process. When they eventually process a task item, they either invoke a callback service to signify completion, or update the item’s status so that the controlling process can proceed to the next step.

The automated services in the system should by now be easy to identify.

1. Get quote decision: This is best implemented on a Rules Engine, which is a specialised Service Container.

2. Make payment: This is a canned service provided by a payment gateway through a proprietary interface. A Broker will be required to convert it to a service that conforms to the insurer’s data model.

3. Register quote: The mainframe needs a record of every quote provided, and the only way to access the mainframe is through a COBOL copybook-over-MQ interface. A Broker is required to act as an adapter and transformer to present a true service interface for this function.

4. Convert quote to policy: This is another mainframe function that a Broker is required to convert to a true service.

The underwriter as a human actor participates in the quote process through an inbox, and a Data Service makes this inbox readily accessible to the quote process as well as the underwriter’s own front-end application. The following service operations are required on the underwriter’s inbox:

1. Refer quote: When the Rules Engine fails to provide a clearcut decision, the quote process must escalate the decision to the human underwriter by placing the quote request in their inbox.

2. Get referrals: The underwriter’s front-end application needs to query the inbox at regular intervals to see the list of task items in it, and the Data Service provides this function.

3. Update decision: The underwriter’s front-end application updates the status of the task item (the quote request) with their decision.

4. Get decisions: The quote process periodically checks the underwriter’s task list to retrieve the decisions that have been made.
Similarly, the broker is also a human actor who needs an inbox. The Data Service that wraps this inbox needs to provide two functions:

1. Notify quote: When the quote process finds that an underwriter has made a decision on a quote request, it places that quote (or the refusal to quote) into the broker’s inbox.

2. Get quotes: The broker’s own front-end application will need to query their inbox periodically to see which quote requests have been responded to.

The processes, services and operations in the system are listed below:

<table>
<thead>
<tr>
<th>Process</th>
<th>Service</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Get quote</td>
<td>Quote Engine</td>
<td>A1. Get quote decision</td>
</tr>
<tr>
<td></td>
<td>Underwriter Inbox</td>
<td>A2. Refer quote</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A3. Get referrals</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A4. Update decision</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A5. Get decisions</td>
</tr>
<tr>
<td></td>
<td>Policy</td>
<td>A6. Register quote</td>
</tr>
<tr>
<td></td>
<td>Broker Inbox</td>
<td>A7. Notify quote</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A8. Get quotes</td>
</tr>
<tr>
<td>B. Purchase policy</td>
<td>Payment</td>
<td>B1. Make payment</td>
</tr>
<tr>
<td></td>
<td>Policy</td>
<td>B2. Convert quote to policy</td>
</tr>
</tbody>
</table>

The diagram below illustrates how these processes and service operations form part of the insurance quotation system.
Part V – Conclusion and Next Steps

This white paper has provided you, the Solution Architect, with a concise and practical way to “do SOA”. As a Solution Architect, your ability to apply SOA principles in your solution designs crucially impacts your organisation’s agility and ongoing operational costs.

By now, you understand that SOA has a technology as well as a data layer, and have grasped the core concepts that underpin each of these layers.

At the technology layer, there are 3 core components that are most frequently used in SOA-based designs:

- The Service Container
- The Broker
- The Process Coordinator

There are other supporting components as well. You should by now have understood the appropriate context to use each of them.

At the data layer, there are four key principles to ensure that systems remain loosely-coupled.

- All dependencies should be made explicit. No implicit dependencies should be allowed to remain.
- All legitimate dependencies between systems should be in the service contract and the rest eliminated.
- The message data model should be only loosely coupled to the domain data model. Mapping or association should be used to link the two rather than generation or derivation, since the latter techniques are a form of tight coupling.
- Message data models should have an intermediate level of granularity centred around business domains. An all-encompassing data model is too ambitious and expensive and also likely to fail. An excessively narrow one will not deliver the benefits that come from standardising business vocabulary.

These simple yet powerful ideas are key to effective SOA design, and you now have these conceptual tools in your repertoire of skills. The two vertical industry examples we worked out should have given you a level of comfort in understanding how to apply these concepts. We hope this enables you to hit the ground running with your next project, so you can intuitively design a solution that conforms to SOA principles.

We have expressly designed this method so that you will be able to apply these skills equally effectively, regardless of the SOA tool suite your organisation uses. Of course, if you have not yet committed to a product suite, you should know that WSO2 has a full line-up of products that correspond to the core and supporting technology components described in this paper. However, a description of these products belongs in a separate document altogether! Watch this space.

For more information about WSO2 products and services, please visit [http://wso2.com](http://wso2.com) or email [bizdev@wso2.com](mailto:bizdev@wso2.com)