

Chapter 1 – An Architectural look at SOA

| | | |
|-------|--|----|
| 1.1 | Software Architecture | 2 |
| 1.1.1 | Constraints & Architecture Principles | 4 |
| 1.1.2 | Patterns and Anti-Patterns | 5 |
| 1.1.3 | The Software Architecture Requirements | 6 |
| | Quality attributes and Scenarios (identifying which patterns to apply) | 7 |
| | Scenarios Structure | 10 |
| 1.1.4 | Technology's impact on Architecture | 11 |
| | Mapping technologies to Architecture | 11 |
| | Technology & Architecture | 13 |
| 1.2 | Challenges of distributed systems | 14 |
| 1.2.1 | Playing in a Distributed Environment | 14 |
| 1.2.2 | Being a distributed System | 16 |
| 1.3 | SOA - an Architecture Style | 18 |
| 1.3.1 | What is SOA | 19 |
| 1.3.2 | SOA Benefits | 21 |
| 1.3.3 | Business Aspects of SOA | 22 |
| 1.4 | Summary | 23 |
| 1.5 | Bibliography | 25 |

Several years ago I stumbled on a little book called "Mots d'Heures, Gousses, Rhames" that supposedly had these medieval French poems - here is one:

*Lit-elle messe, moffette,
Satan ne te fête,
Et digne somme coeurs et nouez.
À longue qu'aime est-ce pailles d'Eure.
Et ne Satan bise ailleurs
Et ne fredonne messe. Moffette, ah, ouais! (Rooten, 1980)*

I was going to buy that book for a Francophile friend of mine but even with my limited French I noticed something was a little off – it took me a while, but then it all clicked when I remembered I read something similar once (think about it for a while and - if you don't see what I mean check the footnote¹)

¹ "Mots d'Heures, Gousses, Rhames sounds a lot like "Mother Goose Rhymes" when read out loud and you probably know "lit-elle mess moffette" as "little miss muffet"...

The anecdote serves as a good analogy for the importance of software architecture and the difference between architecture and the code and detailed design. Code and detailed design, like the words in the poem, are about local decisions. But it isn't enough to try to understand things in the small. You need to take a look at the big picture to understand how things work together and to ensure that you know where you are heading. Unless you put careful thought into it from several perspectives, you will just end up with a bunch of bad French poems that don't mean anything

Software architecture is very similar to this in the sense that it tackles the big overall requirements that has the most effect on the quality of the solution. Software architecture is also similar in the sense that you have to look it at from several perspectives (or view points) to really understand all of it.

This book is about an architectural look at SOA, and the patterns and anti-patterns described here are like small architectural Lego bricks which you can use and put together to create complete SOA solutions. In order to understand which of the patterns we need to use (in a particular solution) and which we may need to avoid we need to understand the architectural implications of SOA in general. More importantly, we need to know how to identify which patterns apply for a particular SOA instance – the solution you are working on today.

The aim of this chapter is to put SOA in the context of software architecture and provide you with some of the tools that can help you identify the relevance of patterns to a specific architecture. Before we delve into the details of the SOA, we will briefly explain what software architecture is and follow that with a description of software architecture requirements and how you can identify them. We will then provide a background for SOA that covers the architectural challenges of distributed computing in an Enterprise environment. Following that we will take a look at SOA from an architectural perspective and how it addresses the challenges of distributed computing.

1.1 Software Architecture

You hear a lot about software architecture and there are many books that touch this subject one way or another (This book is one of them). I'll try to explain what architecture is, and then elaborate on the different input that the architect can and should use to design an architecture.

I wish I could just quote a well accepted definition for software architecture, unfortunately, there is none which is universally accepted. Here are a few examples of the most prominent definitions:

Garland and Shaw, in their book *An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering*, volume I, published one of the first definitions of software architecture and explained that software architecture is something beyond “regular” design or in their words: *“As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem [i.e. Architecture – ARGO]. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives”*.

Another important definition was made in 2000 as the IEEE published standard 1471 “Recommended Practice for Architectural Description of Software-Intensive Systems”. The IEEE committee emphasized the importance of architectural decisions (calling them fundamental). IEEE’s definition is: “*Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution*”

The last example is a definition by Bass, Clements, and Kazman, in their book *Software Architecture in Practice* 2nd edition (published in 2003). Bass et al emphasised the structural elements of architecture. *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*

The three elements the previous quotes (and many other definitions) have in common are:

- *Architecture is there* – whether we explicitly define it or not (we can always reverse engineer it and describe the current situation).
- *Architecture is about components and their boundaries* – we usually don't need all the components but we need to understand the major ones.
- *Architecture is about the relationships and interactions of the components* - again, we are not interested in every minute detail but the general flows and their implications

furthermore, looking at the above definitions we can understand that when we describe an architecture it is probably a good idea to explain the rationale behind the decisions and alternatives weighted – both to help others (maintainers, new comers etc) as well as to help make sure we didn't miss something important.

The last insight we can fathom from the definitions is that architecture is relatively early – not because we want to – but rather because, unfortunately, we have to. Architectural decisions affect the fundamental behavior of the system and though we should try to postpone these decisions (Fowler, 2003) we more often than not, find out that we can't and thus need to make them rather early.

Based on these observations (and my experience working as an architect) here is my definition of software architecture that we are going to use throughout the whole book

Software Architecture Definition (Rotem-Gal-Oz, 2006)

Software architecture is the collection of the fundamental decisions about a software product/solution designed to meet the project's quality attributes (i.e. requirements). The architecture includes the main components, their main attributes, and their collaboration (i.e. interactions and behavior) to meet the quality attributes. Architecture can and usually should be expressed in several levels of abstraction (depending on the project's size).

If an architecture is to be intentional (rather than accidental), it should be communicated. Architecture is communicated from multiple viewpoints to cater the needs of the different stakeholders.

There are several inputs that the architect needs or can choose to work with in order to produce a viable software architecture (as shown in Figure 1.1): Principles, patterns/anti-patterns, quality attributes (requirements) & Technologies, and. The following sections will take a deeper look at each one of them.

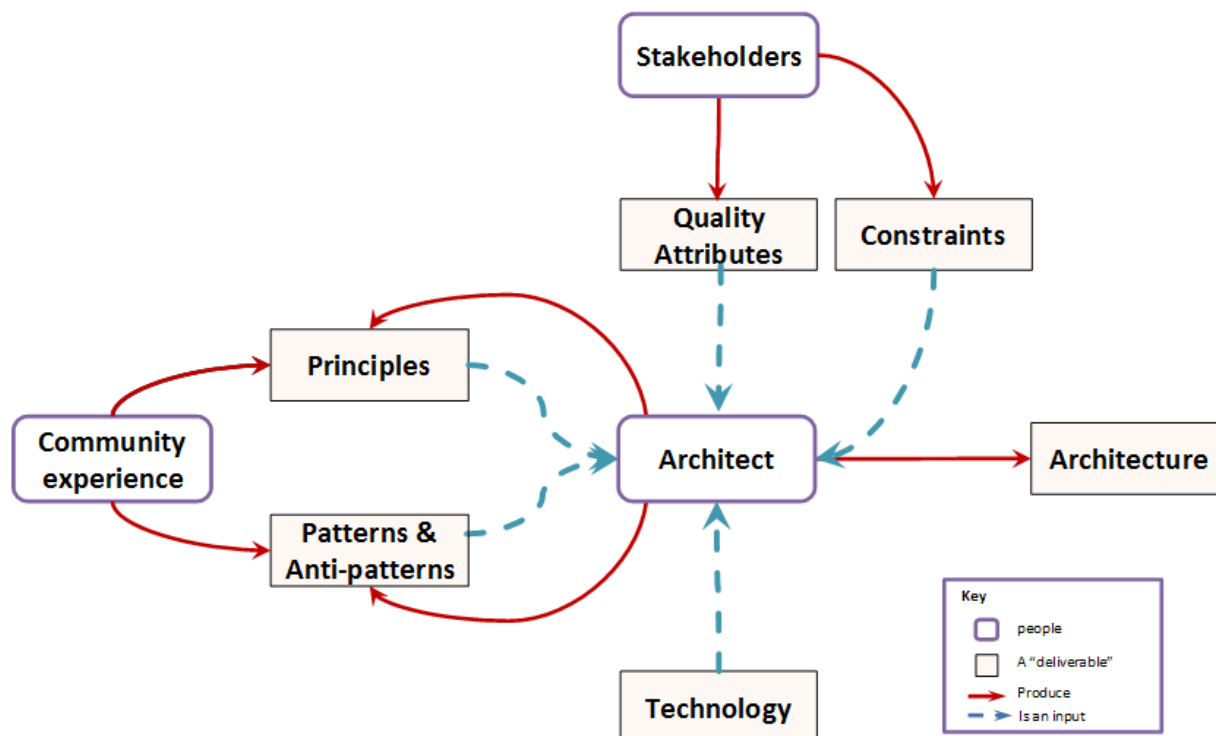


Figure 1.X software Architecture inputs. The Architect uses various inputs to design the architecture. First and foremost the architect works with the constraints and quality attributes (requirements) she can gather from the stakeholders. She can augment these inputs by drawing on her and the community's experience to add principles, patterns and anti-patterns. Lastly there are the possibilities and constraints imposed by available technologies. The architect, then, analyze m prioritize and balance all these inputs to produce an architecture for the problem at hand.

Let's take a deeper look at the different inputs that architects can use to design an architecture

1.1.1 Constraints & Architecture Principles

Once you have a good understanding of the problem space (the sort of system you are going to design and build) you will want to draw on your experience (or on other's experience) to reduce the number of possible solutions to 2-4 options you can evaluate and consider. Two tools to accomplish that are constraints and principles.

Constraints and principles are not directly related to the theme of this book (they are mentioned here for completeness) – so I won't go into too much detail. In short, constraints are requirements by stakeholders that limit the solution space. There can be different kind of constraints, for example the chief architect might limit your technological options ("only use Java on JBOSS for this solution")and, managers can limit the complexity of your solution ("The project should finish in 30 days") etc.

Principles are similar to constraints in the sense that they help you reduce the solution space. They are different on two counts: The first is the source Principles come from your experience or from a collective experience of the development community (as found in books, articles, blogs, forums etc.). The second difference is that principles are not imposed on you, rather these are choices you make Principles are used in the early stages of the architecture design to help us focus on directions that have worked before for similar systems, for example a principle might be to use a distributed database (vs. a federated one) or to design the solution for scale out (horizontal scaling) and not for scaling up. You can see an example for documenting principles in figure 1.2.

| | |
|-------------------------------|--|
| Principle Name | <i>Scale horizontally</i> |
| Description | <i>System should be designed to scale horizontally – to grow the system should add more computers rather than add processors/memory etc.</i> |
| Rationale and Benefits | <i>We don't know the maximum size that we want the system to scale to – this allows us to scale as far as we want without being constrained by the maximum size of the chosen hardware. Increment in sensible cost increments.</i> |
| Implications | <i>Need to implement methods for sharing processing across many identical servers in the same tier. Need stateless processing.</i> |
| Alternatives | <i>Grow vertically – not chosen as there would be a ceiling to growth of a server and potentially major migration to 'bigger' server.</i> |
| Scope and Exceptions | <i>Entire system, possibly excluding data buses.</i> |
| Principle Name | <i>COTS Based</i> |
| Description | <i>The system architecture will be based on standard, commercially available software products and infrastructure.</i> |
| Rationale and | <i>This would simplify the development and ongoing maintenance of the</i> |

Figure 1.x Architecture principles help reduce the solution space. Principles are “good practices” that, from the architect’s experience, are good candidates for the current problem. It is usually good to document the rationale and implications of the principles that are used so that you know their effect on your architecture should you choose to implement them.

1.1.2 Patterns and Anti-Patterns

The second input to architecture are patterns and anti-patterns. Experience shows that no single solution that fits everywhere or in Fred Brooks' words there is "no silver bullet". So while best practices are a fiction, good practices do exist (as do bad practices that we need to avoid) the difference is that we need a context to determine when to use them – this is exactly what patterns are : solutions for specific contexts.

I will not go into further details here regarding Patterns and Anti-Patterns (being the theme of the book they are covered in more depth at the introduction for the book). At this point, it is only important to think of patterns as architectural building blocks you can use when you design a solution and that specific

patterns or anti-patterns cover specific architectural requirements. Now let's look at the input referred to as "Quality Attributes"

1.1.3 The Software Architecture Requirements

There are two types of requirements for software projects: functional and non-functional requirements. Functional requirements are the requirements for what the solution must do (which are usually expressed as use cases or stories). The functional requirements are what the users (or systems) that interact with the system do with the system (fill in an order, update customer details, authorize a loan etc.).

Non-Functional requirements are attributes the system is expected to have or manifest. These usually include requirements in areas such as performance, security, availability etc. A better name for non-functional requirements is "Quality Attributes" (which is the term used in the software architecture definition). Table 1.1. presents formal definitions from IEEE standard 1061 "Standard for a Software Quality Metrics Methodology" for quality attributes (and few related terms).

| Quality Attribute | Definition |
|-------------------------|--|
| Quality attribute | A characteristic of software, or a generic term applying to quality factors, quality subfactors, or metric values. |
| Quality factor | A management-oriented attribute of software that contributes to its quality. |
| Quality subfactor | A decomposition of a quality factor or quality subfactor to its technical components. |
| Metric value | A metric output or an element that is from the range of a metric. |
| Software quality metric | A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality. |

Table 1.1: Quality Attribute definition from IEEE 1061 "Standard for a Software Quality Metrics Methodology"

Most of the requirements that drive the design of a software architecture comes from system's quality attributes. The reason for this is that the effect of quality attributes is usually system-wide (e.g. you wouldn't want your system to have good performance only in the UI – you want the system to perform well no matter what) - which is exactly what software architecture is concerned with. Note however, that few requirements might still come from functional requirements². The question is how do we find out what those requirements are?

The answer to that is also in the software architecture definition. The source for quality attributes are the stakeholders.

Definition: Stakeholder

A stakeholder is just about anyone who has a vested interest in the project. A typical system has a lot of stakeholders starting from the (obvious) customer, the end-users (those people in the customer organization/dept that will actually use the software) and going to the operations personnel (IT – those who will have to keep the solution running), the development team, testers, maintainers, management. In some systems the stakeholders can even be the shareholders or even the general public (Imagine for example, that you build a new dispatch system for a 911 center).

² Design has the ratios reversed i.e. most of the requirements for design come from functional requirements and a few requirements might come from the quality attributes.

One of the architect's roles is to analyze the quality attributes and define an architecture that will enable delivering all the functional requirements while supporting the quality attributes. As can be expected, sometimes quality attributes are in conflict with each other – the most obvious examples are performance vs. security or flexibility vs. simplicity and the architect's role is to strike a balance between the different quality attributes (and the stakeholders) to make sure the overall quality of the system is maximized.

Contextual solutions (e.g. this book's patterns) can be devised to solve specific quality attributes need. However saying that a system needs to have "good performance" or that it needs to be "testable" doesn't really help us know what to do. In order for us to be able to discern which patterns apply to specific quality attribute, we need a better understanding of quality attributes besides the formal definition, something that is more concrete.

The way to get that concrete understanding of the effect of quality attribute is to use scenarios. Scenarios are short, "user story"-like prose that demonstrate how a quality attribute is manifested in the system using a functional situation

Architecture and functional requirements

Architecture is mostly influenced by quality attributes (so called "non-functional" requirements). Functional requirement mostly affect design. Quality attributes drive the ground rules, the component types and permissible interaction and the functional requirement drive the instances. We can take an example from SOA - Designing by quality attributes you might come to the conclusion that you need process agility and decide to use the Orchestrated Choreography pattern (chapter 7). You can map that to a specific BPM tool (see technology mapping later this chapter). You can then design which processes to implement by analyzing the functional requirement (a loan process, a debit process etc.) The functional requirement analysis is a design activity and as such not in the scope of this book.

Quality attributes and Scenarios (identifying which patterns to apply)

Quality attributes scenarios originated as a way to evaluate software architecture. The Software Engineering Institute developed several evaluation methodologies, like Architecture Tradeoff Analysis Method (Clements, Kazman and Klein, 2002) that heavily build on scenarios to contrast and compare how the different quality attributes are met by candidate architectures. ATAM (and similar evaluation methods like LAAAM which is part of MSF 4.0) suggest building a "utility tree" which represent the overall usefulness of the system. The scenarios serve as the leafs of the utility tree and the architecture is evaluated by considering how the architecture makes the scenarios possible.

I found that using scenarios and the utility tree approach early in the design of the architecture³ can greatly enhance the quality of the architecture that is produced (Rotem-Gal-Oz, 2005). When you examine the scenarios you can also prioritize them and better balance conflicting attributes.

The scenarios can be used as an input to make sure the quality attributes are actually met. Furthermore you can use the scenarios to help identify the strategies (or patterns) applicable to make the scenarios possible (and thus ensure the quality attributes are met) within the system.

³ I published this as part of a framework of activities that an architect can take to build successful architectures. This framework is called SPAMMED Architecture Framework (or SAF for short) and it summarizes some of the things discussed in this chapter like identifying Stakeholders, Principles, Quality attributes and more

To help you make better use of the patterns in the book, while you architect your next solution, each of the patterns has a short section with sample scenarios that can lead you to consider utilizing it. Also Appendix A contains the list of scenarios (sorted by quality attribute) and the list of applicable patterns.

Figure 1.3 shows an excerpt from an actual utility tree of a SOA based Naval C4I2⁴ System I architected. The diagrams shows how the overall “utility” (or usefulness” of the system is derived from key quality attributes (Performance, Security etc.). Each of the quality attributes has sub categories (e.g. Performance is broken into latency, data loss etc.). Each sub category is demonstrated by a scenario that is expected to be manifested by the system (if it is to provide that quality attribute).

⁴ C4I2 – Command, Control, Communications, Computers, Intelligence & Interoperability

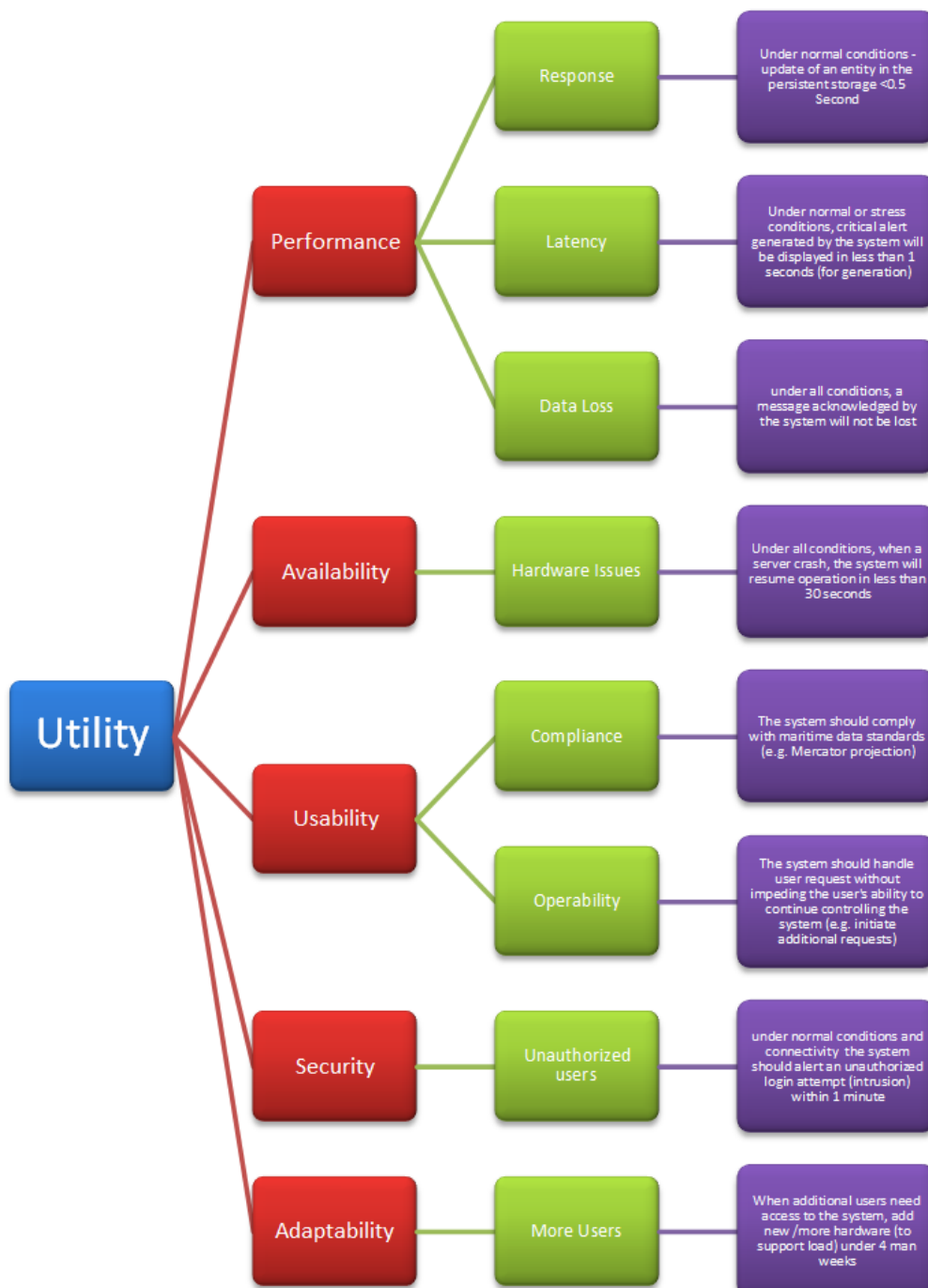


Figure 1.3 Utility Tree from an actual system (excerpt). The root of the tree is the overall usefulness of the system. Level 1 has the quality attributes, Level 2 has sub-categories of the quality attributes and the scenarios column gives examples for how it is expected for the system to behave if the sub-categories were fulfilled (these are the expectations by the stakeholders)

While figure 1.3 above is impressive, usually, a utility tree would be detailed using non-graphical tools such as Excel. Figure 1.4 shows (an excerpt) of the utility tree in figure 1.3 as it actually was written.

| Level 0 | Level 1 | Level 2 | Scenario |
|---------|-------------|----------|---|
| Utility | | | |
| | Performance | | |
| | | Response | |
| | | | Under normal conditions - update of an entity in the persistent storage <0.5 Second |
| | | Latency | |
| | | | Under normal or stress conditions, critical alert generated by the |

Figure 1.4 Utility Tree in Excel (excerpt). This is the same utility tree shown in Figure 1.3 written in excel form. Level 0 is the overall usefulness of the system. Level 1 has the quality attributes, Level 2 has sub-categories of the quality attributes and the scenarios column gives examples for how it is expected for the system to behave if the sub-categories were fulfilled (these are the expectations by the stakeholders)

The most important part of the utility tree are the scenarios, since, as I mentioned earlier, the scenarios are the requirements the architect uses to design the architecture. The following section gives some more detail on how to write scenarios.

Scenarios Structure

Scenarios are expressed as statements that have 3 parts: a stimulus, a context and a response. The stimulus is the action taken (by the system / user/ other system / any other person); response is how the system is expected to behave when the stimulus occur, and the context specifies the environment or conditions under which we expect the to get the response. For example in the following scenario: "When you perform a database operation , under normal condition, it should take less than 100 miliseconds."

- "Under normal condition" is the context
- "When you perform a database operation" is the stimulus
- "it should take less than 100 millisecond" is the response expected from the system.

Here are few additional examples for quality attribute scenarios:

- Performance → Latency → under normal conditions a client consuming multiple services should have latency less than 5 seconds.
- Security → Authentications → Under all conditions, any call to a service should be authenticated using X.509 certificate

When you consider just the first scenario, you can understand that a normal consumer will need to make multiple calls to services, and that we'll need to make sure the latency per call is relatively small. The second scenario tells us that we need to re-authenticate every call (which adds to the overall latency). To make sure both scenarios can be met you need to consider implementing something like "single-sign-on" for services (see TicketServer pattern). Another example is the following scenario that can lead you to thinking about the "legacy bridge" pattern:

- Interoperability → Legacy Support → Under all circumstances, billing records should be reported to

the mainframe.

Identifying quality attributes paves the way to design the software architecture, once you've accomplished that you also need to consider what technologies you will use to implement the solution and the impact of these choices on your architecture.

1.1.4 Technology's impact on Architecture

The software development landscape in general and the SOA landscape in particular has a lot of technologies and new ones are launched almost every day. For SOA you've got WSDL, a horde of WS-* standards, SOAP, BPEL and many vendor specific tools such as ESBs, SOA management solutions, XML firewalls. This section talks about the relation between architecture and technology and the influence technology choices can have on the architecture.

This section includes references to a lot of different technologies. The technologies themselves however, are not the issue here and it shouldn't matter if you don't know all of them. What's important are the complexities incurred when all these (and other) technologies have to play together and help deliver a solution in a given architecture

Mapping technologies to Architecture

Architecture is seemingly technology agnostic and to an extent it actually is. You can really **design** architectures that are not depended on technology – the problem, however, is that you can rarely actually **build** a solution that is not technology depended (well, maybe if you have a severe case of NIH syndrome⁵). What usually happens is that besides designing an architecture, you also need to map (i.e. decide) what parts of the architecture will be implemented using which specific technologies.

For example Figure 1.5 describes a layer diagram of an e-government solution. It has a lot of different layers (UI layer composed of Rich UI and Web UI; Business logic layer composed of a façade Activities, Business rules, workflows and human workflows etc.) which provide a potential for a technology bazaar.

⁵ NIH Syndrome – Not Invented Here Syndrome – The tendency to reject things that were developed by other people. For example, I once worked in a startup where the CEO wanted everything developed in-house. I do mean everything – it got to the point where I spent time designing an developing hover effect for buttons on the UI, where the core business was actually a proprietary ROLAP engine for business intelligence...

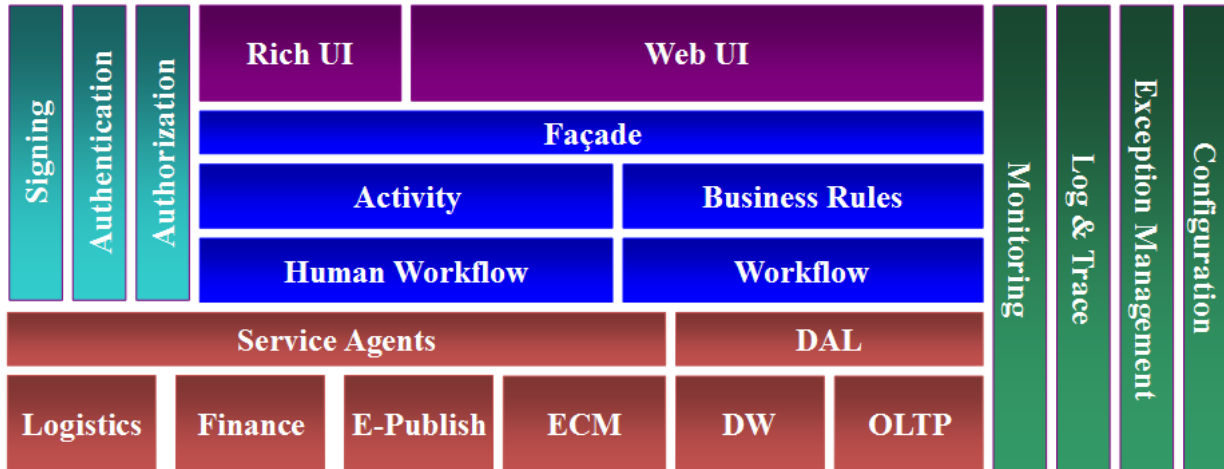


Figure 1.5 Layers Diagram of an e-government project. The figure shows a “classical” 3-tier architecture (UI, Business logic, data) sub-divided into layers (e.g. the business logic is further divided to a façade on top of activities, business rules, workflow and human workflow). In addition the architecture has some cross cutting concerns like monitoring, logging , configuration and the like which require a system-wide solution.

Indeed diagram Figure 1.6 shows a technology mapping to a .Net centric solution (i.e. not all the technologies are .Net ; e.g. Documentum, CA Unicenter, SAP etc.; but most of them are). The mapping in Figure 1.6 is just one mapping of almost infinite number of possibilities. Each and every "box" in diagram Figure 1.5 can be mapped to a completely different technology. For example for monitoring we can map Tibco Hawk, Tivoli, HP-Openview and Microsoft’s MOM 2005; for workflow we can have Windows Workflow Foundation, Flux (Java), Skelta Workflow.Net; for web user interface we could have struts (Java), ASP.net 1.1, Atlas (ASP.Net AJAX implementation), servlets (Java) ; and the list goes on and on.

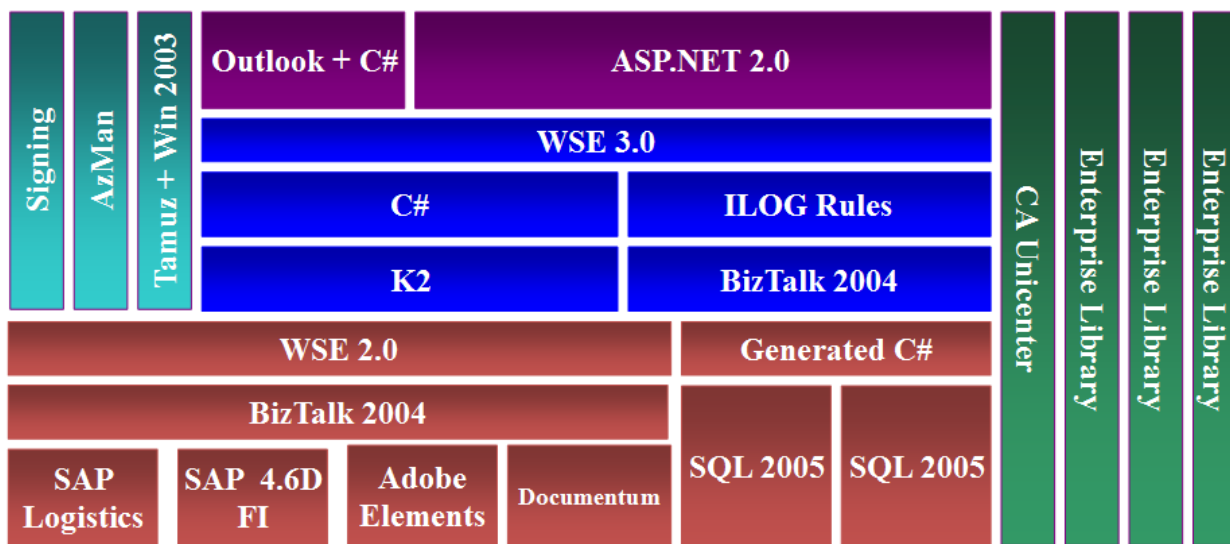


Figure 1.6 Technology mapping for the layers in figure 1.5. Each box in figure 1.5 is mapped to a technology. For example

the “Rich UI” box in figure 1.5 is mapped to a C# solution hosted in Outlook; the Monitoring concern in figure 1.5 is mapped to CA Unicenter etc.

This technology salad raises some very interesting and very important questions. First on the technology level: Can all these technologies live together? Are they aligned with each other? Are they compatible with each other? Some matches are obvious (you will not take ASP.NET with an Apache web-server) but others choices might not be that obvious e.g. do we want to use the built-in business rules engine of Microsoft's BizTalk or do we want to go with another vendor (like Ilog or Blaze). Another example is the service agents. The technologies of the other services are mostly Java centered – Maybe a better choice is to go with WSIT (Web Service Interoperability Technology - a JAX-WS extension for interoperability between Java web-services and Windows Communication Foundation web-services).

So how does these technologies affect the architecture?

Technology & Architecture

The technology level considerations mentioned above are just the tip of the iceberg. The real problems begin when we consider technology's implications on the architecture. Technology solutions are also software products and as such they have their own architectures. This means, unfortunately, that any given technology has its own constraints, assumptions and limitations. In the example above in the service agent layer technology limitations added an additional layer (since we need some way to bridge the .NET world of the application with the Java world of the services it consumes).

The technology limitations and constrains can have even more severe effects on our ability to support the quality attributes required from the architecture. For example, can a specific technology (say BizTalk 2004 in the above example) withstand our project's response requirements? and at what cost (how will it affect our hardware architecture). Furthermore the more technologies a solution mixes, the more complex it becomes to make them all play together. Things that look fine on paper, have all those little discrepancies when you actually try to connect them (or as a wise man once said, the difference between theory and practice is that in theory they are the same...).

The obvious solution to the problems mentioned above is to use as few technologies as possible for any given solution. The two problems with that are that if you opt-out of a technology and need to write it by yourself the costs will probably be higher. The second problem is that any non-trivial solution will need quite a few technologies no matter how hard you try to cut their number.

If we accept that multiple technologies is an integral part of our architectures we need :

- Try to make sure the technologies we choose are compatible with each other
- Examine the implications of choosing an architecture on our architectural decisions and then either revise the architecture accordingly (and reevaluate) or try to find another technology

For example a decision to use some ESB product will jive well with architectural decisions to implement location transparency pattern or the serviceBus pattern – but a technology choice on Windows Communication Foundation will make it considerably harder to implement those same patterns.

Another example: if you are decide to use RestLet framework (an open source java framework that implements the REST architectural style (Fielding, 2000)), the Request/Reply pattern will be a natural match. You can also implement the Firewall and gateway patterns easily (using the library's filter and

router objects respectively), but you'd have a much harder time implementing the ServiceBus pattern mentioned above or the Transactional Handling pattern.

Where applicable, patterns/anti-patterns have a section in this book that looks at the possible technology mappings and mis-mappings and discusses some of the implications of choosing these technologies.

Up to here. I defined some building blocks such as quality attributes and technology mappings that will serve us later when we'll discuss patterns and anti-patterns. Also I've provided a definition for software architecture and I'll spend one of the next sections examining SOA in the light of this definition. Before I'll do that let's take a look at distributed systems which is the reality for many modern systems and the foundation laying in the heart of SOA.

1.2 Challenges of distributed systems

SOA is an architecture style which is suited for distributed systems (vs. object orientation for example which assumes locality). The purpose of chapter 1 is to put SOA in context. We've already covered architecture (as the basis for the A in SOA), the next step, before explaining SOA is to understand the motivation and the type of problems that SOA answers. Section 1.2 provides a brief introduction to distributed systems and the challenges involved in designing and building them.

In the days of mainframe computing - life was relatively simple, everybody had those green terminals, you mainly had batch jobs running and anyway everything just ran on the mainframe. One drawback was that you had no control over the machine - you had to go and ask the mainframe "prophets" for CPU time. Another drawback (though people didn't know that at that time) was that the programs were rather simple and didn't deliver much value (vs. modern software).

Mainframes in an SOA world

One side note regarding mainframes, though people have been talking for years on the demise of mainframes - SOA actually brings new blood to platform as SOA makes it easier to leverage mainframes as large servers and treat them as just another technology that needs to be integrated with the rest of the business (see Orchestrated Choreography and Legacy Bridge patterns)

There are two categories of distributed system challenges. The first is that a system is part of a distributed environment (and has to interact with other systems over a network) and the second category is systems which are distributed as part of their internal structure (i.e. the system is a distributed system). Lets examine them one by one .

1.2.1 Playing in a Distributed Environment

I will spare you the detailed history of computing, but I'll skim over a few highlights. In the late 1960s the first distributed systems were introduced, the most famous example is ARPANET which started at 1969 as a 4 node network (Britton, 2000) (which evolved over the year to what we call "the internet") During the 1970's we saw an explosion of distributed systems network architectures (e.g. networks architectures like SNA from IBM and mini-computers like Digital's PDP-11 led the revolution). Technology advancement tell only part of the story since as the years go by the balance shifts back and

forth between centralization and distribution⁶ for additional reasons e.g.: organizational changes - bringing IT closer to the business (the departmental level). Let's take a closer look at this.

As computing got decentralized departments and other business entities started developing their own systems, forming what is known as applications silos or stovepipe systems (each application has its own database, server and clients and isn't connected to any other system). This worked for awhile but as business processes evolved the remote printing and file transfer capabilities (originating in the 1970s) were not enough and businesses found it was needed to share data between applications. The applications were connected point-to-point (as needs were discovered) and new technologies were developed to help make these connections (RPC, ETL, FTP, distributed transactions etc). The result of that was even a bigger problem of literally hundreds of interconnected systems (for large enterprises) or in other words a complete spaghetti. Figure 1.7 shows how a typical enterprise might look like. The figure shows 3 departments, each with 2-3 systems and how these systems are intertwined by different technologies (on-line interfaces, file based interfaces, ETL and direct database connections). The problem with this approach is that it is very brittle. Changing one system can have a huge ripple effect as systems which are not directly connected might stop working (or work in unexpected ways) i).

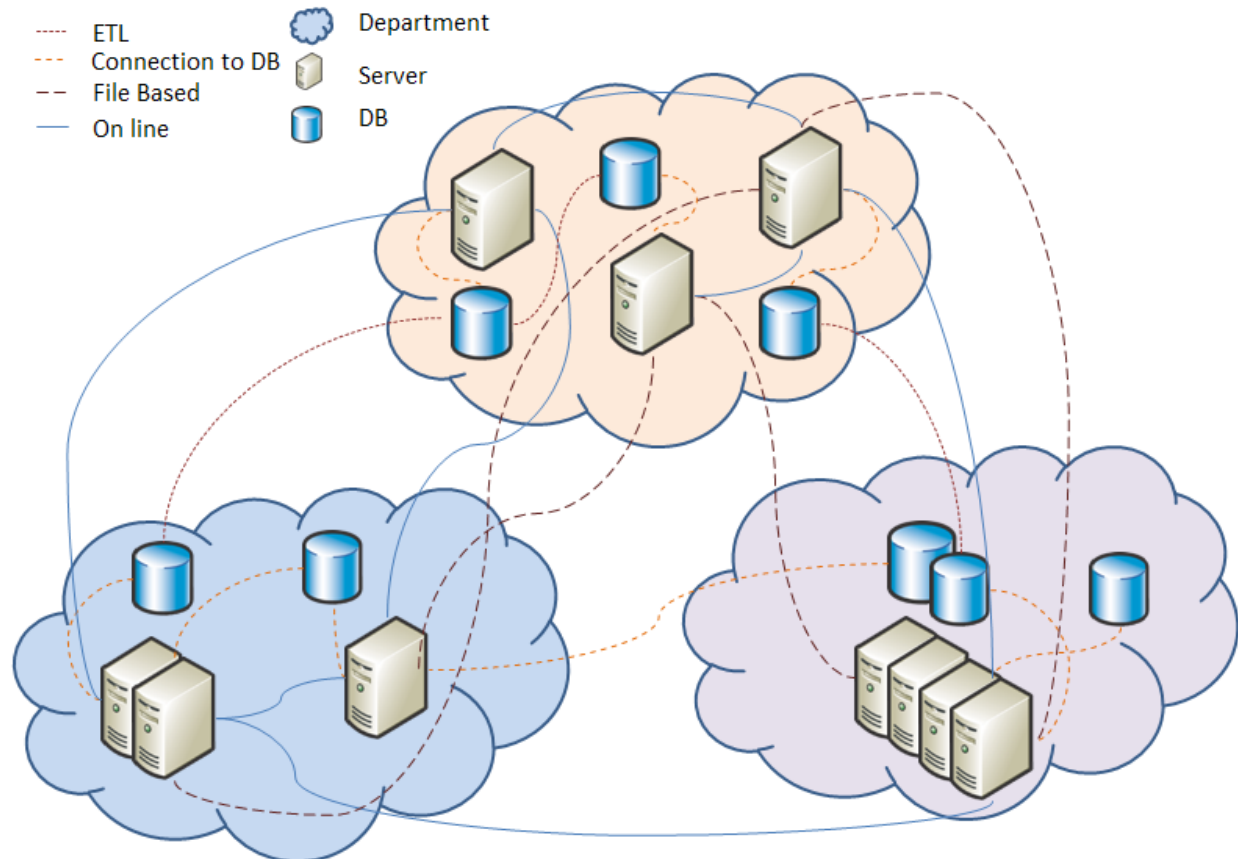


Figure 1.7 Typical Enterprise systems integration spaghetti. Each department builds its own systems. As people use the

⁶ By the way, in case you are wondering, the balance now shifts back to centralization – but with a few twists, on the organizational level you have IT teams dedicated to specific business processes – for all the organization and on the technology front you have technologies like virtualization that lets you co-locate different servers on the same machine

systems they find they need information from other systems and point-to-point integration emerges. The diagram show 4 types of point-to-point integrations: ETL (Extract, Transform Load) which is a DB to DB relation, On-line and File Based which are application to application relations and direct connection to a DB which is an Application to database relation. (there are additional relation types like replication, message based etc.)

One attempt to solve these problems was EAI (Enterprise Application Integration) but it never really succeeded mostly because EAI is data centric and not process centric which means it cannot keep up with business process change – not to mention that EAI solutions are expensive to build and maintain (Arora, 2005). Thus, one of the challenges we, as architects, have today is trying to make sense of this systems spaghetti and make the systems work together and adapt the systems to the changing business processes.

SOA in the large i.e. partitioning the enterprise into services and trying to align the services with the business needs is trying to cope with this set of challenges. This book, however focuses on how SOA in the small (building the services and making them work together) helps solve the second class of challenges – those that come from systems being distributed.

1.2.2 Being a distributed System

The second category of distribution challenges are the ones that exist when a system in itself is distributed. As individual systems grow more complex (i.e. had more functionality), more robust (e.g. had to cope with requirements like 99.999% availability) and as the systems have to respond to changes mentioned above (both technological and business requirements). The systems themselves have become distributed. Again, on the technology front, we have a lot of (attempted) solutions to the distribution problem – technologies like RPC, CORBA, DCOM, distributed objects and the like - Unfortunately the underlying architectural styles (procedural programming at first and Object Orientation later) that were (and still are) in wide use have fundamentally flawed assumptions (in regard to distributed systems) of assuming locality . Trying to stretch Object Oriented Architectures into the distributed systems world is difficult (which is why we see so many so called “3-tier” architecture which are brittle and have poor performance). Architects (especially those new to distributed systems) tend to make false assumptions that are hard to avoid and expensive to fix. The next section details these assumption to help explain while building a distributed system is not a trivial task. The Fallacies of Distributed Computing

In 1994, Peter Deutsch, a sun fellow at the time, drafted 7 assumptions architects and designers of distributed systems are likely to make. These assumptions prove wrong in the long run, which results in all sorts of troubles and pains for the solution and architects who made the assumptions. Three years later (circa 1997) James Gosling (another Sun fellow and the father of Java) added the last fallacy. These "fallacies" are now collectively known as the "The 8 fallacies of distributed computing" (Hoogen, 2004) and are shown in Figure 1.8.

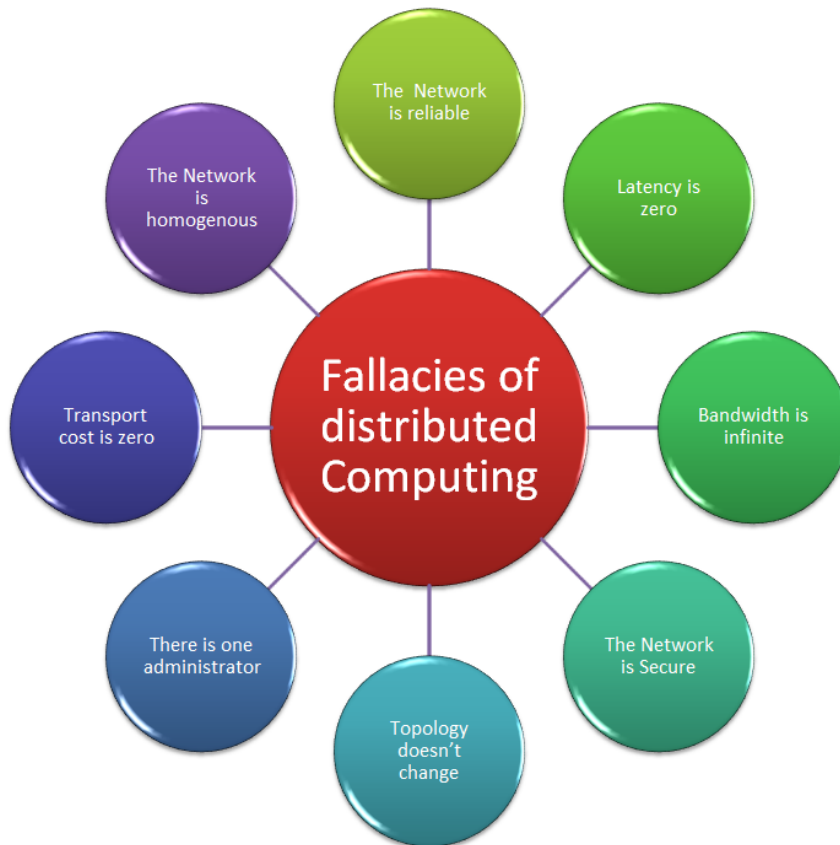


Figure 1.8 The fallacies of distributed computing Table 1.2 The 8 fallacies of distributed computing. The fallacies are assumptions developers and architects tend to make about distributed system that prove to be wrong in the long run (and are hard to fix)

Table 1.2 below details the fallacies in Figure 1.8 and explains why it is wrong to assume each of them.

| Fallacy | Why |
|----------------------------|---|
| The network is reliable | when you have local reference to an object and it is not null you can just call it and it is there. When you want to talk to a remote object/process you can no longer assume that |
| Latency is zero | Always remember that even if your solution only runs on a dedicated 1Gbit LAN the latency is still much larger than direct memory access – and the numbers deteriorate quickly as you access remote systems or nodes over congested LANs or over the internet . |
| Bandwidth is infinite | While bandwidth gets better and better all the time so does our bandwidth needs (we now have VOIP, multi media, office documents, etc. that also flows on the wire) |
| The network is secure | You would think that today people would be aware that they need to consider security – unfortunately this is rarely true. Time to market demand and other pressures still make people patch security in at the last minute or only rely on peripheral security. I personally witnessed even Military mission critical systems that shipped with mediocre security or with known loop holes not to mention how security fares for commercial run-of-the-mill applications. |
| Topology doesn't change | Once an application is deployed, given enough time the IT department will change all of its environment (protocols, hardware, security and what not) |
| There is one administrator | one for each "thing" that is – there's the applicative DBA, the system DBA , the Linux sys-admin, the active-directory expert, the ERP person(s) etc. not to mention the interaction with external partners, suppliers etc. (see for example the Mushup pattern) who have their own administrators. |
| Transport cost is zero | there are inherent costs for keeping servers up and running, maintaining the network, doing backups etc. Any additional node (computer) complicates the solutions resulting in increase of the baby sitting needed. |
| Network is homogenous | does ETL and EIS ring a bell? Most enterprises have more than a single platform. When I think about it, even my home network has Linux and Windows machines, an appliance (DVD player with wireless access) ,a PDA and a (small) NAS. |

Table 1.2 The 8 fallacies of distributed computing. The fallacies are assumptions developers and architects tend to make about distributed system that prove to be wrong in the long run (and are hard to fix)

There are many other inherent complexities (realities) associated with distributed system, which you also need to consider, for example:

- You can be sure to expect a certain level of entropy in the system - Sites are never fully synchronized (unless you stop new data from pouring in)
- There's a problem of shared state between the dispersed systems or nodes - if the data caches and shares is not treated with care it can cause serious discrepancies
- It very very hard to be able to scale indefinitely
- Monitoring is difficult – you will probably only be able to Observing global state if you use designated control messages

So what do we do? Martin Fowler once said – the first rule of distributed computing is don't (Fowler01, 2003)– but since that is rarely an option we need something else.

Enter SOA...

1.3 SOA - an Architecture Style

Unlike Object Orientation, which, as mentioned above, builds on a premise of object locality. SOA was drafted on the premise that services are distributed. This means you expect other services to be remote. This doesn't mean that we no longer need to consider the fallacies and realities mentioned above – it just means that we have better tools to handle them.

Take security for example. Using SOA does not mean security will be magically implemented. You still need to consider it - and consider it carefully. However most of the technologies associated with SOA take security into account (e.g. ESBs or WS-* – with WS-Security, WS-Trust etc.). Furthermore SOA separate policy from substance (more on that later), i.e. the fact that a service can talk with you doesn't mean that it is willing to do that. The meaning of this is that if you follow the SOA principles (which I'll explain below), you not only think about security you will also externalize several aspects of it. You can read more about security related aspects of SOA in chapter 8 (Security Patterns)

Standards based contract is another example of how SOA can help us avoid the fallacies – in this case the assumption about network being homogenous. The fact the we only used a standardized and well-defined channel to interact with neighboring services means we don't care as much about the other service platform, language or underlying technology.

In the following sections I'll explain what SOA is, take another look at the quality attributes from the perspective of SOA, and explain why the business aspects of SOA are not in the scope of this book.

1.3.1 What is SOA

One problem that plagues computer science is that of terminology overloading. A useful "thing" emerges somewhere (be that a methodology, architectural style, technology) and as (marketing?) people see it gain momentum, more and more (related and unrelated) other things take on the same name. One recent example is the word "agile" (for example, I recently saw a demonstration of a tool that is heavily process centric that was dubbed "agile" just because the developers use their IDE to enter task estimate and progress reports and not project).

Another example for overloaded term is SOA, Thus, I guess that the first thing to do before we delve into defining what SOA is to take a brief look at what SOA isn't. Table 1.3 below details popular misconceptions about SOA and explains why they are wrong.:

| Popular misconceptions about SOA | Why this definition is incomplete/incorrect (??) |
|---|--|
| SOA is a way to make IT and the business more aligned | No that's not it - Better IT and business alignments is something we want to achieve using SOA – but it isn't what SOA is. |
| SOA is any program that has a "web service" interface | wrong again. For one we can implement SOA with other technologies (A nice example is the Open Services Gateway Initiative- OSGI, which defines a java based service platform www.osgi.org). Furthermore exposing a method as a web-service can also be used (and many times is) to create procedural-like RPC which is very far from SOA concepts and direction (see also NanoServices antipattern in Chapter 11) |
| SOA is a collection of technologies (SOAP / REST/ WS-I...) | This is a general case of the misconception above. Still, while some technologies are identified with SOA or make a good fit for SOAs. SOA is more at the architectural level. As was mentioned above. SOA can be implemented regardless of technology. |
| SOA is a reuse strategy (i.e. a way to make the elusive promise of reuse a reality at last) | The larger the granularity of a component the harder it is to reuse it. Nevertheless SOA will allow your "services" to evolve over time and adapt (i.e. not start from scratch every time). |

Table 1.3 SOA misconceptions and why they are wrong.

As you can see from table 1.3 there are many misconceptions, the question is then what SOA is? SOA is an architectural style for distributed components. Or as Roy W. Schulte and Yafim V. Natiz defined it back in 1996: " a style of multitier computing that helps organizations share logic and data among multiple applications and usage modes." (Schulte et al., 1996). The SOA style defines the

following components: Service, End Point, Message, Contract, Polity & Service Consumer. Since SOA is an architectural style it also places constraints on how the components can interact – as depicted in Figure 1.9.

Definition: Architecture Style.

Architectural style to architecture is what design pattern is for design. The Software Engineering Institute defines (SEI, 2006) an architectural style as "A specialization of element and relation types, together with a set of constraints on how they can be used."

In case you are wondering where are other components usually associated with SOA like service bus, repositories, workflows etc. These are all important concepts (and the patterns for them are described later in the book), however these concepts are "only" ways to implement SOA and they are not at the core of the SOA style.

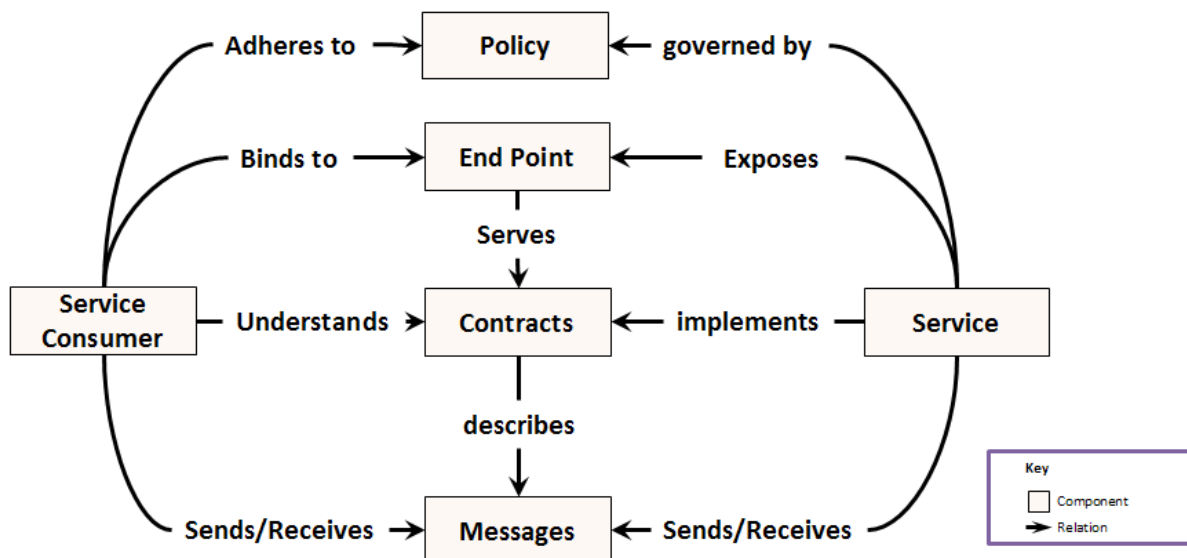


Figure 1.9 SOA components and their relations. Apart from the obvious Service. SOA has several other components: contract that the service implements, End points, where the service can be contacted, messages that are moved back and forth (between the service and its consumers); policies that the service adhere to and consumers that interact with the service

Lets take a deeper look at each of the components that are part of the SOA architecture style

Service

The core component of an SOA is of course the Service. The Service should provide a high cohesion and distinct business function. Services should be coarse grained pieces of logic (see also the nanoservices anti-pattern). A Service should implement at least all the functionality promised by the contracts it exposes.

Note on business aspects of SOA

There are many books and resources that discuss business modeling (or business process modeling) and how to divide the enterprise business into services. The focus of this book, however, is how to build these services once you identify them.

Contract

The collection of all the messages supported by the Service is collectively known as the service's contract. The contract can be unilateral, meaning a closed set of messages the service chooses to provide. A contract might be multilateral (or bilateral) ie between a closed group of parties. The contract is the interface of the Service.

End Point

An address, a URI, a specific place where the service can be found. A specific contract can be exposed at a specific endpoint.

Message

The unit of communication in SOA is the message. Messages can come in different forms and shapes for example http GET messages (REST style), SOAP messages, JMS messages and even SMTP messages are all valid message forms.

The differentiator between a message and other forms of communication (e.g. plain RPC) is that messages have both a header and a body. The header is usually more generic and can be understood by infrastructure and framework components without understanding (and coupling to) every message type.

The existence of the header allows for infrastructure components to route reply messages (e.g. correlated messages pattern) or handle security better (see Firewall pattern).

Messages are of course a very important part of SOA, however messaging patterns have been thoroughly covered by other books – most notable among them is Enterprise Integration Patterns (EIS) by Gregor Hohpe and Bobby Wolf. Nevertheless this book also touches a few of the messaging patterns where I believe the SOA perspective enhances the more generic perspective used in EIS (in the SOA sense) for example see Request/Reply and Correlated Messages patterns.

Policy

One important differentiator between Object Orientation or Component Orientation and SOA is the existence of policy. If an interface (contract in SOA lingo) separates specification from implementation. Policy separates dynamic specification from static/semantic specification. Policy represents the (run-time updatable) conditions for the semantic specification availability for service consumers. Policy specifies dynamic properties like security (encryption, authentication, Id etc.), auditing, SLA etc.

Service Consumer

Any software that interacts with a service (sends or receive messages). Consumers can be client applications or other "neighboring" services their only requirement is that they bind to an SOA contract.

1.3.2 SOA Benefits

SOA has, almost inherently, a lot of architectural benefits, in regard to many quality attributes. I think a lot of the importance and usefulness of SOA comes from the introduction of well-defined boundaries on coarse grained areas of the enterprise i.e. the move from object and application soup to

more disciplined communication. Fewer connectors means less maintenance, reduced assumptions and by that increased flexibility as shown in Figure 1.10.

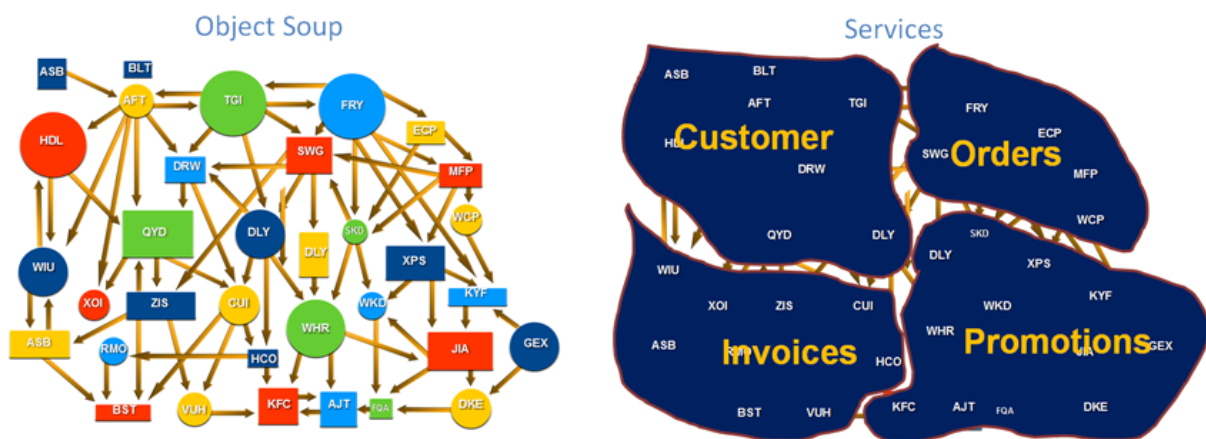


Figure 1.10 from Object Soup to Well formed Services. One of the ideas behind SOA is to set explicit boundaries between larger chunks of logic. Where each chunk represents a high cohesion business area. This is an improvement on the more traditional approach, which more often than not results in an unintelligible object soup

Some of the advantages of services are:

- Reusability—not in the sense of "write once integrate anywhere" but rather in the sense that "don't through everything out when you need different functionality"
- Changeability or upgradability of system—Isolating the internal structure of a service from the rest of the world lets you make changes more easily. You only need to adhere to the contracts you published.
- Maintainability—is also increased by the ability to assign specialized IT teams that own a specific service across the enterprise (i.e. vertical specialization) this absolve the IT organization from trying to pursue the myth of homogenous enterprise while still driving down operational costs.

Policy based communications are also greatly enhances the maintainability and adaptability of SOA based solutions. The fact the several aspects like security, monitoring etc. are configurable moves some of the responsibility from the development team to the IT staff and makes life easier for both parties.

Don't downplay the emphasis I put on better maintenance experience, remember that software spends most of its existence in maintenance. just think about the Y2K craze where systems that were age old (i.e. being maintained) and the effort that was put into make them all work. As Robert L. Glass said (Glass, 2006) "software maintenance is a solution, not a problem" – SOA greatly helps make this a reality.

What about the other quality attributes (performance, scalability, availability etc.) ? Well, that is exactly what this book is about – making sure we can implement an SOA in a way that will enhance the inherent benefits while adding the other quality attributes we need.

1.3.3 Business Aspects of SOA

When I defined SOA, I talked about terminology overloading and said SOA is a good example for the confusion that this overloading generates. We can especially see that in regard to the business value of SOA. If you've read just one article on SOA, you've probably heard the term "SOA increases [or is a tool

to create better] business alignment". The term SOA is used to describe both cases. To prevent confusion while reading the book. I'll define the a new term "SOA initiative". SOA initiative is the project (which is going to use SOA, whose aim is to increase business agility, increase business alignment etc.) and I'll use the term SOA to refer to the architectural style.

Aligning business and IT, drafting contracts etc. are all enterprise architecture issues (i.e. architecture of the enterprise and not software architecture). These activities require specialized processes (such as Business Process Modeling (BPM) etc.) and a lot of business understanding.

There are many books that talk about all the aspects of the SOA initiatives – this isn't one of them.. The book's scope is on the software architectural aspects of SOA and technological implications of these aspects and not on the design effort and methods.

1.4 Summary

Before we move on to the patterns and anti-patterns (which is why your reading this book) let have a quick review of what was covered here.

The first thing we covered was software architecture, how it is driven mostly by quality attributes (so called "non-functional" requirements) and how we can use the simple technique of utility trees to elevate and prioritize quality attributes. I've also explained how the utility tree scenarios can be helpful in identifying patterns.

The second part of the chapter looked at some of the challenges that are inherent for distributed systems, and how object oriented architectural style is lacking in dealing with these challenges.

Lastly, I demonstrated how SOA can help solve distributed systems challenges and put SOA in the context of software architecture and provided a definition of SOA and its main components. The chapter finished with the benefits SOA carries for some of the important quality attributes like maintainability and adaptability and the relation to the business aspect of SOA.

This chapter covered a lot of issues very briefly. The idea was not to teach you software architecture or SOA in 5 minutes, but rather to make sure we have a common vocabulary when we venture forward into the discussion on patterns. I thought this is very important in the light of the many overloads Architecture, distributed systems and SOA have.

If you are interested in learning more on some of the issues discussed in this chapter I recommend you pick up one or more of the following resources:

| Area | Resource name | Why |
|---------------------|--|---|
| Architecture | Software Architecture in Practice 2 nd edition – Paul Clements et. al. | A solid book on software architecture and how to go about designing one |
| Architecture | The SPAMMED Architecture Framework – Dr. Dobb's magazine, October 2006 – Arnon Rotem-Gal-Oz | An article I wrote that explains the activities and processes needed for creating successful architectures |
| Distributed Systems | IT Architecture & Middleware: Strategies for building large and integrated systems – Chris Britton | Provide a good look a the history of distributed systems and the inherent difficulties that they inflict. It is a very thorough book, the only problem is that it ends just before the SOA era. |

| | | |
|------------------------------------|---|---|
| Fallacies of Distributed Computing | "Fallacies of Distributed Computing Explained" (http://www.rgoarchitects.com/Files/fallacies.pdf) | A paper I wrote which explains how the fallacies are still relevant today, |
| SOA | Enterprise SOA: Service Oriented Architecture Best Practices - Dirk Krafzig et. al. | One of the best books on SOA. Provides a very good introduction on the subject |
| SOA | Service-Oriented Architecture (SOA): A Planning and Implementation Guide for Business and Technology - Eric A. Marks, Michael Bell | Takes a look at the business perspectives of SOA and provides a completely different (and complementary) angle at SOA (vs. this book) |
| | | |

1.5 Bibliography

- **Arora Sandeep** EAI, BPM and SOA [Online]. - 9 December 2005. - <http://www.soainstitute.org/articles/article/article/eai-bpm-and-soa/news-browse/1.html>
- **Bass Len, Clements Paul and Kazman Rick** Software Architecture in Practice, 2nd edition [Book]. - [s.l.] : Addison Wesley Professional, 2003
- **Beer S.** The Heart of Enterprise [Book]. - Chichester : John Wiley & Sons, 1979
- **Britton Chris** IT Architectures and Middleware: strategies for building large, integrated systems [Book]. - [s.l.] : Addison-Wesley, 2000
- **Clements Paul, Kazman Rick and Klein Mark** Evaluating Software Architectures: Methods and Case Studies [Book]. - [s.l.] : Addison-Wesley Professional, 2002
- **Fielding Roy Thomas** Architectural Styles and the Design of Network-based Software Architectures [Online] // UNIVERSITY OF CALIFORNIA, IRVINE. - 2000. - <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>
- **Fowler Martin** Who needs an Architect [Article] // IEEE Software . - September/October 2003. - 11-13
- **Fowler Martin** Errant Architectures [Journal] // SD Magazine. - [s.l.] : CMP, April 2003. - 4 : 11. - 38-41
- **Garlan David and Shaw Mary** An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering, volume I. [Book]. - [s.l.] : World Scientific Publishing, 1993
- **Glass Robert L.** Software conflict 2.0: The art and science of software engineering [Book]. - Atlanta, Georgia : Developer.* books, 2006
- **Hoogen Ingrid Van Den** Deutsch's Fallacies, 10 Years After [Journal]. - [s.l.] : Sys-Con Media, 8 Jan 2004. - 1 : 9
- **IEEE ANSI/IEEE Std 1471-2000**, Recommended Practice for Architectural Description of Software-Intensive Systems [Book]. - [s.l.] : IEEE, 2000
- **IEEE IEEE 1061-1998** Standard for a Software Quality Metrics Methodology [Book]. - [s.l.] : IEEE, 1998. - ISBN 0-7381-1510-X
- **Rooten Luis d'Antin van** Mots d'Heures, Gousses, Rhames [Book]. - [s.l.] : Penguin, 1980. - ISBN 0-14005730-7
- **Rotem-Gal-Oz Arnon** RGO Architects [Online] // www.rgoarchitects.com. - 2005. - www.rgoarchitects.com/saf
- **Rotem-Gal-Oz Arnon** Dr. Dobb's Portal - Architecture & Design [Online] // www.ddj.com/dept/architect. - April 2006. - http://www.ddj.com/blog/architectblog/archives/2006/04/looking_the_sof.html?cid=GS_blog_arnon
- **Schulte Roy W. and Natis Yefim V.** SPA-401-068: "'Service Oriented' Architectures, Part 1 [Report]. - [s.l.] : Gartner