



Systems Engineering at MITRE

SERVICE-ORIENTED ARCHITECTURE (SOA) SERIES

Leveraging Federal
IT Investment
Using SOA

Executive Version

.....
Geoffrey Raines

THE BIG PICTURE: Service-Oriented Architecture (SOA) builds on computer engineering approaches of the past to offer an architectural approach for enterprise systems oriented around the offering of services on a network of consumers. SOA, as implemented through the common Web Services standards, gives Federal senior leadership teams a path forward, allowing for incremental and focused improvement of their IT support systems.

Leveraging Federal IT Investment Using SOA *Executive Version*

Geoffrey Raines

Introduction

Similar to the nation's Fortune 500 leadership, today's Federal leadership teams often find themselves facing significant IT investment and portfolio challenges. They have inherited a computing infrastructure that is often not uniform, and whose technologies span the recent history of computing. The IT infrastructures tend to have the following characteristics:

- **Diverse environments:** Mainframe systems, client/server systems, and multi-tier Web-based systems sit side by side, demanding operations and maintenance resources from a technology marketplace in which the cost of niche legacy technical skills continues to rise. The portfolio of systems is generally written in a number of different software development languages such as COBOL, Java, assembly, and C, requiring heterogeneous staff skill sets and experience in a variety of commercial products, some of which are so old that they no longer offer support licenses.
- **Complex business logic:** The systems often conform to a set of complex business logic that has developed over a number of years in response to evolving legal requirements, Congressional

reporting mandates, changes in contractor teams, and refinement of business processes.

While some systems are new and robust, many are brittle and hard to modify, relying on technical skills not common in the marketplace that become increasingly more expensive. The maintenance tail on these systems is surprisingly high and competes for resources with required new functionality.

- **Inconsistent interfaces:** Interfaces between systems have grown up spontaneously without enterprise planning, over many years. The interfaces are the result of one-off negotiations between various parts of the organization and have been designed using many varied technologies during the organization's IT history, following no consistent design pattern. Recent enterprise architecture efforts have documented the enterprise interfaces in diagrams that resemble a Rorschach inkblot test.
- **Limited sustainment budgets:** Even without the continuous downward pressure on IT budgets brought by competing national requirements and the view that IT should be increasingly viewed as a commodity, there are not enough budget resources or human resources to recast the portfolio of systems to be modern and robust in one action. David Longworth writes, "According

to analysts at Forrester Research, there are some 200 billion lines of COBOL, the most popular legacy programming language, still in use. Nor is it going away: maintenance and modifications to installed software increase that number by five billion lines a year. IBM meanwhile claims its CICS mainframe transaction software handles more than 30 billion transactions per day, processes \$1 trillion in transaction values, and is used by 30 million people.”¹ Given budget constraints, an incremental approach seems to be required.

A Path Forward

SOA, as implemented through the common Web Services standards, offers Federal senior leadership teams a path forward, given the diverse and complex IT portfolio that they have inherited, allowing for incremental and focused improvement of their IT support systems. With thoughtful engineering and an enterprise point of view, SOA offers positive benefits.

Language neutral integration—Web-enabling applications with a common browser interface became a powerful tool during the 1990s. In the same way that HTML defined a simple user browser interface that almost all software applications could create, Web Services define a programming interface available in almost all environments. The HTML interface at the presentation layer became ubiquitous because it was easy to create, being composed of text characters. Similarly, the foundational contemporary Web Services standards use eXtensible Markup Language (XML), which again is focused on the creation and consumption of delimited text. The bottom line is that regardless of the development language your systems use, your systems can offer and invoke services through a common mechanism.

Component reuse—Given current Web Service technology, once an organization has built a software component and offered it as a service, the rest of the organization can then utilize that service. Given proper service governance, including items such as service provider trust, service security, and reliability, Web Services offer the potential for aiding the more effective management of an enterprise portfolio, allowing a capability to be built well once and shared, in contrast to sustaining redundant

systems with many of the same capabilities (e.g., multiple payroll, trouble ticket, or mapping systems in one organization). Reuse, through the implementation of enterprise service offerings, is further discussed below.

Organizational agility—SOA defines building blocks of software capability in terms of offered services that meet some portion of the organization’s requirements. These building blocks, once defined and reliably operated, can be recombined and integrated rapidly. Peter Fingar stated, “Classes, systems, or subsystems can be designed as reusable pieces. These pieces can then be assembled to create various new applications.”² Agility, the ability to more rapidly adapt a Federal organization’s tools to meet their current requirements, can be enhanced by having well-documented and understood interfaces and enterprise accessible software capabilities.

Leveraging existing systems—One common use of SOA is to encapsulate elements or functions of existing application systems and make them available to the enterprise in a standard agreed-upon way, leveraging the substantial investment already made. The most compelling business case for SOA is often made regarding leveraging this legacy investment, enabling integration between new and old systems components. When new capabilities are built, they are also designed to work within the chosen component model. Given the size and complexity of the installed Federal application system base, being able to get more value from these systems is a key driver for SOA adoption. David Litwack writes, “The movement toward Web Services will be rooted not in the invention of radical new technology, but rather in the Internet-enabling and re-purposing of the cumulative technology of more than 40 years. Organizations will continue to use Java, mainframe and midrange systems, and Microsoft technologies as a foundation for solutions of the future.”³

Of course, SOA as a concept has existed for many years, and communications between service consumers and providers have been implemented with a number of protocols and approaches before Web Services. Web Services standards have brought renewed contemporary interest in SOA because of their use of textual XML and its ability to be generated and consumed in diverse computing platforms.

The benefits mentioned above, however, accrue only as the result of comprehensive engineering and a

meaningful architecture at the enterprise level. SOA as a service concept in no way eliminates the need for strong software development practices, requirements-based lifecycles, and an effective enterprise architecture. While SOA done right offers valuable benefits, SOA without structured processes and governance will lead to traditional software system problems.

Increasing Span of Integration

SOA and its implementing standards, such as the Web Services standards, come to us at a particular point in computing history. While several key improvements, such as language neutrality, differentiate today's Web Service technologies, there has been a long history of integrating technologies with qualities analogous to Web Services, including a field of study often referred to Enterprise Application Integration (EAI). One of the key trends driving the adoption of Web Services is the increasing span of integration being attempted in organizations today. Systems integration is increasing both in complexity within organizations and across external organizations. We can expect this trend to continue as we combine greater numbers of data sources to provide higher value information. Ronan Bradley writes, "CIOs often have difficulty in justifying the substantial costs associated with integration, but, nevertheless, in order to deliver compelling solutions to customers or improve operational efficiency, sooner or later an organization is faced with an integration challenge."⁴

Drawing Parallels—"Past Is Prologue"⁵

During the 1970s electronics engineers experienced an architectural and design revolution with the introduction of practical, inexpensive, and ubiquitous integrated circuits (ICs). This revolution in the design of complex hardware systems is informative for contemporary software professionals now charged with building enterprise software systems using the latest technologies of Web Services in the context of SOAs.

Like SOA, the IC revolution was fundamentally a distributed, multi-team, component-based approach to building larger systems. Through the commercial marketplace, corporations built components for

use by engineering teams distributed around the world. Teams of engineers created building blocks in the form of IC components that could then be described, procured, and reused.

Like software services, every IC chip has a defined interface. The IC interface is described in several ways. First, the chip has a defined function—a predictable behavior that can be described and provides some value for the consumer. Next the physical dimensions of the chip are enumerated. For example, the number and shape of pins is specified. Further, the electronic signaling, timing, and voltages across the pins are specified. All these characteristics make up the total interface definition for the IC. Of course, software services do not have an identical physical definition, but an analogous concept of a comprehensive interface definition is still viable. Effective software components also possess a predictable and definable behavior.

Introducing and using ICs included the following considerations:

- **Who pays?** Building an IC chip the first time requires a large expenditure of resources and capital. The team who builds the IC spends considerable resources. The teams who reuse an IC, instead of rebuilding it, save considerable time and expense. A chip might take \$500K to build the first time and might be available for reuse in a commercial catalog for \$3.99. The creation of the chip the first time involves many time-consuming steps, including requirements analysis, behavior definition, design layout, photolithography, testing, packaging, manufacturing, and marketing.⁶ The team who gets to reuse the chip instead of rebuilding it saves both time and dollars. At the time, designs of over 100,000 transistors were reported as requiring hundreds of staff-years to produce manually.⁷
- **Generic or specialty components?** Given the amount of investment required to build a chip, designs were purposely scoped to be generic or specific, with particular market segments and consumer audiences in mind. Some chips only worked for very specific problem domains, such as audio analysis. Some were very generic and intended to be used broadly, like a logic multiplexer. The bigger the market, and the greater the potential for reuse, the easier it was for a manufacturer to amortize costs against a broader base, resulting in lower costs per instance.

- **Increased potential design scope:** By combining existing chips into larger assemblies, an engineer could quickly leverage the power of hundreds of thousands of transistors. In this way, IC reuse expanded the reach of the average engineer, allowing the engineer to leverage resources and dollars spent far in excess of the local project budget.
- **Design granularity:** The designer of an IC had to decide how much logic to place in a chip to make the chip most effective on the marketplace. Should the designer create many smaller function chips, or fewer larger function chips? Families of chips were often built with the intention of their functions being used as a set, not unlike a library of software functions. Often these families of chips had similar interface designs, such as consistent signal voltages.
- **Speed of integration:** As designers became familiar with the details of component offerings, and by leveraging pre-built functions, the speed at which an “integrated” product, built of many components, could come to market was substantially increased.
- **Catalogs:** When the collection of potential ICs offered became large, catalogs of components were then created, and classification systems for components were established. Catalogs often had a combination of sales and definitive technical information. The catalogs often had to point to more detailed resources for the technical audiences that they sold components to.
- **Testing:** Technical documents defined the expected behavior of ICs. Components were tested by both the manufacturer and the marketplace. Anomalous behavior by ICs became noted in errata in technical specifications.
- **Engineering support:** IC vendors offered advanced technical labor support to customers in the form of Application Engineers and other technical staff. Helping customers use the products fundamentally supported product sales.
- **Value chains:** Value chains consume raw components and produce more complex, value-added offerings. ICs enabled value chains to be created as collections of chips became circuit boards, and collections of circuit boards became products.
- **Innovation:** ICs were put together in ways not anticipated by their designers. Teams that designed chips could not foretell all the possible uses of the chips over the years. Componentized logic allowed

engineers to create innovative solutions beyond the original vision of component builders.

One might ask, “Were electrical engineers successful with this component-based approach?” Certainly the marketplace was populated by a very large number of offerings based in some part on ICs. Certainly many fortunes and value chains were created. The cost effectiveness of the reuse approach was validated by the fact that it became the predominant approach of the electronics industry. In short, electronic offerings of the time could not be built to market prices if each chip, specification, module, or component had to be re-fabricated on each project. Reuse, through component-based methods, enabled by new technologies, led this revolution. Yet, the transformation took a decade to occur.

An SOA Analogy

In many ways the IC chip revolution described above is analogous to the effort underway with Web Services today. Clearly Web Service components have analogous interfaces definitions, and defined and documented behaviors that provide some benefit to a potential consumer. One can also reasonably expect that the team producing the Web Service will incur substantial expenses that consumers of the service will not. For example, high reliability requirements for the operation of a service and its server and network infrastructure can be a new cost driver for the provider. To continue the analogy, collections of service offerings are becoming sufficiently large to require some librarian function to organize, catalog, and describe the components. While many SOA projects use Universal Description, Discovery, and Integration (UDDI) for this purpose, other reasonable options exist. Enterprise integration engineers are realizing the ability to more rapidly combine network-based service offerings and a new paradigm, sometimes referred to a “mashup,” is demonstrating the speed at which integration can now occur.⁸ Value chains of data integration are already occurring in the marketplace. A data integrator can ingest the product of multiple services and produce a service with correlated data of greater value. Finally, it is also safe to say that service providers may be surprised at how their services get integrated over time, and they may be part of larger integration that they could not

have foreseen during the original design. (Also note that this same component-based approach is now being examined for genetics work as well. The same interface definition, behavior, cataloging, and reuse discussions are currently occurring, creating a new genetic sub-field known as synthetic genetics.)⁹ In summary, many aspects of the current SOA efforts follow similar component-based patterns, and many of the benefits realized historically by the IC revolution will be potentially realized by SOA efforts.

Reuse

Historic source code reuse—During the 1980s many organizations, including the Department of Defense (DoD), attempted to reuse source code modules with little success. For example, during the DoD’s focus on the Ada language, programs were established to reuse Ada language functions and procedures across projects.¹⁰ The basic reuse premise outlines a process where a producer of a source code module would post the source code to a common shared area along with a description of its purpose and its input and output data.¹¹ At that point, staff from another project would find the code module, download it, and decide to invoke it locally in their source code, and actually compile it into their local libraries and system executables. As an example, the Ada Quality and Style Guide states that, “One of the design goals of Ada was to facilitate the creation and use of reusable parts to improve productivity. To this end, Ada provides features to develop reusable parts and to adapt them once they are available.”¹² For example, Project A might create a high-quality sorting function, and Project B could then compile that function into their own software application.

Though well intentioned, the actual discovery and reuse of the source code modules did not happen on a large scale in practice. Reasons given for the lack of reuse at the time included lack of trust of mission-central requirements to an external producer of the source code, failure to show a benefit to the contractor “reuser” implementing later systems, inadequate descriptions of the behavior of a module to be reused, and inadequate testing of all the possible outcomes of the module to be reused.¹³ All in all, the barriers to reuse were high.

Service reuse—The danger in describing the use of services as “reuse” is that the reader will assume we mean the source code reuse model of the 1980s

described above. In fact, the nature of service reuse is closer to the model of the reuse of ICs by electrical engineers described above, though still having common issues of trust, defined behavior, and expected performance. In plain terms, reuse in the service context means not rebuilding a service, but rather the using again, or invoking, of a service built by someone else.

The enterprise as a whole saves resources every time a project decides to reuse a current software service, rather than creating redundant services based on similar underlying requirements, and adding to an agency’s maintenance portfolio. Since a system’s maintenance costs often exceed the cost to build them, over their lifetime, the enterprise saves not only in the development and establishment cost of a new service but also in the 20-plus-year maintenance cost over the service’s lifecycle. One web vendor stated, “Web Services reuse is everything: on top of the major cost savings . . . , reuse means there are fewer services to maintain and triage. So reuse generates savings—and frequency of use drives value in the organization.”¹⁴ However, we should not assume a straight-line savings, where running one service is exactly half as costly as running two services, because the cost of running a service is also impacted by the number of service consumers. Consolidation can make the remaining service more popular, with a greater demand on resources.

Reuse of a service differs from source code reuse in that the external service is called from across the network and is not compiled into local system libraries or local executables. The provider of the service continues to operate, monitor, and upgrade the service as appropriate. Thanks to the benefits of contemporary Web Service technologies, the external reused service can be in another software language, use a completely foreign multi-tiered or single-tiered machine architecture, be updated at any time with a logic or patch modification by the service provider, represent five lines of Java or five million lines of COBOL, or be mostly composed of a legacy system written 20 years ago. In these ways service reuse is very different from source code reuse of the past.

Some aspects of reuse remain unchanged. The consumer of the service still needs to trust the reliability and correctness of the producer’s service. The consumer must be able to find the service and have adequate documentation accurately describing the

behavior and interface of the service. Performance of the service is still key. ZDnet stated, “Converging trends and business necessity—above and beyond the SOA “vision” itself—may help drive, or even force, reuse. SOA is not springing from a vacuum, or even from the minds of starry-eyed idealists. It’s becoming a necessary way of doing business, of dispersing technology solutions as cost effectively as possible. And, ultimately, providing businesses new avenues for agility, freeing up processes from rigid systems.”¹⁵

Mature SOAs should measure reuse as part of a periodic portfolio management assessment.¹⁶ Actional wrote, “Reuse is not only a key benefit of SOA, but also something that you can quantify. You can measure how many times a service is being used and how many processes it is supporting, thus the number of items being reused. This enables you to measure the value of the service.”¹⁷ The assessment of reuse can be effectively integrated into the information repository used for service discovery in the organization, the enterprise catalog.

SOA as an Enterprise Integration Technology

Enterprise Application Integration (EAI) is a field of study in computer science that focuses on the integration of “systems of systems” and enterprise applications. With the span of attempted systems integration and data sharing continually increasing in large organizations, the EAI engineering discipline has become increasingly central to senior leadership teams managing portfolios of applications.

The fundamental EAI tenets are based on traditional software engineering methods, though the scale is often considerably larger. While the traditional software coder focused on the parameters that would be sent to, and received from, a function or procedure, the EAI engineer focuses on the parameters that are exchanged with an entire system. The traditional coder might have been writing 100 source lines of code (SLOC) for a function, while the EAI engineer might be invoking a system with a million SLOC and several tiers of hardware for operational implementation. However, the overall request/response pattern is the same, and the logic issues like error recovery must still be handled gracefully in either case.

SOA can be considered another important step in a 30-year history of EAI technologies. “SOA

eliminates the traditional ‘spaghetti’ architecture that requires many interconnected systems to solve a single problem.”¹⁸ SOA’s ability to run logic and functions from across a network is not new. Recent examples include Enterprise JavaBeans (EJB) by Sun Microsystems Inc., Common Object Request Broker Architecture (CORBA) by the Object Management Group, and Component Object Model (COM), Distributed Component Object Model (DCOM), and .NET from the Microsoft Corporation. The various methods have differed in the ease with which integration could occur from a programmer’s point of view, the methods for conveying runtime errors, ports required to be open on a network, the quantity of enterprise equipment to operate, and general design approaches to fault tolerance when failures occur.

Like owners of many other systems-of-systems environments, decision makers for command and control systems and intelligence systems have an opportunity to leverage SOA to better enable more rapid integration and reconnection of system components. Services can be developed from legacy data sources and existing investment in procedural logic. Aggregation and correlation services can combine the output of more fundamental services to add value for consumers. Finally, registries can detail the ensemble of IT services that an organization will maintain as a portfolio.

Conclusion

SOA offers Federal leadership teams a means to effectively leverage decades of IT investment, while providing a growth path for new capabilities. SOA provides a technical underpinning for structuring portfolios as a collection of discrete services, each with a definable customer base, acquisition strategy, performance levels, and a measurable operational cost.

A key current challenge for many Federal organizations is the structuring of their IT portfolio around a component-based service model and enforcing sufficient standards within their own organizational boundaries, which can be quite large. As the span of attempted integration continues to grow, the challenge of the next ten years will be enabling that integration model to bridge multiple external organizations that undoubtedly will be using disparate standards and tools.

References

- ¹ Loosely Coupled, David Longworth, “Service reuse unlocks hidden value”
<http://www.looselycoupled.com/stories/2003/reuse-ca0929.html> 29 Sept. 2003
- ² Peter Fingar et al., “Next Generation Computing: Distributed Objects for Business”, SIGS Books & Multimedia, New York., 1996
- ³ Internet World Magazine, David Litwack, “Web Services Has the Biggest Hype Machine Behind it of any Technology Today. Here is Why You Should Be Excited Anyway”
<http://iw.com/magazine.php?inc=060102/06.01.02ebusiness1.html> 1 June 2002
- ⁴ GDS InfoCentre, Ronan Bradley, “Agile Infrastructures”
<http://gdsinternational.com/infocentre/artsum.asp?mag=184&iss=150&art=25901&lang=en> 28 March 2008
- ⁵ William Shakespeare, The Tempest
- ⁶ Intel, “How Chips Are Made”
<http://www.intel.com/education/makingchips/preparation.htm> 28 March 2008
- ⁷ Design World, Electronic Design, C. Panasuk, “Silicon Compilers Make Sweeping Changes in the VLSI”, Sep 20 1984, pp. 67-74.
- ⁸ Programmable Web, “Mashup Dashboard”
<http://www.programmableweb.com/mashups> 28 March 2008
- ⁹ International Genetically Engineered Machine Competition (IGEM), “Registry of Standard Biological Parts”
http://parts.mit.edu/registry/index.php/Main_Page 28 March 2008
- ¹⁰ Department of Defense, Ada Joint Program Office, “Ada 95 Quality and Style Guide”
http://www.adaic.com/docs/95style/html/sec_8/ 28 March 2008
- ¹¹ B. W. Boehm, et al., “An environment for improving software productivity.” Computer, June 1984.
- ¹² Ada Joint Program Office
- ¹³ Will Traez, “Software Reuse: Motivators and Inhibitors.” Proceedings of COMPCON S’87, 1987.
- ¹⁴ Progress Actional, “Web Services and Reuse”
<http://www.actional.com/resources/whitepapers/SOA-Worst-Practices-Vol-I/Web-Services-Reuse.html> 28 March 2008
- ¹⁵ ZDnet, Joe McKendrick, “Pouring cold water on SOA ‘reuse’ mantra”
<http://blogs.zdnet.com/service-oriented/?p=699> 30 August 2006
- ¹⁶ Eric Roch, “SOA Service Reuse”
<http://blogs.ittoolbox.com/eai/business/archives/SOA-Service-Reuse-14699> 28 March 2008
- ¹⁷ Progress Actional, “Web Services and Reuse”
<http://www.actional.com/resources/whitepapers/SOA-Worst-Practices-Vol-I/Web-Services-Reuse.html> 28 March 2008
- ¹⁸ Ebiz, Dr. Chris Harding, “Achieving Business Agility through Model-Driven SOA”
<http://www.ebizq.net/topics/soa/features/6639.html> 29 January 2006

MITRE

www.mitre.org

©2009 The MITRE Corporation
All Rights Reserved
Approved for Public Release
Distribution Unlimited
Case Number: 08-1643

