Using Service-Oriented Architectures for Evolvable Software Systems

Dr. Vincent A. Schmidt
Air Force Research Laboratory / HECS
2698 G Street
Wright-Patterson AFB, OH 45433
Voice: 937-255-8363; Fax: 937-255-6555
Vincent.Schmidt@wpafb.af.mil

Abstract

Evolvable Software Systems (ESS) are software systems designed to adapt to changes in usage and in the work environment. It is especially important to build adaptability into enterprise applications in order to extend the lifetime and reduce the overall cost of deployed software. One method worth considering is the use of Service-Oriented Architectures (SOA), which have become very popular for implementing business software solutions due to their modularity and flexibility. This paper presents arguments in favor of implementing enterprise-level command and control software systems as Evolvable Software Systems using SOA practices.[1]

1 Introduction

The United States Air Force continues to have an interest in the science of software system development. By conservative estimates, over fifty percent of the cost (in money, functionality, schedule, and risk) of fielding new systems is associated with software. To save time, money, and related resources, it is necessary to find ways of reducing the costs associated with the software life cycle. One reasonable approach to reduce this cost is to extend the life of deployed systems by using technologies that allow the applications to adapt as the nature of work changes.

Various technological and methodological concepts have been introduced as a hedge against the changing nature of software needs. One of the most popular contemporary approaches expresses the system in terms of network-centric operations, which promote significant interoperability among loosely coupled systems. This interoperability is expressed using standardized data formats and common network-accessible repositories. Embracing net-centricity reduces the overhead associated with generating new interfaces as new systems come online and expect to share data, since specific interfaces don't have to be developed for each pair of data-sharing systems.

Unfortunately, even the incorporation of net-centric concepts is often not enough. The nature of the work and work requirements frequently evolve and change over time. As this occurs, the existing systems become obsolete, no longer supporting the work requirements. To postpone (or even avoid) obsolescence and continue a productive lifecycle, software needs to be able to adapt to support work as it changes over time. This article investigates one promising technology that could be employed to support such changes: the use of Service-Oriented Architectures (SOA).

2 Background

Evolvable Software Systems (ESS) are systems capable of adapting to change in a dynamic fashion. The umbrella defining ESS is vast, and there are many classifications and categories of ESS. Some ESS operate with complete autonomy, while others manage user-initiated changes. Evolutionary changes can range from optimizations almost undetectable to the user to a complete overhaul of the user interface. The types of changes, the degree to which a user is involved, and other issues define the category of ESS. In the Cognitive Systems branch of the Air Force Research Laboratory (AFRL/HECS), the Progressive

[1]This document cleared for public release by AFRL Public Affairs, AFRL-WS-06-0282.

Software System (PSS) is the currently most appropriate type of ESS related to branch research. Progressive Software Systems automate the adaptation of low-risk modifications based on software usage and interface requirements, while encouraging the users to initiate appropriate higher risk modifications. (Reference [5] discusses PSS in more detail.) The concepts illustrated in this article are not specific to PSS, so the terminology refers generically to ESS.

The desire to create evolvable software is not new; it has existed in many forms and has been known by various names. Some evolutionary paradigms are intended to be fully automatic, while others merely support generic upgrades, integrated manually. The objective has always been to create maintainable high-quality software that is easy and inexpensive to modify when needed. This goal has been attempted through ideals (code reuse, modularity), software design methodologies (formal design, agile design, object-oriented design), programming styles (keyboard composition, complete system specification), development language (C, Java, scripting languages, interpreted languages), and other methods. Each has met with some success. Technology improvements continue to make evolvable software more practical, allow new ideas to be investigated, and regenerate and reintroduce old ideas once more. Even with limited past success, the target of designing evolvable and adaptable software remains an important goal in the software development community.

For enterprise application software[2] to be effective, it must be designed and implemented to support the work environment. In fact, the objective of all enterprise software should be to support the work. (Software has historically been data-centric or user-centric, instead of work-centric.) Software flexibility is the key to allowing software to be used as a work tool, versus enslaving the users to a specific software "solution." The intent is to produce loosely coupled, well-integrated systems that can be easily modified or upgraded. In an ideal environment, these upgrades are remotely or automatically managed, or can be performed in the field by non-technical users without interrupting system usage. Traditional solutions typically include the development and integration of field-replaceable modular components. These components are often explicitly coded as static (compiled) modules, but could also be integrated as interpreted operations such as macro functions or configuration options.

Alternatively, the availability of increasingly higher bandwidth allows the incorporation of network-distributed solutions to software capability management issues. Data and functionality can be distributed among local and non-local computing nodes. This distribution need not be accomplished at a modular level: even individual components can be distributed. Distributing system operations has the potential to promote greater flexibility in enterprise software, since the remote components are generally loosely coupled and can be upgraded (hardware, software, or both) with minimal direct impact to system configuration. This is particularly important to ESS; components must be able to be "easily" exchanged or enhanced without having to re-engineer and redeploy the application.

Examples of technologies and software protocols supporting various levels of distributed computing are easy to find, and serious research in distributed, parallel, and real-time computing is a subculture of its own. Low-level and platform-independent APIs include Remote Procedure Call (RPC) and Java's Remote Method Invocation (JMI). Similarly, mid-level libraries such as PVM and MPI, and high-level packages such as the Common Object Remote Broker Architecture (CORBA), have also been used to produce interoperable, platform- and language-independent software solutions. There are many other technologies that could also be used to support ESS designs (see [6] for more examples).

These approaches can support many of the needs of evolvable software systems. One of the most promising contemporary technologies for implementing ESS takes advantage of Web services — Service-Oriented Architectures (SOA).

3 Discussion

3.1 SOA Concepts

The Service-Oriented Architecture (SOA) concept deals with the availability and presentation of resources. SOA is about providing functionality and data as independent and remotely accessible stateless services. Services are designed to be reused, and promote reuse. SOA-based software systems typically take advantage of a loose coupling of separate, stateless web services distributed across multiple computers.

[2]For our purposes, we define enterprise software in contrast with other types of software: desktop applications, operating systems, embedded solutions, etc. Enterprise software tends to support a specific work domain.

From a computational perspective, a service-oriented architecture relies on web services and network connectivity to promote interoperability. Unlike traditional web pages, shopping baskets, and Javascript applications, these web services are generally intended for machine-to-machine interactions, and not to the human consumer (although human-readable interfaces are generally made available for simple testing). Web services are similar to earlier distributed capabilities advertised using Remote Procedure Call (RPC), but with a slightly different distribution of implementation and execution overhead[3].

The popular model of SOA is divided into two segments: the directory of services and the service providers. In theory, the enterprise application connects to a unified directory to request a type of service, and then is redirected to a machine providing the requested type of service. In practice, no single "grand unified" directory service has emerged; vendors and service providers are promoting their own local "unified" directory services. (From an implementation standpoint, even the specific directory service request currently has to be embedded into the source code, so we are not "googling" for services, but making a specific request.)

Once a service provider is identified and the interface is defined (using the Web Services Description Language — WSDL), the enterprise application uses the SOAP[4] protocol (embedding the service request into the HTTP protocol) to connect to the (stateless) service provider. Both the request and response messages are encoded using the Extensible Markup Language (XML) and passed along within a SOAP package. The transaction is completed with the response is returned to the requester. Since all communications are done via HTTP, only HTTP-related security and access issues need to be resolved to support the communications overhead. (No additional servers or holes in the firewall are required.)

Technologists agree that the SOA paradigm is especially well-suited for modeling and implementing business rules. This is mainly due to the flexibility and modularity of the code base: specific business relationships and operations are implemented as small stateless components, capable of being revised or removed quickly as the need arises. Business logic is precisely the core component of the typical enterprise application.

3.2 SOA Deployment

Figure 1 depicts what many consider to be a typical configuration of an SOA enterprise application, simplified to omit the directory service component. Figure 1(A) shows the application deployment. As shown in the figure, a portion of the application remains on the user's local machine, while one or more remote machines host servers providing an array of applicable web services. If the majority of the application is provided by web services, the local component is small and lightweight. From a software design perspective, Figure 1(B) illustrates how integrating SOA concepts is simply achieved through appropriate protocol accesses to the web servers providing the desired services.

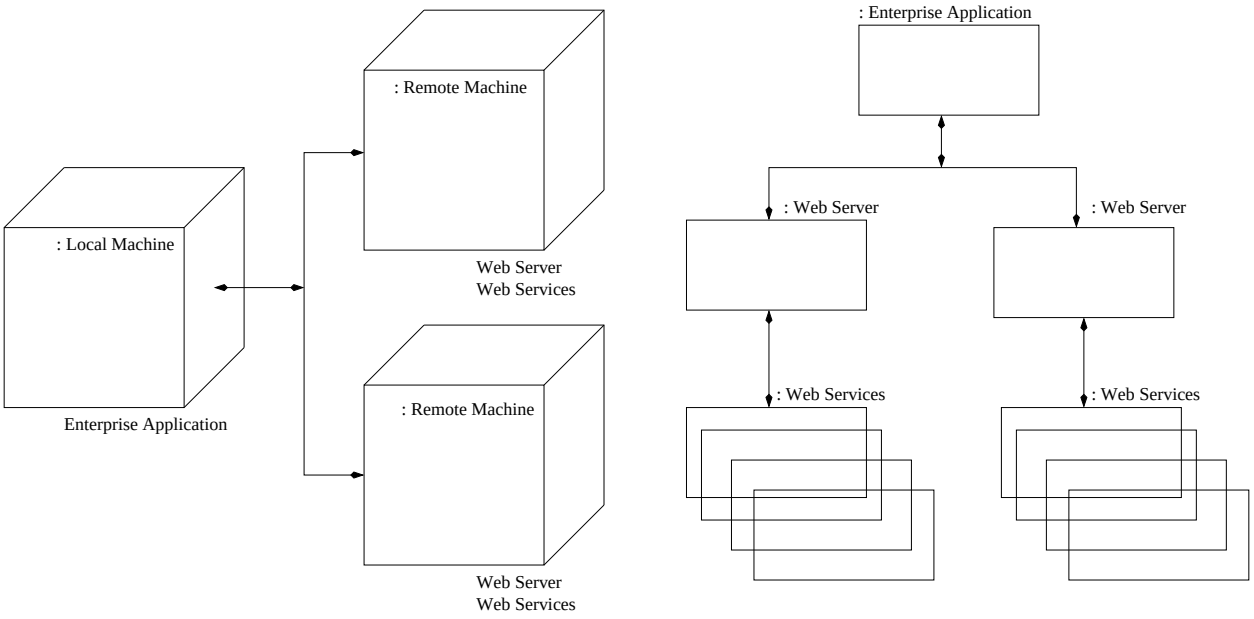
The strategy for building enterprise solutions for the US Air Force is to generate systems that not only meet the work requirements, but are capable of dynamically transforming to service the changing work environment over time. Such an Evolvable Software System can certainly leverage SOA techniques by using services provided by the user's local computer and other machines on the network. Specific remote sites and alternate systems could provide core system capabilities as modular web services. Even if the system is designed to ensure critical capabilities are locally available, remote web services could be designed to meet certain communication needs, such as alerts for upgrades and security fixes, metadata updates, and configuration management functionality. Since web services promote common interfaces and interoperability, this type of design easily fits into the network-centric concept.

As an example, consider an application designed to schedule and report scheduled aircraft maintenance activity, and track aircraft currently undergoing maintenance. A lightweight enterprise application might rely on a collection of web services to obtain data regarding current aircraft inventory infor-

[3]At first glance, SOA looks eerily conceptually similar to Remote Procedure Call (RPC), a software networking and communication technology that has been in use for a couple of decades. Although the specifics of the protocols differ, many aspects of these paradigms are strikingly similar. Those already familiar with RPC have a head start understanding and integrating SOA.

The feature of SOA that sets it apart from technologies such as RPC and Java's RMI is the reliance of SOA on the existing ubiquitous HTTP protocol, coupled with the transfer of data encoded as XML. This allows web services to be provided by existing web servers. (Of course, encoding all messages as XML also substantially increases the communications overhead involved. This is arguably a small price to pay for flexible enterprise software.)

[4]According to www.wikipedia.com, SOAP was originally acronymic for Simple Object Access Protocol, but the acronym has been dropped as of SOAP version 1.2, with emphasis on object interoperability rather than mere object access.



(A) Deployment Diagram

(B) Object Diagram

Figure 1: Distributed SOA Configuration

mation, currently scheduled maintenance, unanticipated maintenance, and expected completion dates. A different set of services might define functions, calculations, and reporting capabilities that describe the business logic. The application running on the local machine might only be responsible for generating appropriate displays based on the work requirements. Such a configuration promotes significant flexibility and reuse, since both the data and required computations are not coded directly into the application. As new data sources become available, the web services providing the dataset and business rules can be updated dynamically to take advantage of the new information; no site-specific software upgrades would be required.

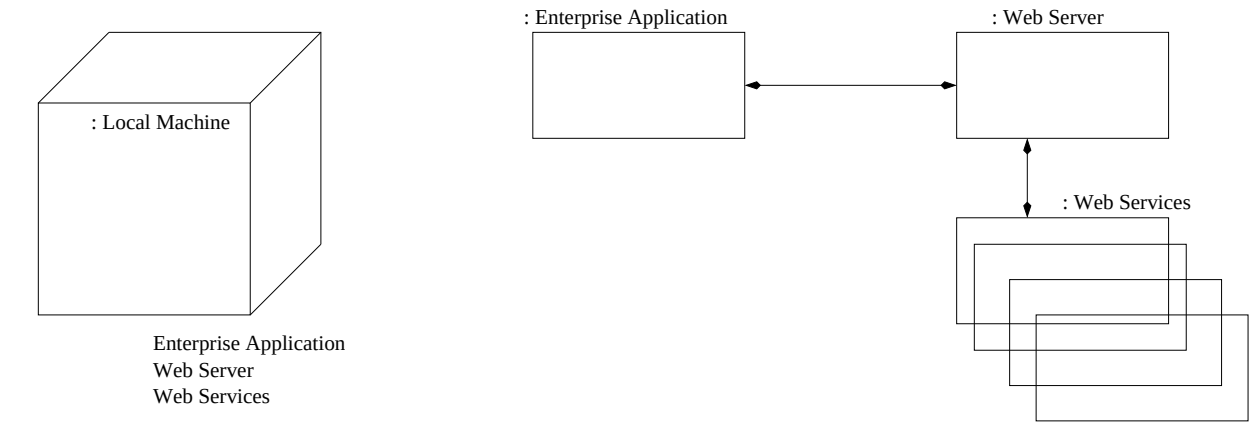
Although typical implementations of service-oriented architectures use web services as the mechanism for sharing data and functionality, it is not strictly necessary to distribute the services across machines. In many cases, a single machine could execute both the enterprise application and the service provider, as illustrated in Figure 2. The deployment diagram in 2(A) identifies a single local machine running the application, web server, and all services. From an implementation standpoint, Figure 2(B) shows the application communicating to the service provider using the same protocols that would be used in a distributed configuration. The only difference is, the centralized configuration is completely self-contained, not requiring additional computers.

Using the previous aircraft maintenance example, the services could all be moved to the local machine to more closely adhere to the centralized SOA configuration. In this case, these services might need to be modified to query local databases, with data tapes loaded into local databases at scheduled intervals. If each service implements a specific subset of business logic for the enterprise application, these services could be controlled as individual software components for ease of maintenance.

3.3 SOA Advantages for ESS

Evolvable Software Systems must be designed to allow for a clear and efficient path supporting a progression of capability as the nature of work changes and matures. SOA methods may play a significant role in providing these facilities. Benefits to integrating web-service-based SOA techniques into enterprise-level ESS applications include, from a software engineering perspective:

- Enforced modular system design — Services are implemented as independent stateless operations. Each web service provides the solution to a specific logical requirement. This kind of



(A) Deployment Diagram (B) Object Diagram

Figure 2: Centralized SOA Configuration

layout promotes modular design. Each service module is designed to capture a specific business rule.

- Standardized interfaces — The interfaces developed to access web services are standardized. Any development suite can be used to generate an application or a new web service, as long as the standard interfaces are adopted by the new service.
- Language independence — There is no requirement to create web services using a specific computer programming language. Web services can be developed using the most appropriate language for implementing the business logic, as long as the inputs can be passed from the application through the web server to the servicing module, and the module can return the results in the standardized format.

Service-Oriented Architectures provide an exceptional mechanism for supporting the evolution and rapid responses to fast-paced changes in business rules:

- Service reuse — Because individual services are stateless and designed to meet specific business logic, a library of services can be quickly built over time across multiple servers. (The concept of reuse is frequently raised in the software engineering community, but actual reuse is often tedious and difficult. There is no silver bullet, but perhaps SOA can push reuse forward.) Generic services can be reused and streamlined. A library of common services or service patterns will emerge as web services become more popular and available.
- Ease of upgrading services — Modules can be replaced, upgraded, extended, or even ignored as business requirements change. When new requirements are close the the original design, existing services can be modified to meet the new requirements. New services can be added dynamically (ignoring existing services) if the requirements are sufficiently different from the original module design. Services should be configuration-controlled to track and manage module changes.
- Greater control of data — Business logic is distributed among nodes, not contained locally within the application. This means that enterprise data is not contained locally. The implication is that processes involving (i.e.) multi-level security can be implemented more easily at the enterprise application level, since modules would not be called until login information is validated. This limits the data and information displayed and retained within the local application.

Web services offered through Service-Oriented Architectures also promote good technology management:

- Dynamic upgrades — Web services are executed impulsively: that is, the service is initially inactive, then is executed, and becomes inactive again until another request is made. This usage pattern allows web service modules to be upgraded *in situ* between impulses, without requiring the web server or any enterprise application

to be explicitly recompiled or undergo extensive restarts.

- Redundancy reduces risk — If multiple services located on different network nodes can provide the same business logic, one node can provide the required solution if another becomes unavailable or is slow to respond. Business logic redundancy reduces the risk that a service cannot be provided, and distributes the work among multiple nodes.

The US Air Force already has a substantial investment in adopting network-centricity in order to ensure interoperability among new and existing applications and data generation platforms. Since a web-service instantiation of SOA operates using commonly accepted standards already on the net, integrating these services into enterprise ESS applications is a logical solution that fully supports network-centric concepts.

Service-Oriented Architectures and the use of web services are not the only way of integrating ESS concepts into enterprise software. Methods that employ modular and dynamically reconfigurable software configurations are candidate approaches, and the distributed computing community also offers useful solutions. Certainly, Internet and web-based networking technologies may be ideal contributors due to their loosely coupled structure. For example, using techniques such as AJAX[5] can potentially extend the life and reduce the lifecycle costs of C2 applications by improving usability and performance.

Regardless of the approach used to implement and field evolvable software, there are open research issues to be addressed. One obvious example is configuration management. If enterprise software is delivered to several sites, and each site manages its work environment differently over time, the software will need to adapt in different ways to support the changing work. Even if the work across sites remains similar, the software will certainly evolve in different ways. When should these systems be rebaselined? How are the different configurations managed? How are they tracked? What is the impact of web service changes to each site? A combination of technology and policy must be used to resolve these research issues.

[5]AJAX, Asynchronous Javascript and XML, is not a specific technology, but a technique combining aspects of Javascript and XML to improve the user's Web User Interface experience by reducing the number and frequency of certain types of time-consuming web page refreshes. This also reduces server calls, which reduces bandwidth usage.

4 Conclusions

It is important to stress that the strategy of developing ESS solutions is not a new software development paradigm. ESS embraces existing methods in an effort to capture best-practice approaches supporting the development of modular, flexible, and upgradeable software. Established software design methods and paradigms (object-oriented, for example) can all be used to create evolvable systems. Ultimately, the mission of ESS is to promote the evolution of the software as the work requirements change, allowing the software to continue supporting the work.

Continuing research includes facets of ESS such as formal design expectations, configuration management issues, practical implementation techniques, development paradigms, and other related areas. We anticipate that the software development community will be enthusiastic about using evolvable design concepts to generate useful, practical, and reusable software. Truly evolvable software systems will be able to meet the customers' work requirements long after the original work environment has changed beyond recognition.

Disclaimer

The views expressed in this paper are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

References

[1] David S. Alberts, John J. GArstka, Richard E. Hayes, and David T. Signori. *Understanding Information Age Warfare.* CCRP Publication Series, Washington, DC, USA, 2003.

[2] David S. Alberts, John J. Garstka, and Frederick P. Stein. *Network Centric Warfare: Developing and Leveraging Information Superiority.* CCRP Publication Series, Washington, DC, USA, 1999.

[3] David S. Alberts and Richard E. Hayes. *Power to the Edge.* CCRP Publication Series, Washington, DC, USA, 2003.

[4] Simon R. Atkinson and James Moffat. *The Agile Organization: From Informal Networks to Complex Effects and Agility.* CCRP Publication Series, Washington, DC, USA, 2005.

[5] Kendall O. Conrad and Vincent A. Schmidt. Progressive software systems: Dynamic software for a dynamic workplace. In *Proceedings of the Ninth IASTED Conference on Software Engineering and Applications*, Phoenix, AZ, November 14–16 2005.

[6] Kendall O. Conrad and Vincent A. Schmidt. Practical technologies for implementing evolvable software systems. In *Proceedings of the 2006 International Conference on Software Engineering Research and Practice (SERP'06)*, Las Vegas, NV, 2006.

[7] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.