

# The SOA Magazine

## Feature Article



### Refactoring Considerations for Service-Enabling Applications

by CP Jois

Published: March 13, 2008 (SOA Magazine Issue XVI: March 2008)

[Download this article as a PDF document.](#)

*Abstract: Service-based computing is taking traction, and why not, services hold the visionary promise of transforming the manner in which software is built, distributed and managed. The standards are evolving and are making services more mature and absolutely more implementable than ever before. According to a Gartner forecast from several years ago, by the end of 2008 SOA will provide the basis for 80 percent of development projects. This is a remarkable prediction considering it wasn't so long ago that many in the industry were complaining of a lack of mature SOA platforms.*

*To truly achieve a state of service-orientation, you need to apply many of the best practices that have been documented by industry experts. But, doing so within the context of specific business requirements can be challenging.*

*In the realm of an enterprise contemplating a services-based engineering approach to developing solutions for the business the one factor to note is that most, if not all, business software solutions are evolutionary in nature. This means that they leverage existing software investments and enhance them to deliver increased business value. It is at this point that the services approach encounters perhaps its greatest obstacles because we, as service architects, are then forced to think in terms of the trade-offs of carrying out a foreign SOA methodology.*

*This article provides a practitioner's perspective on these issues with an emphasis on refactoring.*

#### **Introduction: Taking the First Step**

Re-orienting oneself to think about software application development as "assembly" rather than "ground-up development" is a significant shift. It is in many ways similar to the challenge practitioners went through when object-oriented programming emerged.

Accomplishing this transition is fundamental to leveraging the value that services-based computing can bring about. The literature on the subject is rapidly expanding and multiple approaches have proposed how enterprises can get started with many suggesting that pilot projects are the best way to take that first step.

A move toward establishing a service-oriented environment includes work on many different fronts. Essential to getting a services-based solution developed, amongst other things, is having a set of services to use and reuse. Secondly, in order to use a collection of services effectively together, we need a means of composing them.

Erl in his treatise on services-based architectures [REF-3] effectively demonstrates the different approaches to engineering a service-oriented solution by exploring top-down, bottom-up and hybrid methods. Krafzig, Banke and Slama [REF-4] describe the impacts of these approaches on various aspects of solution engineering and team management.

Typically, a team decomposes a candidate solution top-down to identify the set of services needed to form the solution (or grows that list from the bottom-up). In one of his follow-on works that focuses on the design of services [REF-5], Erl

refers to this as identifying "service candidates".

Once the team is past identifying what services are needed to render the business requirements, a services manifest becomes available. At this point it is important to determine from where these services will be sourced. Decisions need to be made as to whether the services will be newly built or if existing software would be leveraged to create them (and, if existing software is being leveraged, whether it is leveraged in full or just in part).

Regardless of the outcome of these decisions, some of the required services will need to be developed from legacy applications by repurposing the logic within them. And this where the problem begins to take form. Legacy environments come in different shapes and sizes – different platforms, programming languages, protocols, architectures and so on. Leveraging these intrinsically means finding ways to leverage the functionality embedded within them and exposing them as services that can be effectively used by new solutions. This typically requires that existing legacy applications be refactored.

## **The Importance of Refactoring**

While refactoring existing software investments into services is a reasonable first step, it is often a difficult one, for a variety of reasons.

Cost tops the list. Corporate finance departments need a business case to support projects. Refactoring not only costs money but a good measure of it. So how does a team effectively justify the costs of enabling existing software as services when not only will the benefits of doing so be accrued over time but will also be hard to quantify?

Refactoring efforts are functions of application size and structure. The larger the application, the more complex the refactoring activity. The structure of an application to be refactored is an equally important factor. If the application is not well designed, chances are that the refactoring effort will be not very clean and will become extra time consuming.

As an example, imagine routines that embody business rules reflecting an employee union contract at a large company. Now further imagine these routines redundantly coded throughout the application (anyone out there that's worked with large enterprise applications knows that that this is not uncommon). These types of legacy characteristics make the task of refactoring a serious challenge and also highlight the critical importance of domain knowledge.

We need also to understand that refactoring in this context essentially means unbundling of a software application. The shift to client-server brought about a change from the monolithic software programs. Distributed multi-tier application models extended that further. But in each case while the size of the monolith decreased, it retained the core character of a program which bundled multiple capabilities within.

With service-centric computing we don't have a monolith of any size bundling capabilities any more. In the world of service-orientation, each bundle is a capability – unique and autonomous. Even if we simply stayed with static service models, the benefits brought about by moving to services as the standard implementation medium are huge. Once again, since refactoring essentially means unbundling, identification of business logic that can qualify being viewed as a capability is critical to its success. In most situations, applications that are worth refactoring will often be those that are critical to a business anyway, making the task of refactoring it all the more difficult.

## **Service Interoperability**

It is relevant to emphasize at this point that one of the high level benefits of using services is achieving interoperability. Interoperability in turn essentially refers to the sharing of data [5].

Let's take the example of a large insurance company that carried out an SOA project last year. Despite the involvement of focused steering groups and committees, the discussions around granularity became so long-drawn that the overall timeline of the candidate project was severely impacted.

Many enterprises underestimate the education, effort, thinking and learning curve involved in getting this right. And it's worthy to note that in the case of this insurance company, only two legacy applications were part of the SOA project's scope, meaning that they provided the primary source from which business services were to be derived (eventually, 16 services were created).

Helbig and Scherдин [REF-6] validate this point in their description of work at a large logistics provider wherein an evolutionary approach was pursued first to transform the architecture and application landscape prior to embarking on

a services initiative.

One has to remember that the primary goal of a project team is to develop a software solution to satisfy a business need. Refactoring requirements can come about as a cascade activity arising out of the need for enabling certain legacy business logic as services. The business group asking for a new service-oriented solution will likely not have an appetite to listen to all the reasons why service-enabling an existing application involves a process filled with speed bumps.

Matthias Kaiser of SAP Research [REF-7] makes a reference to work done by Norbert Bieberstein and his colleagues in developing guidelines for overcoming several problems of complex monolithic systems. Described in those guidelines are useful pointers to refactoring existing applications into services.

Refactoring existing business logic is but one aspect of putting a service-oriented solution together. Architecture, performance, infrastructure, tools - all these considerations follow with a host of additional issues that demand consideration.

### **Transforming Maintenance Cycles into Refactoring Opportunities**

What does this all mean? Well, the lesson here is that if an organization intends to make the transition toward SOA; it is not likely going to be successful just picking a candidate project and focusing on refactoring with a service-centric perspective.

So, what then is truly needed? The answer, to a great extent, lies in readiness. While 'being ready' is a large topic, there are certain aspects of readiness which, when established early, can make a huge impact on how your first service-oriented solution will be built.

Readiness entails multiple areas – business logic, legacy refactoring, service infrastructure, organizational changes, education, new perspectives on quality, and more. Let's begin by discussing readiness in the context of service-enabling existing applications.

One path to readiness when it comes to leveraging what's residing in existing applications is to extract defect and enhancement handling workflow logic. This allows an organization to establish an environment where service creation becomes an on-going process.

To elaborate further, this approach seeks to turn the traditional maintenance lifecycle into a "value creating activity". Software maintenance engineers are essentially asked to engage in refactoring activities each time they touch code (i. e. each time they open up code to fix a defect or handle enhancement requests). The engineering team is asked to perform small but definite movement towards localizing business logic, extracting it and further exposing it as a service (s).

As small as each individual activity may seem, the impact of collective action over a period of time is huge. So significant, in fact, that the very economics of refactoring are augmented.

It is important to ensure that refactoring is focused on areas of business logic that matter. For this it is essential that solution owners have a deep understanding of which aspects of the solution are impacted most via change cycles. Tracking change metrics provides insight into areas that undergo most change, thereby indicating whether it is worth exposing them as services. The advantages of integrating the task of refactoring into maintenance cycles is that it adds only a few steps to the process but produces lasting value in the form of services that then allow for repeated composition.

Erl's reinforcement of this point in his work on service design [REF-3] and further his practice-related recommendations are very encouraging. Erl speaks to the above topic in the section where he covers how service-orientation ultimately brings about increased organizational agility but can, in the short term, potentially slow down tactical software delivery. To summarize, maintenance cycles provide a great opportunity to renew and revitalize an application portfolio.

### **Refactoring and Service-Centric Computing**

One of the visionary promises of service-oriented computing is the level of runtime-flexibility provided by the on-

demand "composability" of services.

Ultimately, in the world of business, automation is a primary means of increasing a company's competitive advantage. Hardware is available to anyone, but the uniqueness of an automated business process can be preserved and leveraged in response to specific business requirements.

However, there is more to competitive automation than just designing effective automation systems. The reality is that the very same processes and functions embodied within these systems will almost always change frequently over time. When that happens, the systems need to change also. How well they can be adapted to business change is the true indicator of the extent to which automation can continue to elevate competitiveness because, as any entrepreneur knows, timing is critical to business success.

System designs have historically been inherently rigid. Despite several advances in tools, architecture and programming languages, software development teams are not expected to build in the levels of flexibility required to rapidly facilitate future changes.

Ideally what is needed is the ability to perform 'zero-programming business process change' – the ability to refactor business processes without resorting to impactful software programming changes.

Service-oriented computing is the software engineering community's answer to this, and it all starts with the creation of services. However, for services to be really useful, they need to be composed into a chain of services that collectively automate a business process.

This leads us to service composition, a topic of deep research in itself. Brian Blake's perspective on composition [REF-1] is educative and thought-provoking and Erl's work on service composition design [REF-5] is particularly interesting because the situations referred to in his text are based on actual practice. Hughns and Singh [REF-2] further describe QoS guidelines for service architectures and make special reference to the issue of runtime service transaction blocks.

In a recent IEEE article on services computing it stated Charles Petrie's vision for Web services as "... that of a worldwide wizard, who knows at any time, in any situation, everywhere, how to serve its customers best". Service composition and the optimality of service chains represent the next step toward achieving this state. However, to achieve optimum service composition design requires that each service effectively encapsulate its underlying business logic. When that logic resides within legacy applications, the extent to which a service can actually be effective is often determined by the extent to which that logic was successfully refactored in support of service-orientation.

## Conclusion

The road to engineering a services based solution can be complex, especially when it comes to solutions that intend to render repeatable value in an enterprise already filled with legacy systems. Critical to achieving excellence in an SOA project is paying close attention to the task of refactoring those legacy environments in support of individual services, because each service eventually becomes a building block of a service-oriented solution.

## References

- [REF-1] "Decomposing Composition: Service Oriented Software Engineers", M Brian Blake, IEEE Software 0740-7459/07
- [REF-2] "Service-Oriented Computing: Key Concepts & Principles", Hughns & Singh, IEEE Internet Computing 1089-7801/05
- [REF-3] "Service-Oriented Architecture: Concepts, Technology, and Design", Thomas Erl, Prentice Hall/PearsonPTR, [www.soabooks.com](http://www.soabooks.com)
- [REF-4] "Enterprise SOA: Service-Oriented Architecture Best Practices", Dirk Krafzig, Karl Banke & Dirk Slama, Prentice Hall
- [REF-5] "SOA Principles of Service Design", Thomas Erl, Prentice Hall/PearsonPTR, [www.soabooks.com](http://www.soabooks.com)
- [REF-6] "Creating Business Value through Flexible IT Architecture", Johannes Helbig and Alexander Scherдин, IEEE Computer
- [REF-7] "Toward the Realization of Policy-Oriented Enterprise Management", Matthias Kaiser, IEEE Computer 0018-9162/07
- [REF-8] "Task Driven Computing", Zhenyu Wang & David Garlan, Carnegie Mellon University, CMU-CS-00-154

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL



[Home](#) [Past Issues](#) [Contributors](#) [What is SOA?](#) [SOA Glossary](#) [SOA School](#) [SOA Books](#) [About](#) [Legal](#)

Copyright © 2006-2008  
SOA Systems Inc.