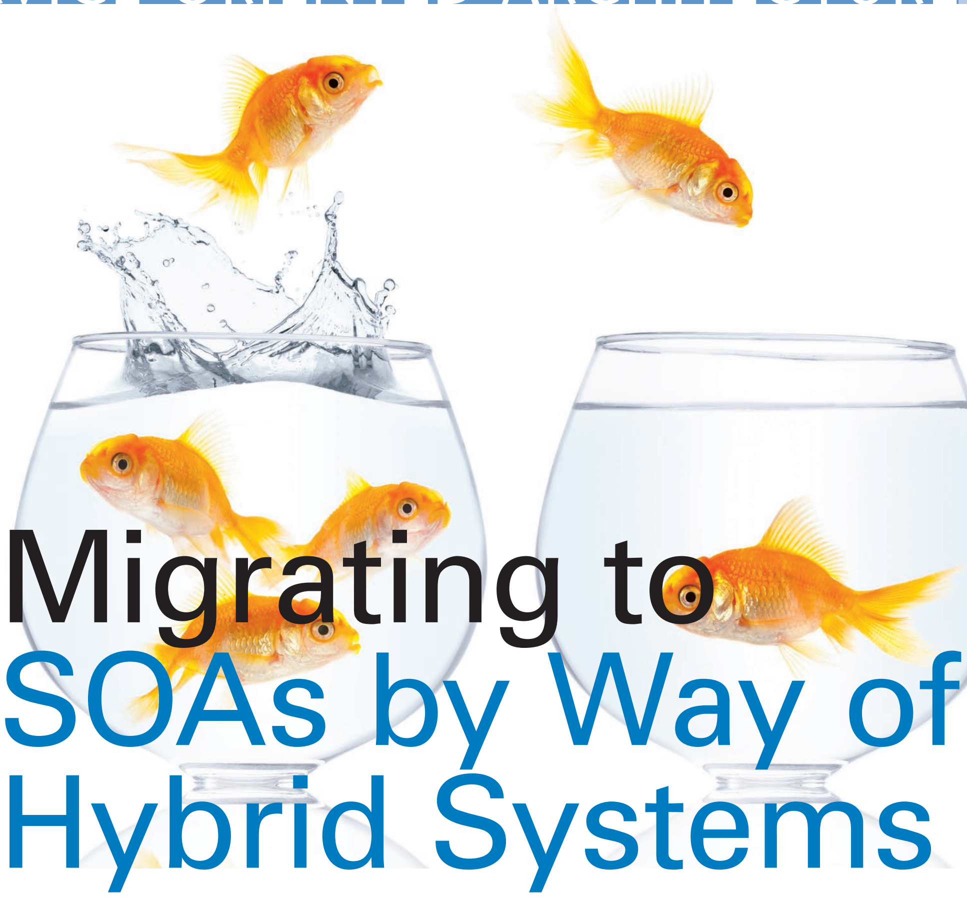


Migrating to SOAs by Way of Hybrid Systems

John Hutchinson, Gerald Kotonya, James Walkerdine, Peter Sawyer, Glen Dobson, and Victor Onditi

A progressive-evolution strategy for migrating systems to service-oriented architectures should minimize the risk to investments in existing software systems while letting businesses exploit the benefits of services.

Software systems let many businesses stay competitive, creating a relationship between a business's success and its software systems' fitness for purpose. To remain fit for purpose, systems must evolve, and a given software system's usefulness depends on how well it reflects the needs of its changing environment.[1]

You might view service-oriented architectures as just another phase in this evolution. But SOAs represent a major shift in how organizations implement and, potentially more importantly, deliver new business functionality to users. The flexibility offered by services and SOAs poses a real threat to investments in existing software systems. A pragmatic solution is for businesses to migrate their systems to SOAs by developing hybrid systems through a process of *progressive evolution*.

Providing adequate processes and tools for achieving this evolution hasn't been a particularly active area of research within the community. In this article, we look at some of the issues and motivations, and describe some approaches that might contribute to the development of suitable methods. We also examine some of the remaining challenges.

Adopting SOAs

SOAs present a compelling vision for businesses. Conceptually, services bring together a layer of business functionality and a layer of technological implementation. From a business perspective, services are about the appropriate packaging of functionality and flexibility. Capturing system knowledge in a way that's appropriate for both business users and developers is difficult.[2] However, services provide a mechanism for packaging functionality in a meaningful unit for development, provision, sale, and consumption. This combines with a business model that affords services a high degree of flexibility to both providers and consumers. This means that businesses can become more responsive to individual customer and market needs.

Major shifts in how business functionality is packaged and offered threaten to make existing systems obsolete. The impact of these shifts is compounded for services because they appear to offer freedom from such a legacy tie-in: if another service provider offers a new improved service, you simply change service providers.

How should we migrate existing systems—often providing core, or even critical, business functionality—to SOAs? Although there's probably no single answer, it's necessary to unpack the likely motivations for businesses wishing to adopt SOAs. For many businesses, services' true value isn't the possibility of dynamic service discovery and late binding. Instead, it's the ability to rationalize their existing systems into chunks of business functionality that they can reconfigure easily and quickly to exploit new business opportunities. In other words, the relative immaturity of the standards for service discovery and service-centric system engineering isn't necessarily an impediment to SOA adoption. The real hindrance is the lack of methods for the daunting task of unraveling existing systems' architectures.

Although the technical challenges of refactoring a substantial, mission-critical system are considerable, the associated business challenges are just as great. Companies must determine which business processes, supported by existing or legacy systems, should be liberated as services in a SOA. In some cases, when these processes are implemented as services, businesses will identify new revenue streams associated with providing their services to external consumers. Conversely, businesses will also identify which of their business processes don't align with their core competencies, making them prime candidates for outsourcing (essentially equating to "dogs" in a growth-share matrix). In a SOA world, replacing in-house services with third-party services should be seamless and painless.

We believe that progressive evolution is the best way to migrate existing systems to SOAs. This process might involve many intermediate stages, in which an organization integrates core existing systems into SOAs. Initially, this might involve adding functionality as a service, but progressively, obsolete functionality will be replaced by more independently implemented services. We characterize the many forms of intermediate systems as hybrids.

In a SOA world, replacing in-house services with third-party services should be seamless and painless.

What Is a Hybrid System?

In principle, hybrid systems combine services with nonservice elements. A strict definition of a service is not particularly easy to come by, nor is it particularly helpful because, as Alan Brown and his colleagues note, the aggregation of likely features is what generally defines a service best, including:[3]

- *Coarse grained*. Services usually deal with more varied information and support more functionalities than similar components.
- *Interface-based design*. A single service might implement multiple interfaces, and multiple services might implement the same single interface.
- *Discoverable*. Services are published to make them discoverable at design or runtime using their interface descriptions or other specifications.
- *Single instance*. A provider can replicate components to match the number of requests, but services are only single entities with which many clients can interact.
- *Loosely coupled*. Services use document-based message exchange (such as XML) to support interoperability and reduce coupling.
- *Asynchronous*. Services usually don't wait for replies, so although not compulsory, services typically use asynchronous communication.

Other characterizations of a service are possible. Clemens Szyperski stresses that "a service is not the software," continuing, "The entire tower of abstractions, right down to the physical machine, still doesn't deliver a service … A software service is the pairing of an operating agent and infrastructure with the software itself, implementing the service functionality and offering it through some interface."[4]

Szyperski's characterization of a service establishes three enabling components—an operating agent, the infrastructure, and the software—for delivering service functionality via some interface. These components are separate from the principles, attributes, and features described earlier. They're essential if functionality is to be implemented using the service model.

We don't base our characterization of service-oriented computing (SOC) on "necessary and sufficient conditions," but on a rather nebulous collection of principles, likelihoods, and tendencies. As such, it might reflect SOC's real-world relevance, rather than it being an artificial, technology-led construct. It does, however, complicate the identification of hybrid systems. We can consider deviations from the norms of SOC in terms of provision, process, and technology.

Provision

Service provision is an important aspect of SOC. We're particularly concerned with the provider–consumer relationship and the assumption that these two entities are separate. Assuming they are, the other elements that Szyperski considers integral to the service model—the software and infrastructure—are part of the provider's operation. Consequently, the consumer (or service-centric system developer) has no control over these aspects of the service's provision. Therefore, potential consumers must satisfy themselves that a candidate service offers the required function, or operations, and they must also reason about the service provision's nonfunctional properties (including service-level agreements and the provider's reputation) although they can't directly influence them.

Merging the provider and consumer roles is an important consideration for determining what constitutes a hybrid system. It removes from the consumer the need to interact with third-party, or off-the-shelf, services and might contribute to the migration of an existing system to a SOA.

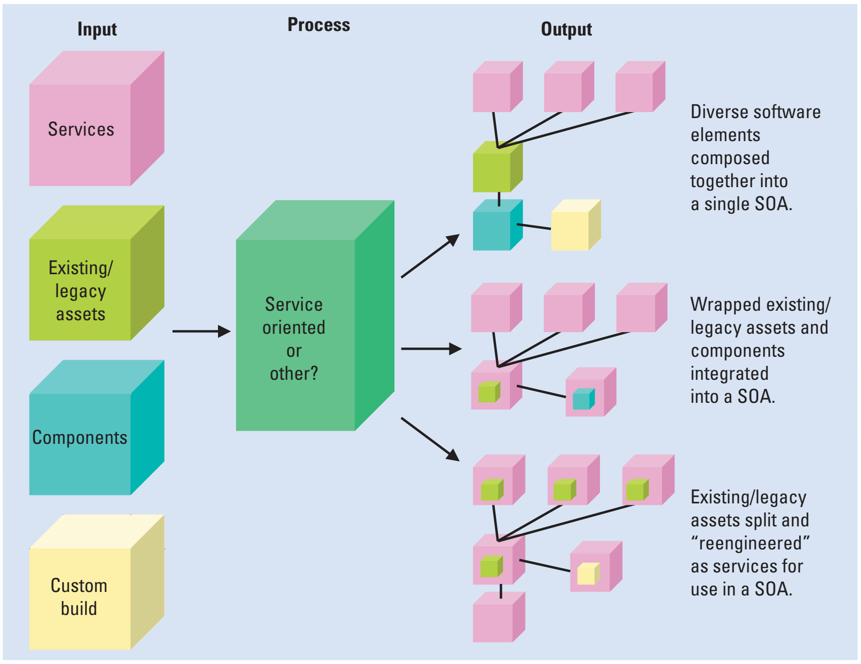
Process

Two distinct processes are associated with SOC: service engineering and service-centric system engineering.

Service engineering. No real expectations exist in SOC about how services should be developed, only an expectation that services exhibit certain behavior types. Where the service-engineering process involves making an existing or legacy system capable of operating in a service environment, the developer might not be able to address all of these expectations, with possible implications for how the resulting service will be used. If a provider developed a service this way and offered it to external consumers, service specification might become an issue. The most obvious examples of the likely deviations include communication and state issues. An existing or legacy application element might be designed to interact in a stateful way and interact synchronously with other application elements. Developers might be able to buffer this communication to achieve a more service-based interaction style, but it might not be possible, and the resulting service might pose problems for later composition.

Service-centric system engineering. Several aspects of service-centric system engineering might deviate from SOC norms. These include conflating the provider and consumer, the discovery and binding processes, and the related issue of integrating nonservice elements into the developed system. The service model assumes that services are published to a repository, where potential users can discover them and bind to them when needed.

Binding is linked to the technologies used. However, it's also related to the integration of nonservice system elements, such as legacy systems or software components. In some cases, the norms of service interaction (such as asynchronous communication and document-based messaging) place unacceptable overheads on the system being developed, and developers must use other technologies for the required functionality. A similar scenario is possible when services provide additional or exceptional functionality in an otherwise nonservice-centric system (for example, providing exchange-rate information in an online purchasing system). In both scenarios, the process used to develop the intended system might significantly differ from those used to develop purely service-based systems.



Input
Process
Output
Services
Existing/ legacy assets
Components
Custom build
Service oriented or other?
Diverse software elements composed together into a single SOA.
Wrapped existing/ legacy assets and components integrated into a SOA.
Existing/legacy assets split and "reengineered" as services for use in a SOA.

1 **Possibilities for hybrid system development.**

Table 1. Evolved system types.

	Provider/consumer relationship	
Existing system	**Same**	**Different**
As is	Ad hoc (type 1)	Hybrid (type 2)
Converted to services	Hybrid (type 3)	Service-oriented computing (type 4)

Technology

SOC isn't defined purely in terms of the technologies it uses. Service technologies can serve simply as an implementation mechanism for a system developed without the use of third-party services. Alternatively, service providers can implement their services, according to the norms we outlined, using novel technologies. Therefore, technology isn't particularly useful for defining hybrid systems. Although this is counterintuitive, it reflects the subtlety of what constitutes a service and a service-centric system.

Hybrid systems in practice

Although we can characterize hybrid systems as those deviating from service model norms in terms of provision, process, or technology, in practice we'll likely face a more limited set of scenarios (see Figure 1). Businesses will have existing systems, or products, that are central to their core function and will require some new functionality to better support their current business needs. Thus, developers will need to address two questions:

- Will the existing system be turned into services?
- Will an external provider supply the new functionality, which we assume to be a service?

Although obviously a simplification, the answers to these questions gives rise to four different types of system, as Table 1 shows.

Type 1. Essentially, we're concerned here with service technologies as an implementation mechanism only. Developers modify existing system elements, but only to enable integration, so these systems can be classified as ad hoc. Service technology adoption can represent a first step into an SOC world.

Type 2. This type of hybrid system imposes stricter adherence to SOC norms and expectations. You can't adjust the externally provided service to overcome difficulties, so you might have to significantly modify the existing system to make it compatible. This strategy might deliver some of the off-the-shelf benefits of services and SOAs, but it won't result in the difficult adoption of a genuine business-service culture. Core business functionalities will remain static and fixed, but the benefits of using externally provided services might generate enthusiasm for and commitment to further SOA adoption.

Type 3. Added services to existing systems (for example, wrapping systems to offer functionality as a set of service-based operations) for use with internally provided services suggests a much greater commitment to SOC than type 1 systems. However, control over provision and consumption still affords greater flexibility in the face of problematic difficulties (for example, the resulting services' statefulness). The key factor in a type 3 system is whether the process of creating the service is a lip-service provision of a service interface or a thorough realignment of an existing system provision with identified business services. In the latter case, the use of internally developed services to extend functionality is incidental to the commitment to adopt SOA.

Although development-related problems (such as coding and compilation) arguably might not be relevant in a SOA using third-party services, these problems came as developers sought solutions for the underlying architectural mismatches.

Type 4. This system type represents a wholehearted commitment to adopting SOC within an organization, especially if it represents the culmination of the business-service-analysis process described earlier.

Progressive evolution could be a gradual shift from a type 1 system, through types 2 and 3, to a type 4 system. Whether such a strategy would deliver the necessary business benefits would depend on the circumstances, but, for some businesses, it might represent a lower risk migration route to SOAs.

Architectural Mismatch Challenges

David Garlan and his colleagues performed the reference work on architectural mismatches when integrating independently developed systems.[5] Not all of this work is directly applicable to services and hybrid systems, but the lessons learned might be relevant.

The problems encountered included code bloat, poor performance, the need to modify existing components, the need to reimplement existing functions, unnecessarily complex code, and error-prone construction processes. Although development-related problems (such as coding and compilation) arguably might not be relevant in a SOA using third-party services, these problems came as developers sought solutions for the underlying architectural mismatches (that is, from conflicts between the architectural assumptions made by the various components).

To understand architectural mismatch, it's helpful to view a system as a set of components (the system's high-level computational and data-storage entities) and connectors (the interaction mechanisms among the components)—a view that's relevant to service-based systems.

Garlan and colleagues concluded that assumptions relate to four main items:

- *The components' nature.* For example, inadequate documentation of the requires interface can lead to false assumptions about the infrastructure.
- *The connectors' nature.* Examples include assumptions made about interface semantics and protocols.
- *The global architectural structure.* In SOC, examples might include assumptions made about orchestration or choreography.
- *The construction process.* For example, assumptions about the instantiation order.

When service provider and consumer are separate, tremendous scope exists to make the kinds of assumptions that we describe here, and that separation is inherent in much of the promise of services and SOA.

Process Approaches and System Reengineering

Researchers have described several methods and strategies for evolving systems that are partly applicable to the problem of migrating existing systems to SOAs. Here, we briefly summarize three approaches that address different issues.

Renaissance

In appreciation of both the functionality offered by existing systems and the investment they represent, the Renaissance method presents a set of maintenance strategies that put reengineering above replacement.[6] This is the implicit foundation of any approach that proposes progressive evolution. Identifying the dilemma between maintenance and replacement,[7] the method stresses that system reengineering can effectively mitigate the costs and risks associated with replacement—especially with a view to ongoing system development.

Renaissance lists six evolution strategies:

- *Continued maintenance*—accommodating change in a system, without radically changing its structure, after it's been delivered and deployed.
- *Revamp*—transforming a system by modifying or replacing its user interfaces. The system's internal workings remain intact, but it appears to have changed to the user.
- *Restructure*—transforming a system's internal structure without changing any external interfaces.
- *Rearchitecture*—transforming a system by migrating it to a different technological architecture.
- *Redesign for reuse*—transforming a system by redeveloping it, using some of the legacy system components.
- *Replace*—totally replacing a system.

While not directly applicable to evolving an existing system to SOA, these strategies suggest that understanding the range of available techniques will be a valuable resource. However, Renaissance lacks an explicit recognition of the business context. Thus, the evolution strategies are primarily selected on technical and organizational grounds (for example, system knowledge and documentation availability).

Compose

Because of the similarities between components and services, a process for evolving an existing system using COTS components is a good candidate for evolving systems to SOAs. Two of us (Kotonya and Hutchinson) have used the Component-Oriented Software Engineering (Compose) method to evolve a legacy freight-tracking system to support the demanding requirements of a company's larger customers.[8] This method includes the following aspects:

- It interleaves planning and negotiation, development and verification. Compose does this because many of the challenges of using COTS components stem from limitations of available documentation. Verification embeds activities that check system viability at every stage, while negotiation allows for corrective action.
- It incorporates a viewpoint-oriented requirements approach.[9] Viewpoints provide an excellent mechanism for modeling legacy system elements and other concerns as service consumers.
- It uses the notions of service providers and service consumers to model the system being developed. It uses required services to map system requirements to available components.

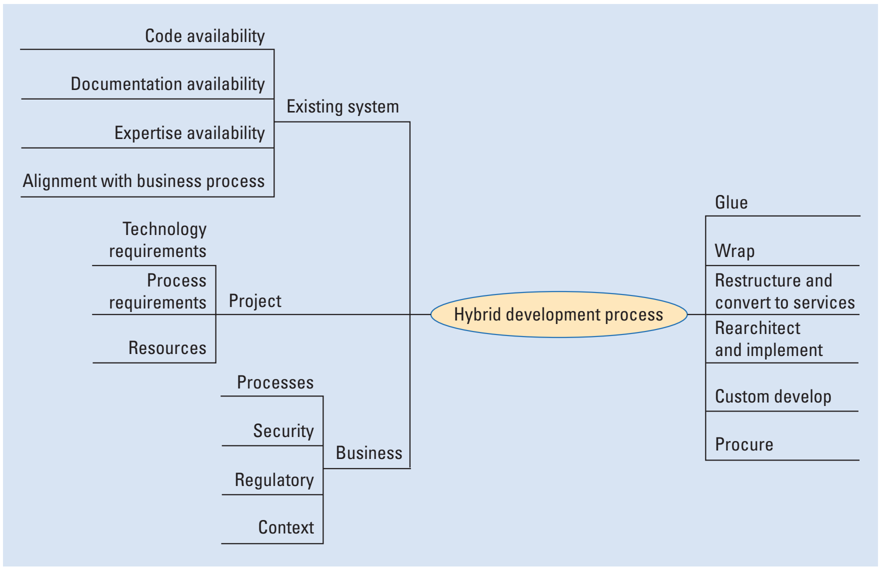
Because of these features, developers can use Compose to model an existing system as a series of refined subsystems that provide and consume services. The resulting model can then be used as, essentially, a roadmap for progressive evolution. However, it doesn't explicitly address the development activity's entire business context.

Service-oriented solutions approach

SOSA's principle contribution is to explicitly recognize that external reasons exist for wanting to adopt a SOA.[10] It identifies several issues.

Critical business issues. Organizations considering a SOA solution do so because they've identified critical business issues.

- The organization develops the system to implement a business strategy, not as an end in itself.
- The technical problem's details aren't important in themselves, only insofar as they affect the business.
- The organization might ultimately reject a viable technical solution for business reasons



2 Migration strategies and the contextual features that will guide their use.

(for example, it's expensive or too slow); similarly, business priorities might favor an inelegant technical solution.

Business process improvement. This provides the rationale for the development activities and involves modeling the existing process, determining the changes to be made to solve the relevant critical business issues, and an explicit attempt to estimate the return on investment associated with the proposals.

Enterprise service architecture. This is effectively a plan for the organization's business services bus.

Once developed, the ESA can act as a road map for an incremental, or progressive-evolution, process in which functionality provided by existing or legacy systems is moved to service-based provision. However, SOSA is primarily intended for companies that intend to implement their SOA using bespoke development. As such, it explicitly addresses neither the challenges of using third-party services, nor the process of providing service interfaces to existing systems.

Combining the approaches

These three approaches present some interesting perspectives on the migration-to-SOA challenge. None of them addresses all of the challenges; however, together they highlight many of the important issues.

For a business to fully engage in migration to SOA, it must be prepared to convert its existing systems into services, because these systems support the core business processes. SOSA presents some pointers for achieving this process. The enterprise service model, if adequately mapped onto existing or legacy system functionalities, goes some way toward identifying a business's key processes. However, SOSA doesn't offer a mechanism for providing such a mapping.

Renaissance provides some important pointers for determining the viability of reengineering an existing system into services. If developers used the SOSA ESA as further input to the Renaissance process, it might provide useful insights into the service creation process's feasibility.

You could use Compose to model an ESA and map service definitions onto components that can deliver those services. As such, you could use it to express an ESA that one or more existing systems deliver. However, it doesn't explicitly support identifying business services provided by such systems.

Migration strategies

Table 1 distinguishes between existing systems used as is and those converted to services in a wholesale fashion but it's too simplistic to represent concrete migration strategies that organizations could apply in practice. A major challenge facing the community is to identify practical strategies on the continuum between using existing systems as is and consuming third-party services, and understanding the factors that will lead to the appropriate selection in a given context. In practice, these strategies will amount to practical solutions and partial solutions reached in specific contexts. However, we'd expect common types of solutions to recur. Figure 2 illustrates some possible strategies.

Glue represents the least invasive attempt to make an existing system usable in a SOA context. It amounts to, for example, intercepting and redirecting a system call using some form of adapter. Organizations typically use this strategy when an existing system will consume a service to augment its functionality. An organization might use this approach to introduce new functionality, or replace failing or obsolete functionality, when buying a service from a third-party supplier. Alternatively, it might be appropriate when in-house development adopts a service model for delivery (for example, as a long-term strategy) and integration is required.

Wrap is a more general attempt to make an existing system usable as part of a SOA. An interface component—adapter or facade—must be developed that mediates all communication between the existing system and the outside world. This is an appropriate strategy for a system whose operations resemble those of a service. This approach might be used, for example, to offer services to external consumers or business partners.

The *restructure and convert to services* approach involves separating out the code supporting different business processes and making them available as separate services. Much of the original source code will be reused, but only those operations supporting current and future business needs will be made available in this way. This strategy represents an enormous commitment to supporting SOA. It might be more appropriate for companies who develop or support software systems than for the system users. However, highly specialized users who can't procure software off-the-shelf when adopting SOAs might also use it.

In the *rearchitect and implement* strategy, the existing system's form isn't amenable to restructuring or appropriate for significant reuse. However, it embodies important business logic and resource elements that must be retained. An organization develops new services to exploit these resources, which conform to some of the existing business processes. Clearly, the key difference between this and converting the existing systems to services is the availability and reuse of significant amounts of code. This strategy might therefore be appropriate when system documentation is available, but not significant amounts of usable code.

Custom develop is most appropriate when the match between the existing system and important business processes isn't close, but some functionality must be retained. We envisage new services to provide this functionality. These services might still use existing resources, such as data. An obvious implication is that the existing system no longer properly supports the business processes using it. This would clearly constitute a serious business failing, so it might be more sensible to use this strategy when integrating diverse systems following a merger or acquisition.

Procure discovers third-party services to deliver the required functionality. In the context of existing system reuse, this strategy is only appropriate if there are considerable problems associated with reusing some or all of the existing system and the required services are readily available from an external supplier.

Clearly, these strategies represent different approaches to reusing existing systems as well as different opportunities to do so. In other words, the prevailing context will determine which strategy an organization will adopt. In practice, this will likely mean that the unique combination of factors related to the particular migration activity (for example, whether code is available for the existing system, what resources are available, and whether regulatory constraints prohibit the use of third-party services) will make a given set of solution types, or strategies, available in the context. An organization's attitude toward the existing system, its commitment to SOA adoption, and the available strategies will determine its choice of a particular strategy. Furthermore, in practical circumstances, the prevailing factors will likely result in a combination of strategies—some gluing along with some procurement, for example, representing a possible low-risk venture into service consumption.

While considering migration strategies, we noted that the procurement of third-party services represented one end of the continuum. In part, this is because shifting from an existing system to an entirely new service-based system, while representing total commitment to services and SOAs, might also represent too high a risk for the business concerned. Careful selection of appropriate strategies will let businesses balance the benefits of using services in a SOA with the security of relying on tried and trusted systems. In other words, the required new system is a development of the existing system, so the new system's implementation is essentially a development of the existing system's implementation. In this way, an organization might migrate a substantial existing system to a SOA through a series of relatively discrete development cycles, or by progressive evolution. As we've suggested, this might start as a low-risk consumption of noncritical external services to provide new, or replace obsolete, functionality. This strategy's success, combined with an increasingly mature service marketplace, might make further service consumption an attractive option when further updating of the existing system is needed, requiring that some part is turned into services, or rearchitected in the absence of code. Ultimately, the business might have a system that's composed entirely of third-party services and yet never replaced more than a small part of the provided functionality at one time.

Migrating an existing system to a SOA is as much a business challenge as it is a technical one, which is why suitable processes must incorporate appropriate business modeling. One thing is clear: the need for such processes will only increase as SOC adoption increases and businesses realize that they must not be left behind.

The real challenge for the community is to identify and document patterns of SOA introduction and existing or legacy system migration, because some strategies can be applied in multiple contexts. Certainly, the subtleties of each development scenario will mean that no one-size-fits-all solution will exist, but systematic analysis of the prevailing system, project, and business factors must surely lead to migration strategies that can be applied reliably, robustly, and efficiently. IT

Acknowledgments

We performed this work as part of the European IST funded Service Centric System Engineering Project (http://secse.eng.it).

References

1. M.M. Lehman and L. Belady, *Program Evolution: Processes of Software Change*, Academic Press, 1985.
2. D. Dhungana et al., "Architectural Knowledge in Product Line Engineering," *Proc. 32nd Euromicro Conf. Software Eng. and Advanced Applications*, IEEE CS Press, 2006, pp. 186-197.
3. A. Brown, S. Johnston, and K. Kelly, *Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications,* tech. report, IBM, 2002; www.ibm.com/developerworks/rational/library/510.html.
4. C. Szyperski, "Components and Web Services," sdmagazine.com, 1 Aug. 2001; www.sdmagazine.com/documents/s=7208/sdm0108c.
5. D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch: Why Reuse is So Hard," *IEEE Software*, vol. 12, no. 6, 1995, pp. 17-26.
6. I. Warren and J. Ransom, "Renaissance: A Method to Support Software Systems Evolution," *Proc. 26th Ann. Int'l Computer Software and Applications Conf.* (Compsac), IEEE CS Press, 2002, pp. 415-420.
7. K. Bennet, "Legacy Systems: Coping with Success," *IEEE Software*, vol. 12, no. 1, 1995, pp. 19-23.
8. G. Kotonya and J. Hutchinson, "A COTS-Based Approach for Evolving Legacy Systems," *Proc. 6th IEEE Int'l Conf. COTS-based Systems* (ICCBSS 07), IEEE CS Press, 2007, pp. 205-214.
9. G. Kotonya and J. Hutchinson, "Viewpoints for Specifying Component-Based Systems," *Proc. Int'l Symp. Component-based System* (CBSE 07), LNCS 3054, Springer, 2004, pp. 114-121.
10. "Hybrid System Development," Service Centric System Engineering (SeCSE) Project (IST 511680) Document A3.D7, 2006; http://secse.eng.it.

John Hutchinson *is a research associate in the computing department at Lancaster University. Contact him at hutchinj@comp.lancs.ac.uk.*

Gerald Kotonya *is a senior lecturer in the computing department at Lancaster University Contact him at gerald@comp.lancs.ac.uk.*

James Walkerdine *is a research associate in the computing department at Lancaster University. Contact him at walkerdi@comp.lancs.ac.uk.*

Peter Sawyer *is a senior lecturer in the computing department at Lancaster University. Contact him at sawyer@comp.lancs.ac.uk.*

Glen Dobson *is a research associate in the computing department at Lancaster University. Contact him at dobsong@comp.lancs.ac.uk.*

Victor Onditi *is a research associate in the computing department at Lancaster University. Contact him at onditi@comp.lancs.ac.uk.*