Fifteen Years of Service-Oriented Architecture at Credit Suisse

Stephan Murer and Claus Hagen

Credit Suisse has been an adopter of SOA principles and patterns since the beginnings of this architectural style, even before the term appeared. The authors reflect on the financial institution's journey from using tightly integrated mainframe programs to open SOA services, emphasizing the importance of interface contracts and service governance in corporate IT. —*Olaf Zimmerman, Associate Editor*

THE SERVICE-ORIENTED ARCHITECTURE (SOA) style has become a mainstream pattern for the design of large distributed systems over the past decade. The main idea behind SOA is to design a system as a network of interacting services. Each service provides clearly specified functionality over a well-defined interface.[1]

Several hundred million lines of self-developed code and many off-the-shelf applications support the complex global banking business. Like many global banks, Credit Suisse runs a very large application landscape—in this case, almost 6,000 different applications. To make things even more complicated, this landscape is also very tightly integrated, a necessity for most banks to be able to provide highly efficient "straight through" automated processing and to allow the calculation of the aggregated risk exposure at any time.

Due to the nature of the business, banking is traditionally one of the first adopters of new information technology. With several thousand software developers on the payroll at a typical global institution, banks actively develop systems that must accommodate everything from 30-year-old legacy code to the latest mobile application. The sheer size of the landscape, the technical and architectural heterogeneity, and the need for dynamic development and tight integration create a very challenging environment for application integration.

Credit Suisse strategically responded to these challenges by placing integration architecture in the spotlight, emphasizing the decomposition of the overall IT system into clearly defined subsystems decoupled through SOA.[2] This article reports on Credit Suisse's journey over the past 15 years. Why 15? Because 15 years ago, two events fundamentally challenged the traditional enterprise architecture: one, there was a need to replace existing systems because they had reached the end of their useful life cycle, and two, it became clear that, with the Internet, banking services had to be offered via new technical channels that were largely incompatible with existing enterprise technology.[3]

The '90s Challenge: Opening the Mainframe

In the early 1990s, Credit Suisse's platform in Switzerland was still using the typical systems architecture of the 1960s and 1970s: a tightly integrated collection of mainframe programs sharing a common set of

0740-7459/14/$31.00 © 2014 IEEE

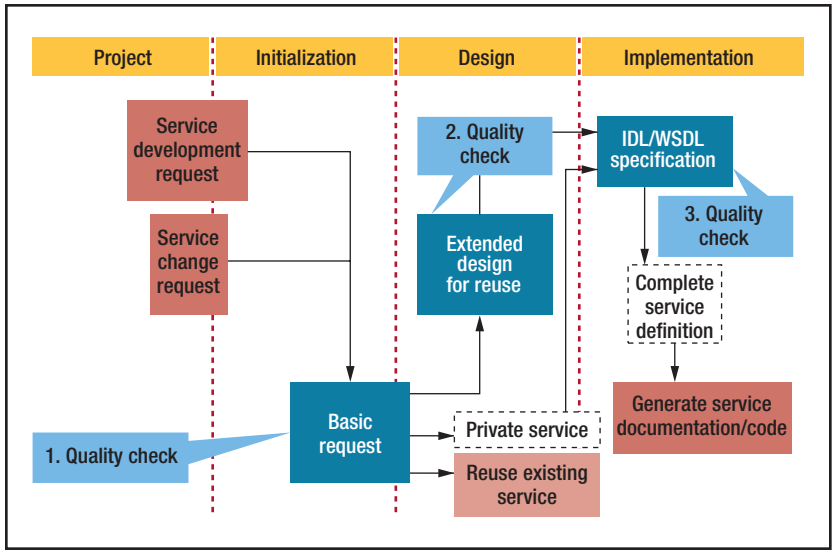

FIGURE 1. Quality assurance in the Credit Suisse Information Bus. Bottom-up design combined with top-down governance.

central databases. It had some pockets of 1980s technology—"fourth-generation" code, generator-based development, and some client–server applications running on PCs—but access to most applications happened through terminal screens. Moreover, certain parts of the landscape had simply reached the end of their life cycle.

Attempts to replace the whole core system with a new, modern one had failed: neither the planned purchase of standard software nor a custom development project based on Smalltalk (one of the first object-oriented languages to gain business acceptance) was successful. Replacing the platform piecewise was also difficult because it was so tightly integrated through shared code and data. Up to 80 percent of the effort was spent on application integration in projects trying to build distributed applications. With Internet banking emerging, we recognized that delivering a new Web-based transaction channel would require completely different technologies than the ones available in the bank.

At this point, Credit Suisse reconsidered its whole IT strategy and decided for a managed evolution of the platform instead of a big bang: piece-by-piece replacement to keep the risks manageable and to maintain the necessary strategic flexibility. In particular, the plan was to build modern Web front ends on top of existing rich back-end capabilities. For this to be successful, we recognized the need for a strong integration architecture bridging technical heterogeneity and supporting well-defined interfaces between applications—and with that, the concept of the Credit Suisse Information Bus was born. This bus, an "enterprise nervous system" linking all the major subsystems through services, was essential for our strategy because it allowed us to stepwise-evolve single subsystems without a major impact on all the others.

Building the Information Bus

After a careful evaluation of distributed computing infrastructures available on the market, we opted for the Common Object Request Broker Architecture,[4] mainly because CORBA had the most compelling story for defining interfaces independent of technology at the time, but no implementation for the mainframe existed. Together with a vendor, we started porting a Unix implementation to the mainframe, which included adapting it to the PL1 programming language.[5]

> *Lessons learned in this phase*: Technological challenges exist but can be solved. After 15 years of useful life, CORBA is currently being retired and replaced with Web services. It's interesting to see that many of the initial performance challenges have appeared again with the switch to the new technology.

Service Governance

A fundamental design principle was that clients could only access data on the mainframe through service interfaces on the Credit Suisse Information Bus. So, we opted for a bottom-up, demand-driven approach with top-down quality assurance (see Figure 1).

Projects build services as they need them. In an early phase, projects request new or extended interfaces when they need data and no sufficient interface exists. Integration architects review the "basic request" in a preliminary quality check step and come up with one of three possible conclusions:

- A service fulfilling the need already exists, so the project should use it. For instance, a

stock-trading system might plan to build an equity code service without knowing that this information is already offered by some other application.
- There's no reuse potential for the requested service, so the project can build a private service to be used exclusively within the application. This often happens with larger applications that are inherently service oriented.
- There's reuse potential for the requested service that requires an extended design for broader usability. For instance, the equity code service might be extended to deliver standard ISIN codes in addition to internal codes for each currency.

In a second quality check, a group of design specialists reviews this extended design and clears it for implementation. A final quality check makes sure that the service implementation will fulfill nonfunctional quality attributes, such as performance.

Lessons learned in this phase: It took us a while to find the right balance between quality control strictness and the required flexibility and agility for projects that build services. We solved this by focusing the review on those decisions that have the biggest impact on the overall system. For instance, we check that a service isn't redundant to others and is designed in a reusable way, rather than discussing every single data field of its signature.

Adoption of the Architecture

Figure 2 shows how the Credit Suisse Information Bus has been adopted. Although it's been available since 1998, project managers were reluctant to build services and use them. Only after several large performance- and stability-critical applications adopted the Credit Suisse Information Bus did trust in the development community increase. Functionality exposed on the bus rose from 2002 until 2007, when most services had been made available and growth slowed. On the client side, adoption took off seriously after 2003. With more than half of all functionality available on the bus, big clients (large program packages using several hundred services and generating millions of service calls a day) finally fully adopted the service architecture. After that, it took until about 2009 before essentially every client accessed the mainframe through the service layer.

Our strategic objective was to expose all data and functionality on the mainframe through services. Most of the ones that have been built so far allow distributed applications to access and manipulate data—be it master data (customer information), reference data (currency codes), or business data (account information). A smaller group provides access to business functions, such as the initiation of a payment, submission of a trade order, or calculation of tax amounts.

An important objective has been to foster reuse of services among multiple applications. The average reuse factor is four, meaning that four different applications use each service. Reuse is very uneven. While roughly 100 applications use some services, about half have only one consumer. Not surprisingly, the services with the highest reuse deliver our reference and customer data.

The high number of services (and their comparatively low reuse) is a consequence of our demand-driven approach. We're convinced that an ideal design could have been done with about half the services for the same functionality. But this is easy to say in hindsight. When we started the initiative, we weren't able to tell which services should be built with what priority.

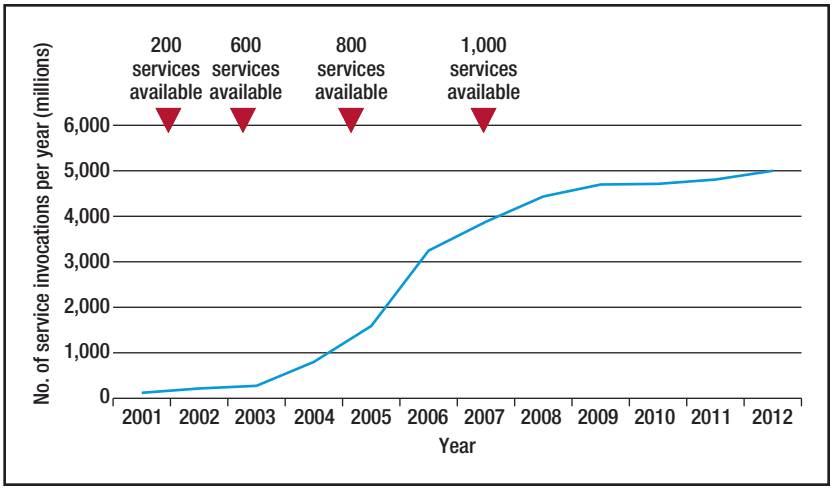

FIGURE 2. Service availability and usage growth on the Credit Suisse Information Bus. Widespread adoption after the availability of a critical mass of services.

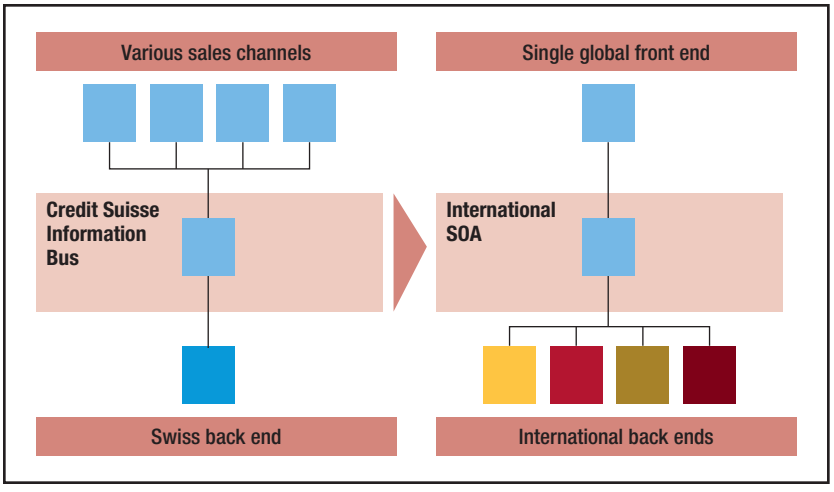

FIGURE 3. Target architecture for international SOA. A single front-end accesses multiple back-ends through services.

Lessons learned overall: It took four years from the strategic decision until the whole organization was fully committed to the plan. This is because it takes a while to fully deploy complex middleware technology in an environment that's sensitive to performance, stability, and security. The real challenge has been the stamina and governance needed to make this strategic decision pervasive—it took 10 years before everybody accessed the mainframe through the service layer. We debated whether this is unusually slow, but we're now convinced that this is normal for this kind of organization and application landscape. On the business side, the SOA approach has helped revolutionize user interfaces. Nobody accesses the mainframe through terminal screens anymore. Credit Suisse has built several Internet channels on top of the service layer, from a simple electronic banking application in the beginning to today's sophisticated mobile banking.

The International Service Architecture

The Credit Suisse Information Bus's success inspired the application of SOA in other strategic contexts. One such case was the international private banking business. A typical entry strategy into a new market is to set up a local subsidiary based on a locally purchased banking system that covers product and regulation specifics. Often, the front ends included with the local system are quite cumbersome and hard to learn, which results in the relationship managers not maintaining customer contact data as well as they should. Our strategic approach was to deploy a best-in-class internal front end from a mature market to the emerging markets. Architecturally, the requirement was to integrate a global front end with several local back ends (see Figure 3). This is similar to what we did with the Credit Suisse Information Bus, but instead of multiple consumer applications sharing a service, a single consumer accesses multiple implementations of the same service.

What seems fairly straightforward at first glance opened a whole new topic for integration architecture in Credit Suisse. We had to become more formal about interface semantics, particularly about the exact semantics for the data passed through the interface. Interestingly, this hadn't really been a problem before. In our earlier work, the implementation and its underlying databases implicitly defined the data semantics. Most clients understood the semantics because they remained the same as before we introduced the services. This was radically different in the international SOA, where each back end was developed completely independently. Although the high-level business objects, such as clients, accounts, and transactions, exist in every back end, their detailed representation is very different between back ends.

The international SOA had two requirements:

- We had to define services based on the least common denominator among all back ends, making an implementation possible on each one.
- Services needed precise semantic definitions because each service implementer had to bridge the semantic gap. Defining semantics was a new challenge that we addressed by defining the hierarchical business object model in Figure 4.

At the enterprise level, the model consists of a dozen abstract objects (customer, contract, product, and so on) and the relationships between them. Domain architects refine objects owned by their domain and add additional objects at the domain

level. For instance, the abstract "product" will be refined to represent a stock trade by adding attributes that describe the stock traded, execution price, and so forth. Further refinement happens at the individual application levels. Consistency rules make sure that objects on the lower levels match up with the upper level. On the logical and physical levels, database attributes or parameters in databases are mapped to corresponding conceptual business objects.[6]

On one hand, this ensures a consistent conceptual model for information stored and passed through interfaces. On the other, it allows a mapping of attributes and parameters with different names and syntax onto common semantic concepts, and vice versa. The key to make this successful is federated governance. There's no way to centralize all the know-how necessary to define a detailed conceptual information model of a large banking application landscape. The second important point is that the model isn't top-down: it's a mix between a top-down conceptual model and a bottom-up mapping of reality. As such, it's possible to cater for the heterogeneity of legacy implementations or off-the-shelf software. This is also the key difference in enterprise-wide data models.

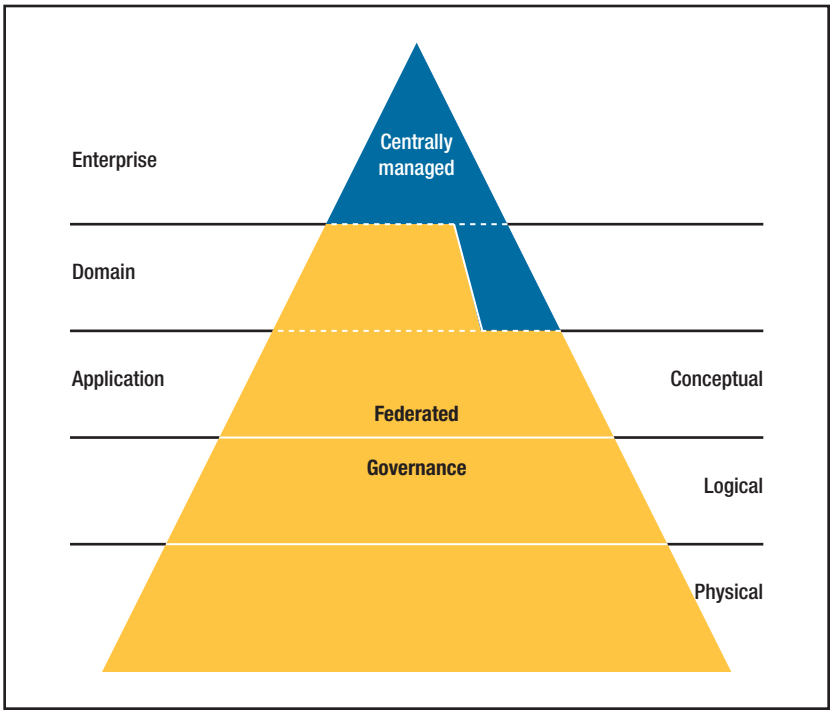

FIGURE 4. Hierarchical business object model. The enterprise level is centrally managed, but the more specific levels have federated management.

The Interface Management System: Supporting Interface Definition and Governance

A service database was an essential part of the SOA strategy from the beginning, but after a few years, it became clear that it wasn't enough. With the growing number of services and development activity, it became difficult to manually manage the governance process. It also became clear that CORBA wouldn't last forever and that capturing service definitions only in the form of CORBA Interface Definition Language (IDL) would create problems for later migrations. Furthermore, we found that we needed a system that would allow managing a service's full life cycle, from initial design to retirement.

We decided to establish a management tool that would let us capture services in an implementation-independent way while simultaneously maintaining various governance processes during the service life cycle. Our first attempt was to buy a SOA repository. Unfortunately, in 2005, the tools available on the market had limited capabilities, especially regarding SOA governance—we wanted the ability to efficiently drive design governance for the federated development of a large amount of services. To achieve this, we built our own service management application. The Interface Management System (IFMS) supports design, governance, implementation, and life-cycle management.

At its core, IFMS is a database that captures a service's complete contract, including the message signatures and all relevant metadata, in an implementation-independent way. Metadata includes semantic information (for example, the main business objects affected by the service and pre- and postconditions), nonfunctional information (such as the service's performance), and implementation-related information (for instance, the platforms and technologies on which the service is available.) The database also

tracks all known consumer applications of a service. This information is then leveraged to generate the technology-specific contracts (for example, using the CORBA IDL or Web Service Definition Language [WSDL]) used to implement services.

In certain cases, we also generate parts of the implementation itself, for example, service skeletons in PL1 or consumer-side abstraction layers in Java. The main benefit of this approach is that it lets us change a service's implementation technology just by using a different generator. We can, for instance, generate CORBA and Web service versions of the same service at the same time and so manage the transition from one technology to the other. This feature has greatly simplified our ongoing CORBA phase-out. The generator approach also greatly reduces the time required for changing existing services, in some cases, up to 75 percent.

> Deep architectural changes in large companies take longer than most people think.

IFMS manages three types of governance processes: the design governance for services, data governance for ensuring consistent data semantics across the service universe, and life-cycle governance ensuring that services can be properly retired at the end of their useful life cycle.

The design governance process in Figure 1 is implemented through a workflow management component that notifies reviewers and owners of tasks to be done and records all intermediate steps and results, such as obligations and their fulfillment. This automated process allows several hundred peer reviews per year with comparatively little overhead and quick turnaround times. A review is typically completed within a week from the time a service is submitted—assuming no design changes are required.

Data governance is closely linked to the business object model. Attributes in a service definition can be linked to business objects, either through a simple relationship indicating that an attribute is related to a particular object or as a direct derivation in which the attribute's type is derived from the business object through a model transformation. The governance process focuses on reviews that ensure semantic clarity and data consistency.

Proper versioning of services is essential to effectively decouple service providers from consumers. The main idea here is that several major versions of the same service are in production at the same time, which lets consumers continue using older versions of a service without the immediate need to migrate to a newer version. It's part of the service provider's contractual obligation to guarantee the availability of multiple versions. While beneficial to service consumers, offering too many versions in parallel puts a cost burden on providers because all versions must be maintained. This is why we defined an upper bound on the number of versions in production. We found three to be a useful upper bound because it limits provider cost while allowing consumers to skip two versions before they have to adopt the latest option. If, for instance, a service has one major version change per year, its consumers have three years before they have to adopt the latest one. Typically, consumer applications will have an upgrade themselves during this time, so the adoption of a new service version can take place as part of normal life-cycle management. This is clearly preferable over a forced migration, which would be much too expensive. In our experience, the three-major-version rule is a good compromise.

While we never intended to build our own SOA repository, we were forced to do so by the lack of mature solutions on the market at the time of development. Even today, although many solutions offer more capabilities than eight years ago, we still see limitations, especially in terms of platform independence (most tools are limited to Web services) and a scalable automation of the governance process. We hope that this will change in the future.

Lessons learned: The reduction of development effort through code generation made describing services in IFMS much more attractive for service developers. This solved an initial problem—developers don't like to document. In the times before IFMS, service descriptions always lagged behind actual implementations and were often inconsistent. Now, the service implementation starts

with documentation in form of the platform-independent specification and the technical interface, as well as parts of the actual implementation.

Looking back over 15 years of enterprise service architecture at Credit Suisse, we've learned a few lessons.

First, deep architectural changes in large companies take longer than most people think. The reason for this is because most projects are risk-averse and only want to adopt a proven approach. Proving a new approach plus the time lag between design decisions and implementation completion adds up to three to four years. After that, depending on the rollout strategy, it could take several years to fully implement a strategy. Patience and stamina are absolutely necessary for success in this field. If your CIO wants to see results within a quarter, SOA or other enterprise architecture approaches aren't worth pursuing.

Second, when thinking about SOA, technology on an enterprise scale is a nontrivial prerequisite, but it's the easier part. Orchestrating the entire organization around SOA, providing a proper semantic framework to create a common language across the organization, and implementing the necessary governance processes are the harder parts, in our opinion.

ABOUT THE AUTHORS

STEPHAN MURER is group chief technology officer of UBS, a global bank headquartered in Switzerland. Prior to his current role, he worked at Credit Suisse as managing director of the firm's information systems architecture. Murer received a PhD in computer science from ETH Zurich. He's a member of the Swiss Informatics Society. Contact him at stephan.murer@ubs.com.

CLAUS HAGEN is head of data and integration architecture at Credit Suisse. His research interests include enterprise architecture, SOA, and business process management. Hagen received a PhD in computer science from ETH Zurich. He's a member of ACM. Contact him at claus.hagen@credit-suisse.com.

References

1. D. Krafzig, K. Banke, and D. Slama, *Enterprise SOA: Service-Oriented Architecture Best Practices*, Prentice Hall, 2004.
2. S. Murer, B. Bonati, and F. Furrer, *Managed Evolution: A Strategy for Very Large Information Systems*, Springer, 2011.
3. S. Murer, "15 Years of Service Oriented Architecture at Credit Suisse," keynote presentation at SATURN 2013 Conf.; www.sei.cmu.edu/library/assets/presentations/murer-saturn2013.pdf.
4. P. Bernstein and E. Newcomer, *Principles of Transaction Processing for System Professionals*, Morgan Kaufmann, 1996.
5. W. Froidevaux, S. Murer, and M. Prater, "The Mainframe as a High-Available, Highly Scalable CORBA Platform," *Proc. IEEE 18th Symp. Reliable Distributed Systems*, 1999, pp. 310–315.
6. C. Batini, S. Ceri, and S. Navathe, *Conceptual Database Design*, Addison-Wesley, 1991.