# Web services-oriented architecture in production in the finance industry

Michael Brandner · Michael Craes · Frank Oellermann · Olaf Zimmermann

**Effective and affordable business process integration is key in the finance industry. With the help of IBM Global Services and IBM Software Group, Sparkassen Informatik, a large German joint-use centre supplying services to 236 individual savings banks, enhanced the integration capabilities of its core banking system (500+ complex functions) through aggressive use of Web services.**

The advanced requirements of this system, e.g. heterogeneous client environment, sub-second response times, 300% traffic growth, and interface complexity, challenged today's Web services implementations. To achieve true interoperability between MS Office XP/.NET and the Java world and to design an SOAP envelope compression scheme were two of the most important issues that had to be solved.

This paper discusses the rationale behind Sparkassen Informatik's decision for Web services, and gives an architectural overview of the integration approach. Furthermore, it features the lessons learned during the implementation of this enterprise–scale solution.

## 1 Introduction

This paper structured into six main sections. The introduction sketches the business problem and resulting requirements. In Sect. 2, we explain the Web services value proposition in this scenario. After that, we outline the solution architecture on a rather high conceptual level.

In Sect. 4, the project approach and results are discussed, while Sect. 5 features the lessons we learned and the best practices we identified during the project. The final section contains our conclusions and gives an outlook to future work.

### 1.1 Sparkassen Informatik: a full service provider for 236 financial institutes

*Sparkassen Informatik GmbH & Co. KG* [14] plays a major role in providing information technology (IT) services to many German savings banks ("Sparkassen"). It is the largest service and data centre in the *Sparkassen Finanzgruppe* in Germany, providing services to 236 individual savings banks. To satisfy the individual business and technical requirements of these banks, Sparkassen Informatik provides them with standard and optional service offerings as well as with unified interfaces to common business transactions.

As such, Sparkassen Informatik is a complete solution provider hosting mission-critical enterprise applications and data stores for the savings banks. At the heart of Sparkassen Informatik's solution stack is their real-time transactional *core banking solution*, based on a CICS® transaction monitor and a DB2® database management system located in a centralized z/OS backend. Furthermore, Sparkassen Informatik allows its customers and partners to flexibly integrate other applications, which are either developed individually or procured on the market place.

DOI 10.1007/s00287-004-0380-2
© Springer-Verlag 2004

M. Brandner
IBM Financial Sector,
Münster, Germany
e-mail: brandner@de.ibm.com

M. Craes · F. Oellermann
Sparkassen Informatik GmbH,
Münster, Germany
e-mail: michael.craes@sparkassen-informatik.de
frank.oellermann@sparkassen-informatik.de

O. Zimmermann
IBM Enterprise Integration Solutions,
Heidelberg, Germany
e-mail: ozimmer@de.ibm.com

The resulting business model, Sparkassen Informatik acting as a *shared service provider* for many different service requestors (the savings banks), inherently leads to a highly distributed, heterogeneous overall IT infrastructure and application landscape. Sparkassen Informatik therefore has to solve the following integration challenges:

- Fast, effective *business process integration* between Sparkassen Informatik and its customers, the savings banks, is the overall goal in this context.
- To achieve this integration, efficient *frontend to backend connectivity* is required – the savings banks operate the end-user frontend applications, Sparkassen Informatik provides the core banking backend.
- The centralized backend has to deal with a highly *heterogeneous frontend landscape*, as the savings banks decide for programming languages and runtime platforms independently of each other.
- Finally, it must be possible to seamlessly *integrate best-of-breed software solutions* available from Independent Software Vendors (ISVs).

### 1.2 Dynamic Interface: a service-oriented integration architecture

Sparkassen Informatik's strategic response to the integration challenges identified in Sect. 1.1 is a comprehensive integration and connector architecture called *Dynamic Interface* ("Dynamische Schnittstelle"). The Dynamic Interface provides standardized and flexible access to a collection of business functions, which are implemented in the backend core banking system. This offering is a key differentiator for Sparkassen Informatik, because it offers savings banks and ISVs a highly convenient way to connect frontend applications to the core banking backend.

The Dynamic Interface consists of two abstraction layers:

- A protocol interface and access layer called *technology platform*
- The *application layer* providing the core banking functions

The technology platform is the glue between client applications and the business functions; the concrete function invocation Application Programming Interface (API) and transport protocol mapping are defined on this layer. For each supported client environment and distribution mechanism, there is a separate technology platform (layer). For example, there are technology layers providing support for Java and a proprietary Remote Procedure Call (RPC) mechanism offering a C API.

The application layer consists of a large set of banking-specific functions, which we refer to as *processes*. Process granularity ranges from Create, Read, Update, Delete (CRUD) operations on core entities such as *Person*, *Account*, *Contract*, or *Product*, to search facilities and more complete use cases such as portfolio overviews, risk calculations and cross-selling functions.

This modular, two-layered interface design allows decoupling the business-oriented application layer from concrete implementation platform. In the case that an additional client programming model has to be supported, only the technology platform is affected.

In summary, the Dynamic Interface solution was established for the integration of client/server applications developed by Sparkassen Informatik. In addition, some savings banks needed to flexibly integrate their own applications purchased from ISVs. Due to the fact that these applications are selected by the savings banks based on the provided functionality rather than the supported platforms, the Dynamic Interface has to support a heterogeneous IT environment and to maintain multiple server-side interfaces.

### 1.3 Challenges and issues

As we have outlined in Sect. 1.2, until the arrival of Web services as an architecture alternative, separate technology platform layer instantiations had to be available to support multiple client programming languages and platforms.

However, it is desirable to minimize the number of required technology layers, as the development and maintenance of server-side support for several different distributed computing technologies is an expensive undertaking. Concepts such as interface description format, service naming, transport protocols, data (un)marshalling and tooling differ from platform to platform, and typically the learning curve to gain all required skills is rather steep.

Furthermore, Sparkassen Informatik is not – and does not want to be – a middleware platform vendor. However, the introduction of any home-grown integration solution makes it necessary to develop tools such as interface description browsers, stub generators and test clients in addition to the runtime integration solution. A solution built on standards makes it possible to buy rather than build such tools.

Finally, the continuous competition in the finance industry is a driving force for the savings banks to enhance the integration facilities to interact with their partners. Upcoming business models require that business processes can interact dynamically across enterprises. For example, many savings banks offer third-party insurance products. Just-in-time access to such insurance policies, which are processed by external insurance companies, must therefore be supported.

These issues forced Sparkassen Informatik to look for a new approach based on open standards. In a joint effort with IBM Software Group and IBM Global Services, Sparkassen Informatik decided to evaluate the potential benefits of Web services technology[1].

## 2 Vision and requirements

Since 1996, Sparkassen Informatik had attempted to consolidate the solution so that only one interface technology could support multiple client platforms. All previously existing technologies – such as a proprietary communication protocol, (D)COM, CORBA, Java, and a home-grown HTTP/XML solution – could either not fulfil this vision or did not meet all requirements of a truly *Services-Oriented Architecture* (SOA).

In contrast, Web services can be characterized as self-contained, modular applications that can be described, published, located and invoked over a common Web-based infrastructure defined by open standards [15]. An early investigation of the *Web Services Description Language* (WSDL) [17] showed that its modular structure, for example distinguishing between abstract port types and concrete protocol bindings, nicely mirrors the two-layered design of the Dynamic Interface (see Sect. 1.2).

Moreover, *SOAP* [13], the underlying messaging format, is designed to be platform- and implementation-neutral, and is built on already established internet standards (HTTP, XML). We detected that in combination with WSDL, which provides a formal interface and access specification, SOAP would be able to improve the existing solutions. In combination, WSDL and an SOAP service provider comprise the desired, unified and standards-based architecture supported by commercially off-the-shelf tooling.

[1] A high increase in the efficiency of IT development projects through the extensive use of Web services was expected. Furthermore, we anticipated that Web services software components – interacting with one another dynamically via standard Internet technologies – would make it possible to connect IT systems whose integration otherwise would require extensive development effort.

### 2.1 High-level requirements

As mentioned earlier, we evaluated the Web services technology with the intention of improving the Dynamic Interface access technologies. The main goals and requirements for the new Web services-based architecture therefore were:

- Minimize the number of required interfaces and middle-tier implementations to support the different existing client component and interface technologies
- Reduce the development effort for the savings banks by minimizing the interface complexity through encapsulation and better integration into existing development tools
- Improve the interface documentation of the existing proprietary HTTP/XML messaging interface, following the design-by-contract philosophy
- Reduce the volume of data transferred between requester and server

Moreover, the following Non-Functional Requirements (NFRs) had to be addressed:

- Heterogeneous service requestor (client) environment in terms of platforms and programming languages, including Java and Java 2 Enterprise Edition (J2EE), Microsoft .NET C#, Microsoft .NET Visual Basic and Visual Basic 6, Perl/PHP.
- Sub-second response times have to be achieved, even if network capacity is low, e.g. 64-kbit ISDN telephone line in rural areas (some application clients are directly used by customer facing staff).
- Scalability is a must-have, as a 300% traffic growth for the Dynamic Interface was observed in recent years (organic growth, mergers).
- And, of course, just as in any other enterprise-level scenario, security requirements such as authentication, authorization, integrity and confidentiality have to be met (sensitive data are transferred).
- Finally, the envisioned solution has to have excellent interoperability, performance and development efficiency characteristics.

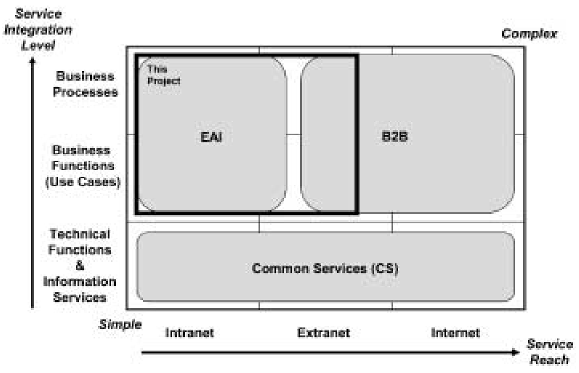

***Fig. 1** Usage scenarios for Web services (source: [19])*

In Sect. 3 we will describe how we map the vision and address these requirements in our solution architecture. However, before doing so, let us first assess how our scenario positions in an overall Web services complexity matrix.

### 2.2 Scenario complexity

Web services solutions differ – on the low end of the spectrum, there are simplistic services such as the ubiquitous `getStockQuote` example. The other extreme is fully automated service value chains, preferably communicating with each other ad hoc or *on demand*, as envisioned in vendor strategies such as IBM e-business on demand [8].

Web services solutions in general address *Enterprise Application Integration* (EAI) and *Business-to-Business* (B2B) scenarios possibly of *Intranet, Extranet* and *Internet* reach. Lower-level *Common Services* can also be made available as Web services – most first-of-a-kind prototypes fall into this category. Implementation complexity varies with integration level and services reach. Figure 1 shows where the Sparkassen Informatik solution resides in the resulting *complexity matrix*:

The business functions and the application layer of the Dynamic Interface are medium to highly complex in terms of their granularity (refer to Sect. 1.2 for a few examples). The savings banks access the Sparkassen Informatik servers via an Intranet, not the Internet. Thus, this solution is a rather advanced Web services usage scenario.

## 3 Solution outline

In the autumn of 2001, we started with a conceptual feasibility study delivering a vision statement, requirements and project goals as well as success criteria. Next, we decided to prove the usability and maturity of Web services implementations in a realistic, production-close environment. We therefore initiated a Proof of Concept (December 2001 to February 2002), which was important for risk minimization, as at project initiation time, Web services were still an emerging technology.

The final production solution was designed and implemented between August and December 2002. In this section, we will highlight its key architecture elements.

### 3.1 Key architectural decisions

As outlined in Sect. 2.1, the lack of standard interface documentation was one of the major business drivers for the project in order to leverage wizards provided by standard development tools. We addressed it by introducing WSDL descriptions for the banking functions. SOAP/HTTP became the message exchange format connecting the client applications with the functions provided by the Dynamic Interface.

Automatic WSDL provisioning from the existing, XML-based function repository is a key feature of the solution. Due to the widespread acceptance of the existing, HTML-based repository frontend, we decided to simply enhance the existing HTML presentation of each business function with the corresponding WSDL description. Therefore, there was no pressing need for introducing a service broker such as a Universal Discovery, Description and Integration (UDDI) repository.

Figure 2 illustrates the resulting three-tiered architecture of the resulting overall solution, providing a single, unified interface to different clients. It also outlines the key role of the metadata repository, which drives code generation for all tiers.

### 3.2 Interface design: generic vs. generated

The solution architecture could immediately satisfy requirement one (one interface) and three (improve documentation) from Sect. 2.1 through the use of SOAP and WSDL. To satisfy requirement two (reduce the development effort) and four (reduce network traffic), we had to carefully model the service invocation interface, and to decide between a model-driven and a generic, document-oriented design style.

A generic, function-independent API requires deep knowledge of each business function and performs most error checking at runtime. On previous projects we had gained the experience that a well-modelled, type-safe API following the com-

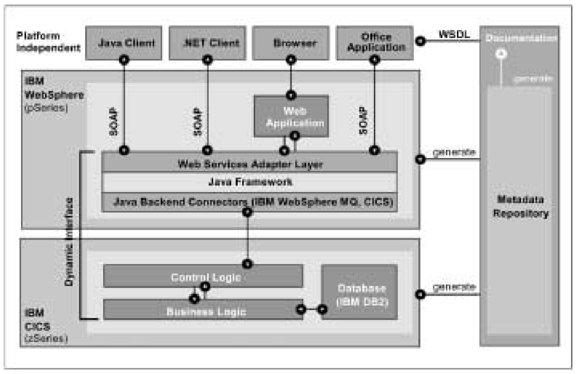

***Fig. 2 Architecture overview of the integration solution***

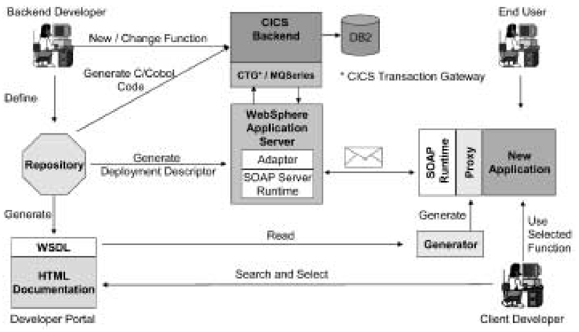

***Fig. 3 Integrated and automated application development process***

mand pattern from Gamma et al. [7], providing a *specific* client interface for each business function – instead of a *generic* one for all – hides complexity and reduces the development effort significantly.

To further satisfy requirement two (to reduce the development effort), we had to design the new Web services API in such a way that a high-level API could easily be generated by WSDL-aware tools; this is an instance of the *remote proxy* pattern [7]. The API also was supposed to hide all technical details of the service implementations and the technology platform from the client developer.

These considerations led us to an operation design with complex, function-specific XML schema definitions for the request and response messages (or input and output parameters, respectively). For each business function, a corresponding *Web service provider bean* was implemented as a J2EE component following a common adapter pattern. The interface signature itself was defined as follows[2]:

**ResultBean = execute(ContextBean, InputBean, WishlistBean)**

The model-driven API was designed in such a way that the `ContextBean` is identical for all functions, representing session parameters such as user and session identification. The other beans are business function specific. The `InputBean`s are responsible for input parameters, the `ResultBean`s for all output parameters and error messages. The `WishlistBean`s consist of indicator fields matching the output parameters, as the client can explicitly ask for a subset of all available result information (in order to reduce network traffic and processing time).

### 3.3 Automated application development environment

The high degree of code generation based on the metadata information stored in the repository and the out-of-the-box integration of Web services into standard tools available in the market results in faster development cycles, better software quality and reduced development costs.

The resulting integrated code generator- and repository-supported development process outlined in Fig. 3 is a key element of the solution architecture. A new business function is first described in the repository. Code generation support is then available on all three tiers. The backend business logic developer is supported by generated database access code; for the middle tier, deployment information and the code for the Web services access layer is created. WSDL service descriptions are generated as well, which can be imported into different client development environments to create service invocation proxies for various programming languages.

### 3.4 Solving the XML/SOAP message verbosity issue

As already mentioned in Sect. 2.1, we had to achieve good response times and avoid bottlenecks even in situations when only low network capacity is available. A related key issue, which we identified at an early stage, is the high amount of XML overhead typically produced by the SOAP runtimes. In our environment, payload to SOAP message size ratio was 1:4 in the best case, sometimes 1:16 and worse.

XML verbosity can be a challenging problem, and there are more related requirements than just

[2] According to the JAX-RPC specification [10], the Java representation of complex XML schema types are nested JavaBean and array structures.

algorithm efficiency. For example, a generic algorithm is desired, no software distribution effort should be introduced, and benefit and overhead (e.g. in terms of CPU consumption) have to be balanced.

Our final countermeasure to the message verbosity concern was to base the implementation on SOAP transport hooks allowing a flexible integration of different data reduction algorithms.

Clients have the flexibility to select the optimal data reduction algorithm for their particular usage scenarios; depending on the usage scenario, data reduction rate, lower network bandwidth and increased CPU cycles have to be balanced. Tests showed that the GNU Zip algorithm is a good solution for many scenarios, especially because inexpensive implementations exist on the most popular platforms and implementation languages. According to our experience, the transferred XML data streams can typically be reduced by 40–50%.

## 4 Project approach and results

As at the beginning of our analysis and design work in October 2001, Web services certainly were undiscovered ground in the enterprise application development world. We therefore applied a three-phased project approach:

- Feasibility study: conceptual groundwork, October to November 2001.
- Proof of Concept, December 2001 to February 2002.
- Release 1 of production system (August 2002 to December 2002); at the time of writing, a Release 2 project had just been completed.

In this paper, we describe the full-scope production solution and the project work related to it. The Proof of Concept project, which implemented a representative subset of the final solution, is featured in a previous report [4].

### 4.1 Overall results

The most important overall projects results are:

- The solution was delivered on time and budget. The system is in production now and well accepted. At the time of writing, there are six client applications already, with more in the pipeline.
- The three-phased project approach mentioned above turned out to be very helpful, because it allowed the team to learn and grow over the various stages; it also helped to mitigate the mutual project risk.
- The SOAP server performance met the requirements. We experienced no significant overhead compared to the proprietary XML/HTTP solution that existed before. Not surprisingly, SOAP engines using SAX parsing outperform those making use of DOM; document/literal communication performs better than rpc/encoded (these two SOAP communication styles and encoding schemes are discussed in detail elsewhere [4, 19]).
- Microsoft to Apache SOAP interoperability was achieved with reasonable testing effort (less than ten person days in Release 1). Some workarounds had to be applied; all required knowledge is public and available at developer forums such as IBM developerWorks [5]. Issues that required workarounds were WSDL imports, implicit vs. explicit typing in SOAP envelope, null values, binary data serialization, and SOAP Section 5 Encoding ambiguities. The work of the Web Services Interoperability (WSI) initiative, whose Basic Profile [18] became available after Release 1 of our solution had gone into production, provides significant further improvements, so that in Release 2, the interoperability testing effort was reduced to being almost negligible.
- The J2EE and Web services support in tools such as IBM WebSphere Studio Application Developer speeds up projects tremendously. For enterprise scale projects, the investment in production-strength tools should be made. Open source tools can be a low-cost alternative for smaller efforts.
- Not all Web services technologies always have to be used. For example, the service repository does not always have to be a UDDI registry; the existing HTML documentation does a perfect job in our case.

Our conclusion from these very positive results is that Web services *are* ready for production use, solving real-world problems in a mature and good enough way. The standards and product stacks certainly still have to be improved and completed, particularly in the higher layers as defined in the article by Ferguson et al. [6]. However, the XML, WSDL, and SOAP core existing today has proven the point.

A decision against a certain element of the technology, e.g. UDDI, or concerns in areas such as security and transactions cannot justify ruling out the entire technology – the modular structure of Web services allows a best-of-breed strategy. Complementary technologies can be used to complete the Web services stack on a per-scenario base. Furthermore, standards bodies and vendors are working on closing the remaining gaps.

When assessing the maturity of Web services, the implementation alternatives should also be considered – for example, is there out-of-the-box support for secure reliable transactions in your home-grown, proprietary distribution technology?

## 4.2 Technical issues

The most prevalent technical issues we encountered and solved during the various stages of the project were:

- Several SOAP to programming language mappings had problems with the *serialization and deserialization of null values*, which are allowed in XML schema (`nillable="true"` attribute) and SOAP. Consider the following scenario: an empty versus a null-valued phone number in the *CustomerMoves* function, an empty phone number indicating that there is no phone in the new home (yet), and a null-valued phone number indicating that the old phone number continues to exist after the move. In the Java world, the problem can be solved because the SOAP/XSD to Java mappings typically are configurable, and there are wrapper classes such as `java.lang.String`. In Microsoft .NET, to the best of our knowledge such features currently do not exist. We had to define a workaround here.
- *SOAP Chapter 5 Encoding*: Until recently, this data model, which for historical reasons is different from XML schema, was the default used by many RPC-oriented code generation tools, especially in the Java world. Unfortunately, the serialization algorithm defined by the SOAP specification is ambiguous and gives the writer many choices, e.g. how to represent arrays. The reader had to be able to understand them all. This caused some extra development effort in our project; in general, it is very hard, if not impossible, for tool vendors to guarantee interoperability. WSI has therefore decided to ban SOAP Section 5 Encoding from its interoperability profile. For these reasons, our Release 2 implementation uses *wrapped document/literal* styled messages [3] rather than rpc/encoded ones.
- *SOAP at its heart is just a messaging format*. The data type encoding is an optional part of the specification. However, from our point of view, a large amount of the value add of SOAP lies in the automatic (de)serialization support. Therefore, the wrapped document/literal mode has emerged as a de facto standard both in the Java and in the Microsoft world; it is already described in the 1.1 version of the JAX-RPC specification [10]. This communication and invocation style should be formally adopted in the WSDL and WSI specification work, and, in the Java world, aligned with the JAX-B work [9].
- *Case (in)sensitivity mismatch*: "The JavaBean specification defines an automatic property name decapitalization algorithm. If there is a getter method called `get_McProperty`, with `Mc` standing for mixed case, the JavaBean specification dictates that the name of the corresponding property is `mcProperty` and not `McProperty`. However, in the WSDL description of the service, `McProperty` might appear as the name for the corresponding field (depending on the tool being used for WSDL creation). This case sensitivity mismatch can introduce an interesting interoperability issue for SOAP RPC parameters. Following the JavaBean decapitalization rules, the property name assumed by Apache SOAP in our example is `mcProperty`. However, a WSDL-aware non-Java client might write an upper case accessor `McProperty` into a request envelope. In this case, a server side Apache SOAP runtime might not find the corresponding bean property during parameter deserialization. Watch out for this problem in any heterogeneous environment. For example, a .NET client obviously does not enforce the JavaBean specification. The best solution to the problem is not to use mixed case names at all; both server side JavaBeans and WSDL/XSD definitions should stay with lower case names and use only the basic Latin characters." [19].

## 4.3 More on interoperability and productivity improvement verification

To prove the interoperability of the solution, early Microsoft .NET and Java test clients were implemented, since these technologies were most widely used by the savings banks. In the Proof of Concept and the Release 1 project, interoperability between .NET and Apache SOAP could be achieved with a few "tweaks" such as explicit sending of data type information and slight WSDL modifications (resolution of schema imports), demonstrating that the suggested approach was valid. In Release 2, the emerging vendor support for the WSI Basic Profile delivered the promise of seamless interoperability.

At a very early project stage, we verified that requirement two (reduce the development effort) could be fully satisfied by testing with the IBM WebSphere Studio Application Developer and the Microsoft .NET development environment. Based on the WSDL files, both tools could generate a proxy, representing the client part of the Web services provider bean as expected. The short development time of new applications reusing the proxies generated by tools demonstrated that the model-driven API we introduced in Sect. 3.2 indeed was easy to use.

## 5 Web services best practices

Following the best practices for a technology is always key to project success; Web services are no exception to this rule. Several excellent articles on this topic have recently been published (for example, [2]).

We harvested the following best practices from this and other projects [19].

### Service modelling

The following service modelling guidelines can be defined:

- Follow the design-by-contract principle.
- Separate concerns, and separate interface from implementation.
- Provide interoperable versions of your WSDL specifications.
- Expose coarse-grained interfaces.
- Avoid complex operation signatures; stay with request-response messages.
- Keep service, method, parameter and type names small and simple.

### Service messaging

The following best practices for service messaging apply:

- Specify the style/use attributes either as document/literal or as rpc/encoded; do not mix the two alternatives in the same Web service.
- Carefully observe the messaging overhead so that countermeasures can be applied early enough.
- Be aware of the trade-off between security and performance requirements.
- Design your Web services to be as stateless as possible.
- Include, but do not rely on the HTTP `SOAPAction` header.
- Try to leverage already existing XML compression features.

### Service matchmaking

In the service matchmaking domain, the following hints should be followed:

- Carefully evaluate which type of UDDI registry (private versus public) is suited for your scenario.
- Consider lightweight alternatives to UDDI.
- Obey the best practices already established by UDDI.org

### SOA and project approach

General advice regarding the SOA is as follows:

- Carefully decide whether Web services are the right technology for your problem at hand.
- Apply standards pragmatically; follow the 80–20 rule.
- Use stateless session EJBs as provider type if J2EE is your implementation platform and EJBs exist in the overall architecture.
- Resist the temptation to be over-creative.
- Design for performance, and apply performance measurement best practices.
- Test early and often.

Unfortunately, a more detailed coverage of these best practices exceeds the scope of this article. Refer to pages 527 to 534 in the paper by Zimmerman et al. [19] for such a discussion of all best practices listed here, as well as several additional ones.

## 6 Conclusions and future work

In this paper, we described how Sparkassen Informatik implemented a services-oriented architecture consisting of standardized business functions (processes) to be reused in new applications in a flexible and channel-neutral manner.

The Dynamic Interface, an integration platform providing a set of APIs and interface layers and allowing access to business function and data, is the foundation of this architecture. In a three-phased staged project, Sparkassen Informatik, with the help of IBM, developed a successful Web services-enabled Dynamic Interface.

### 6.1 Engagement summary

The Web services enablement of the Dynamic Interface is a key component providing the glue between applications and business functions and defining the supported platforms and implementations. The integration approach determines the level of flexibility and complexity. We decided on Web services for the following reasons:

- Web services make simple, self-describing modular applications available on the web.
- Web services are independent of any component model, implementation language, transport protocol, operating system and platform.
- Web services separate interface and implementation (encapsulation).

- Web services are based on open standards and can be invoked over Internet infrastructure.
- Web services may optionally be published and searched in a repository.
- Web services are supported by standard development tools.

Concrete benefits the Web services solution brings to the table in our context are:

- *Write once, use everywhere*: It is no longer required to write custom, platform-specific code, true interoperability between platforms is achieved.
- *Design by contract*: standard interface description of business components, integration with standard development tools, no software distribution
- *Improved client interface*: A model-driven, business function-specific Application Programming Interface (API) replaces low-level XML and HTTP coding.

Key architectural decisions in this project were:

- *Service modelling and granularity*: General advice is to model as coarse-grained as possible, the service boundary should reflect a business process (or activity). In our case, lower-level CRUD and search functions as well as higher-level services are exposed. We decided for a process model-driven, generator-supported service invocation interface.
- *SOAP runtime and API*: Our client API is JAX-RPC. As SOAP runtimes, we worked with Apache SOAP 2.2, Apache Axis, and an optimized IBM implementation of JAX-RPC/JSR 109 called WebSphere 5.0.2 SOAP.
- *SOAP communication style and encoding*: We support both rpc/encoded and document/literal, but over time we will move away from rpc/encoded due to its conceptual flaws such as usage of an outdated, obsolete data model (which is different from XML schema) and inherent ambiguities (which cause interoperability problems).
- Regarding service matchmaking, an XML/HTML service repository (and frontend) is already in place at Sparkassen Informatik. Therefore, we do not use UDDI, even if a business need for a central service broker/directory exists.
- The success of a project is driven mainly by general architectural decisions such as choice of hardware and operating system platform (we chose IBM pSeries and AIX for the middle tier) and the quality of the XML parser.

### 6.2 Outlook

Sparkassen Informatik is committed to continue to support and enhance the Web services solution described in this article. Here is an outlook to future activities:

- Functional enhancements further improving client developer productivity.
- Continued WSI and other standards compliance (WSI, W3CWSA).
- Declarative and descriptive process flow execution (higher-level services) available through the Business Process Execution Language for Web Services (BPEL4WS) and related modelling tools and runtimes [1, 11].
- Leveraging the emerging Web services security (WS-security) standards and their implementations as defined by the OASIS consortium [12][3].

### 6.3 In retrospect: web services – holy grail or deja vu?

Are Web services the holy grail of distributed computing [19]?

- Web services are a relatively simple, programming language-neutral and interoperable communication technology.
- Web services are available today – at low initial costs and minimal risk.
- Many business scenarios such as EAI, B2B and common services are supported.
- Comprehensive and mature tool support is available; huge productivity gains can be realized.

Or did you experience déjà vu when you first heard about Web services, or reading this article?

- Web services merely provide an API instead of a browser GUI for access to Web applications.
- The remote proxy pattern, interface descriptions and a service directory (registry) are key concepts.
- Well-known technologies such as HTTP and XML are leveraged.
- There is a semantics (vocabulary) issue. The specification lifecycle is quite typical; gaps in the technology exist and are currently being closed.

[3] Currently, security requirements such as integrity, confidentiality, authentication and authorization are addressed on the network, on the transport and on the application layer.

In a nutshell, our experience suggests that both provocative statements are true...

## About the authors

Michael Brander is a Consulting IT Architect in IBM's Financial Industry Sector. In his role as Client IT Architect, he supports his customers in pre- and post-sales situations, and is responsible for strategic enterprise architecture consulting activities.

Michael Craes is Project Manager and Technical Consultant at Sparkassen Informatik. He leads the Dynamic Interface development projects, and is the product manager both for the core interface as well as its Web services extension.

Frank Oellermann is a Technical Consultant at Sparkassen Informatik, responsible for the Dynamic Interface solution in general and its generators in particular. Furthermore, he runs the technical education programme the Dynamic Interface team provides to its customers.

Olaf Zimmermann is a Consulting IT Architect in IBM's world-wide Enterprise Integration Solutions team. Over the last few years, he has conducted numerous Web services-related engagements. Olaf is an author of the text book *Perspectives on Web Services* [19], and contributed to IBM ITSO Redbooks such as *Web Services Wizardry with WebSphere Studio Application Developer* [16].

## Acknowledgements

The project featured in this article project certainly was a team effort. In particular, Reinhold Nolte, Guido Ranft, and Torsten Verstappen (all of Sparkassen Informatik), as well as Martin Fleming, Sven Milinski, Frank Strecker, and Thorsten Stumpf (IBM), all contributed tremendously to the project success and deserve massive thanks and appreciation.

## References

1. Business Process Execution Language for Web Services Version 1.1, available from http://www.ibm.com/developerworks/webservices/library/ws-bpel
2. Brown, K., Reinitz, R.: Web services architectures and best practices, IBM developerWorks 2003, http://www.ibm.com/developerworks/websphere/techjournal/0310_brown/brown.html
3. Butek, R.: Which style of WSDL should I use?, IBM developerWorks 2003, http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl
4. Craes, M., Oellermann, F.: Getestet und für geeignet befunden, XML & Web Services Magazin 1 (2002) (overview of Sparkassen Informatik Web services proof of concept, in German)
5. IBM developerWorks portal. Articles, tutorials, sample code, links to trial versions of software and open source assets. http://www.ibm.com/developerworks/webservices
6. Ferguson, D.F., Storey, T., Lovering, B., Shewchuk, J.: Secure, reliable, transacted web services, IBM and Microsoft 2003, http://www.ibm.com/developerworks/webservices/library/ws-securtrans
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns – Elements of reusable object-oriented software. Addison-Wesley 1995, ISBN 0201633612
8. IBM e-business on demand overview, available from http://www.ibm.com/e-business/index.html
9. Java XML Binding, available via http://java.sun.com
10. Java XML API for Remote Procedure Calls (JAX-RPC), available via http://java.sun.com
11. Leymann, F., Roller, D., Schmidt, M.T.: Web services and business process management, IBM Systems Journal 41(2) (2002)
12. OASIS consortium, http://www.oasis-open.org
13. Simple Object Access Protocol (SOAP) 1.1, W3C Note 08 May 2000, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/
14. Sparkassen Informatik on the Internet, http://www.sparkassen-informatik.de
15. Web Services Architecture, W3C, available from http://www.w3.org/2002/ws
16. Wahli, U., Tomlinson, M., Zimmermann, O., Deruyck, W., Hendriks, D.: Web services wizardry with WebSphere studio application developer, IBM Redbook 2002, ISBN 0738423351
17. Web services description language (WSDL) 1.1, W3C Note 15 March 2001, http://www.w3.org/TR/wsdl
18. Web Services Interoperability Initiative, http://www.ws-i.org
19. Zimmermann, O., Tomlinson, M., Peuser, S.: Perspectives on web services – Applying SOAP, WSDL and UDDI to real-world projects. Berlin Heidelberg New York Tokyo: Springer, 2003, ISBN 3540009140, http://www.perspectivesonwebservices.de