Second Generation Web Services-Oriented Architecture in Production in the Finance Industry

Olaf Zimmermann
Sven Milinski
IBM Software Group, IBM Global Services
Gottlieb-Daimler-Strasse 12
68165 Mannheim, Germany
+49 171 970 6341
ozimmer@de.ibm.com
milinski@de.ibm.com

Michael Craes
Frank Oellermann
Sparkassen Informatik GmbH & Co. KG
Nevinghoff 25
48147 Münster, Germany
+49 251 288 3456
michael.craes@sparkassen-informatik.de
frank.oellermann@sparkassen-informatik.de

ABSTRACT

Effective and affordable business process integration is a key concern in the finance industry. A large German joint-use centre, supplying services to 237 individual savings banks, enhanced the integration capabilities of its core banking system, consisting of more than 500 complex functions, through aggressive use of Web services.

Advanced requirements such as heterogeneous client environment, sub-second response times, 300% traffic growth, and interface complexity did challenge today's Web services implementations. To achieve true interoperability between Microsoft (MS) Office™/.NET™ and Java™, and to implement more than 500 Web service providers in a short time frame were two of the most important issues that had to be solved. The current, second release of this solution complies with the Web Services Interoperability (WS-I) Basic Profile 1.0. Leveraging the Basic Profile reduced the development and testing efforts significantly.

This report discusses the rationale behind the decision for Web services, and gives an architectural overview of the integration approach. Furthermore, it features the lessons learned and best practices identified during the design, implementation and rollout of the solution.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures – *domain-specific architectures.*

General Terms

Design, Standardization, Performance, Security.

Keywords

Application Server, Business Process, CICS, Compression, Core Banking, Enterprise Application Integration, HTTP, Service-Oriented Architecture, SOAP, Software Architecture, UDDI, Web Services, WebSphere, Web Application, WSDL, XML, XML schema.

1. INTRODUCTION

Sparkassen Informatik GmbH & Co. KG [18] provides Information Technology (IT) services to German savings banks. Supporting 237 individual savings banks, Sparkassen Informatik is one of the largest service and data centers in Germany. To satisfy the individual business and technical requirements of the savings 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 the solution stack is a real-time transactional *core banking solution*, which is 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.

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 is exposed to the following integration challenges:

- Fast, effective and inexpensive *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.
- It must be possible to seamlessly *integrate best-of-breed software solutions* available from Independent Software Vendors (ISVs).

- Any selected integration platform must promote quality factors such as ease of client access and resiliency, and business-level benefits such as flexibility and agility.

1.1 Dynamic Interface: A Service-Oriented Integration Architecture

Sparkassen Informatik's strategic response to its integration challenges is a comprehensive integration and connector architecture called *Dynamic Interface*. The Dynamic Interface provides standardized and flexible access to a collection of business functions, which are implemented in a core banking backend. 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 called *technology platform* and *application layer*. 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 is 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, which comes as 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 platforms. In case an additional client programming model has to be supported, only the technology platform is affected.

1.2 Challenges and Issues

As we have outlined in Section 1.1, until the arrival of Web services as an architecture alternative, a separate technology platform layer instantiation had to be available for each client programming language and platform to be supported.

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 (DC) technologies is an expensive undertaking. DC concepts such as interface descriptions, service naming and lookup, 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. Selecting a solution built on open 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. A ubiquitous integration technology is required to provide such access upon demand.

All 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 the Web services technology.

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 fulfill this vision or did not meet all requirements of a truly *Service-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 which is defined by open standards. An early investigation of the *Web Services Description Language* (WSDL) 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 introduced in Section 1.1.

Moreover, *SOAP* [17], the underlying messaging format (not an acronym according to version 1.2 of the specification), is designed to be platform- and implementation-neutral, and built on already established Internet standards such as HTTP and XML. We detected that in combination with WSDL, which provides a formal and language-independent interface and access specification, SOAP would be able to improve the existing solution.

In combination, WSDL definitions and a SOAP service provider comprise the desired, unified and standards-based architecture supported by commercially-off-the-shelf tooling. Introducing either SOAP or WSDL alone would not have yielded sufficient return on investment for the required design and development effort.

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

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

4. Reduce the volume of data transferred between requester and server.
5. Support very large numbers of deployed services (about 100 new functions to be deployed every year).

Rather challenging Non-Functional Requirements (NFRs) had to be addressed: first and foremost, the service requestor (client) environment is very heterogeneous 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, as well as Perl and PHP.

Some application clients are directly used by customer-facing staff. Hence, sub-second response times have to be achieved, even if network capacity is low – for example, 64 kbit ISDN telephone lines are used in certain rural areas.

Moreover, scalability is a must-have, as a 300% traffic growth for the Dynamic Interface could be observed in recent years (organic growth, mergers). And, just as in any other enterprise-level scenario, security requirements such as authentication, authorization, integrity and confidentiality have to be met (sensitive data is transferred). Finally, the envisioned solution has to have excellent interoperability, performance and development efficiency (usability) characteristics.

3. PROJECT APPROACH AND SOLUTION OUTLINE

Starting from the vision and the requirements outlined in the previous section, we employed a staged approach to craft a Web services-enabled solution architecture for the Dynamic Interface.

In fall 2001, we started with a conceptual feasibility study, or *project definition workshop*, 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* (PoC) project, which ran from December 2001 to February 2002. The PoC was important for risk minimization, as at project initiation time Web services still were an emerging technology. The final *production solution* was designed and implemented between August and December 2002 [2]. The current second release went live in July 2004.

3.1 Key Architectural Decisions

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 Sparkassen Informatik 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 an 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) registry.

Figure 1 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.

SOAP provides connectivity between the client tier and the WebSphere® Application Server (WAS) middle tier, which has pure gateway character. The interface between the middle tier and the backend is provided by the IBM CICS Transaction Gateway (CTG) and IBM WebSphere MQ®.

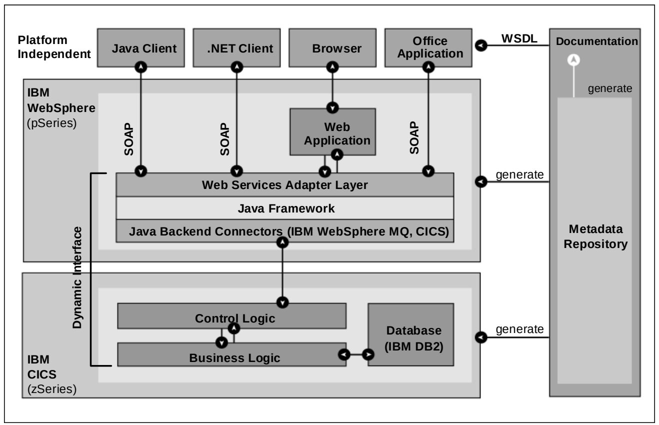

Figure 1. Architecture Overview of the Integration Solution

3.2 Interface Design: Generic vs. Generated

The solution architecture could immediately satisfy Requirements 1 (have a unified interface) and 3 (improve documentation) from Section 2 through the use of SOAP and WSDL. To satisfy Requirements 2 (reduce the development effort) and 4 (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-modeled, type-safe API following the command pattern [10], 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 1 (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 [10]. The API also was supposed to hide all technical details of the service implementations and the technology platform from the client developer.

These considerations lead 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 an adapter pattern [10]. The interface signature itself was defined as follows:

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 like user and session identification. The other beans are business function specific. The InputBeans are responsible for input parameters, the ResultBeans for all output parameters and error messages. The WishlistBeans 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 Largely Automated Development Process

In response to Requirement 5 from Section 2 (large number of services), we envisioned that a high degree of code generation based on the metadata information stored in the repository, along with the out-of-the-box integration of Web services into standard products available on the market, could result in faster development cycles, better software quality, and development cost reductions.

The resulting integrated code generator- and repository-supported development process, which is outlined in Figure 2, is a key element of our solution architecture:

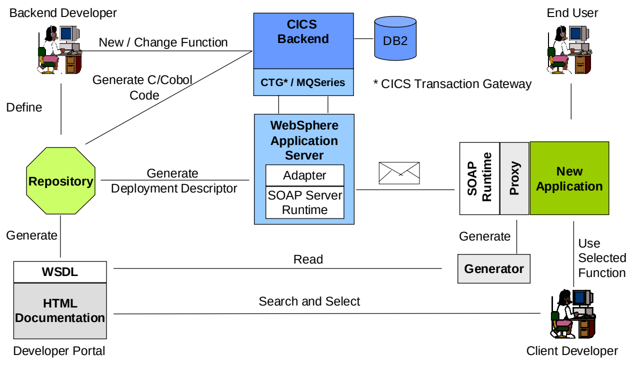

Figure 2. Integrated Application Development Process

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 Service Deployment: One vs. Many

Another challenge was how to rapidly deploy the large amount of existing processes as Web services in the middle tier of the solution. Implementing the respective Web service providers on a one-by-one base would have been rather development and maintenance resource intense, as new processes are added continuously.

We therefore investigated two architecture alternatives:

- Developing a single, generic Web service provider implementation, as well as a set of custom deserializers to mediate the incoming requests (structured according to the generated client-side interface WSDL contract described in Section 3.2), to this generic Web service. This variation of the façade and command patterns [10] was our original approach, minimizing the amount of coding required on the middle tier.
- Generating specific Web service implementation classes for all processes, as well as corresponding Web services deployment descriptor entries. In the second release, we decided to use this approach, because at design time the JAX-RPC [14] support for custom serialization was not sufficient. Such support would have been required for implementing the first alternative.

Figure 3 illustrates the server-side components implementing the second alternative. *WAS 5.0.2 Web service engine* represents the IBM WebSphere SOAP engine supporting JAX-RPC; *xxxBindingImpl* are the more than 500 Web service implementation classes.

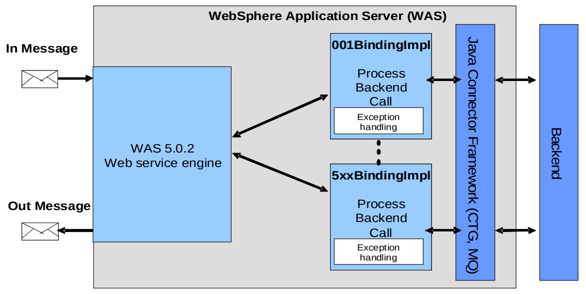

Figure 3. Server-side Component Model

3.5 Message Verbosity Countermeasures

As already mentioned as a requirement in Section 2, we had to achieve good response times and avoid bottlenecks even in situations when only low network capacity was available. A related key issue, which we identified at an early stage, is the high amount of XML overhead typically produced by SOAP runtimes. There is a clear performance penalty for using SOAP given our measured numbers for payload vs. SOAP message size (see below). However, SOAP was an architectural imperative given that the requirement was to deliver application interoperability with the heterogeneous client environment (.NET, Java, and other platforms to be supported, see Section 2).

In our environment, payload to SOAP message size ratio was 1:4 in the best case and sometimes significantly worse. For example, we measured a 1:16 ratio for a *SearchPerson* request during the early project stages, when using the rpc/encoded communication style and the Apache SOAP 2.2 implementation (which employs a rather verbose serialization scheme, always adding explicit type information for each XML element in the envelope).

XML verbosity can be a challenging problem, as there are more related requirements than just a need for algorithm efficiency. For example, a generic algorithm is desired, no software distribution effort should be introduced, and benefit and overhead have to be balanced (e.g., data reduction rate and reduced network bandwidth versus increased CPU consumption).

Our solution for the message verbosity problem in the first production release was to base the implementation on SOAP *transport hooks* allowing a flexible integration of different data reduction algorithms. Clients had the flexibility to select the optimal data reduction algorithm for their particular usage scenarios.

However, tests showed that the GNU zip algorithm is a good solution for many scenarios, especially because inexpensive implementations exist for the most popular platforms and implementation languages. Based on our experience, the transferred XML data streams can typically be reduced by 40-50%. In the second release, we therefore decided to concentrate on one algorithm, now making use of the built-in response compression capabilities provided by the Web server and SOAP engines in use.

4. PROJECT RESULTS

The Web services-enabled Dynamic Interface is in production now and well accepted, delivering the expected business benefit of providing a universal, low-cost, frontend-to-backend integration technology that can easily be enhanced (about 100 new services/processes are released every year). At the time of writing, six client applications use the new interface already, with many more being in the pipeline.

Client developers appreciate the new high-level API, and experience the desired productivity gain. Over time, the Web services interface will replace all existing proprietary ones; for the time being, a transition and coexistence phase is ongoing.

4.1 Project Approach

Regarding the project approach, the three-phased approach outlined in Section 3 turned out to be very helpful, because it allowed the team to learn and grow over the stages and to mitigate the mutual project risks.

On all three project stages, we delivered in time and on budget.

4.2 Technical Aspects

Microsoft to Java interoperability for SOAP was achieved with reasonable testing effort (less than ten person days in the first release). On the WSDL level, we had to come up with several workarounds; all required knowledge is public and available at developer forums such as IBM developerWorks [11]. Issues requiring workarounds were WSDL import statements and XML namespaces, implicit vs. explicit typing in the SOAP envelope, null values, binary data serialization, and SOAP Section 5 Encoding ambiguities (see below). The work of the Web Services Interoperability (WS-I) initiative [21], whose Basic Profile became available after our first release had gone into production, provides significant further improvements, so that in the second release, the SOAP interoperability testing effort reduced to being almost negligible. To define WSDL consumable both by Microsoft and by Java tools remained a rather tedious task even on the second release.

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 tend to outperform those making use of DOM, and *document/literal* styled communication performs better than the *rpc/encoded* mode (these two SOAP communication styles and encoding schemes are discussed in detail in [6]).

Web services tool support speeds up projects significantly. For enterprise scale projects, the investment into production-strength tools generally available as products should be made. Open source tools can be a low-cost alternative for smaller efforts.

Not all Web services technologies have to be used in each and every project. For example, the service repository does not always have to be a UDDI registry. The existing XML-based process repository does a perfect job in our case; WSDL service descriptions can be generated from the information contained in the repository. To enable build-time service discovery, we simply had to enhance the existing HTML frontend to the XML process repository.

4.3 Issues and Countermeasures

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 automatic (de)serialization support. Due to the issues pertaining to the rpc/encoded communication style, 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 [14]. This style should be formally adopted in the WSDL and WS-I specification efforts, and, in the Java world, be aligned with JAX-B [13].

The SOAP Chapter 5 Encoding has conceptual flaws. 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, for example 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. WS-I has therefore decided to ban the SOAP Section 5 Encoding from its interoperability profile. For these reasons, in our second release we use wrapped document/literal styled messages rather than rpc/encoded ones.

Null values frequently cause tools and runtimes to fail. 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 wrapper classes such as `java.lang.String` and `java.lang.Integer` exist. In Microsoft .NET, to the best of our knowledge such features currently do not exist for simple types. We had to define a workaround here.

4.4 Lessons Learned

Our conclusion from these positive results is that Web services *are* ready for production use, solving real-world problems with a mature and stable base technology stack. The standards and product stacks certainly still have to be improved and completed, particularly in the higher layers as defined in [9]. However, the

XML, WSDL, and SOAP core existing today has proven its point.

The success of a Web services project to a large extent is driven by the general architectural decisions such as choice of an appropriate hardware and operating system platform, as well as non-technical factors such as management of expectations and good teamwork. All practices established on other application development projects can be fully leveraged.

A decision against a certain element of the technology, e.g., UDDI, or concerns in areas such as security and transactions, can not justify ruling out the entire technology – the modular organization of the various Web services specifications allows a best-of-breed strategy. Complementary technologies can be used to complete the Web services stack on a per-scenario base.

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 distributed computing technology?

5. WEB SERVICES BEST PRACTICES

The best practices in this section directly originate from our experiences gained in this project. We can only briefly introduce a small subset of these practices here; the text book *Perspectives on Web Services – Applying SOAP, WSDL, and UDDI to Real-World Projects* [24], features all of them in much more detail, along with additional ones originating from other projects.

Follow the design-by-contract principle for service modeling. We recommend always describing services in WSDL and XML schema to decouple client and server development. For example, the only communication link between our client and our server implementation teams was WSDL document exchange via e-mail (PoC) and service repository (production releases). Consider developing your own WSDL generator if many similar processes have to be supported or a server-side function repository exists (as in our case). An indication that a custom generator might be a good idea is that developers copy-and-paste extensively [24].

Elements of Service-Oriented Analysis and Design (SOAD) [22] starts a more comprehensive discussion of service modeling, suggesting an interdisciplinary approach combining elements from Business Process Modeling, Enterprise Architecture Frameworks and Object-Oriented Analysis and Design.

Select service messaging styles based on interoperability characteristics and tool support. The `style` and `use` attributes in the WSDL specification can be set to document/literal or rpc/encoded [6]. Historically, rpc/encoded had better tool support, and interoperability could be achieved in most cases (we used this style successfully in our first release). However, due to various ambiguities pertaining to the rpc/encoded (de)serialization rules, WS-I now prohibits this style, so that document/literal has become the preferred style supported by many tools (we changed the generated interfaces to document/literal in our second release).

Carefully evaluate which service matchmaking strategy fits your needs. Using UDDI on the (public) Web is problematic not for technical, but organizational reasons. Issues such as business model, data quality, and trust have to be answered. For these reasons, we believe that UDDI is most useful in intranet and extranet scenarios where the user groups are well known [24]. As we support more than 500 Web services, introducing a private UDDI registry would have been perfectly justified in our case. We would have done so if a metadata repository had not already been in place.

Apply standards pragmatically; follow the 80-20 rule. It is not required to always use all elements of a technology. Furthermore, we recommend upgrading to higher specification levels only if there is a concrete need, and not for its own sake (for example, even in the second release we decided for SOAP 1.1 rather than SOAP 1.2). Unnecessary, distracting changes can be minimized this way. The 80-20, or keep-it-simple, rule also helps to achieve interoperability [24].

6. CONCLUSIONS AND OUTLOOK

In this report, we described how we designed and implemented a Web service-oriented architecture consisting of standardized business functions (processes) to be assembled in custom applications in a flexible and channel-neutral manner. The resulting Web services enablement of the Dynamic Interface is a key building block in the enterprise architecture of Sparkassen Informatik, providing the glue between applications and business functions.

We decided for Web services because they are independent of any component model, implementation language, transport protocol, operating system and platform (*loose coupling* promoted). Interface and implementation are separated from each other (facilitating *encapsulation*). Furthermore, Web services are based on open standards and can be invoked over existing Internet/intranet infrastructures. There is comprehensive Web services support in modern software development tools such as IBM WebSphere Studio Application Developer™ and Microsoft Visual Studio .NET™.

Concrete benefits the Web services solution brings to the table in our context are: *Design by contract*: WSDL provides a standard interface description of business components; there is off-the-shelf support in standard development tools (no software distribution required). *Improved client interface*: A WSDL- and XML schema-driven, business function-specific API is available, which allows coding against generated convenience proxies rather than lower-level XML and HTTP libraries. The availability of WSDL and tool support for it was one of the main drivers for our decision towards Web services – SOAP alone would not have been enough. As envisioned, WSDL played a key role to the success of the project. *Write once, use everywhere*: It is no longer required to write custom, platform-specific code; true interoperability between platforms is achieved via SOAP.

We took the following key architectural decisions:

- *Service modeling 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 (PoC and first release), an optimized IBM implementation of JAX-RPC/JSR 109 called WebSphere 5.0.2 SOAP (second release), and Apache Axis (client-side).
- *SOAP communication style and encoding*: we supported both rpc/encoded and document/literal in the first release, but moved away from rpc/encoded for the second release 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) already is in place. Therefore, we do not use UDDI, even if a business need for a central service broker/directory exists.

For the future, Sparkassen Informatik is committed to continue to support and enhance its Web services solution. To maintain WS-I and other standards compliance [21] is a continuous activity. Several functional enhancements are planned, further improving client developer productivity. Support for additional Web service provider platforms and multiple transport protocols exposed through the same client interface could evolve the solution into a full-blown, distributed *Enterprise Service Bus* (ESB).

Another area of investigation is declarative and descriptive process flow execution (composition of higher-level services), as available through the *Business Process Execution Language for Web Services* (BPEL) [3,15] and BPEL modeling tools and runtimes. As of today, the assembly of processes into end-user applications is application specific and typically hard-coded on the client side; a predefined set of high-level services is orchestrated in the backend (programmatically rather than declaratively).

Finally, an additional option would be to leverage the emerging Web Services Security (WS-Security) standards and their implementations as defined by the OASIS consortium [16]. Currently, all security requirements such as integrity, confidentiality, authentication and authorization are fully addressed on the network layer, on the transport layer, and on the application layer.

Our conclusion from our encouraging project results is that the Web services core technologies, namely XML, WSDL, and SOAP, are ready for production use, and able to solve real business problems. On the other hand, the support for the higher layers of the overall Web services stack (e.g., process choreography and security) still has to improve; two key challenges for any related standardization and product development effort are to maintain backward compatibility and not to break the original simplicity of the approach. For example, we expect future versions of specifications such as the WS-I Basic Profile and JAX-RPC not to cause any major reengineering efforts for the existing users of this technology.

In our case, client-side usability is the benchmark: our Web services-based process interface at all times has to be easier to code against and better maintainable than the existing proprietary approaches in order to justify the decision for Web services in the long term. It only makes sense for Sparkassen Informatik to buy rather than build middleware such as SOAP runtimes and WSDL tooling if these products meet these high standards, as well as general quality factors such as completeness of standards support, seamless interoperability, API stability, and robustness – today, as demonstrated in this project, and in the future.

7. ACKNOWLEDGMENTS

This paper is the result of a joint project effort. In particular, we would like to recognize the following individuals: Michael Brandner (IBM); Jiri Andress, Martin Fleming, Dr. Karsten Johannsen, Frank Strecker, and Thorsten Stumpf (also IBM); Reinhard Nolte, Guido Ranft, and Torsten Verstappen (Sparkassen Informatik).

8. REFERENCES

[1] Adams, J., Koushik S., Vasudeva G., Galambos G., *Patterns for e-business – A Strategy for Reuse*, IBM Press, 2001.

[2] Brandner M., Craes M., Oellermann F., Zimmermann O., *Web Services-Oriented Architecture in Production in the Finance Industry*, Informatik-Spektrum 02/2004, Springer-Verlag, 2004.

[3] *Business Process Execution Language for Web Services Version 1.1*, available from http://www.ibm.com/developerworks/webservices/library/ws-bpel

[4] Brown K., Reinitz R., *Web Services Architectures and Best Practices*, IBM developerWorks 2003, http://www.ibm.com/developerworks/websphere/techjournal/0310_brown/brown.html

[5] Buschmann F., Meunier R., Rohnert H., Sommerlad P., and Stal M., *Pattern-Oriented Software Architecture – a System of Patterns*. Wiley, 1996

[6] Butek, R., *Which style of WSDL should I use?*, IBM developerWorks 2003, http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl

[7] Component Based Development and Integration (CBDI), Insight for Web Service & Software Component Practice, http://www.cbdiforum.com

[8] Endrei M., et al., *Patterns: Service-oriented Architecture and Web Services*, IBM Redbook, April 2004, http://www.redbooks.ibm.com

[9] 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

[10] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[11] IBM developerWorks portal. Articles, tutorials, sample code, links to trial versions of software and open source assets. http://www.ibm.com/developerworks/webservices

[12] IBM e-business on demand overview, available from http://www.ibm.com/e-business/index.html

[13] JAX-B, Java XML Binding, available via http://java.sun.com

[14] Java XML API for Remote Procedure Calls (JAX-RPC), available via http://java.sun.com

[15] Leymann F., Roller D., Schmidt, M. T., *Web Services and Business Process Management*, IBM Systems Journal, Vol. 41, No 2, 2002

[16] OASIS consortium, http://www.oasis-open.org

[17] *Simple Object Access Protocol (SOAP) 1.1*, W3C Note 08 May 2000, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[18] Sparkassen Informatik on the Internet, http://www.sparkassen-informatik.de

[19] Wahli U., Tomlinson M., Zimmermann O., Deruyck W, Hendriks D., *Web Services Wizardry with WebSphere Studio Application Developer*, IBM Redbook, 2002

[20] *Web Services Description Language (WSDL)*, W3C Note, http://www.w3.org/TR/wsdl

[21] Web Services Interoperability Initiative (WS-I), http://www.ws-i.org

[22] Zimmermann O., Krodgdahl P., Gee, C. *Elements of Service-Oriented Analysis and Design*, IBM developerWorks 2004, http://www.ibm.com/developerworks/webservices/library/ws-soad1

[23] Zimmermann O., Müller F., *Web Services project roles – The team perspective*, IBM developerWorks 2004, http://www.ibm.com/developerworks/webservices/library/ws-roles

[24] Zimmermann O., Tomlinson M., Peuser S., *Perspectives on Web Services – Applying SOAP, WSDL and UDDI to Real-World Projects*, Springer-Verlag, 2003