



Principles of SOA Design

A Whitepaper from Cape Clear Software Inc.

Table of Contents

INTRODUCTION	3
DEFINING A BUSINESS SERVICE.....	3
THE WSDL DESIGN VIEW.....	5
LOOSE-COUPLING.....	6
LOOSE-COUPLING MECHANISMS	7
CAPE CLEAR PRODUCT SUPPORT FOR SOA	9
FURTHER RESOURCES.....	9

Principles of SOA Design

Copyright © 2004 Cape Clear Software Inc. All rights reserved. This document is not intended for production and is furnished as is without warranty of any kind. All warranties on this document are hereby disclaimed including the warranties of merchantability and fitness for a particular purpose.

Trademarks

Cape Clear, Cape Clear Studio, Cape Clear Server, Cape Clear Data Interchange and Cape Clear Manager are trademarks of Cape Clear Software Inc. in the United States and other countries.

All other company, product, and service names mentioned in this document are trademarks of their respective owners.

CPV-DOC-3031

Introduction

This article discusses how Web Service design principles can be used to support a Service Oriented Architecture (SOA). In particular, the following topics are covered:

- Defining a Business Service – what makes for a sensible Service Definition in the context of a SOA.
- The WSDL Design View – the mechanics of formally defining a Service interface using XML and WSDL.
- Loose-Coupling – introducing flexibility into the system by allowing Service invocation using a variety of document formats.

Familiarity with Web Service technologies (SOAP, WSDL, etc.) is assumed throughout.

Defining a Business Service

As companies increasingly seek to expose applications to internal and external users, it is critical that the exposed interfaces are well defined, easy to use and meaningful to the service consumer. This means that they must more closely reflect business concepts and requirements (documents, processes) rather than low-level technical concepts (APIs, data types & platforms). Interfaces exposed in this way are referred to as Business Services and the overall approach is known as a Service Oriented Architecture (SOA). The following outlines factors to be considered when designing a Business Service.

Granularity

An often-asked question in (distributed) system building is what is a sensible interface size or granularity for an entity within the system? In previous models for entities read objects, components or classes. In SOA they are Business Services used within the context of a business process that will need to be understood by business users. Therefore a Business Service interface should typically reflect real-world business documents: forms, orders, contracts, and invoices. Note that this is a much coarser granularity than previous integration models, which focussed on low-level programming APIs. Earlier RPC or message-passing approaches used

simple messages flowing between complex API endpoints. In the SOA case we have larger, richly typed, self-contained documents flowing between coarse endpoints.

Re-use

In many cases, it may be best to model Services based on existing document formats. For a Financial Services application this may be a SWIFT MT100 money movement, or in a manufacturing application this could be a SAP R/3 order. Note that these may not be SOAP or even XML documents, but historically they do represent a sensible message or document within the context of a business process. Now, technologies such as Cape Clear Data Interchange support both transport and data transformation of these formats to SOAP. Therefore, existing business processes can be re-used and integrated into a SOA model. This approach, which allows a Service to be invoked by a range of documents including legacy formats, is known as loose-coupling. A more detailed description of loose-coupling is provided later in this paper.

Interface is not the same as Implementation

The choice of outward-facing Business Service definition doesn't exclude the (re-)use of fine-grained components, based on a Java/.Net/CORBA model, within the physical Service implementation itself. However, this should be transparent to the client using the Service (i.e. the Service definition should avoid references to low-level objects within the Service implementation).

State

A document exchanged between Business Services should be self-contained — it should be possible to establish context from the document itself both in terms of identity and state within a business process. Documents within a SOA should not rely on external state management such as external component references. This is relevant to both synchronous (typically HTTP) and asynchronous (JMS) models. In the former, load-balancing at the HTTP-level is often used for scalability with application state, therefore maintained at the database rather than the Web Service layer — hence Service implementations tend to be stateless. In the JMS case, transactions could take hours or even days with the documents outliving individual system process lifecycles. Again, avoidance of implementation references within the document structure is advisable.

The WSDL Design View

One of the implications of the Business Services principle is that service designers need to be able to define the Service interface without reference to existing technical API's. This runs contrary to many Web Service platform implementations in which the API (Java/EJB interface) is used as the starting point for Web Service implementation and designers must "code first" in order to derive the service. While attractive for some applications there are many drawbacks to this approach, even in the case of simple RPC-style applications:

- WSDL created from source code is less strongly typed than WSDL that is created from the original XML schema. There are a large number of schema features that are not supported by automatic WSDL generators (e.g. enumerations, range checks, pattern matching, cardinality rules). Therefore, a lot of the value of an XML-based approach in terms of tighter data definition is lost.
- Creation of Web Services from source code implementations can lead to interoperability issues across platforms. For example, a service that uses Java remote references is not interoperable with a .NET-based client. .NET simply has no understanding of what to do with a java remote reference. Using a Web Services design methodology these interoperability issues can be avoided since both client and server are working from a common set of XML schema types.

The WSDL Design view, starting with an XML schema (pre-existing or created from scratch) and importing this into the WSDL to build the final Service specification, also brings the following benefits:

- If you can create WSDL first then you can use automated tools to generate the client and server-side code that implements the service. This is a considerable productivity improvement since it reduces massively the amount of code that has to be written. This is particularly true of complex interfaces where it is difficult to "code first". Generating a coded interface for every data type in a complex schema saves a considerable amount of work.
- Separation of design and development — the service designers can specify requirements as WSDL and pass these to developers for implementation.

- Service producers and consumers can work in parallel. By defining the Service interface as WSDL, client-side developers can work independently of server-side developers. The client-side and server-side do not even need to use the same development language thus significantly increasing the number of developers who can participate in distributed application development.
- WSDL is portable across Web Service frameworks thus it is possible to effectively decouple the service design and implementation of Web Services.

Loose-Coupling

Years of experience with distributed systems have led to the inevitable conclusion that it is practically impossible to eradicate incompatibilities between interfaces through standardization. Despite the best intentions, and investment in standardization initiatives, there will always be problems due to political, technical, geographical or philosophical boundaries. Add to this the perennial bugbear of versioning. The notion of loose-coupling takes a different approach: don't depend on standardization — assume incompatibility and put in place mechanisms to easily rectify it.

In the Web Services domain, loose-coupling allows a Service to be accessed using any arbitrary document format, regardless of syntax, providing the physical business data is present within, or can be determined from, the document contents.

Hence, the brittleness of earlier models (e.g. DCOM) that relied on client and service being aligned right down to the byte-level is removed.

Self-Healing Software?

One common misconception is that loose-coupling will support the notion of 'self-healing' software. Nobody is claiming that all incompatibilities can be fixed through loose-coupling. If a Service definition is updated to require an additional piece of business information that cannot be determined from the contents of the document then the client is going to need to be updated to provide that information.

Loose-coupling can however provide dramatic time and cost savings by rectifying the annoying and expensive syntactical differences that typically plague interoperability.

The following outlines some of the incompatibility issues that can be addressed using loose-coupling.

- Filtering
 - Reject input that does not meet certain requirements e.g. parameter bounds
- Renaming
 - Fix clashes in nomenclature e.g. 'trade_date' and 'TradeDate'
- Assembly
 - Compile output is from different parts of the input including string manipulation
- Calculation
 - Compute output as a function of the input
- Completion
 - Fill out missing fields using sensible defaults
- Rearrangement
 - Sort or re-order lists and arrays (e.g. sort a chronological list into alphabetically order)
- Restructuring
 - Repair structural (syntax) incompatibilities. This can include conversion between XML and text data formats
- Conditional transformations
 - Determine the document transformation to be performed based on the input. This is particularly useful for versioning.
- Enhancement
 - Add missing type information
- Lookups
 - Query a look-up table or database to translate input content e.g. look-up a customer ID to fill-out a Shipping Address.
- Transport Incompatibility
 - Map between transports – HTTP, JMS, SMTP, Filesystem

Loose-Coupling Mechanisms

While many vendors propose and support the idea of loose-coupling, very few actually offer a concrete mechanism for this doing this! Cape Clear however has been at the forefront of this loosely-coupled approach and has provided commercial products to physically support such a model since early 2000. The company takes the view that it is **impossible to really benefit from a SOA strategy without loose-coupling**. Therefore any Business Integration product set based on Web Services would be incomplete if it did not offer a deeply integrated XML mapping and transformation capability.

At its core, the Cape Clear approach is based on the native XML transformation mechanism, XSLT. Considering that XML provides the underpinning for Web Services this is a somewhat

obvious choice. Alternative approaches to loose-coupling such as late-binding using untyped 'wildcard' elements within a document such as `xsd:any` are not a useful option. This simply pushes the mapping and transformation back into the implementation code. Therefore some of the business semantics become implicit within the implementation – which somewhat defeats the aims of explicit service definition in the first place! The more business meaning is published within the service definition, the easier it is to understand and therefore use.

For the beginner, although XSLT provides an extremely powerful transformation capability, it also has a considerable technical learning curve. This is somewhat at odds with the value proposition of a loosely-coupled approach espoused above. If all that's being proposed is that low-level XSLT 'code' is being substituted for say Java-code to fix syntactical incompatibilities then the savings are minimal.

Cape Clear addresses this by providing powerful graphical mapping tools and wizards to allow users to create the required transformations without needing to resort to low-level XSLT programming – the XSLT is automatically generated. Hence, all that's required is knowledge of the business context and document structure rather than an understanding of the underlying XML technologies. This means that mappings can be created in **hours rather than weeks** resulting in the significant productivity savings promised above.

Cape Clear Data Interchange takes this one step further to address not just XML to XML mapping but also the conversion from various text formats to XML as well as transport mapping (e.g. HTTP to JMS). Therefore, it is no longer necessary to restrict Service invocation to a SOAP message corresponding exactly to the Service WSDL. It could just as easily be the contents of an Excel Spreadsheet or data lifted via scheduled file pick-up from an AS/400 application. Thus, Cape Clear Data Interchange effectively extends the domain of applicability of XSLT beyond just the world of XML documents.

Cape Clear Data Interchange therefore influences the design view for SOA considerably. By providing an easy means for mapping and transforming legacy data it allows Business Service developers to map legacy Business processes and documents directly to a SOA-based approach.

Cape Clear product support for SOA

The Cape Clear Business Integration Suite supports a SOA-based approach the whole way through design, development, integration and deployment. The Business Integration Suite is comprised of Cape Clear Studio, Cape Clear Server, Cape Clear Data Interchange and Cape Clear Manager.

- Cape Clear Studio provides a WSDL Editor to enable top down service design without requiring the user to understand the intricacies of WSDL or XML Schema syntax.
- Cape Clear Data Interchange provides graphical mapping tools that allow developers to rapidly construct transformations to rectify inconsistencies in data formats between versions.
- Cape Clear Server supports simultaneous deployment of multiple Service versions that facilitates backwards and forwards compatibility when used in conjunction with the integrated mapping and transformation capability.
- Cape Clear Manager provides a configurable, content-based transformation mechanism that can query details of the inbound request to determine version and apply a suitable transformation to reconcile any interoperability issues. These rules could be used to check the contents of individual XML elements or to query the document namespace to evaluate version number directly.

A full architecture is available on our website at <http://www.capeclear.com/products/>.

Further Resources

The Big Tour – a look at how one company used Web Services to integrate their suppliers, customers, and internal systems:

<http://www.capeclear.com/products/tour/index.shtml>

Clear Thinking – Insights into the changing world of integration:

http://www.capeclear.com/clear_thinking.shtml

Cape Clear's Web Services Developer Community:

<http://www.capescience.com/>

Cape Clear Software Inc.

Website: <http://www.capeclear.com/>

US toll-free: (888) 227 3439

Europe: +44 20 8899 6020

E-mail: info@capeclear.com