The SOA Magazine

Feature Article

Service Error Content Patterns (Part I)

by Ronald Murphy

Published: July 30, 2009 (SOA Magazine Issue XXX: July 2009)

Download PDF

Digg This • De.licio.us • Slashdot • Technorati • StumbleUpon • Google Bookmark

Abstract: Exceptions are a necessary evil in the world of service development and one that must be controlled and planned for. Allowing exceptions just to happen can have numerous negative consequences, especially when relying upon some of the more common fault handling options made available to us with current Web-based technologies. In this two-part article series we'll explore several successful techniques for controlling runtime error conditions and ensuring that their occurrence does not unnecessarily (or chaotically) disrupt the business process flow. The techniques form actual development patterns that can be applied to different types of services based on different types of service technologies.

Introduction

The name of SOAP's earlier cousin, XML-RPC, nicely captures the dual nature of SOAP itself: in one sense, this Web service technology is focused on XML and document exchange (via the document-literal binding), and in another sense it is a remote programming mechanism (via the RPC binding). Nowhere is this duality more evident than in error modeling. Are errors just a kind of data to be carried in our document-like responses, or are they part of the procedural control flow? Looking just at the SOAP standard alone, we seem to have an easy answer: Errors should be expressed as SOAP faults. But a look into the history of programming reveals a more complicated situation with a variety of trade-offs, as we'll examine in this article.

XML-RPC and SOAP arrived at the crest of an era of very large scale programming - the 1990's. This period brought us operating systems exceeding 10 million lines of code, programming call stacks nested a hundred or more lines deep, and other technology wonders. To cope with the accompanying complexity, the industry favored languages such as C++, Java, and C#. In all these languages, the exception - shall we say - raised itself, as a solution toward unwinding the maddeningly deep call traces and helping impose order and predictability on error handling.

Before exceptions, programmers had to pass an output error container over and over as a parameter in a deeply nested set of method calls - a programming faux pas referred to as stamp coupling. Or they had to store errors into a globally available structure, threatening thread safety and introducing dependencies on that global structure across large spans of code. Both approaches were becoming increasingly out of favor due to the rise of object orientation and multi-threaded programming. The solution: the exception pattern of error propagation and context unwinding, reminiscent of operating system signals and the even older "alert" concept of signaling systems. The exception concept was so compelling that even older languages like Basic were retrofitted with it.

The software layering we see in large systems highlighted another issue which we call the "foreign error" problem. At any particular error boundary, you might expect callers to code to the error structures of the software they were calling. But the further removed the layers, the less likely it was that the higher level software would know the exact nature of an error. In multi-layer systems, higher level calling layers somehow had the obligation to pass, convert, or understand error structures they had no business knowing about. You

could try to standardize the typing or structure, but with vendors around the globe contributing software to programs, there wasn't much hope. We needed something built into the language. Enter the Exception as a language type, serving to model errors by way of a well-defined type hierarchy. By being guaranteed an Exception base type, you had the assurance that you could code to some "lowest common denominator" error base class.

The C++ programming language became the first truly mainstream industry language offering the exception features of a throw-catch pattern and exception typing, although the roots of exceptions can be traced to Pascal, CLU, and original academic concepts [REF-1]. The approach in C++ presented a quantum leap in the manageability of errors in large scale systems. Java and C# followed the lead of C++ in their error handling design. Java included one slightly controversial refinement (first introduced in CLU) through the use of checked exceptions - thereby distinguishing "expected" or registered exceptions from "unexpected" exceptions. The core idea was for methods to declare their expected exceptions with a throws clause on the method, aiding callers. This has been criticized [REF-2] as essentially re-introducing stamp coupling by requiring throws clauses to be repeated across the signatures of the whole method dependency graph of a program. The throws replicates itself like a chain letter. However, the expected/unexpected distinction itself does have some useful aspects, which we will later revisit in the context of Web services.

At about the same time as the mainstream exception revolution of C++ and Java, programming framework vendors and other parties were trying to build a better RPC / CORBA with web-friendly content, XML, and internet-friendly network transports such as HTTP. With exceptions at the height of their popularity, ignoring them in the Web service standards would be unthinkable, especially for vendors offering these features prominently in their environments [REF-3]. So, SOAP was ratified by the W3C with the SOAP fault as the only standard error propagation mechanism, and mappings from SOAP faults to the corresponding programming models were carefully devised, as in the JAX-WS standard [REF-4]. SOAP faults became the direct analogue of programming exceptions, and the WSDL "fault" declaration on WSDL operations became the equivalent of a Java method "throws" clause - "checked" exceptions.

After this historical tour through error handling in the programming and WSDL-SOAP worlds, let's consider some practical issues.

Limitations of the Exception Model

The biggest strength of exceptions can also be their biggest weakness, one which is well known to anyone who has worked with them: Exceptions fundamentally change the processing flow. In programming environments, you can catch exceptions; what you cannot do is go back and un-throw them. The code flow through the called logic is fatally altered, like one of those cars you recently drove by on the freeway shoulder with a crunched up front end.

There are two main consequences to this finality. First, the programming model does not allow for reporting multiple error conditions very well. Since our method call terminates errors after the first and only exception thrown can't be accumulated - at least not without workarounds, like putting off the throw until all errors have been detected and collected.

In Web services, faults represent only one class of a variety of fatal error conditions, a fact which further impairs their practical use. According to the SOAP/WSDL compatibility profile WSI-BP, R1126 [REF-5], "An INSTANCE MUST return a "500 Internal Server Error" HTTP status code if the response envelope is a Fault." But many real world servers return 500 errors in all kinds of horrible situations such as out of memory errors, buggy code, or improperly initialized systems. Because of this, in practical use, many clients give very cursory handling to faults. In some cases, they will just lump all the HTTP 500 errors together, giving end users a very vague message about the problem.

The data model for exceptions further ignores the return of multiple errors as a single event. There are no language features or common programming patterns for multi-error exceptions. Exceptions typically carry and localize a single message string with human-readable text. There is no standard composite exception that can carry subordinate exceptions. Essentially, exceptions are "singularities", in both the grammatical and mathematical sense! Similarly, SOAP and WSDL fault modeling, and programming mappings like JAX-WS, really specify one payload (for example, one JAX-WS fault bean) per fault. You could design a payload that contains multiple errors, but that is not part of the standard; it becomes service-specific data modeling.

A second and perhaps more significant limitation of the fault/exception approach: any response data you were building up as part of your result processing will be lost. By definition, if an exception is thrown, you won't be seeing any result value from the method you called. This becomes a real issue if you actually have a mixture of results and errors - for example, if your errors are really more like warnings.

We might be tempted to treat warnings as a different kind of animal: maybe they're just data, not errors. But this quickly tangles us up in practical issues and contradictions. First, many real world programs treat warnings and errors quite similarly: They show the messages to end users, and make a fatal/non-fatal decision based on whether any errors exist. Warnings and errors are cut from the same red flag-waving cloth, typically modeled the same way. Second, business rules may change warnings to errors or vice versa. Separating them impairs flexibility to change.

Third, the context of usage alone may influence whether a given problem is a warning or error. Say I have an operation to list an item for sale, and another operation to re-list a previously listed item that didn't sell. If the item ID for the re-list operation is not found, that's probably an error - fatal (Figure 1). But in a bulk operation that can re-list 10 items at a time, it might be only a warning if one of them is not found (Figure 2). We should be free to represent error data using a similar structure for errors and warnings, and then to simply make a severity decision based on the business use case.

Figure 1: Single request with error.

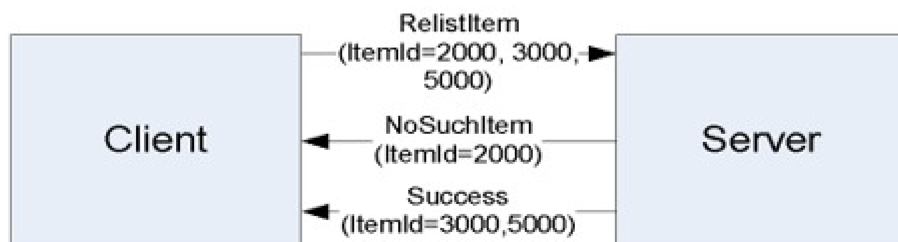


Figure 2: Multiple request elements, combination of error and success.

The third limitation of exception modeling derives from adopting a declarative type system as a way of expressing our errors. The exception hierarchy helped solve the "foreign error" problem, but this issue was replaced by a more subtle one involving type evolution. In any larger distributed deployment, we tend to have older consuming software layers that were coded to a specific type system with a fixed set of known types. As their providing software layers add new types, the older clients don't know about these types unless they are rebuilt and redeployed.

For example, in Figure 3, a client is originally coded to expect a response with ElementX of a particular type. If a server later is updated to also return ElementY of a new type, the client will not understand this new type until it is updated to incorporate the later version of the service's schema, its type definition system.

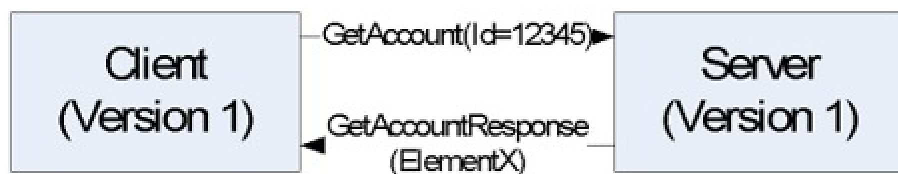
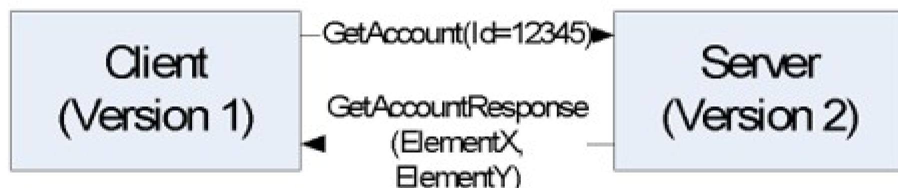


Figure 3: client getting unknown type from newer server.



This type evolution problem will be familiar to developers who have worked with client-server architectures - and it affects exceptions as much as any other type. This applies to exceptions also - and particularly, remote exceptions! Older clients cannot have type knowledge of newer faults or exceptions. These newer

errors must be treated as if they are foreign, by using the "lowest common denominator" approach.

Recognizing this, the Web service standards have provided such a "lowest common denominator" error. The generic SOAP fault is specified to have a faultcode (broad SOAP category of error such as client problem, server problem, etc.), faultstring (readable message), and detail (the custom type information for your error).

The error evolution problem is accommodated by Web services extended standards such as the Web Services Interoperability Organization Basic Profile (WSI-BP). Rule R2742 sagely advises: "An ENVELOPE MAY contain [a] fault with a detail element that is not described by a soapbind:fault element in the corresponding WSDL description." This allows for unmodeled "foreign" faults, such as those coming from a newer version of a service to an older client. Older clients will at least have visibility of a generic fault object, although they will not be able to interpret your specific fault detail types.

Rule R2742 is helpful in keeping client software loose in its expectations for arriving faults. But considering it more closely, it has a very interesting consequence for interoperability: Web services can throw faults that are not declared in the WSDL! The WSI-BP standard makes it clear that R2742 is intended only to solve the foreign error problem, and admonishes us: "A Web service description should include all faults known at the time the service is defined". However, the normative word is "should", and many well known industry Web services such as those of eBay, Amazon, and Microsoft MapPoint declare no faults of any kind in their WSDL definitions.

From the programming language perspective we established earlier in this article, undeclared faults represent a kind of "unchecked exception" similar to those of Java. Indeed, runtime Java exceptions are diligently converted into generic SOAP faults by SOAP runtimes such as the JAX-WS reference implementation.

Modeling Patterns for Service Errors

At this point we have a clear understanding of the motivation, advantages, and limitations of the exception / fault approach to error modeling. The main choices available to us in designing WSDL for errors are the same choices we have in modern programming environments:

- Purely fault-based modeling: We can faithfully adhere to the WSI-BP profile and model all "known" service errors as declared faults in the service WSDL (standard fault usage)
- Purely response-based error modeling: We can avoid WSDL fault declarations, and model our service errors within response data ("Response resident errors").

In addition, we examine some attempts to blend the two approaches:

- Merged response/fault structures: the incredible multi-purpose result container!
- Mixed usage, in which responses carry the errors in specifically prescribed situations, and faults handle the remainder of cases.

WSDL-Declared Faults

WSDL faults are declared as SOAP messages, just like the input and output data for an operation. A typical pattern for any SOAP message is to declare a top-level element as the message, and then feature this element in a wsdl:message declaration and feature the wsdl:message in an operation declaration, as in the following example:

```
<xs:element name="ItemNotFound">
  <xs:complexType>
    <!-- type information… -->
  </xs:complexType>
</xs:element>

<wsdl:message name="ItemNotFound">
  <wsdl:part name="ItemNotFound" element="tns:ItemNotFound"/>
</wsdl:message>

...
```

```
<wsdl:operation name="FindItem">
    <wsdl:input ... />
    <wsdl:output ... />
    <wsdl:fault message="tns:ItemNotFound"/>
</wsdl:operation>
```

(A more complete elaboration of this example is given in at the end of this article.)

For any fault-based modeling approach, we recommend using a base class to hold fault information that is common across all your services.

```
<xs:element name="ItemNotFound">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="tns:BaseServiceFaultType">
        <xs:sequence>
          <!-- specific fault data here -->
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```

The base class should have error information in a standardized format, as shown below; we'll visit this more closely later.

```
<xs:complexType name="BaseServiceFaultType">
  <xs:sequence>
    <xs:element name="Error" type="tns:ErrorType"/>
  </xs:sequence>
</xs:complexType>
```

The main advantage of WSDL faults derives from their "first class" status as dedicated SOAP message types. For Web services that emphasize interoperability with tooling, network infrastructure, and standardized clients, the advantage can be compelling: Interoperating software will understand that errors are in flight when they see faults. Intermediate systems such as enterprise service buses can log and report errors, and standard test tools will know that an error was returned.

Development tooling is also your friend in this area. Exceptions and faults are well integrated with one another in most mainstream SOAP programming environments. In fact, faults have a way of coming out of the woodwork in SOAP runtime engines, as we'll discuss in an upcoming section.

Another advantage of using faults in your error modeling is that faults can be specifically declared against each operation. For example, our "findItem" operation above declares "ItemNotFound" which is specific to that operation. However, it becomes cumbersome to declare large numbers of distinct fault messages, and it's usually more appropriate to use this to declare types of faults (with particular message structures) rather than instances of faults. Sophisticated operations may throw dozens of particular errors relating to various validation or processing problems, and WSDL isn't really an effective way to catalog all of these - at least not using individual wsdl:fault declarations. This actually calls out one of the limitations of WSDL as a Web service modeling language: It doesn't provide a built-in way to detail behavioral aspects of Web services, which include business level validation, specific error instances, and algorithmic behavior.

The limitations of using WSDL-based faults are the following:

1. The solution is incomplete, because of the error evolution problem.
2. The lowest common denominator built into SOAP is very minimal indeed. You aren't guaranteed much more than a single string identifying the error.
3. Multiple errors are not explicitly addressed by the approach.
4. Mixtures of errors and normal response output are not possible.

Conclusion

In part two of this article series we'll introduce the actual patterns that help us overcome the limitatoins we've just described.

References

[REF-1] John B. Goodenough, Structured exception handling, Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages, p.204-224, January 20-22, 1975, Palo Alto, California.
[REF-2] http://www.mindview.net/Etc/Discussions/CheckedExceptions
[REF-3] http://msdn.microsoft.com/en-us/library/aa480514.aspx
[REF-4] Currently at version 2.0, JSR 224, http://jcp.org/en/jsr/detail?id=224
[REF-5] http://www.ws-i.org/Profiles/BasicProfile-1.1-2006-04-10.html

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL

Copyright © 2006-2009 SOA Systems Inc.