Serendipitous Reuse

Steve Vinoski • *Verivue*

According to the dictionary on my Macbook Pro, the word *serendipity* means "the occurrence and development of events by chance in a happy or beneficial way." The dictionary entry also explains that the term comes from an old fairy tale, "The Three Princes of Serendip," in which the title characters repeatedly make pleasant discoveries of things for which they're not specifically looking. It's probably safe to say that we've all experienced serendipity at one time or another and, like the three princes, gladly welcome it whenever it comes our way.

In October 2007, I read a blog entry by Stu Charlton of BEA Systems (www.stucharlton.com/blog/archives/000165.html) in which he used the term serendipity to describe a style of integration and reuse that the Web encourages. He contrasted this with the "planned" style, which is the approach that enterprise architects typically prefer for reuse and integration.

A few weeks later, I serendipitously met Stu at the QCon 2007 Conference in San Francisco. I mentioned that I thoroughly appreciated how he had used the term in his blog. To me, this single word speaks volumes about the nature of the Web and why it's become such an indispensable tool within our daily lives. For example, serendipity is the best explanation for Web mashups, in which the capabilities of unrelated Web sites are combined to create new sites that provide benefits beyond those that the original developers had intended or even considered. Stu explained that he'd borrowed the term from Roy Fielding of Day Software, who also happens to be the coauthor of the HTTP 1.0 and 1.1 specifications and the creator of the Representational State Transfer (REST) architectural style. In searching the Web for the terms "Fielding" and "serendipity," I found several references to the following quote:

> Engineer for serendipity. —Roy Fielding

Some might find this quote puzzling, given that by definition, we can't plan for serendipity. If we're wise, we never assume that serendipity will come along just in time whenever we need it. So is it really possible, as Fielding suggests, to make a given situation more amenable to serendipity? Is it feasible to arrange the primary elements of an area such as enterprise integration in a way that encourages the development of beneficial applications that the enterprise architects never dreamed of?

Inhibiting Serendipity

Before examining how we might nurture serendipity, let's explore how we often unwittingly hinder it. It turns out that software development and integration practices that are widely viewed as advantageous to the enterprise can actually inhibit integration and reuse — not only of the serendipitous kind but even of the planned kind.

Enterprise architects tend to favor practices and approaches based on a service-oriented architecture (SOA). Many approaches to connecting software systems, including but not limited to Corba, many Java-based integration systems, and .NET, generally follow the best practices of SOA. For example, each promotes the use of specialized interfaces or contracts for describing the systems and services being connected. Service contracts define the operations that services furnish for client applications to invoke. These contracts also define the data types that the services expect as input and return as results.

1089-7801/08/$25.00 © 2008 IEEE

Examples of languages that allow for the definition of specialized interfaces include the Web Services Description Language (WSDL) and the Corba Interface Definition Language (IDL). Yet, regular programming languages such as Java and C# increasingly perform double duty as contract-definition languages as more developers attach special metadata, in the form of extra language declarations called annotations, to normal interfaces and classes. These annotations can help tools generate the glue needed to plug objects of these types into service messaging and distribution platforms.

The overall goal with service contracts is to make the coupling between callers and services clear and unambiguous. The SOA community widely accepts that service contracts help separate interface from implementation. Another assumed benefit is that contracts let developers implement callers and services in separate languages, but this isn't always true, especially given the rate at which programming languages are replacing traditional interface definition languages. Using a regular programming language to define supposedly abstract interfaces greatly increases the chances that language-specific constructs will leak into the contract.

The benefits of contracts might seem obvious, but specialized service contracts and interfaces also have downsides. One negative aspect is that each specialized interface effectively represents a specialized protocol between a caller and a service. To invoke a service, a caller must incorporate details of each specific operation defined in the service's contract. In other words, the specialized interface forces each calling application to include custom code specific to the operations it wants to call. Calling applications must also be cognizant of each contract's "implied workflow," which is the order in which the service's operations were designed to be invoked.

The more specific the service interface, the less likely it is to be reused, serendipitously or otherwise, because the likelihood that an interface will fit what a client application requires shrinks as the interface's specificity increases. Highly specialized interfaces inhibit general reuse because only purpose-built clients can invoke them. Should requirements change — and they will — modifying such highly specialized services and clients to fulfill the new requirements can be costly because of the high degree of coupling between them.

- different message exchange patterns;
- registering and finding services and their metadata;
- configuration, management, transactions, and security services; and
- business process orchestration and workflow.

Given that market pressures force vendors to support as many relevant standards as possible, service platforms can wind up bloated and overly complicated. Even if they never use all the platform features, customer applications still pay for the underlying platform complexity.

The benefits of contracts might seem obvious, but specialized service contracts and interfaces also have downsides.

Typical SOA platforms also inhibit serendipitous reuse. They often include vendor-specific services and extensions designed to make customer applications easier to develop, or provide features that go beyond the relevant industry standards. But however well intentioned, such extensions and add-ons often end up forcing all participating applications — clients and services, alike — to use the same underlying platform or risk interoperability problems. This situation leads to the dreaded "vendor lock-in."

The problems don't end there. Platforms can inhibit reuse even if they avoid extensions and stick only to industry standards because the standards themselves also subscribe to the school of interface specialization, such that every different aspect of distributed system interaction requires a specific interface. Standards end up publishing special interfaces for a wide variety of areas, such as

Much of the weight and complexity of service interface specialization and platform implementations stem from the Remote Procedure Call (RPC) ancestry of these systems and approaches. RPC is intended to let programmers use familiar procedure, function, and object method invocation approaches to invoke remote services. Although researchers and developers long ago discredited local/remote transparency, which is the trick of pretending that a distributed invocation is the same as a local one, today's SOA platforms often tend to promote distributed system development centered on programming languages and integrated development environment (IDE) tools. This development style forces idioms and patterns that are appropriate only for local applications — especially the idiom of crafting specialized interfaces for each service — to be unwittingly projected onto the distributed systems domain. The

layers of complexity required to maintain the resulting leaky illusion of local/remote transparency are reminiscent of the convoluted equations that pre-Copernican astronomers used to explain how the Sun and other planets revolved around the Earth.

Reuse Begets Reuse

If the proliferation of specialized interfaces inhibits reuse, reducing interface differentiation should increase it. Software frameworks are a prime example of this phenomenon because extending a framework normally requires developers to provide implementations that conform to existing framework interfaces rather than inventing their own. Developers achieve reuse both by extending the framework to the particular problem they're trying to solve and by making their specialized implementations available through the framework to other users.

When it comes to software integration and service architectures, one alternative to specialized service interfaces is the uniform interface — one of the REST architectural style's constraints, defined to help induce desired architectural properties. Systems that conform to the uniform interface constraint gain several advantages:[1]

- System resources adhere to the same semantics for each operation in the uniform interface, thus simplifying client applications by eliminating the need for custom code to support specialized interface semantics.
- Developing resources means designing your implementation to fulfill the uniform interface and its expected semantics, essentially eliminating the development phase required for designing separate interfaces for each resource, with their specialized semantics and implied workflow.
- Error handling is typically a source of significant variance between interfaces as interface designers individually cook up their own data structures and exceptions for reporting problems. Under the uniform interface constraint, however, error handling also gains uniformity.
- Intermediation becomes highly practical because intermediaries can understand the uniform interface semantics just as well as resources and clients can. For example, a uniform interface can specify which calls are *idempotent* (that is, can be called repeatedly without side effects) and which aren't. Resources can include cache control information in responses to idempotent operations, so that developers can easily insert caches between a client and the resources it uses without breaking the client or needing to specialize the caches for the invoked resources. Along the same lines, introducing monitoring intermediaries makes watching over client–resource interactions almost trivial.
- Without the presence of numerous specialized interfaces, overall system simplicity increases, which typically decreases the number of defects.
- Interface versioning issues are significantly reduced, though not entirely eliminated.
- The overall system becomes much more extensible.

The primary trade-off of the uniform interface constraint is a possible decrease in efficiency for some clients due primarily to having to deal with more general data formats. However, this seems well worth it, given the overall gains the constraint yields in simplicity, visibility into system interactions, and extensibility.

It's important to note, though, that "uniform" doesn't mean "as generic as possible." We could easily abuse the notion and create a completely generic interface consisting of a single operation:

```
BagOfBytes
processThis(BagOfBytes);
```

The problem is that such a generic operation is necessarily semantically weak, which means that clients can have no expectations regarding what it might do. Clients have to assume it isn't idempotent and that its results aren't cacheable. Visibility via intermediation also becomes impractical. Idempotency, caching, visibility, and other properties can be critical to system performance and scalability. They're so valuable that it's practically unthinkable to develop distributed systems without considering them. Moreover, this interface's lack of semantics makes reusing it just as difficult as reusing a specialized interface.

The Web is a REST-based system, and its uniform interface consists of HTTP's verbs — primarily `GET`, `PUT`, `POST`, and `DELETE`. Together, they represent a fine balance between generality and specificity, between arbitrary openness and iron-fisted control. They're broadly applicable, but they also help uphold specific Web architectural properties. Rather than being chosen by accident, these verbs represent the judicious application of constraints to induce desired architectural properties, as Fielding details in his thesis.[1]

Control

Enterprise architecture is primarily about control. Architects put rules in place hoping to achieve application consistency across the enterprise, increase the chances for reuse, and cut costs. Unfortunately, as I've mentioned, allowing the proliferation of ad hoc interfaces creates a losing battle for the architect trying to gain buy-in for such rules. Many such

architects are SOA enthusiasts who intentionally ignore REST and its elements, constraints, properties, and relationships. Ironically, if applied properly, REST could yield the very consistency and reuse they seek.

Following REST would also reduce the need to continuously rein in those enterprise developers who don't necessarily see the same value in the rules the architect wants to enforce. Perhaps most unfortunate, though, is the fact that few enterprise architects realize that they're quite unlikely — even as talented as they might be — to devise something as well-balanced, broadly applicable, and inherently powerful as REST; whatever they come up with will almost certainly come up short in various technical areas. This, in turn, makes their quest for control and buy-in that much more difficult.

The Web is, in effect, an expansive application framework. If you implement your resources to obey its uniform interface and other constraints and plug those resource implementations into the Web framework, they'll be fairly consistent as well as instantly usable by a wide variety of clients. This applies not only to the World Wide Web but also to intranets and enterprise webs as well. If your resources conform to the broadly agreed Web interface, chances are quite good that unexpected clients will serendipitously reuse them in unforeseen ways.

It's highly ironic that many enterprise architects seek to impose centralized control over their distributed organizations. In many cases, such centralization is a sure recipe for failure. A proven framework based on a well-constrained architectural style like REST allows for decentralized development that, because of the architectural constraints, still yields consistency. The Web itself is proof that this form of "control without controlling"[2] works. In the long run, this approach is far more likely to achieve what architects seek than trying to enforce collections of ad hoc governance rules.

Reuse the Web and you thereby increase the chances that what you put there will be reused. Roy Fielding is right: you *can* engineer for serendipity.

References

1. R.T. Fielding, *Architectural Styles and the Design of Network-Based Software Architectures*, doctoral dissertation, Dept. of Computer Science, Univ. of California, Irvine, 2000.
2. P.M. Senge, *The Fifth Discipline: The Art & Practice of the Learning Organization*, Currency, 1994.

Steve Vinoski is a member of the technical staff at Verivue. He's also a senior member of the IEEE and a member of the ACM. You can reach him at vinoski@ieee.org.