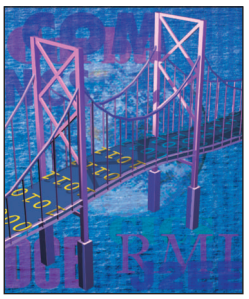

A Time for Reflection

Steve Vinoski • *IONA Technologies*

It's hard to believe that it's already 2005. Traditionally, a new year's arrival brings reflections on the previous year. Such reflection can often result in positive changes and improvements in our lives. Software reflection, a technique or approach that makes software self-aware, is similar to human reflection, although not as subjective or complicated. Proper use of software reflection can produce flexible, adaptable applications. In a sense, reflective applications, like reflective people, are capable of dynamic self-improvement.

Because integration requires flexibility and adaptation, reflection is growing in popularity for software, such as Web services, that glues applications together. Understanding reflection basics and how to apply them to your own applications can ease integration nightmares. Much has been written about reflective techniques, so I won't try to cover the topic in its entirety here. Instead, I'll focus on a problem that reflection can help solve: dynamic invocation as an approach for working across disparate type systems. (For more information about reflection in general, visit the reflective middleware section of *IEEE Distributed Systems Online*; http://dsonline.computer.org/middleware/RM.htm.)

This problem pops up frequently when integrating disparate systems: each system has its own idea of a type system, and invocations in each are based on assumptions about that type system's ubiquitousness. Such assumptions make it inherently difficult to perform invocations and send messages between systems.

Signature-Based Polymorphism

Many of us are familiar with polymorphism's utility. It lets us separate interface from implementation in our applications, which reduces coupling between modules and lets us extend applications without making changes across the whole code base. Most polymorphism uses are based on interface inheritance. Invoking a base interface method on a target object whose interface is derived from the base works because, by definition, the derived interface supports the base interface.

Yet, interface-based polymorphism is not a panacea. With popular languages such as Java and C++, polymorphism suffers from a scalability problem. Specifically, it doesn't work across multiple interface or type hierarchies developed separately in such languages. Borrowing from an old and somewhat contrived example, assume we have a lottery interface hierarchy and a gun interface hierarchy, and each supports a method named `draw`. With interface-based polymorphism, we can write code to invoke either the gun version or the lottery version, but the same code can't invoke the two different types. For real applications, you encounter this limitation whenever you try to use similar but separately developed interface hierarchies in a single application. You end up requiring similar but separate — and essentially duplicated — code to handle invocations for each interface hierarchy.

Signature-based polymorphism provides one answer to these limitations. Emerald, a highly innovative and influential distributed system from the 1980s, relied extensively on signature conformance for polymorphism.[1] The idea behind signature conformance (which also appears in nondistributed systems[2]) is to allow polymorphic behavior based solely on method signatures rather than on groups of methods defined and overridden in inheritance-related interfaces. For example, if our lottery and gun hierarchies' `draw` methods had the same signature, the same code could correctly invoke both using signature-based polymorphism.

Signature-based polymorphism hasn't become a common feature in popular programming lan-

 Published by the IEEE Computer Society 1089-7801/05/$20.00

guages, but generic programming is based on its essence.[3] For example, the C++ Standard Template Library (STL) makes assumptions about the properties — but not the interface types — of parameters passed to C++ template classes and functions. For example, a forward iterator for a container class is expected to provide only two capabilities: element-by-element forward advancing and dereferencing for reading and writing individual elements. There is no abstract base forward-iterator class that defines pure virtual functions that all concrete forward iterators must inherit and override.

This conformance-based approach allows a normal C++ pointer as well as a specialized C++ class with overloaded increment and dereference operators — two very distinct types — each to fulfill the forward-iterator needs for the same template class or function. Code based on generic programming techniques simply assumes that each template parameter conforms to the desired behavior and properties, without requiring subtype relationships to achieve polymorphism.

Although signature-based polymorphism and generic programming are incredibly useful approaches for integrating disparate code, they generally rely on compile-time checking and method binding that is too static for many applications. They aren't suitable for applications that must use application metadata to construct dynamic invocations at runtime. Despite its flexibility, generic programming doesn't work for joining disparate systems that can't be compiled together into the same application. However, I mention generic programming here because it illustrates a key benefit of reflection: applications can work around the somewhat artificial obstacles that strict interface inheritance and subtyping rules often impose.

Reflection in Corba

Fundamentally, Corba exists to support application integration — for both static and dynamic applications. In the static approach, developers write object interface definitions in the Corba Interface Definition Language (IDL), a C++-like declarative language, and then use those IDL definitions to generate code in their chosen programming language. They implement their objects based on the generated code, which is emitted by an IDL compiler. The compiler uses rules specified by the Object Management Group (OMG), stewards of the Corba standard, for mapping IDL constructs to different programming languages. The most popular language mappings are for C++ and Java, although standard mappings also exist for PL/I, Smalltalk, and Python.

Despite its flexibility, generic programming doesn't work for joining disparate systems that can't be compiled together into the same application.

We call this code-generation approach static because the generated code and the application code written against it strongly tie to the original IDL definitions, and the application code can't be used against other IDL definitions, even if they're structurally equivalent. The code is essentially tied to the interface-based polymorphism that IDL brings to the picture. Despite this lack of flexibility, most Corba applications are based on the static approach, mainly due to ease of programming and to overarching performance concerns.

Corba's support for multiple programming languages makes it quite flexible; it can combine applications written on different platforms and in different languages. Thus, it can solve the problem of integrating disparate systems with separate type system notions. In Corba, such a solution typically involves creating a set of IDL definitions that abstract the services offered by the systems being integrated, and then combining code generated from those IDL definitions with the application code for each system.

Unfortunately, that's not the end of the story. What happens when a single Java or C++ application must invoke operations in two or more sets of IDL definitions? One approach is simply to build the application with static knowledge of both IDL definitions, but that doesn't scale for large IDL definitions or for large numbers of IDL definition sets. Even with two sets, the application requires distinct code for invoking each set to address type incompatibilities. Interface-based polymorphism strikes again.

Fortunately, Corba also supports dynamic applications. With this approach, applications depend on the Dynamic Invocation Interface (DII) provided by the Object Request Broker (ORB) and the root interface of Corba's object interface inheritance hierarchy, which is named `CORBA::Object`. Every Corba object supports this interface because every IDL interface implicitly derives from it. One operation that `CORBA::Object` provides lets an application construct a dynamic request object for any operation that the target object supports. For example, if you write an application that depends on an operation named `print`, you can dynamically ask any object supporting an operation by that name to construct a request letting you invoke it.

The key part of such dynamic re-

quest construction is that the application doesn't need any compile-time knowledge of the object's specific IDL interface. The only compile-time knowledge it requires is that of the `CORBA::Object` base interface. Unlike its static counterpart, dynamic Corba invocation has no dependencies on IDL interface-inheritance hierarchies, so applications avoid the problems associated with interface-based polymorphism. Thus, dynamic Corba applications are far more flexible and adaptable than their static counterparts.

At runtime, a dynamic Corba application can dynamically discover information necessary for request construction, such as operations' names, arguments, and return types. Unfortunately, Corba has a problem with runtime metadata discovery. The standard specifies that IDL metadata is managed by an interface repository (IFR) service. If a Corba application wants to invoke an operation on a previously unknown object, it must first consult the IFR to determine that object's interface metadata, and then use the metadata to dynamically construct the desired requests.

Although this sounds simple, it doesn't work well in practice. One big problem is that almost nobody deploys an IFR as part of a production Corba application, because it's yet another server that requires computing resources and must be monitored and maintained. Static Corba applications avoid these costs because they don't need IFRs. Another problem with the IFR approach is that it's too easy for the metadata to become outdated with respect to the actual Corba object definitions if someone changes an object's interface but forgets to update the related IFR definitions.

A better approach to discovering object metadata is *direct object introspection*, which is nothing new. Reflective programming languages such as CLOS, Smalltalk, Java, and Python have supported it for years. Indeed, reflection and introspection go hand in hand; you use introspection to obtain metadata about an object or application, and then you use that metadata as part of creating and carrying out reflective operations. Unfortunately, Corba didn't support object introspection until recently.

In November 2003, I began an effort in the OMG to add object introspection to Corba. A year later, a standard approach to Corba introspection is entering a finalization phase for official adoption into the Corba specification sometime in 2005.[4] This will enable truly reflective Corba applications that can simply ask a Corba object for its metadata — rather than requiring IFR access — and use it to dynamically construct requests independent of IDL interface-inheritance hierarchies. Object introspection will also ease the integration of Corba systems with other technologies, notably Web services.

Drawbacks

Using reflective techniques to integrate disparate systems has some problems. For one thing, the code required for introspection and dynamic request construction can be jarringly different from — and much more complicated than — regular code. To call a function with normal programming-language syntax, the developer simply writes the function name and passes arguments to it, also by name. With reflection, you must find the metadata for the function or operation to be invoked, and use it to dynamically construct appropriate values to pass as arguments. This can be arduous even for relatively simple types. Worse, you often have to perform the dynamic invocation using syntax or calls that look nothing like normal invocations. The relative complexity of reflective programming, which often results in applications consisting of one or two orders of magnitude more lines of code than similar static application code, tends to mean that only sophisticated programmers practice it.

Another problem, which relates directly to the dynamic invocation problem we're discussing, is that separately developed systems don't normally offer operation signatures that match coincidentally. This is a likely reason that signature-based polymorphism has never caught on. Thus, reflection code that's very specific to a particular operation is unlikely to be widely reusable. You can avoid this problem by taking the generic programming approach and specifying what particular operations and features generic or reflective code expects from its parameters and targets, without requiring them to inherit from a particular class hierarchy. This is essentially how the JavaBeans component architecture works (http://java.sun.com/products/javabeans).

What I've described here isn't really new, but it reflects many of the common problems and arguments we see in today's Web services world. We invented SOAP in an attempt to get around the problems of requiring ubiquitous object models and types in distributed systems. Universal type systems simply don't scale. Systems based on static code generation remain popular because they're an easy choice. Static approaches map artifacts from the system level — such as service interfaces and operations — into programming language entities that we can use and manipulate using normal programming approaches. Thus, static systems appear to be easier to build and safer to use as a basis for distributed integration applications.

Unfortunately, few real-world systems are static. Almost inevitably, changes in business requirements wipe out any development advantages gained from hard-coding static interfaces and types at the programming-language level. Thus, the Web services community is increasingly moving away from remote procedure call (RPC) and interface-oriented approaches,

because they perpetuate the problems with types, interfaces, and operation invocations I outlined here.

Instead, Web service developers have been moving toward a model of passing XML documents around to simple – although not quite uniform – service interfaces. This simplified approach scales reasonably well because it mirrors the way humans interact in day-to-day business activities. This approach can be more difficult than writing static programs, but making developers' lives easier is not the goal. Even though it's harder to write reflection code, programmers are increasingly using reflective techniques to develop distributed services because the resulting systems are far more flexible and adaptive than their static counterparts, resulting in cost-effective applications that clearly and measurably solve business problems.

Will Web services development stay on this path, or will unforeseen innovations in 2005 result in a new direction altogether? Perhaps we'll reflect on that next year.

References

1. A. Black et al., "Distribution and Abstract Types in Emerald," *IEEE Trans. Software Eng.*, vol. 13, no. 1, 1987, pp. 65–76.
2. E.D. Granston and V.F. Russo, "Signature-Based Polymorphism for C++," *Proc. Usenix C++ Conf.*, Usenix Assoc., 1991, pp. 65–79.
3. K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.
4. Object Management Group, *Corba Reflection: OMG Request for Comments*, OMG document mars/2004-08-12, 2004; www.omg.org/cgi-bin/apps/doc?mars/04-08-12.pdf.

Steve Vinoski is chief engineer of product innovation for IONA Technologies. He's been involved in middleware for 17 years. He is the coauthor of *Advanced Corba Programming with C++* (Addison Wesley Longman, 1999), and he has helped develop middleware standards for the Object Management Group (OMG) and World Wide Web Consortium (W3C). Contact him at vinoski@ieee.org.