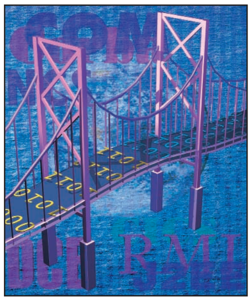
Invocation Styles



Steve Vinoski • *IONA Technologies* • *vinoski@ieee.org*

To obtain my BS in electrical engineering, I had to pass several general engineering courses. I recall taking two such courses back-to-back in my third year. The first class, statics, dealt with topics such as calculating how much load various structures such as bridges could handle, based on both their construction and the physical characteristics of the materials used to make them. The second class, dynamics, had us analyzing various characteristics of systems in motion, such as acceleration, velocity, and friction. The topics are quite different, yet complementary. Understanding one but not the other would make for an incomplete engineering knowledge base, which is why both classes were required for all engineering disciplines.

The terms *static* and *dynamic* also exist in middleware, but they apply to very different concepts: we most often use them to describe different service-invocation styles — the nature of the information that a middleware application requires to properly invoke a given service. Just as engineering statics and dynamics play an important role in analyzing and designing real-world mechanical systems, robust middleware applications require static and dynamic styles of service invocation.

To invoke a service, you need the following information:

- *Service address* — where to contact the service. The address might be something as simple as a TCP host name and port number, or something more complex such as a Corba interoperable object reference (IOR) or a message queuing end-point identifier.
- *Service contract* — what you're supposed to send to the server and what, if anything, it is supposed to return to you. For message-oriented systems, we normally specify contracts in terms of documents, such as purchase orders or requests for quotes. For remote object systems, contracts are generally synonymous with object interfaces because they tell you what operations your applications can invoke and what the parameters and return type are for each operation.
- *Service semantics* — what the service actually does. Technically, semantics are part of the contract, but I've separated them out because the contract portion can easily be specified in a form, such as a Java interface or an XML schema, which applications can read and process. Unfortunately, the same cannot be said for semantics. Despite the considerable effort invested in developing machine-understandable semantics, much work remains before they become another everyday part of the average programmer's toolkit.

In other words, what you need to know to invoke a service is: what to send to it, where to send what you're sending, and what the service will do for you. I already discussed issues surrounding the "where to send" aspect in a previous column on service discovery.[1] Here, I'll focus on "what to send."

Static Invocation

Applications that employ static invocation have a priori knowledge of what to send to a particular service instance. For example, consider the following Corba interface definition language (IDL) interface for an employee phone book service:

```
module HumanResources {
  struct Employee {
    string name;
    string address;
    string location;
    string organization;
    string phone_number;
  };
  typedef sequence<Employee>
    EmployeeSeq;
  interface PhoneBook {
    exception NoMatch {
      string name;
    };

    EmployeeSeq lookup(
      in string surname
    ) raises(NoMatch);
  };
};
```

Our PhoneBook interface provides a single operation named lookup, which takes the employee's surname as a string and returns a sequence of structures, each of which represents an employee with that surname. If no employees have the surname, the lookup operation throws a NoMatch exception.

A Java application could invoke an instance of the PhoneBook service using code like this:

```
try {
  Employee[] data =
    phonebook.lookup("Smith");
} catch (NoMatch ex) {
  System.err.println("Employee "
    + ex.name + " not found");
}
```

Compiling this code into the Java application gives it static knowledge of the PhoneBook interface, the lookup operation, the Employee structure, and the NoMatch exception. With Corba, the developer adds this static knowledge to the application by compiling it with additional code generated by an IDL compiler from the original IDL definition. Armed with this static knowledge, the application can invoke the lookup operation as if it were just another normal Java method call (well, almost[2]). Static invocations like this one work well for developers writing them in statically typed programming languages such as Java and C++. As the example shows, the IDL definitions are mapped into the application programming language in a way that makes using them as natural as possible for the application developer.

Unfortunately, this example's simplicity is misleading. In reality, the developer has created a strong coupling between the application and the PhoneBook service, such that the application will stop working if the PhoneBook service ever changes. For example, the developer who wrote the original HumanResources module might decide later that separating an employee's name into distinct first, middle, and surnames would make certain operations easier and, thus, change the Employee struct to:

What you need to know to invoke a service is: what to send to it, where to send what you're sending, and what the service will do for you.

```
module HumanResources {
  struct EmployeeName {
    string first;
    string middle;
    string last;
  };
  struct Employee {
    EmployeeName name;
    // the rest same as before
```

As soon as the developer recompiles the PhoneBook service against this new IDL definition and redeploys it, all applications using the service that were compiled against the original IDL definition will break. The applications expect the service to return a sequence of Employee structures in which the name field is a string, not another structure. The IDL compiler generates different marshaling code for these two different Employee structures. The service's marshaling code will return the name field as a structure comprising first name, middle name, and last name as three strings, whereas the application's marshaling code will think the name field is a single string. Upon receiving the return value from the invocation, the application will attempt to demarshal the field following the first name field as if it were the address field, not the middle name field. The application will eventually fail because the service has sent back more data than was expected.

You could argue that the service developer shouldn't have changed the name field definition, and should — at a minimum — receive a stern warning about silently breaking all the applications using the service. That's true, but alternative approaches aren't all that appealing: Rather than changing the original Employee structure, the service developer could have

- Added a new structure named Employee2, and added a new operation named lookup2 to the PhoneBook interface (because IDL doesn't support overloading) to return a sequence of Employee2;
- Created a new PhoneBook2 interface that inherits the original PhoneBook interface and provides the new lookup2 operation that returns Employee2, which avoids changing the original PhoneBook interface contract; or
- Created a new module HumanResources2 that contains the modified Employee structure and PhoneBook interface.

As you can see, the problem comes down to versioning. Static systems generally do not handle versioning very easily: the developer usually has to build it into the code explicitly, by tacking version numbers onto the names of modules, interfaces, or structures. If you've ever maintained a system in which seemingly arbitrary version numbers are part of the names of types and methods, you already know how confusing it can be.

Dynamic Invocation

Unlike static invocation applications, which have all necessary invocation information built into them, applications that use dynamic invocation discover all the invocation information they need at runtime. There are various ways to do this, but all revolve around some form of metadata management. For example, systems such as Java and C# support *reflection*, which lets you dynamically load new classes, create instances of them, and invoke their methods, all on the fly and without any compile-time knowledge of those classes in your application.

Of course, reflection is nothing new, but Java's popularity and C#'s growing popularity have put it into more programmers' hands than ever before. Java's support for reflection has spawned some interesting applications, such as *mock object* implementations that avoid the need to manually write code to stub out dependent classes for unit tests. (For an example, see Peter Morgan's mock objects approach; www.iona.com/devcenter/mock_object/.)

Not surprisingly, middleware systems that provide some form of metadata management also tend to support dynamic invocation. For example, instead of a static invocation approach, we could use Corba's dynamic invocation interface (DII) to invoke the lookup operation in the `PhoneBook` example. (Normally, I would show the code for using the DII in this fashion, but given that invocations made through the DII generally require an order of magnitude more lines of code than their static counterparts,[3] I do not have enough space in this column.)

A general DII invocation requires the application to create a `Request` object, and then modify it to fill in the details of each argument, as well as any return value. Handling these details is straightforward if the arguments all have simple types, like strings, but it gets painful with more complex types such as the `Employee` structure from the `PhoneBook` example. Applications require facilities for creating or examining complex types such as these in terms of the simple types they're composed of. For example, a dynamic application would have to deal with the `Employee` structure in terms of the string fields that comprise it. In Corba, the Dynamic Any facility provides this functionality: when the service invocation returns, the application uses the same facilities to dynamically examine any return values.

Applications that employ dynamic invocation do not have service-contract compile-time knowledge, so they avoid versioning issues that can plague static applications. However, this benefit does not come for free. Dynamic invocation applications tend to be slower and larger than their static counterparts because they use general facilities to manipulate all types and thus cannot rely on the programming language or compiler for help. Furthermore, data values that dynamically create instances of complex types on the fly must come from somewhere other than the application, which cannot know all the semantics associated with the types and values. For this reason, dynamic invocation applications tend to be human-driven with GUIs that let the application query the user for field values and display values for user interpretation and consumption.

One issue with Corba's support for dynamic invocation is that its metadata facilities are not built in; they are provided in the form of an extra service called the interface repository (IFR). While this approach works, it requires Corba interface developers and objects to remember to populate the IFR with the metadata. If they forget to do so, other developers can't write dynamic invocations against those objects without finding the original IDL definitions and adding them to the IFR. In general, metadata that must be added after the fact essentially defeats the purpose of dynamic invocation facilities.

Where Does All This Leave Us?

Static invocation suffers from tight coupling and versioning problems, while dynamic invocation applications can be too big, slow, and complex. Fortunately, while these statements are generally true, they do not apply in all circumstances. In Corba's early days, there were strong disagreements between the static invocation and dynamic invocation camps, as if the approaches were mutually exclusive. Thankfully, neither camp prevailed, and Corba supports both approaches, as do other middleware systems that have come along since then, including Web services. As a result, many successful middleware applications deployed today use a combination of invocation approaches. Static invocations are easy to develop and, with proper attention to versioning issues, can be easy to maintain. Dynamic invocations are invaluable for applications such as debuggers, interceptors, and bridges between disparate middleware systems, which tend to have to deal with many different types on the fly. In other words, you need both approaches. It looks like the engineers got it right again.

References

1. S. Vinoski, "Service Discovery 101," *IEEE Internet Computing,* vol. 7, no. 1, Jan./Feb., 2003, pp. 69–71.
2. J. Waldo et al., *A Note on Distributed Computing*, tech. report SMLI TR-94-29, Sun Microsystems Laboratories, Mountain View, Calif., 1994; www.sunlabs.com/technical-reports/1994/abstract-29.html.
3. S. Vinoski and D. Schmidt, "Dynamic Corba, Part 1: The Dynamic Invocation Interface," *C/C++ Users J.*, July 2002, www.cuj.com/documents/s=7981/cujcexp 2007vinoski/.

Steve Vinoski is chief engineer of product innovation for IONA Technologies. He has been involved in middleware for 15 years. He is the coauthor of *Advanced Corba Programming with C++* (Addison Wesley Longman, 1999), and he has helped develop middleware standards for the OMG and W3C.