Architectural Implications of Classes

Classes have a number of architectural implications that are worth exploring in some detail.

On this page:

- Classes without Operations
- Classes with Operations
- Classes for Persistence

Classes without Operations

Classes that do not use inheritance and only have data members (whether self-referential or not) pose no architectural problems: they simply are values that are marshaled like any other value, such as a sequence, structure, or dictionary. Classes using derivation also pose no problems: if the receiver of a derived instance has knowledge of the derived type, it simply receives the derived type; otherwise, the instance is sliced to the most-derived type that is understood by the receiver. This makes class inheritance useful as a system is extended over time: you can create derived class without having to upgrade all parts of the system at once.

Classes with Operations

Classes with operations require additional thought. Here is an example: suppose that you are creating an Ice application. Also assume that the Slice definitions use quite a few classes with operations. You sell your clients and servers (both written in Java) and end up with thousands of deployed systems.

As time passes and requirements change, you notice a demand for clients written in C++.

For commercial reasons, you would like to leave the development of C++ clients to customers or a third party but, at this point, you discover a glitch: your application has lots of classes with operations along the following lines:

Slice

```
class ComplexThingForExpertsOnly {
    // Lots of arcane data members here...
    MysteriousThing mysteriousOperation(/* parameters */);
    ArcaneThing arcaneOperation(/* parameters */);
    ComplexThing complexOperation(/* parameters */);
    // etc...
};
```

It does not matter what exactly these operations do. (Presumably, you decided to off-load some of the processing for your application onto the client side for performance reasons.) Now that you would like other developers to write C++ clients, it turns out that your application will work only if these developers provide implementations of all the client-side operations and, moreover, if the semantics of these operations exactly match the semantics of your Java implementations. Depending on what these operations do, providing exact semantic equivalents in a different language may not be trivial, so you decide to supply the C++ implementations yourself.

But now, you discover another problem: the C++ clients need to be supported for a variety of operating systems that use a variety of different C++ compilers. Suddenly, your task has become quite daunting: you really need to supply implementations for all the combinations of operating systems and compiler versions that are used by clients. Given the different state of compliance with the ISO C++ standard of the various compilers, and the idiosyncrasies of different operating systems, you may find yourself facing a development task that is much larger than anticipated. And, of course, the same scenario will arise again should you need client implementations in yet another language.

The moral of this story is not that classes with operations should be avoided; they can provide significant performance gains and are not necessarily bad. But, keep in mind that, once you use classes with operations, you are, in effect, using client-side native code and, therefore, you can no longer enjoy the implementation transparencies that are provided by interfaces. This means that classes with operations should be used only if you can tightly control the deployment environment of clients. If not, you are better off using interfaces and classes without operations. That way, all the processing stays on the server and the contract between client and server is provided solely by the Slice definitions, not by the semantics of the additional client-side code that is required for classes with operations.

Classes for Persistence

Ice also provides a built-in persistence mechanism that allows you to store the state of a class in a database with very little implementation effort. To get access to these persistence features, you must define a Slice class whose members store the state of the class.

See Also

- Freeze