Interfaces, Operations, and Exceptions

The central focus of Slice is on defining interfaces, for example:

Slice

```
struct TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
};

interface Clock {
    TimeOfDay getTime();
    void setTime(TimeOfDay time);
};
```

This definition defines an interface type called `Clock`. The interface supports two operations: `getTime` and `setTime`. Clients access an object supporting the `Clock` interface by invoking an operation on the proxy for the object: to read the current time, the client invokes the `getTime` operation; to set the current time, the client invokes the `setTime` operation, passing an argument of type `TimeOfDay`.

Invoking an operation on a proxy instructs the Ice run time to send a message to the target object. The target object can be in another address space or can be collocated (in the same process) as the caller — the location of the target object is transparent to the client. If the target object is in another (possibly remote) address space, the Ice run time invokes the operation via a remote procedure call; if the target is collocated with the client, the Ice run time uses an ordinary function call instead, to avoid the overhead of marshaling.

You can think of an interface definition as the equivalent of the public part of a C++ class definition or as the equivalent of a Java interface, and of operation definitions as (virtual) member functions. Note that nothing but operation definitions are allowed to appear inside an interface definition. In particular, you cannot define a type, an exception, or a data member inside an interface. This does not mean that your object implementation cannot contain state — it can, but how that state is implemented (in the form of data members or otherwise) is hidden from the client and, therefore, need not appear in the object's interface definition.

An Ice object has exactly one (most derived) Slice interface type (or class type). Of course, you can create multiple Ice objects that have the same type; to draw the analogy with C++, a Slice interface corresponds to a C++ class definition, whereas an Ice object corresponds to a C++ class instance (but Ice objects can be implemented in multiple different address spaces).

Ice also provides multiple interfaces via a feature called *facets*.

A Slice interface defines the smallest grain of distribution in Ice: each Ice object has a unique identity (encapsulated in its proxy) that distinguishes it from all other Ice objects; for communication to take place, you must invoke operations on an object's proxy. There is no other notion of an addressable entity in Ice. You cannot, for example, instantiate a Slice structure and have clients manipulate that structure remotely. To make the structure accessible, you must create an interface that allows clients to access the structure.

The partition of an application into interfaces therefore has profound influence on the overall architecture. Distribution boundaries must follow interface (or class) boundaries; you can spread the implementation of interfaces over multiple address spaces (and you can implement multiple interfaces in the same address space), but you cannot implement parts of interfaces in different address spaces.

Topics

See Also

- Classes
- Facets and Versioning