Pass-by-Value Versus Pass-by-Reference

As we saw in Self-Referential Classes, classes naturally support pass-by-value semantics: passing a class transmits the data members of the class to the receiver. Any changes made to these data members by the receiver affect only the receiver's copy of the class; the data members of the sender's class are not affected by the changes made by the receiver.

In addition to passing a class by value, you can pass a class by reference. For example:

Slice

```
class TimeOfDay {
    short hour;
    short minute;
    short second;
    string format();
};

interface Example {
     TimeOfDay* get();  // Note: returns a proxy!
};
```

Note that the `get` operation returns a *proxy* to a `TimeOfDay` class and not a `TimeOfDay` instance itself. The semantics of this are as follows:

- When the client receives a `TimeOfDay` proxy from the `get` call, it holds a proxy that differs in no way from an ordinary proxy for an interface.
- The client can invoke operations via the proxy, but *cannot* access the data members. This is because proxies do not have the concept of data members, but represent interfaces: even though the `TimeOfDay` class has data members, only its *operations* can be accessed via a the proxy.

The net effect is that, in the preceding example, the server holds an instance of the `TimeOfDay` class. A proxy for that instance was passed to the client. The only thing the client can do with this proxy is to invoke the `format` operation. The implementation of that operation is provided by the server and, when the client invokes `format`, it sends an RPC message to the server just as it does when it invokes an operation on an interface. The implementation of the `format` operation is entirely up to the server. (Presumably, the server will use the data members of the `TimeOfDay` instance it holds to return a string containing the time to the client.)

The preceding example looks somewhat contrived for classes only. However, it makes perfect sense if classes implement interfaces: parts of your application can exchange class instances (and, therefore, state) by value, whereas other parts of the system can treat these instances as remote interfaces.

For example:

Slice

```
interface Time {
    string format();
    // ...
};

class TimeOfDay implements Time {
    short hour;
    short minute;
    short second;
};

interface I1 {
     TimeOfDay get();           // Pass by value
     void put(TimeOfDay time);  // Pass by value
};

interface I2 {
    Time* get();                // Pass by reference
};
```

In this example, clients dealing with interface `I1` are aware of the `TimeOfDay` class and pass it by value whereas clients dealing with interface `I2` deal only with the `Time` interface. However, the actual implementation of the `Time` interface in the server uses `TimeOfDay` instances.

Be careful when designing systems that use such mixed pass-by-value and pass-by-reference semantics. Unless you are clear about what parts of the system deal with the interface (pass by reference) aspects and the class (pass by value) aspects, you can end up with something that is more confusing than helpful.

A good example of putting this feature to use can be found in Freeze, which allows you to add classes to an existing interface to implement persistence.

See Also

- Self-Referential Classes
- Freeze