Passing Interfaces by Value

Consider the following definitions:

Slice

```
interface Time {
    idempotent TimeOfDay getTime();
    // ...
};

interface Record {
    void addTimeStamp(Time t); // Note: Time t, not Time* t
    // ...
};
```

Note that `addTimeStamp` accepts a parameter of type `Time`, not of type `Time*`. The question is, what does it mean to pass an interface *by value*? Obviously, at run time, we cannot pass an an actual interface to this operation because interfaces are abstract and cannot be instantiated. Neither can we pass a proxy to a `Time` object to `addTimeStamp` because a proxy cannot be passed where an interface is expected.

However, what we *can* pass to `addTimeStamp` is something that is not abstract and derives from the `Time` interface. For example, at run time, we could pass an instance of the `TimeOfDay` class we saw earlier. Because the `TimeOfDay` class derives from the `Time` interface, the class type is compatible with the formal parameter type `Time` and, at run time, what is sent over the wire to the server is the `TimeOfDay` class instance.

See Also

- Pass-by-Value Versus Pass-by-Reference