Versioning with Facets

We described several ways that a user might try to solve the versioning problem. A negative aspect of all these approaches is that they change the type system in intrusive ways. In turn, this forces unacceptable programming contortions on clients. Facets allow you to solve the versioning problem more elegantly because they do not change an existing type system but extend it instead. We saw this approach once already, where we added date information about a file to our file system application without disturbing any of the existing definitions.

In the most general sense, facets provide a mechanism for implementing multiple interfaces for a single object. The key point is that, to add a new interface to an object, none of the existing definitions have to be touched, so no compatibility issues can arise. More importantly, the decision as to which facet to use is made at run time instead of at compile time. In effect, facets implement a form of late binding and, therefore, are coupled to the type system more loosely than any of the previous approaches.

Used judiciously, facets can handle versioning requirements more elegantly than other mechanisms. Apart from the straight extension of an interface as shown earlier, facets can also be used for more complex changes. For example, if you need to change the parameters of an operation or modify the fields of a structure, you can create a new facet with operations that operate on the changed data types. Quite often, the implementation of a version 2 facet in the server can even re-use much of the version 1 functionality, by delegating some version 2 operations to a version 1 implementation.

On this page:

Facet Selection

Given that we have decided to extend an application with facets, we have to deal with the question of how clients select the correct facet. The answer typically involves an explicit selection of a facet sometime during client start-up. For example, in our file system application, clients always begin their interactions with the file system by creating a proxy to the root directory. Let us assume that our versioning requirements have led to version 1 and version 2 definitions of directories as follows:

Slice

```
module Filesystem { // Original version
    // ...

    interface Directory extends Node {
        idempotent NodeSeq list();
        // ...
    };
};

module FilesystemV2 {
    // ...

    enum NodeType { Directory, File };

    class NodeDetails {
        NodeType type;
        string name;
        DateTime createdTime;
        DateTime accessedTime;
        DateTime modifiedTime;
        // ...
    };

    interface Directory extends Filesystem::Node {
       idempotent NodeDetailsSeq list();
       // ...
    };
};
```

In this case, the semantics of the `list` operation have changed in version 2. A version 1 client uses the following code to obtain a proxy to the root directory:

C++

```
// Create a proxy for the root directory
//
Ice::ObjectPrx base = communicator()->stringToProxy("RootDir:default -p 10000");
if (!base)
    throw "Could not create proxy";

// Down-cast the proxy to a Directory proxy
//
Filesystem::DirectoryPrx rootDir = Filesystem::DirectoryPrx::checkedCast(base);
if (!rootDir)
    throw "Invalid proxy";
```

For a version 2 client, the bootstrap code is almost identical — instead of down-casting to `Filesystem::Directory`, the client selects the "V2" facet during the down-cast to the type `FilesystemV2::Directory`:

C++

```
// Create a proxy for the root directory
//
Ice::ObjectPrx base = communicator()->stringToProxy("RootDir:default -p 10000");
if (!base)
    throw "Could not create proxy";

// Down-cast the proxy to a V2 Directory proxy
//
FilesystemV2::DirectoryPrx rootDir = FilesystemV2::DirectoryPrx::checkedCast(base, "V2");
if (!rootDir)
    throw "Invalid proxy";
```

Of course, we can also create a client that can deal with both version 1 and version 2 directories: if the down-cast to version 2 fails, the client is dealing with a version 1 server and can adjust its behavior accordingly.

Behavioral Versioning

On occasion, versioning requires changes in behavior that are not manifest in the interface of the system. For example, we may have an operation that performs some work, such as:

Slice

```
interface Foo {
    void doSomething();
};
```

The same operation on the same interface exists in both versions, but the *behavior* of `doSomething` in version 2 differs from that in version 1. The question is, how do we best deal with such behavioral changes?

Of course, one option is to simply create a version 2 facet and to carry that facet alongside the original version 1 facet. For example:

Slice

```
module V2 {

    interface Foo {    // V2 facet
        void doSomething();
    };
};
```

This works fine, as far as it goes: a version 2 client asks for the "V2" facet and then calls `doSomething` to get the desired effect. Depending on your circumstances, this approach may be entirely reasonable. However, if there are such behavioral changes on several interfaces, the approach leads to a more complex type system because it duplicates each interface with such a change.

A better alternative can be to create two facets of the same type, but have the implementation of those facets differ. With this approach, both facets are of type `::Foo::doSomething`. However, the implementation of `doSomething` checks which facet was used to invoke the request and adjusts its behavior accordingly:

C++

```
void
FooI::doSomething(const Ice::Current& c)
{
    if (c.facet == "V2") {
        // Provide version 2 behavior...
    } else {
        // Provide version 1 behavior...
    }
}
```

This approach avoids creating separate types for the different behaviors, but has the disadvantage that version 1 and version 2 objects are no longer distinguishable to the type system. This can matter if, for example, an operation accepts a `Foo` proxy as a parameter. Let us assume that we also have an interface `FooProcessor` as follows:

Slice

```
interface FooProcessor {
    void processFoo(Foo* w);
};
```

If `FooProcessor` also exists as a version 1 and version 2 facet, we must deal with the question of what should happen if a version 1 `Foo` proxy is passed to a version 2 `processFoo` operation because, at the type level, there is nothing to prevent this from happening.

You have two options for dealing with this situation:

- Define working semantics for mixed-version invocations. In this case, you must come up with sensible system behavior for all possible combinations of versions.
- If some of the combinations are disallowed (such as passing a version 1 `Foo` proxy to a version 2 `processFoo` operation), you can detect the version mismatch in the server by looking at the `Current::facet` member and throwing an exception to indicate a version mismatch. Simultaneously, write your clients to ensure they only pass a permissible version to `processFoo`. Clients can ensure this by checking the facet name of a proxy before passing it to `processFoo` and, if there is a version mismatch, changing either the `Foo` proxy or the `FooProcessor` proxy to a matching facet:

C++

```
FooPrx foo = ...;               // Get a Foo...
FooProcessorPrx fooP = ...;     // Get a FooProcessor...

string fooFacet = foo->ice_getFacet();
string fooPFacet = fooP->ice_getFacet();
if (fooFacet != fooPFacet) {
    if (fooPFacet == "V2") {
        error("Cannot pass a V1 Foo to a V2 FooProcessor");
    } else {
        // Upgrade FooProcessor from V1 to V2
        fooP = FooProcessorPrx::checkedCast(fooP, "V2");
        if (!fooP) {
            error("FooProcessor does not have a V2 facet");
        } else {
            fooP->processFoo(foo);
        }
    }
}
```

Facets Design Considerations

Facets allow you to add versioning to a system, but they are merely a mechanism, not a solution. You still have to make a decision as to how to version something. For example, eventually you may want to deprecate a previous version's behavior; at that point, you must make a decision how to handle requests for the deprecated version. For behavioral changes, you have to decide whether to use separate interfaces or use facets with the same interface. And, of course, you must have compatibility rules to determine what should happen if, for example, a version 1 object is passed to an operation that implements version 2 behavior. In other words, facets cannot do your thinking for you and are no panacea for the versioning problem.

The biggest advantage of facets is also the biggest drawback: facets delay the decision about the types that are used and their behavior until run time. While this provides a lot of flexibility, it is significantly less type-safe than having explicit types that can be statically checked at compile time: if you have a problem relating to incorrect facet selection, the problem will be visible only at run time and, moreover, will be visible only if you actually execute the code that contains the problem, and execute it with just the right data.

Another danger of facets is the opportunity for abuse. As an extreme example, here is an interface that provides an arbitrary collection of objects of arbitrary type:

Slice

```
interface Collection {};
```

Even though this interface is empty, it can provide access to an unlimited number of objects of arbitrary type in the form of facets. While this example is extreme, it illustrates the design tension that is created by facets: you must decide, for a given versioning problem, how and at what point of the type hierarchy to split off a facet that deals with the changed functionality. The temptation may be to "simply add another facet" and be done with it. However, if you do that, your objects are in danger of being nothing more than loose conglomerates of facets without rhyme or reason, and with little visibility of their relationships in the type system.

In object modeling terms, the relationship among facets is weaker than an *is-a* relationship (because facets are often not type-compatible among each other). On the other hand, the relationship among facets is stronger than a *has-a* relationship (because all facets of an Ice object share the same object identity).

It is probably best to treat the relationship of a facet to its Ice object with the same respect as an inheritance relationship: if you were omniscient and could have designed your system for all current and future versions simultaneously, many of the operations that end up on separate facets would probably have been in the same interface instead. In other words, adding a facet to an Ice object most often implies that the facet has an *is-partly-a* relationship with its Ice object. In particular, if you think about the life cycle of an Ice object and find that, when an Ice object is deleted, all its facets must be deleted, this is a strong indication of a correct design. On the other hand, if you find that, at various times during an Ice object's life cycle, it has a varying number of facets of varying types, that is a good indication that you are using facets incorrectly.

Ultimately, the decision comes down to deciding whether the trade-off of static type safety versus dynamic type safety is worth the convenience and backward compatibility. The answer depends strongly on the design of your system and individual requirements, so we can only give broad advice here. Finally, there will be a point where no amount of facet trickery will get past the point when "yet one more version will be the straw that breaks the camel's back." At that point, it is time to stop supporting older versions and to redesign the system.

See Also