The Versioning Problem

Once you have developed and deployed a distributed application, and once the application has been in use for some time, it is likely that you will want to make some changes to the application. For example, you may want to add new functionality to a later version of the application, or you may want to change some existing aspect of the application. Of course, ideally, such changes are accomplished without breaking already deployed software, that is, the changes should be backward compatible. Evolving an application in this way is generally known as *versioning*.

Versioning is an aspect that previous middleware technologies have addressed only poorly (if at all). One of the purposes of facets is to allow you to cleanly create new versions of an application without compromising compatibility with older, already deployed versions.

On this page:

Versioning by Addition

Suppose that we have deployed our file system application and want to add extra functionality to a new version. Specifically, let us assume that the original version (version 1) only provides the basic functionality to use files, but does not provide extra information, such as the modification date or the file size. The question is then, how can we upgrade the existing application with this new functionality? Here is a small excerpt of the original (version 1) Slice definitions once more:

Slice

```
// Version 1

module Filesystem {
    // ...

    interface File extends Node {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    };
};
```

Your first attempt at upgrading the application might look as follows:

Slice

```
// Version 2

module Filesystem {
    // ...

    class DateTime extends TimeOfDay {  // New in version 2
        // ...
    };

    struct Times {                      // New in version 2
        DateTime createdDate;
        DateTime accessedDate;
        DateTime modifiedDate;
    };

    interface File extends Node {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;

        idempotent Times getTimes();    // New in version 2
    };
};
```

Note that the version 2 definition does not change anything that was present in version 1; instead, it only adds two new types and adds an operation to the `File` interface. Version 1 clients can continue to work with both version 1 and version 2 `File` objects because version 1 clients do not know about the `getTimes` operation and therefore will not call it; version 2 clients, on the other hand, can take advantage of the new functionality. The reason this works is that the Ice protocol invokes an operation by sending the operation name as a string on the wire (rather than using an ordinal number or hash value to identify the operation). Ice guarantees that any future version of the protocol will retain this behavior, so it is safe to add a new operation to an existing interface without recompiling all clients.

However, this approach contains a pitfall: the tacit assumption built into this approach is that no version 2 client will ever use a version 1 object. If the assumption is violated (that is, a version 2 client uses a version 1 object), the version 2 client will receive an `OperationNotExistException` when it invokes the new `getTimes` operation because that operation is supported only by version 2 objects.

Whether you can make this assumption depends on your application. In some cases, it may be possible to ensure that version 2 clients will never access a version 1 object, for example, by simultaneously upgrading all servers from version 1 to version 2, or by taking advantage of application-specific constraints that ensure that version 2 clients only contact version 2 objects. However, for some applications, doing this is impractical.

Note that you could write version 2 clients to catch and react to an `OperationNotExistException` when they invoke the `getTimes` operation: if the operation succeeds, the client is dealing with a version 2 object, and if the operation raises `OperationNotExistsException`, the client is dealing with a version 1 object. However, doing this can be rather intrusive to the code, loses static type safety, and is rather inelegant. (There is no other way to tell a version 1 object from a version 2 object because both versions have the same type ID.)

In general, versioning by addition makes sense when you need to add an operation or two to an interface, and you can be sure that version 2 clients do not access version 1 objects. Otherwise, other approaches are needed.

Versioning by Derivation

Given the limitations of the preceding approach, you may decide instead to upgrade the application as follows:

Slice

```
module Filesystem {     // Version 1
    // ...

    interface File extends Node {
        idempotent Lines read();
        idempotent void write(Lines text) throws GenericError;
    };
};

module FilesystemV2 {   // New in version 2
    // ...

    class DateTime extends TimeOfDay {
        // ...
    };

    struct Times {
        DateTime createdDate;
        DateTime accessedDate;
        DateTime modifiedDate;
    };

    interface File extends Filesystem::File {
        idempotent Times getTimes();
    };
};
```

The idea is to present the new functionality in an interface that is derived from the version 1 interface. The version 1 types are unchanged and the new functionality is presented via new types that are backward compatible: a version 2 `File` object can be passed where a version 1 `File` object is expected because `FilesystemV2::File` is derived from `Filesystem::File`. Even better, if a version 2 component of the system receives a proxy of formal type `Filesystem::File`, it can determine at run time whether the actual run-time type is `FilesystemV2::File` by attempting a down-cast: if the down-cast succeeds, it is dealing with a version 2 object; if the down-cast fails, it is dealing with a version 1 object. This is essentially the same as versioning by addition, but it is cleaner as far as the type system is concerned because the two different versions can be distinguished via their type IDs.

At this point, you may think that versioning by derivation solves the problem elegantly. Unfortunately, the truth turns out to be a little harsher:

- As the system evolves further, and new versions are added, each new version adds a level of derivation to the inheritance tree. After a few versions, particularly if your application also uses inheritance for its own purposes, the resulting inheritance graph very quickly turns into a complex mess. (This becomes most obvious if the application uses multiple inheritance — after a few versioning steps, the resulting inheritance hierarchy is usually so complex that it exceeds the ability of humans to comprehend it.)
- Real-life versioning requirements are not as simple as adding a new operation to an object. Frequently, versioning requires changes such as adding a field to a structure, adding a parameter to an operation, changing the type of a field or a parameter, renaming an operation, or adding a new exception to an operation. However, versioning by derivation (and versioning by addition) can handle none of these changes.
- Quite often, functionality that is present in an earlier version needs to be removed for a later version (for example, because the older functionality has been supplanted by a different mechanism or turned out to be inappropriate). However, there is no way to *remove* functionality through versioning by derivation. The best you can do is to re-implement a base operation in the derived implementation of an interface and throw an exception. However, the deprecated operation may not have an exception specification, or if it does, the exception specification may not include a suitable exception. And, of course, doing this perverts the type system: after all, if an interface has an operation that throws an exception whenever the operation is invoked, why does the operation exist in the first place?

There are other, more subtle reasons why versioning by derivation is unsuitable in real-life situations. Suffice it to say here that experience has shown the idea to be unworkable: projects that have tried to use this technique for anything but the most trivial versioning requirements have inevitably failed.

Explicit Versioning

Yet another twist on the versioning theme is to explicitly version everything, for example:

Slice

```
module Filesystem {
    // ...

    interface FileV1 extends NodeV1 {
        idempotent LinesV1 read();
        idempotent void write(LinesV1 text) throws GenericErrorV1;
    };

    class DateTimeV2 extends TimeOfDayV2 {
        // ...
    };

    struct TimesV2 {
        DateTimeV2 createdDate;
        DateTimeV2 accessedDate;
        DateTimeV2 modifiedDate;
    };

    interface FileV2 extends NodeV2 {
        idempotent LinesV2 read();
        idempotent void write(LinesV2 text) throws GenericErrorV2;
        idempotent TimesV2 getTimes();
    };
};
```

In essence, this approach creates as many separate definitions of each data type, interface, and operation as there are versions. It is easy to see that this approach does not work very well:

- Because, at the time version 1 is produced, it is unknown what might need to change for version 2 and later versions, *everything* has to be tagged with a version number. This very quickly leads to an incomprehensible type system.
- Because every version uses its own set of separate types, there is no type compatibility. For example, a version 2 type cannot be passed where a version 1 type is expected without explicit copying.
- Client code must be written to explicitly deal with each separate version. This pollutes the source code at all points where a remote call is made or a Slice data type is passed; the resulting code quickly becomes incomprehensible.

Other approaches, such as placing the definitions for each version into a separate module (that is, versioning the enclosing module instead of each individual type) do little to mitigate these problems; the type incompatibility issues and the need to explicitly deal with versioning remain.

See Also

- Slice for a Simple File System