# Connections

ZeroC's Newsletter for the Ice Community

*Issue Number 8, November 2005*


## At a Loose End…

Over the past few weeks, an interesting discussion has been taking place on Steve Vinoski's and Ted Neward's blogs. Much of the discussion was about whether CORBA has failed and if web services or REST are any better. Predictably, the discussion didn't progress very far before the words "loose coupling" were mentioned.

Loose coupling has become the Holy Grail of distributed computing in the past few years. Why is this? The answer, in a nutshell, is *versioning*: in real life, applications do not hold still; instead, they need to evolve, provide new functionality, change existing functionality, and remove old functionality. As people using CORBA found out, evolving an application is difficult. It is especially difficult if different versions of an application must be able to coexist and interoperate. And providing such backward compatibility is essential because of the difficulty of simultaneously upgrading all deployed clients and servers.

First of all, let me state that the main culprit for versioning problems is naïve system design. *Of course* it is difficult to evolve an application if all types and client–server interactions are hard-wired without any negotiation steps (for example, to acquire a bootstrap object that supports the desired version). A few well-chosen interfaces that allow clients to negotiate a version with a server and to acquire objects that are at the desired version can go a long way toward mitigating the problem.

But, putting design flaws aside, what people in the CORBA world kept stumbling over is the inability to change types or interfaces because that invalidates the on-the-wire contract between client and server, and so loses backward compatibility. WS and REST proponents immediately latched onto this point, and we ended up with XML as an encoding on the wire. It is XML that is supposed to provide the sought-after loose coupling and allow applications to evolve more easily. ("The receiver can always ignore the bits it doesn't understand.")

On closer examination, this argument turns out to be severely flawed. (For some of the reasons, check the abovementioned discussions as well as this issue's article on facets.) In fact, using WS or REST, clients and servers are as tightly coupled as ever. The problem is in the nature of the beast: without an a-priori agreement, communication between clients and servers is impossible. The agreement goes far beyond mere issues of representation: whether the data is encoded as binary or as XML is irrelevant because the interesting problems raised by versioning are at the semantic level, not the syntactic one.

The situation is rather ironic: not only does XML fail to provide the loose coupling that we so dearly want, but it also fails to provide it at a truly stupefying cost in bandwidth and CPU cycles. And, to add insult to injury, WSDL, which is usually used to specify interfaces for WS, is in direct conflict with loose coupling: it nails down specific types just as much as any other interface definition language. To actually get loose coupling, the developer has to abandon WSDL and explicitly parse the incoming XML. But that burdens the application with the need to perform type checking at run time that, otherwise, would be performed by the middleware at compile time.

Which brings me to facets… Facets are an elegant and simple mechanism that allows Ice applications to evolve gracefully, without a need to weaken or otherwise interfere with an application's type system, without a need to compromise static type safety, and without a need to sacrifice bandwidth and CPU cycles on the altar of loose coupling. I suggest you give facets a closer look—when it comes to evolving your application, they won't leave you at a loose end.

Michi Henning
Chief Scientist

## Issue Features

### FreezeScript

Matthew Newhook continues his series of articles by showing you how to add user profiles to the chat application, as well as how to use FreezeScript to migrate database contents to work with a new version of the application.

### Can a Leopard Change its Spots?

In this article, Michi Henning explains how you can version your Ice application with the help of facets.

## Contents

# FreezeScript

*Matthew Newhook, Senior Software Engineer*

## Introduction

This article, we add support for personal profile information to the user object, and demonstrate how to use FreezeScript to migrate a database after changing a Slice definition.

## Personal Profile

Most online chat systems have a profile section where a user can enter their personal information. Let's add the following information to the user object:

- First name and last name
- Country/Region
- Province
- City
- Date of birth
- Marital status
- Gender
- Occupation
- Personal information
- Email address
- Home page

At this point, we need to choose a representation for this information. There are two choices: individual variables for each field or an un-typed string representation, such as a series of string–string pairs. Why would we consider an un-typed representation? The reason is versioning. Suppose the user profile for the deployed application looks something like this:

```
// Slice
struct UserProfile
{
    string lastName;
    string firstName;
    // ...
};
```

In a later version of the application, you want to add a new field, `zipCode`:

```
// Slice
struct UserProfile
{
    string lastName;
    string firstName;
    string zipCode;
    // ...
};
```

Doing this forces you to redeploy the new clients because the old clients do not expect the `zipCode` to be present in the structure. To avoid having to redeploy all clients, an alternative that is often suggested is to use an un-typed representation, such as a collection of key–value pairs. For example:

```
// Slice
dictionary<string, string> ProfileDetails;
```

In your code, you use the name of the variable as the key to store the value of the variable:

```
// C++
ProfileDetails pd;
pd["lastName"] = "Newhook";
```

Adding the `zipCode` field now does not change the type system because the zip code becomes an additional entry in the dictionary, so all the existing clients still work. You may think that this neatly solves your versioning problem but, unfortunately, reality is rarely that cooperative. It is true that old clients continue to work, but only because they can ignore the `zipCode` field; but, in many cases, old clients do have to know how to deal with new data and cannot just ignore it. As an example, consider a Boolean field `displayEmailAddress`. This field is used by the application to determine whether to show or hide the email address of users. For this to work, all the old clients must be updated (at least if you want them to indeed respect people's privacy).

The dictionary approach also has another drawback: we have no type information for the lookup values—they are all strings, regardless of their real type. Of course, we can encode a Boolean value as `true` or `false`, which is fine until someone mistakenly uses `0` or `1`, or `yes` or `no`. If this happens, it is highly likely that the code that looks up the value will parse the value incorrectly and then do the wrong thing. This loss of static type safety is the biggest drawback of the key–value approach to versioning and you should avoid using it. (Ice provides a better way to deal with versioning in the form of facets, which are the topic of Michi's article in this issue.)

So, for the moment, we'll ignore the versioning issue and look at the Slice definitions to support user profiles:

```
// Slice
enum ProfileGender
{
    GenderNotSet,
    GenderUndisclosed,
    GenderMale,
    GenderFemale
};

enum ProfileMaritalStatus
{
    MaritalStatusNotSet,
    MaritalStatusUndisclosed,
    MaritalStatusMarried,
    MaritalStatusSingle
};

class User
{
    // ...
    string lastName;
    string firstName;
    string countryRegion;
    string province;
    string city;
    int dobDay;
    int dobMonth;
    int dobYear;
    bool displayAge;
    ProfileMaritalStatus maritalStatus;
    ProfileGender gender;
    string occupation;
    string personalInfo;
    string emailAddress;
    bool displayEmail;
};
```

How do we pass this information around? One option is to add methods like these:

```
// Slice
class User
{
    nonmutating string getLastName();
    nonmutating string getFirstName();
    nonmutating string getCountryRegion();
    // ...
};
```

Consider a possible use case for these methods—a web form that users edit to update their details. With a separate operation to retrieve the value of each field, the code will need to make fifteen separate remote invocations to populate the form, which is extremely inefficient. It is far better to make a single invocation and get all of the information at once. This suggests an API such as:

```
// Slice
struct UserProfile
{
    string lastName;
    string firstName;
    string countryRegion;
    string province;
    string city;
    int dobDay;
    int dobMonth;
    int dobYear;
    bool displayAge;
    ProfileMaritalStatus maritalStatus;
    ProfileGender gender;
    string occupation;
    string personalInfo;
    string emailAddress;
    bool displayEmail;
};

class User
{
    // ...
    nonmutating UserProfile getProfile();

    UserProfile profile;
};
```

Once the user has edited the profile details, the client needs update the stored profile with the new values. Again, it is easiest to have a single operation that accepts all the details, as follows:

```
// Slice
class User
{
    // ...
    idempotent void setProfile(
        UserProfile profile);
};
```

The implementation of this is very simple. We add to `UserI.java`:

```
// Java
synchronized public UserProfile
getProfile(Ice.Current current)
{
    return profile;
}

synchronized public void
setProfile(UserProfile p, Ice.Current current)
{
    profile = p;
}

public
UserI(UserManagerI manager, String id, String pw)
{
    // ...
    profile = new UserProfile();
    profile.maritalStatus =
        ProfileMaritalStatus.MaritalStatusNotSet;
    profile.gender = ProfileGender.GenderNotSet;
}
```

Note that, if you were to add methods that mutate the contents of the member variable `profile`, you must ensure that the marshaling of the `UserProfile` data in `getProfile` is thread-safe. (See Mark Spruiell's article Thread-Safe Marshaling in Issue 2 of Connections for a detailed discussion of this topic.)

After changing the user class definition, you need to wipe the Freeze databases. The reason is that, for efficiency reasons, Freeze databases contain a binary representation of the data. If the class definition changes, attempts to read the old data will fail with a `MarshalException` or give unexpected results. As an example, consider the following:

```
// Slice
struct Foo
{
    int x;
    int y;
};
```

The generated C++ marshaling code for this looks something like:

```
void
Foo::__write(::IceInternal::BasicStream* __os)
const
{
    __os->write(x);
    __os->write(y);
}

void
Foo::__read(::IceInternal::BasicStream* __is)
{
    __is->read(x);
    __is->read(y);
}
```

Freeze stores data in the same format as it is marshaled, so the database entry for an instance of this structure contains four bytes containing the value of `x`, followed by four bytes containing the value of `y` (in little-endian byte order). Suppose we now change the structure to:

```
// Slice
struct Foo
{
    int y;
    int x;
};
```

The generated code for this is:

```
void
Foo::__write(::IceInternal::BasicStream* __os)
const
{
    __os->write(y);
    __os->write(x);
}
```

```
void
Foo::__read(::IceInternal::BasicStream* __is)
{
    __is->read(y);
    __is->read(x);
}
```

If you were to use the generated code for the new definition to read the contents of a database written using the old definition, the values of `x` and `y` would be swapped!

## FreezeScript

Fortunately, Ice provides a solution to this problem without requiring us to delete the existing databases: FreezeScript.

Using the `transformdb` tool, you can migrate a database to work with updated Slice definitions. By comparing the old Slice definitions with the new Slice definitions using a set of user defined rules written in XML, `transformdb` makes whatever changes are necessary to the data contained within the database to support the new Slice types. In the above example, the FreezeScript rules would swap the values of `x` and `y`.

`transformdb` supports two modes of transformation: default transformation and custom transformation. Let's first look at what a default transformation does to our user objects. (The discussion that follows assumes that you have both the code for this issue and the code for Issue 7 installed.) First, clean the issue 7 demo databases:

```
$ cd demo.7
$ rm -rf db/node/* db/registry/*
```

Then run the issue 7 demo, create a user called "`foo`" with password "`bar`", and shut down the server:

```
$ admin
==> add foo bar
==> quit
$ icepackadmin --Ice.Config=config.icepack  -e
shutdown
```

Next, copy the databases into the issue 8 demo directory:

```
$ cd ../demo.8
$ cp -r ../demo.7/db .
```

This creates the situation we just discussed, namely, that the new version of the application ends up with a database written by the previous version of the application.

The FreezeScript tool `dumpdb` allows you to inspect the contents of a Freeze database if you have its Slice definitions available. We can use `dumpdb` to show the contents of the user manager database as follows:

```
$ dumpdb -e -I/opt/Ice/slice -I../demo.7 --load
../demo.7/User.ice db/node/servers/UserManager/
dbs/UserManager user
```

```
Key: struct ::Ice::Identity
{
  name = 'c0:a8:1:65:c502b70:102e7965480:-7fff'
  category = 'user'
}
Value: struct ::Freeze::ObjectRecord
{
  servant = class ::Chat::User (object #0)
  {
    userId = 'foo'
    password = 'bar'
    callback = ::Chat::InvitationCallback*(nil)
  }
  stats = struct ::Freeze::Statistics
  {
    creationTime = long(1111986952343)
    lastSaveTime = long(9266)
    avgSaveTime = long(9266)
  }
}
```

Cool huh? This shows the user you previously created. Now you can try this with the issue 8 demo `User.ice`:

```
$ dumpdb -e -I/opt/Ice/slice -I. --load User.ice
db/node/servers/UserManager/dbs/UserManager user
c:\Ice-2.1.0-VC60\bin\dumpdb.exe: ../../include\
Ice/BasicStream.h:104: Ice::UnmarshalOutOfBoundsE
xception:
protocol error: out of bounds during unmarshaling
```

As predicted, this does not work: the user object has changed and now contains a `UserProfile` data member. When the run time tries to read the data for the profile, it encounters the end of the stream and raises this exception. Now we'll try the default transformation:

```
$ transformdb -e --include-old /opt/Ice/slice
--include-old ../demo.7 --old ../demo.7/User.
ice --include-new /opt/Ice/slice --include-new .
--new User.ice db/node/servers/UserManager/dbs/
UserManager user newdb
```

This transforms the database and puts the result in a directory called "`newdb`". Let's dump the content of this new database:

```
$ dumpdb -e --load User.ice newdb user
Key: struct ::Ice::Identity
{
  name = 'c0:a8:1:65:c502b70:102e7965480:-7fff'
  category = 'user'
}
Value: struct ::Freeze::ObjectRecord
{
  servant = class ::Chat::User (object #0)
  {
    userId = 'foo'
    password = 'bar'
    callback = ::Chat::InvitationCallback*(nil)
    profile = struct ::Chat::UserProfile
    {
      lastName = ''
      firstName = ''
      countryRegion = ''
      province = ''
      city = ''
      dobDay = int(0)
      dobMonth = int(0)
      dobYear = int(0)
      displayAge = bool(false)
      maritalStatus = ::Chat::ProfileMaritalStatus
(MaritalStatusMarried)
      gender = ::Chat::ProfileGender(GenderMale)
      occupation = ''
      personalInfo = ''
      emailAddress = ''
      displayEmail = bool(false)
    }
  }
  stats = struct ::Freeze::Statistics
  {
    creationTime = long(1111986952343)
    lastSaveTime = long(9266)
    avgSaveTime = long(9266)
  }
}
```

Amazing, almost like magic! Unfortunately, there is one glitch: for the `maritalStatus` and `gender` fields, the values provided by the default transformation are wrong. To fix this, we need to re-transform the database with a custom migration descriptor. To assist in writing a custom migration descriptor, `transformdb` can generate a descriptor that does the default transformation. Since the default is mostly correct, we'll use it as the basis of our migration. The first step is to generate the default descriptor:

```
$ transformdb -e --include-old /opt/Ice/slice -
-include-old ../demo.7 --old ../demo.7/User.ice
--include-new /opt/Ice/slice --include-new . --new
User.ice -o migrate.xml
$ cat migrate.xml
<transformdb>
  <database key="::Ice::Identity"
   value="::Freeze::ObjectRecord">
    <record/>
  </database>

  <!-- class ::Chat::User -->
  <transform type="::Chat::User">
    <!-- NOTICE: profile has been added -->
  </transform>

  <!-- enum ::Chat::ProfileGender -->
  <init type="::Chat::ProfileGender"/>

  <!-- enum ::Chat::ProfileMaritalStatus -->
  <init type="::Chat::ProfileMaritalStatus"/>

  <!-- struct ::Chat::UserProfile -->
  <init type="::Chat::UserProfile"/>
</transformdb>
```

Now we need to change the rule for initializing `UserProfile`, as follows.

```
// migrate.xml
//...
<init type="::Chat::UserProfile">
  <set target="value.gender"
   value="::New::Chat::GenderNotSet"/>
  <set target="value.maritalStatus"
   value="::New::Chat::MaritalStatusNotSet"/>
</init>
//...
```

This rule says that, when initializing a `UserProfile`, different values should be used for the `gender` and `maritalStatus` fields, namely `::Chat::GenderNotSet` and `::Chat::MaritalStatusNotSet`. Let's try the transformation again with the custom rule:

```
$ rm newdb/*
$ transformdb -e --include-old /opt/Ice/slice -
-include-old ../demo.7 --old ../demo.old/User.
ice --include-new /opt/Ice/slice --include-new .
--new User.ice -f migrate.xml db/node/servers/
UserManager/dbs/UserManager user newdb
$ dumpdb -e -I/opt/Ice/slice -I. --load User.ice
newdb user
Key: struct ::Ice::Identity
{
  name = 'c0:a8:1:65:c502b70:102e7965480:-7fff'
  category = 'user'
}
Value: struct ::Freeze::ObjectRecord
{
  servant = class ::Chat::User (object #0)
  {
    userId = 'foo'
    password = 'bar'
    callback = ::Chat::InvitationCallback*(nil)
    profile = struct ::Chat::UserProfile
    {
      lastName = ''
      firstName = ''
      countryRegion = ''
      province = ''
      city = ''
      dobDay = int(0)
      dobMonth = int(0)
      dobYear = int(0)
      displayAge = bool(false)
      maritalStatus = ::Chat::ProfileMaritalStatus
(MaritalStatusNotSet)
      gender = ::Chat::ProfileGender(GenderNotSet)
      occupation = ''
      personalInfo = ''
      emailAddress = ''
      displayEmail = bool(false)
    }
  }
  stats = struct ::Freeze::Statistics
  {
    creationTime = long(1111986952343)
    lastSaveTime = long(9266)
    avgSaveTime = long(9266)
  }
}
```

Pure gold—our database is ready to rock with the new version of the application!

FreezeScript and `transformdb` are powerful and flexible tools, and this article only scratches the surface of what you can do. Future articles will demonstrate how to do more complex transformations. In the meantime, if you want to know more about FreezeScript, please consult the Ice manual or ask questions on our forums!

# Can a Leopard Change its Spots?

*Michi Henning, Chief Scientist*

## Introduction

In this issue's article on FreezeScript, Matthew touched on the issue of application *versioning*. Versioning refers to the evolution of an application over time to incorporate changed or new functionality. The challenge of versioning is to achieve it in a way that is compatible with already-deployed application components because it is often impossible to upgrade all clients and servers simultaneously. Or, to put it differently, we would like to make our applications change spots a little, depending on who is using it.

The kinds of changes that are required for versioning can range from the simple, such as adding a single new interface, to the quite complex, such as renaming an operation or interface, adding or removing a field from a structure, adding a new exception to an operation, or changing the inheritance structure of interfaces. In general, such changes modify the type system of the application and are not backward (or forward) compatible because they invalidate the on-the-wire contract between client and server.

A more interesting case of versioning (and one that is often ignored) is if, for a later version of an application, the behavior of one or more operations needs to change. In this case, the change does not affect the type system because all interfaces and types stay the same. However, the change may still be incompatible because other application components might rely on the old behavior and might break when presented with the new behavior.

## Conventional Approaches to Versioning

One of the most common approaches to versioning is to incorporate changes by relaxing the type system. The basic idea is to make interfaces and types more generic, so they can accommodate changes. For example, as discussed in Matthew's article, you can model parameters as name–value pairs:

```
// Slice
dictionary<string, string> ParamList;

interface Generic
{
    ParamList genericOp(ParamList pl);
};
```

With this approach, `genericOp` becomes like a Swiss army knife: it can accept an arbitrary number of parameters of arbitrary type, and return an arbitrary number of return values of arbitrary type. Unfortunately, this idea is severely flawed (as are variations of it). One big problem with the approach is that it loses static type safety: a client can easily send too many or too few parameters, or send parameters of the wrong type, but, as far as the compiler is concerned, everything is fine. In other words, the loose type system of this approach means that errors are not detected at compile time but at run time, and the error-detection code has to be provided by the application.

Another big problem with this approach is that all values are encoded as strings, even if the natural type of a value is something else, such as an integer or a structure. This means that the sender must encode values as strings, and the receiver must parse these strings to recover the values in their native type. This not only requires a lot of extra coding effort, but is also wasteful in bandwidth and CPU cycles. (It is far cheaper to send an integer as a binary 4-byte value than to encode it as a string, transmit the string, and decode the string again at the receiving end.) And, of course, the encoding and decoding machinery itself may not be perfect and add yet another potential source of errors.

These problems are indeed serious: errors are detected only if you happen to have test cases that expose them. But even for systems of moderate complexity, it is impossible to achieve complete test coverage, making it likely that errors will be detected only after the application is deployed. Ironically, the complexity of the loose typing approach makes it much more likely for errors to creep into the code, and you may well find that you need to create a new version of the application only to deal with a bug that would not be there had you not used loose typing in the first place.

But, even more seriously, the loose typing approach is flawed because the original idea is flawed. Versioning by loose typing relies on anticipating the need for versioning: the expectation is that the designers of the system will provide loose typing at the points where they expect future changes. Unfortunately, this rarely works in real life: the versioning requirements of an application are typically not known in advance, and designers are not prescient.

Another frequently-advocated approach to versioning is versioning by derivation. The basic idea is to extend an application by deriving new interfaces from old ones, and implementing the new functionality in the derived interfaces. For example, we might have a version 1 interface that provides some functionality:

```
// Slice
interface Thing
{
    void doSomething();
};
```

For version 2, we can add new functionality to a derived interface:

```
// Slice
interface ThingV2 extends Thing
{
    void doSomethingNew();
};
```

Initially, this looks attractive: version 2 clients attempt to down-cast a `Thing` proxy to `ThingV2` proxy; if the down-cast succeeds, they are dealing with a version 2 instance and, if the down-cast fails, they are dealing with a version 1 instance.

Unfortunately, versioning by derivation creates more problems than it solves. For one, once an application has undergone a few revisions, chances are that you will end up with an impenetrable mess of interface inheritance that is incomprehensible to mere mortals. Secondly, and more importantly, versioning by derivation is naïve because it assumes that all future versions of an application are fully backward compatible with all previous versions: versioning by derivation allows you to *add* new functionality to derived interfaces, but it does not allow you to *change* anything. However, real-life versioning requirements are rarely that simple. It is often necessary to add a field to a structure, add a new exception to an operation, or change the type of a parameter, but derivation cannot handle any of these things. Finally, versioning sometimes requires removing functionality that was present in an earlier version, but derivation cannot deal with that either. The best you can do is to throw an exception if a conceptually-removed operation is invoked. But this perverts the type system which, after all, states that the operation exists; unconditionally throwing an exception whenever the operation is invoked is the moral equivalent of having a machine with a button labeled "Don't Press Me"—why put the button there in the first place if no-one is supposed to press it?

Yet another approach to versioning is to explicitly version everything:

```
// Slice
sequence<string> SomeTypeV1;
dictionary<string, string) SomeOtherTypeV1;

interface OtherThingV1
{
    string getName();
};

interface ThingV1 extends OtherThingV1
{
    SomeTypeV1 op(SomeOtherTypeV1 param);
};

sequence<string> SomeTypeV2;
dictionary<string, double> SomeOtherTypeV2;

interface OtherThingV2
{
    string getName();
    double getSize();
};

interface ThingV2 extends OtherThingV2
{
    SomeTypeV2 op(SomeOtherTypeV2 param);
    void otherOp(SomeOtherTypeV2 param);
};
```

Even for this simple contrived example, it is clear that this approach is unspeakably clumsy. It is typically not known what might change at the time version 1 is created, so everything has to be tagged with a version label, even though it may never change. And because every version uses its own separate types, all type compatibility is lost: a version 2 type cannot be passed where a version 1 type is expected (even if it would be compatible) without explicit copying. Worst of all, client code must be written to explicitly deal with every version. In effect, the approach does not create backward compatible versions, but multiple unrelated versions, and application components have to know how to deal with all possible versions and how to convert between them. This quickly leads to incomprehensible (and untestable) code.

A variation of the preceding approach is to version at the module level:

```
// Slice
module V1
{
    sequence<string> SomeType;
    dictionary<string, string) SomeOtherType;

    interface OtherThing
    {
        string getName();
    };

    interface Thing extends OtherThing
    {
        SomeType op(SomeOtherType param);
    };
};

module V2
{
    sequence<string> SomeType;
    dictionary<string, string) SomeOtherType;

    interface OtherThing
    {
        string getName();
    };

    interface Thing extends OtherThing
    {
        SomeType op(SomeOtherType param);
    };
};
```

At first glance, this looks better than the previous attempt because it does not use mangled names that include a version number; instead, the enclosing module carries the version identifier. However, this really is no better than the previous approach because version 1 and version 2 types are completely unrelated, so there is no compatibility or type substitutability of any kind.

## Ice Facets

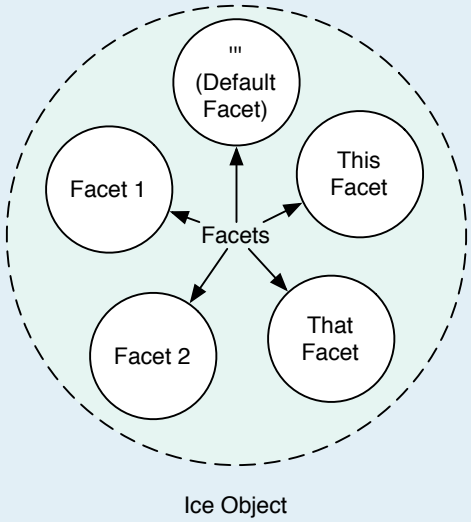
When versioning an application, we must trade off two conflicting desires:

- We want things to be statically type safe, so we would like to change the type system to match each new version.
- We want things to be backward compatible, so we would like to *not* change the type system to match each new version.

Resolving this tension is difficult and, as we saw in the preceding section, naïve approaches create more problems than they solve. To address this issue, Ice provides *facets*. Correctly used, facets allow you to have your cake and eat it too: you can use facets to version an application while retaining both static type safety and backward compatibility.

Facets allow an Ice object to present multiple interfaces to the world. Figure 1 shows a view of an Ice object that consists of five facets, each corresponding to an interface.

Figure 1: Ice Object with Five Facets




Each facet has a name, known as the *facet name*. The names of the facets of an Ice object must be unique within that object. Facet names are arbitrary strings that are assigned by the server. If a server does not explicitly use facets, each Ice object has a single facet, known as the *default facet*, whose name is the empty string. By default, operation invocations made by clients are directed to the default facet.

It is important to note that all facets of an Ice object share the same object identity. (The facet name is not part of the object identity.) If a client holds proxies to two different facets of the same Ice object and compares their identities, they compare equal.

Typically, the facets of an Ice object each will have a different interface (although it is legal for several facets of an Ice object to have the same interface). Normally, each facet is implemented by a separate servant. The Ice object adapter provides operations that allow a server to control the facets of an Ice object:

```
// Slice
namespace Ice
{
    dictionary<string, Object> FacetMap;

    local interface ObjectAdapter
    {
        Object* addFacet(Object servant,
            Identity id, string facet);
        Object* addFacetWithUUID(Object servant,
            string facet);
        Object removeFacet(Identity id,
            string facet);
        Object findFacet(Identity id,
            string facet);
        FacetMap findAllFacets(Identity id);
        FacetMap removeAllFacets(Identity id);
        // ...
    };
};
```

These operations have the same semantics as the corresponding "normal" operations to manipulate the Active Servant Map (ASM), but they also accept a facet name. (The "normal" operations use the default facet name so, in fact, `add(servant)` is equivalent to `addFacet(servant, "")`.) Here is a C++ code fragment that shows how a server can create an Ice object with two interfaces, `File` and `Stat`:

```
// C++
// Create a File instance.
//
Filesystem::FilePtr file = new FileI();

// Create a Stat instance.
//
FilesystemExtensions::StatPtr stat =
    new StatI(/* ...*/);

// Register the File instance as the default
facet.
//
Filesystem::FilePrx filePrx =
    myAdapter->addWithUUID(file);

// Register the Stat instance as a facet with
// the name "Stat".
//
myAdapter->addFacet(stat,
    filePrx->ice_getIdentity(), "Stat");
```

The code registers the `File` instance as the default facet, and adds the `Stat` instance with the facet name `Stat`, using the same identity as for the `File` instance. Even without seeing the corresponding Slice definitions, you can tell from the code what is going on here: the original version uses an interface called `File`, and the new version adds functionality by creating a `Stat` interface in a separate module called `FilesystemExtensions`. This illustrates an important advantage of facets: any type definitions that are relevant to a newer version are completely independent of the type definitions of previous versions, and can even be placed into an unrelated module. This not only avoids polluting the older Slice definitions when creating a newer version, but also avoids unnecessary recompilation. (Those parts of the system that only deal with the original version do not need to be recompiled.)

On the client side, the facet to which an operation invocation is sent is implicit in the proxy for the invocation. By default, operation invocations are directed to the default facet. Clients can

navigate among the facets of an Ice object using a `checkedCast` that specifies the facet name. For example:

```
// C++
// Get a File proxy.
//
Filesystem::FilePrx file = ...;

// Get the Stat facet.
//
FilesystemExtensions::StatPrx stat =
    FilesystemExtensions::StatPrx::checkedCast(
        file, "Stat");

// Go back from the Stat facet to the File facet.
//
Filesystem::FilePrx file2 =
    Filesystem::FilePrx::checkedCast(stat, "");
```

If an Ice object does not provide the specified facet, the `checkedCast` returns `null`. A client can find out which facet is targeted by a proxy by calling its `ice_getFacet` method.

Normally, if a client invokes on a proxy for an Ice object that only has the default facet, the call either succeeds or, if the Ice object no longer exists, raises an `ObjectNotExistException`. With facets, it is possible for the Ice object to exist, but for the specified facet to not exist. An invocation to such a non-existent facet raises `FacetNotExistException`. In other words, clients see `ObjectNotExistException` only if the Ice object for a request truly does not exist, that is, if the Ice object has no facets at all.

## Versioning Using Facets

Suppose we have deployed an application that contains the following interface:

```
// Slice
module Filesystem
{
       // Original version
       // ...
       interface Directory extends Node
       {
              nonmutating NodeSeq list();
              // ...
       };
};
```

The `list` operation returns a sequence of `Node` proxies in version 1. For version 2, we decide to change the `list` operation to return more detail for each node, to avoid having clients call `list` only to immediately make more calls to retrieve the details for each node:

```
// Slice
module FilesystemV2
{
    // ...
    enum NodeType { Directory, File };
    class NodeDetails
    {
        NodeType type;
        string name;
        DateTime createdTime;
        DateTime accessedTime;
        DateTime modifiedTime;
        // ...
    };
    interface Directory extends Filesystem::Node
    {
        nonmutating NodeDetailsSeq list();
        // ...
    };
};
```

Note that we now have two `Directory` interfaces, `Filesystem::Directory` for version 1, and `FileSystemV2` for version 2. For a version 1 client, nothing unusual happens at all. The client can construct a proxy and down-cast to `Filesystem::Directory` as always:

```
//
// Create a proxy for the root directory
//
Ice::ObjectPrx base = communicator()
    ->stringToProxy("RootDir:default -p 10000");

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

A version 2 client bootstraps itself in exactly the same way, except that it down-casts to the `V2` facet:

```
//
// Create a proxy for the root directory
//
Ice::ObjectPrx base = communicator()
    ->stringToProxy("RootDir:default -p 10000");

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

If the down-cast fails, the server does not provide the V2 facet and so must be a version 1 server. In that case, the client can either fall back on the version 1 interface or, once you no longer want to support version 1, print an error message.

Note that the two versions do not interfere with each other and do not make a mess of the type system: each version's Slice definitions are contained in stand-alone modules. Moreover, each client that understands the application up to a particular version number is completely oblivious of any later versions that may since have been added to the application, so later versions do not interfere with earlier ones at the source code level. Also, versioning using facets avoids having to mangle names with version numbers: the two `Directory` interfaces with their respective `list` operations co-exist without problems.

Earlier, I mentioned behavioral changes, in which only the behavior of an operation changes, but not its signature. For example, we may have an operation that, as a side effect, changes the state of an object. Something like:

```
// Slice
interface SomeObject
{
    void changeState();
    State getState();
};
```

A client calls `changeState` to affect a state change and can read the changed state back with `getState`. For version 2 of the application, we might decide that `changeState` has to work differently and produce a state change that differs in some way from version 1. However, this is not necessarily a backward-compatible change because version 1 clients might break if they end up calling the version 2 implementation of the operation and vice-versa. The problem though is that the `changeState` operation has exactly the same signature in both versions. We can deal with this in one of two ways:

- We can add separate facet servants for version 1 and version 2 to the Ice object, and implement the `changeState` operation accordingly in each servant. This is the same approach we illustrated with the preceding example.
- Alternatively, we can register the same servant as both a version 1 and version 2 facet, and check which facet is addressed by the client at run time.

With the latter approach, the client uses a version 1 or version 2 proxy as usual, and the server adjusts its implementation by looking at the `Current` object for the invocation:

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

The `Current` object contains the facet name of the target of the operation in the `facet` member, so the server can choose the correct behavior for each invocation.

## Limits of Versioning

Facets are a surprisingly powerful and flexible mechanism to address the versioning problem. In particular, using facets, you do not need to compromise static type safety by loosening the type system. Another advantage of facets is that they are both simple and non-intrusive: versions are shielded from each other, so the impact of versioning on the code base is limited to those parts of the code that actually need to deal with different versions; the remainder of the code is ignorant of versioning (and does not even require recompilation). This avoids the contortions that conventional approaches suffer because there is no need to interfere with the type system of an existing version when adding a new version. In addition, facets do not extract a performance penalty: the only run-time overheads are storing the facet name in each proxy and sending that name on the wire with each invocation, which are insignificant.

However, despite their elegance, facets do not solve all versioning problems. In particular, you need to worry about the semantics of versioning. For example, if a version 1 object is passed as a parameter to a version 2 operation, you must, in the application code, take explicit action to deal with the situation; the invocation must have appropriate semantics (or raise an exception), neither of which is free in terms of implementation effort. What makes the versioning problem hard (despite the simplicity of facets) is the semantic complexity that can arise from the interactions of types at different version levels with each other. Versioning is often limited by the complexity rising to the point where developers can no longer keep up with it.

There is also a point where it no longer makes sense to version an application. If you look at an application and find that, while each pair of adjacent versions is similar, the first and the final version might as well be considered completely different applications, it is a good sign that versioning has gone far enough. At that point, you should consider dropping compatibility with older versions and refactor the code to reduce complexity.

So, can a leopard change its spots? Yes, thanks to facets, it can. But it's still a leopard and no amount of versioning will turn it into a lion (or turn a payroll application into a first-person shooter). Facets cannot perform miracles but, used judiciously, they go a long way towards allowing you to evolve your application without having to replace every piece of deployed software at once.

## FAQ Corner

In each issue of our newsletter, we present a few frequently-asked questions about Ice. The questions and answers are taken from our support forum at http://www.zeroc.com/vbulletin/ and deal with specific problems that developers tend to encounter, and for which the answer may not be readily apparent from reading the documentation. We hope that you will find the hints and explanations in this section useful.

**Q:** How do I find out how much memory my Ice process really uses?

The simple answer to this question is to look at the resident set size using the task manager (under Windows) or the `ps` command (under Linux). Unfortunately, either way, you are likely to get misleading figures with no practical relevance.

To see why, consider how an operating system such as Windows or Linux allocates physical memory to a process. When the OS starts a process, it initially allocates only a small amount of physical memory to a process (at the very least, one page of memory that contains the instructions at which the process starts executing). As the process runs, it may access virtual memory locations for which the operating system has not allocated a physical page of memory as yet. If this happens, the memory management hardware raises an interrupt (known as a page fault) and transfers control to the OS kernel: in response to the page fault, the kernel suspends the current process, allocates a physical page of memory, initializes that memory with the appropriate text or data, and then resumes the process that caused the page fault. This is known as demand paging. The net effect of demand paging is that the OS does not need to allocate physical memory to a process until that memory is actually needed: if a process never accesses some of its virtual pages of memory, no physical memory is ever allocated for the corresponding pages of virtual memory.

In a demand-paged operating system, it can happen that a process incurs a page fault at a time when all physical memory pages are in use. When this happens, the kernel must choose a page of memory that it can take away from its current process so it can allocate it to the process that caused the page fault. Usually, the page is chosen using a least-recently-used page replacement strategy: the idea is to choose a page for eviction that is unlikely to be used in the near future. However, unless you have a machine that runs a number of large processes, it is quite possible that physical memory actually is never fully in use: with main memory often exceeding a gigabyte, it is entirely possible for all running processes on a machine to actually fit into main memory, so the kernel actually never evicts any physical memory page when servicing a page fault.

If you instrument a process for memory access (pretty much any kind of process will do), you will find that memory access follows the 80/20 rule: a process spends around 80% of its time accessing around 20% of its virtual memory. The 80/20 rule, together with a large main memory size, are responsible for the misleading figures reported by the task manager and the `ps` command: these tools show the total number of memory pages that are physically allocated to a process, even though most of these pages are touched only once during process start-up and will never be touched again (and could be given to another process without any performance penalty).

What we would like these tools to report instead is the working set size of a process, which is the amount of memory that is used frequently. This number is hard to define because it depends on the exact definition of "frequently". Intuitively, it is the number of pages of physical memory required by a process to continue running under its normal workload without incurring page faults. Or, in other words, the working set size is the "real" amount of memory that is used by a process while it is running. The difficulty of determining the working set size is the reason why it is not reported by tools such as the task manager and the `ps` command.

While there is no direct way to measure the working set size of a process, there is simple trick that you can use to find it, at least under Windows: Windows trims the physical memory pages given to a process to the bare bones when you minimize the window the process runs in. To try this out, take an Ice server and start it in its own command window, and access the server from a client running in another window. Finally, make sure that the client is idle for the time being, so it does not invoke more operations in the server. Now start the task manager and look at the size or your server. Most likely, the task manager will report quite a large figure (around 4MB). Of course, that number includes all the pages that are not really used by the server anymore. Now minimize the window the server is running in. You will see the process size drop dramatically, most likely to less than 100kB, because Windows frees up most of the memory pages that are allocated to the process at this point. Unfortunately, that new number is not the working set size either, because Windows is quite aggressive at freeing memory and ends up removing pages that, in normal operation, would be needed. To get around this, go back to the client and exercise the server again, but do it while the server's window is still minimized. You will see the process size jump back up again because Windows allocates physical memory pages to the server as it incurs page faults in response to the client activity. That new size gives you an accurate working set size and represents the "real" memory use for your server. (For a minimal server, that size is at around 600kB.)

Unfortunately, under Linux, there is no simple trick you could use to determine the working set size because there is no way to coerce Linux into artificially trimming the physical memory allocated to a process—the kernel will remove physical pages of memory from a process only once all physical memory is in use. You can force this to happen by writing a program that allocates an array of memory that is as large as the physical memory of

your machine. The program should write at least one byte to every page of the array. You will need super user privileges to run this program because it will most likely require a larger than normal memory limit (see `setrlimit(2)` or the `limit` command of your shell). Running this program causes most of the machine's physical memory to be allocated to it. (It would be polite to not do this on a multi-user machine—using up all available physical memory is a great way to bring a machine to its knees…) If you run the memory-waster program while your Ice client and server are running, per force, the kernel will trim the physical memory of your Ice processes. Now let the memory-waster program exit and use the still-running Ice client to exercise the server. When you now look at the resident set size of the server with `ps`, you will get a figure that accurately reflects the working set size.

Finally, before you go and give your machine a good work-out in this way, consider simply running your server under Windows and determining its working set size there. Chances are that the Linux figure will be within 20% of the Windows figure, so you can use the Windows working set size as an estimate of the Linux working set size.

**Q:** When should I use thread-per-connection instead of thread pool?

"Thread-per-connection" and "thread pool" are the names of the two concurrency models supported by Ice.

Thread pool is the default concurrency model because it is an appropriate choice for most applications. With the thread pool model, the Ice run time devotes one or more threads to servicing requests. The maximum size of a thread pool determines the number of concurrent requests a server is capable of dispatching. Although the default maximum size is one, this value can be increased to satisfy the requirements of your application. During periods of increased activity, the Ice run time will automatically add new threads to the pool, up to the configured maximum size. As activity decreases, idle threads are terminated to conserve resources. You can also specify a minimum size for the thread pool so that Ice will keep at least that many threads active for handling new requests.

As its name implies, the thread-per-connection model dedicates a new thread to each incoming and outgoing connection. This model was initially added to support the requirements of the Glacier2 router; the model makes it easier for the router to defend against certain types of attacks by hostile clients. But there are times when the thread-per-connection model is useful, and sometimes its use is mandatory. For example, since Java does not provide the necessary support for using SSL with a thread pool, the IceSSL plug-in for Java can only be employed by applications that use the thread-per-connection model.

The thread-per-connection model also comes in handy when you need to serialize requests from a client. For instance, suppose that a transaction processing server must ensure that requests are dispatched in the order they are received. If the thread-per-connection model is enabled, only one thread can dispatch the requests received on a connection, and therefore serialization is guaranteed (assuming the client is not sending requests on multiple connections).

This requirement could also be satisfied with the thread pool model, but only if you limit the maximum size of the thread pool to one or, alternatively, if you can guarantee that the client does not send requests from multiple threads and does not send oneway requests. The disadvantage of using a thread pool restricted to one thread is that it serializes requests from all clients, rather than just the requests from a single connection. If you use a larger thread pool, and the client sends requests from multiple threads or uses oneway requests, then the operating system's thread scheduling behavior in the server could cause requests to be dispatched out of order.

One distinct advantage of the thread-pool model over the thread-per-connection model is scalability. Since thread-per-connection creates a new thread for each connection, a program that establishes hundreds of connections also creates hundreds of threads. The use of active connection management to reap idle connections (and therefore the threads associated with them) can mitigate this somewhat, but thread-per-connection clearly does not scale as well as a thread pool. In fact, the ability to set the maximum size of a thread pool allows you to tune an Ice application to match the hardware capabilities of its host. On a multi-processor machine, for example, you may decide to limit the thread pool to the number of physical processors in order to minimize the overhead of thread context switches. No such control is available with thread-per-connection.

Earlier we mentioned a use case in which the serialized nature of requests when using thread-per-connection is an advantage, but it could just as easily be considered a disadvantage in a use case with different requirements. For example, the thread-per-connection model might be an inappropriate choice for a server with long-running operations when the client needs the ability to have several operations in progress simultaneously. There are ways to achieve the client's concurrency requirements while using the thread-per-connection model. One option is to design the client so that it forces new connections to be established, however this tightly couples the client with the server implementation. Another alternative is for the server to use asynchronous dispatch in order to avoid blocking the connection's thread, and use a work queue to execute the requests in separate threads. However, unless the server needs to track the order of requests, the thread pool model provides similar functionality with less effort.

In general, we recommend using the thread pool model, unless your application has very specific requirements that only thread-per-connection can satisfy and scalability is not a concern.