Interface Translation and Implementation Filtering

Mark A. Linton
Silicon Graphics
linton@sgi.com

Douglas Z. Pan
Stanford University
pan@panda.stanford.edu

Abstract

Separating interface from implementation in C++ requires a set of conventions for defining classes. Using an interface definition language, we can ensure that an interface does not contain any implementation details. To simplify the definition of separate interfaces, the translator that generates C++ class declarations should be flexible and convenient to use.

As part of the Fresco user interface system, we have developed an interface translator called Ix. In addition to generating C++ classes and stubs for distributed access, Ix can "filter" implementation code to automate as much of the code as possible. Filtering also gives the programmer more control over where and how the code is generated. We have built an initial implementation of Fresco using Ix, and our experience has been that using Ix has made programming with interfaces easier than using C++ directly.

1. Introduction

Defining an object in terms of its interface as distinct from its implementation is one of the basic tenets of object-oriented programming. C++ provides a mechanism for separating interfaces from implementation, but not a policy with which one can enforce the separation. In addition, an object interface defined in C++ might not be suitable for a remote implementation, that is, a situation where the caller of a member function resides in a different address space than the target object.

The Object Management Group (OMG) has developed the Interface Definition Language (IDL) as a standard way to specify objects that may be accessed remotely. IDL is part of OMG's Common Object Request Broker Architecture (CORBA) that defines the mechanisms for transparent access to objects in a network.

Using IDL to define object interfaces has three potential advantages over using C++:

- IDL enforces a policy of separating interfaces and implementation.
- IDL can be mapped in a natural way to more than one programming language, so an interface can be used conveniently from languages other than C++.
- IDL supports distribution—the sender and receiver of an operation may be in different address spaces and therefore on different machines in a network.

However, a potential drawback is that using IDL (and interfaces in general) might increase the burden on the programmer by requiring knowledge about the interface language, its mapping to C++, and how to use the tool that translates interfaces. All this knowledge must be acquired in addition to doing the actual implementation work.

For the Fresco[3] user interface system, we wanted to use IDL because of its potential advantages but were concerned that we and other programmers would find the burdens unacceptably high. As part of Fresco, we have therefore built a tool, called Ix, to simplify the use of interfaces.

Ix gives the programmer several options for how to translate interfaces to C++ class definitions. One such option is the choice of whether or not to use virtual base class derivation, which allows programmers to avoid the overhead and complexity of virtual base classes in some cases.

In addition to translating interface definitions to C++ class definitions and generating stubs for remote calls, Ix can perform "filtering" on an implementation header or source file to output those parts of the implementation that can be automated. One kind of filtering Ix performs is to generate the function type signature of an operation defined in the interface. Using this feature, a programmer need only enter the function signature once in the interface definition. All other instances of the signature, whether in class definitions or implementation files, can be generated automatically.

We have also built a tool, called I2mif, to simplify the documentation of a collection of interfaces. I2mif creates a document interchange file from a collection of interface files, copying specially-marked comments as well as the interface definitions.

2. Fresco

Fresco defines an object-oriented API for graphical user interfaces that covers functionality in Xlib and Xt and adds support for structured graphics and application embedding. Fresco covers a broad range of functionality—buttons, menus, rectangles, white space, text editors, and movie players can all be Fresco objects.

We decided to specify Fresco in IDL rather than C++ to obtain the advantages of using interfaces. We particularly needed the Fresco specification to be free of any implementation details. Support for multiple languages was attractive though not required. Additionally, distribution is desirable to support application embedding. As user environments become more object-oriented, turning applications into objects, users want distribution to be available for the individual components in a user interface just as is now available for individual applications using the X Window System.

For example, one might want to edit a single document containing text, a spreadsheet, and a drawing, where a separate editor is available for each type of component. The three component editors should be able to run in separate address spaces, perhaps on different machines, while the user should see them composed in a single unit.

An added benefit of using interfaces is the ability to generate run-time type information automatically without modifying the C++ compiler. CORBA defines two mechanisms that rely on run-time type information: narrowing and dynamic invocation. Narrowing effectively queries an object to see if it supports a particular interface and is similar to the C++ dynamic cast operation[7].

Dynamic invocation is a way to interpret a member function call, performing operation lookup and parameter checking at run-time rather than compile time. For Fresco, dynamic invocation is a particularly attractive way to support scripting. To investigate the viability of this approach, we implemented a script interface to Fresco called Dish (Dynamic Invocation SHell) using the Tcl language and interpreter[6]. Dish is a relatively small application (about 1,100 lines of C++) that uses dynamic invocation to evaluate commands that create and manipulate Fresco objects. All Fresco operations defined in IDL are automatically available in Dish, without any special registry or manual setup.

3. Translating interfaces

At first glance, the translation of an IDL interface to a C++ class appears straightforward:

- an IDL operation maps to C++ virtual functions
- an IDL attribute maps to a pair of C++ virtual functions

- IDL data types map either directly to C++ data types or to C++ classes that define the behavior of the IDL data type.

However, the important issue is to what extent the generated C++ class is abstract, that is, whether the generated class contains any data or non-pure member functions.

3.1 Member data

Typically, a generated class will inherit from some common base class. Proxy objects—surrogates that refer to an implementation in another address space—must store the information needed to access the remote implementation. It is tempting to put this data in the common base class object. However, generating a class that has data, either directly or inherited from a common base class, has two significant drawbacks.

The first drawback is that all object instances must carry that data even if they do not need it. This requirement could be a deterrent to using the interface for high-volume objects. For example, Doc[1] is a document editor that represents each visible character as a drawable object. These objects are not accessible remotely, as they are part of the document editor's implementation and not available to other applications. We want to define a common interface for drawable objects, but we do not want to burden the character objects with unnecessary overhead.

The second drawback of member data is that it requires virtual base class derivation because in the case of multiple inheritance a class should only contain a single copy of the base class data. As a consequence, if one wishes to allow for the possibility of multiple inheritance then one must use virtual class derivation everywhere.

3.2 Virtual base classes

Martin[4] uses virtual base classes and multiple inheritance in his approach to separating interface and implementation. Our own experience is that virtual base classes are expensive—at least in the C++ implementations that we use. For Fresco, our concern was that requiring virtual base classes would discourage some users.

The potential costs are best illustrated with an example. The code below defines a common base class named Base, two interfaces A and B, where B is derived from A, and implementation classes Aimpl and Bimpl. The A interface defines an operation "f" and the B interface defines an operation "g." Aimpl and Bimpl implement the A and B interfaces, respectively, where Bimpl wants to reuse Aimpl. That is, Bimpl::f() should just be Aimpl::f(), and Bimpl must hold any state needed for Aimpl. In this first version of the code, all derivation is virtual, and Bimpl inherits from both B and Aimpl.

```
class Base {
public:
    Base();
    virtual ~Base();
};

class A : public virtual Base {
public:
    A();
    ~A();
    virtual void f();
};

class B : public virtual A {
public:
    B();
    ~B();
    virtual void g();
};
```

```
class Aimpl : public virtual A {
public:
    Aimpl();
    ~Aimpl();
    void f();
};

class Bimpl : public virtual B, public virtual Aimpl {
public:
    Bimpl();
    ~Bimpl();
    void g();
};
```

An alternative to using multiple inheritance with virtual base classes is delegation, that is, using an object member instead of a parent class and redefining the operations to invoke the corresponding operation on the member. The code below provides the same interface as the previous example, but with the Bimpl class containing an instance of an Aimpl rather than deriving from Aimpl. Bimpl must also redefine A::f to call a_->f().

```
class Base {
public:
    Base();
    virtual ~Base();
};

class A : public Base {
public:
    A();
    ~A();
    virtual void f();
};

class B : public A {
public:
    B();
    ~B();
    virtual void g();
};

class Aimpl : public A {
public:
    Aimpl();
    ~Aimpl();
    void f();
};

class Bimpl : public B {
public:
    Bimpl();
    ~Bimpl();
    void f();
    void g();
private:
    Aimpl a_;
};
```

Note this approach means a Bimpl pointer can no longer be widened to an Aimpl pointer. However, this distinction is not an issue because Aimpl and Bimpl are both implementation classes and therefore hidden from the application code. Virtual base classes are not necessary even if the interfaces contain multiple inheritance, so long as the generated classes for the interfaces contain no data.

Table 1 shows the cost in memory size of the two approaches compiling with optimization on an Indigo running IRIX 5.1 and a C++ compiler based on Cfront 3.0. The code and data columns are the size in bytes for a file containing the definitions and empty function bodies with the exception of Bimpl::f() in the delegation case, which contains a call to a_->f().

Object	A	B	Aimpl	Bimpl	code	data
Inherit	12	24	24	68	2192	512
Delegate	4	4	4	8	1344	240

Table 1. Code and data sizes in bytes

In this example, delegation is more efficient in terms of memory usage, though less efficient in CPU time because of the extra virtual function call. A sophisticated compiler could, in principle, remove the CPU overhead by noticing that it could fill Bimpl's vtbl entry for f with Aimpl::f and the appropriate offset for the embedded Aimpl object. Regardless, we are willing to trade the CPU time for lower memory usage.

Whether these results are reflective of a particular compiler or applicable to other C++ compilers is not important for our purposes. Our goal is make the use of interfaces convenient and efficient. Since Ix can easily generate either virtual derivation or not, we give the choice of using virtual base classes or delegation to the programmer. Ix also provides a filtering mechanism to make delegation automatic instead of requiring the programmer to code every delegated function. This and other filtering features are described in more detail in Section 4.

3.3 Member functions

Conceptually, the class generated for an interface should have no code as well as no data. However, this approach means that the stubs for remote calls are generated for a subclass, and therefore the stubs for a derived interface must use either virtual base classes or delegation to inherit the stubs for the parent interface. In terms of the previous example, we want the stubs for class B to inherit the stubs for class A.

An alternative approach that we use in Ix is to generate stubs for the functions in the class generated for an interface. For an interface A that defines an operation f, the body of A::f contains the stub to perform a remote call. The stub code performs a virtual function call to access the state necessary to send the object reference to the remote site. The stub object contains a pointer to this state, which must be accessed indirectly anyway to allow different subclasses to use alternate representations of an object reference (it might be desirable to send a copy of the object's state, for example).

4. Filtering

In C++, the signature for every function is written at least twice—once in the class definition and once for the function body. Defining an interface class, whether in IDL or not, adds another definition to this burden. Since we needed an interface translator anyway, we wanted to use the information in the translator to eliminate the burden of repeating function signatures.

We considered several ideas for how to use the translator to generate C++ signatures automatically from IDL. One approach is to generate a file containing empty C++ functions for an interface and let the programmer fill in the bodies. However, this approach does not help if the interface changes after some of the implementation has already been written.

A second approach is to process the implementation files as part of compilation, generating the appropriate function signatures. A separate processing pass is undesirable because it would always slow down compilation, even when the interfaces have not changed. We considered defining C++ preprocessor macros for the signatures, but found that we really wanted to see the expanded signature, not a macro, while we were editing the implementation.

Our approach is to "filter" an implementation header or source file whenever the interface changes. The interface translator first reads the interface definition, then scans the implementation file looking for a line that contains special comments that begin with the characters "//+." Any line not containing these characters is copied as is.

The translator parses the annotations in special comments and generates the appropriate code. For this process to work repeatedly, filtering must eliminate the previously-generated code and copy the annotation comment back out to the new version of the implementation file.

The Ix filtering notation is not intended to be particularly elegant or self-explanatory. Our goals were for filtering to be simple to use, as visibly unobtrusive in the source as possible, and easy to process.

4.1 Class annotations

An implementation class defines some or all of the functions defined on the interface. We use the annotation "interface::op" to indicate that a class definition should include the signature for the given operation. For example, suppose the IDL interface for A is

```
interface A {
    void f();
    long g();
};
```

Before filtering the first time, we could write an implementation class as

```
class Aimpl : public A {
public:
    Aimpl();
    ~Aimpl();

    //+ A::f

    /* other members */
};
```

After the first filtering, the annotation line will appear as

```
    void f(); //+ A::f
```

If the signature for A::f changes, then re-filtering will automatically update the definition in the class. Attributes are similar to functions, except the annotation contains a trailing "=" or "?" to specify the set or get function, respectively.

4.1.1 Generating all operations

A "+" in place of the function name means that the class implements all the functions and attributes defined by the interface. In this case, the translator relies on a line containing only "//+" as an end marker for the lines of generated code. Following the example above, we could have

```
class AnotherAimpl : public A {
public:
    AnotherAimpl();
    ~AnotherAimpl();

    //+ A::+
    //+

    /* other members */
};
```

Filtering the annotation would generate

```
//+ A::+
void f();
long g();
//+
```

As before, re-filtering after changes to the interface will automatically update the class definition. This feature is especially convenient when adding or removing functions to an interface. A "*" can be used instead of "+" to generate inherited functions as well as functions defined by the interface.

4.1.2 Type information for implementation classes

Filtering also provides a mechanism for defining type information for implementation classes. The translator generates type information for interfaces that are used in narrowing and dynamic invocation. Sometimes, one would like to be able to narrow an interface to a specific implementation class. The annotation syntax is "derived : base1[, base2 ...]." For example, if we wanted to narrow to the Aimpl class we could write

```
//+ Aimpl : A
//+
    Aimpl();
    /* other members */
};
```

Filtering this annotation would augment the class specification with type-related member functions

```
//+ Aimpl : A
class Aimpl : public A {
public:
    ~Aimpl();
    TypeObjId _tid();
    static Aimpl* _narrow(BaseObjectRef);
//+
    Aimpl();
    /* other members */
};
```

4.2 Implementation annotations

To define the signature for the body of a function, we use the annotation "C(I::op)" where C is the implementation class name and I is the interface. The translator needs the implementation and interface names because the class definition and implementation may be in separate files and filtering does not perform include file processing.

Continuing the Aimpl example, the initial implementation would be

```
//+ Aimpl(A::f)
{
    /* function body */
}
```

After filtering, the code would be

```
//+ Aimpl(A::f)
void Aimpl::f() {
    /* function body */
}
```

4.2.1 Delegation

The annotation "C(I::+*call*)" generates delegation functions for all the operations and attributes defined on the specified interface. As with class definitions, a "*" instead of a "+" means all operations and all inherited operations.

The "*call*" is the part of the expression for the delegation call excluding the function, but including the member access operation ("." or "->"). For example, if Bimpl is a class that wants to delegate A operations to an Aimpl member "a_" then the filtered code would be

```
//+ Bimpl(A::*a_.)
void Bimpl::f() { a_.f(); }
long Bimpl::g() { return a_.g(); }
//+
```

4.2.2 Controlling support code

The translator also generates support code for the interface classes, including the bodies of the constructor, destructor, and narrow operation. One option is to generate this code in a separate file specified as part of the translation process. We support that option, but we also wanted to give the programmer finer control as to what code is generated and where it is generated.

Filtering makes the custom generation of support code simple. A collection of annotations for an interface specify that certain information should be generated. The annotations are generally of the form "I::%info" where "info" refers to the specific kind of information. The options for info include:

"init" for the constructor and destructor code

"type" for run-time type information

"type+dii" for run-time type information and dynamic invocation support

"stub-externs" for external declarations of stubs for concrete types such as structs

"type-stubs" for stubs for concrete types

"stubs" for remote stubs for operations

"client" for external declarations, type stubs, and operation stubs

"server" for receiving stubs

Several info requests can be separated by commas in a single annotation. An annotation containing only an interface name generates a default set of information, which is currently the initialization and type information.

We use custom code generation in Fresco for two reasons. First, we put the annotations for several interfaces in the same file, reducing the number of implementation files. Second, we avoid generating run-time information for those interface that need not be accessed dynamically, such as the type interface itself.

5. Ix implementation

Ix is about 11,000 lines of C++ code and to date represents roughly half a person-year of effort. Ix parses all IDL constructs, but does not check or generate code for features we have not needed for Fresco, such as contexts.

The Ix implementation is split into the following three main phases:

- Parsing, which is about 3,000 lines of code
- Symbol resolution and semantic checking, which is about 1,500 lines of code
- Code generation and filtering, of which about 3,000 lines are for generation and 1,500 for filtering.

The remaining 2,000 lines implement support data structures and command-line argument processing.

We have been using Ix for the development of Fresco for about six months. Fresco defines about 40 interfaces in IDL and about 100 classes in C++. Currently, about 15,000 of the 35,000 lines in the Fresco library are automatically generated by Ix.

6. Generating documentation

The I2mif program produces FrameMaker Interchange Format (MIF)[2] from comments in IDL source files. The comments are denoted by a line containing either the string "//-" or the string "//.". A line containing the string"//-" followed by additional text denotes the beginning of a definition where the text is the name being defined. A top-level definition ends with a line containing "//-" with no trailing text or a line beginning with a right brace ("}"). A nested definition (such as an operation within an interface) ends at the beginning of another definition or the end of the outer definition.

Lines that begin with "//." indicate text that should be written to the MIF file. Index markers can be specified with the syntax "\marker{text}" where text is the index entry. By default, I2mif creates index entries for names associated with "//-" comments. Formatted text can be specified with the syntax "\emphasis{text}" for italics or "\bold{text}" for a boldface font.

As an example of how this works, here is part of the Transform object interface in IDL:

```
//- TransformObj
interface TransformObj : FrescoObject {
    //. A transform represents a (logically) 4x4 matrix for use in translating coordinates.
    //. A 2-dimensional implementation may store and manipulate a 3x2 matrix rather than
    //. a full 4x4 matrix.

    //- load
    void load(in TransformObj t);
        //. Copy the matrix data from the given transform.

    //- scale, rotate, translate
    void scale(in Vertex v);
    void rotate(in float angle, in Axis a);
    void translate(in Vertex v);
        //. Modify the matrix to perform coordinate scaling, rotation, and translation.
        //. The rotation angle is given in degrees. A 2-dimensional implementation
        //. only implements rotate about the z-axis.
};
```

The formatted output would appear as follows:

TransformObj

interface TransformObj : FrescoObject

A trasnform represents a (logically) 4x4 matrix for use in translating coordinates. A 2-dimensional implementation may store an manipulate a 3x2 matrix rather than a full 4x4 matrix.

load

void load(in TransformObj t);

Copy the matrix data from the given transform.

scale, rotate, translate

void scale(in Vertex v);
void rotate(in float angle, in Axis a);
void translate(in Vertex v);

Modify the matrix to perform coordinate scaling, rotation, and translation. The rotation angle is given in degrees. A 2-dimensional implementation only implements rotate about the z-axis.

Before producing the output interchange file, I2mif sorts all the top-level definitions (interfaces) by name and within each top-level definition also sorts the nested definitions by name. The effect is a "dictionary" of interfaces and operations that is generated automatically from the IDL source. Using I2mif has help tremendously in keeping the interface documentation and source in sync with each other.

7. Conclusions

Separating interface from implementation is an important part of building a software system. Acceptance of using interfaces will depend to a large extent on the ease with which interfaces can be defined and modified.

Ix is a flexible tool for translating CORBA IDL to C++ that we have developed as part of the Fresco project. Our experience with Ix has been that implementation filtering makes programming with interfaces as easy or easier than straight C++. The ability to implement delegation easily has also made it simple for us to avoid the use of virtual base classes.

Overall, our software is more abstractly specified, more powerful, easier to edit, and as efficient as if we had not separated interface and implementation. Rather than spending time on mechanics, using Ix allows one to concentrate on the semantics of interfaces and their possible implementations.

8. Acknowledgment

Steve Churchill implemented I2mif.

9. Availability

Fresco will be available as part of the X11R6 distribution in the spring of 1994, including the source for Ix, run-time library, and the Dish application. This software will be available without restrictions on use. For further information about obtaining Fresco, send electronic mail to linton@sgi.com.

10. References

[1] P. Calder and M. Linton. The Object-Oriented Implementation of a Document Editor. *OOPSLA '92*, Vancouver, British Columbia, Canada, pp. 154-165.

[2] Frame Technology Corporation. *Maker Interchange Format (MIF) Reference Manual.*

[3] M. Linton and C. Price. Building Distributed User Interfaces with Fresco. *Proceedings of the Seventh X Technical Conference*, Boston, Massachusetts, January 1993, pp. 77-87.

[4] B. Martin. The Separation of Interface and Implementation in C++. *Proceedings of the USENIX C++ Conference*, Washington, D.C., April 1991, pp. 51-63.

[5] Object Management Group. Common Object Request Broker Architecture and Specification. OMG Document Number 91.12.1, Revision 1.1.

[6] J. Ousterhout. Tcl: an Embeddable Command Language. *Proceedings of the 1990 Winter USENIX Technical Conference*.

[7] B. Stroustrup and D. Lenkov. Run-Time Type Identification for C++ (Revised). *Proceedings of the USENIX C++ Conference*, Portland, Oregon, 1992, pp. 313-339.