Architecture of the CORBA Component Model

C++ Language Mapping: Client Side

Overview

- Mapping for interfaces and object references
- ORB initialization
- Stringification
- Initial References
- Parameter passing conventions
- Exceptions

Mapping for Interfaces

- Interfaces map to classes

```
// IDL
interface MyObject{
    long get_value();
};
// C++
class MyObject: public virtual CORBA::Object {
    public:
        virtual CORBA::Long get_value() = 0;
};
```

- actual proxy classes are subclasses

Interface Class Usage

- Mapped interfaces are abstract classes and must be instantiated
  - How are proxy objects created?
- Pointers and references to proxy objects must be not be formed
  - Use _ptr types instead

Object Reference Types

- **`I_ptr`** is a pointer-like type
- Proxies are (typically) locally reference counted
  - `CORBA::release(obj)` decrements the reference counter
  - `I::_duplicate` creates a new reference to the same object

```
I_ptr o1 = ...;
I_ptr o2 = I::_duplicate(o1);
CORBA::release(o2);
```

- **`I_var`** is an automatically manages references
  - Constructors invoke _duplicate
  - Destructor invokes CORBA::release
  - _var objects can be used like pointers (typically overload `operator->`)

Life Cycle of Object References

- References are never created in clients; they are obtained somehow instead
  - result/out parameter of operation
  - initial references
  - narrowing
  - nil reference
- Client keeps reference, may duplicate it
- Eventually reference is released
  - local operation; there is no distributed reference counting in CORBA
  - If client wishes that the CORBA object should end its life, it needs to invoke some interface operation

Reference duplication

```
class MyInterface{
    static MyInterface_ptr _duplicate(MyInterface){...}
    //...
};
namespace CORBA{
    void release(MyInterface_ptr); // might accept Object_ptr instead
};
```

Nil References

```
class MyInterface{
    static MyInterface_ptr _nil();
    //...
};
namespace CORBA{
    Boolean is_nil(MyInterface_ptr);
}
```

- nil references need to be duplicated/released just like any other reference
- Invoking operations on nil references is not allowed

Interface Inheritance

- Object references can be widened and narrowed
- Widening does not require type conversions, and does not involve duplication of the reference:

```
Base_ptr base;
Derived_ptr derived = ...;
base = derived;                    // ok, base and derived refer to the same proxy
base = Base::_duplicate(derived);  // base refers to a duplicated reference
```

- Narrowing requires explicit operation, as it may create new stub object
  - involves always a duplication
  - Potentially involves communication with remote server to validate target type
  - May return _nil if narrowing fails

```
derived = Derived::_narrow(base);
CORBA::release(base);
```

ORB Initialization

- Allow CORBA runtime to initialize
- Provide a means for ORB to inspect (and modify) command line arguments
  - Arguments starting with –ORB are reserved for CORBA usage (e.g. –ORBid Mico)

```
// CORBA.h
namespace CORBA {
   ORB_ptr ORB_init(int &, char**,
                    char * orb_identifier = "");
}
// Application
#include <CORBA.h>
int main(int argc, char* argv[])
{
   CORBA::ORB_var the_orb;
   the_orb = CORBA::ORB_init(argc, argv);
}
```

Stringified Object References

- ORB operations: string_to_object, object_to_string

```
namespace CORBA{
    interface ORB{
        char*          object_to_string(Object_ptr p);
        Object_ptr     string_to_object(const char*);
    };
}
```

- Standard stringifications for object references
  - IOR: (interoperable object references); hexified version CDR-marshalled GIOP::IOR
  - corbaloc: URI-style bootstrap object references; simplified information model
  - corbaname: URI-style references to CORBA naming service (CosNaming)
- Stringification is not unique: Multiple stringifications may refer to the same object

The **`Object`** Interface

```
namespace CORBA{
    class Object_ptr;
    class Object{
        Boolean   _is_a(const char* repository_id);
        Boolean   _non_existent();
        Boolean   _is_equivalent(Object_ptr other);
        ULong     hash();
        static Object_ptr _nil();
    };
}
```

Object::_is_a

- Expects repository ids
  - Typical form: IDL:prefix/path/to/definition:version
  - prefix can be controlled by #pragma prefix

  #pragma prefix "omg.org"
  - version can be controlled by #pragma version, defaults to 1.0
- May be implemented locally, or may involve remove communication
  - Can raise system exceptions

Object::_non_existent

- checks whether the reference refers to an existing object
  - TRUE indicates that the object definitely does not exist anymore
  - FALSE indicates that the object definitely exists
  - TRANSIENT system exception indicates that no determination could be made
  - other system exceptions are also possible
- may or may not involve communication with remote object

Object::_is_equivalent

- tests whether one reference refers to the same object as another
  - must be implemented locally (reference identity, not object identity)
  - TRUE guarantees that the references are equivalent
  - FALSE means no determination could be made

Object::_hash

- Allows to put references into hash tables
- Must be implemented locally
- hash value stays constant over life-time of reference

_var types for references

```
class MyInterface_var{
            MyInterface_var();
            MyInterface_var(MyInterface_ptr);
            MyInterface_var(const MyInterface_var&);
  MyInterface_var&  operator=(MyInterface_ptr);
  MyInterface_var&  operator=(const MyInterface_var&);
                    operator MyInterface_ptr();
  MyInterface_ptr   operator->();
  MyInterface_ptr   in()const;
  MyInterface_ptr&  inout();
  MyInterface_ptr&  out();
  MyInterface_ptr   _retn();
}
```

Memory Management for Object References

- Assignment/Construction from _ptr assumes ownership of reference
- Assignment/Construction from _var duplicates reference
- Deletion of _var releases reference

Explicit _var conversion

- .in(): Converts to _ptr
- .out(), .inout(): Convert to _ptr&
- ._retn(): Yields reference for returning result

Widening of _var

- Implicit widening between _var is not supported (may require two user-defined conversions)
  - Implicit widening between _ptr, and mixing _ptr and _var is supported only
- Need to invoke duplicate:

```
Derived_var d = ...;
Base_var b;
b = Derived::_duplicate(d); // may also use Base::_duplicate
```

Mapping for Operations and Attributes

- Operations map to member function with the same name
  - Can invoke operations using ->method for both _ptr and _var
- Attributes map to pair of getter/setter, overloaded on attribute name

```
// IDL
interface Person{
   attribute long age;
};
// C++
class Person{public:
   virtual long age()=0;
   virtual void age(long)=0;
}
```

Parameter Passing Conventions

- Need to take memory management into account
- Need to take fixed-size vs. variable-size into account (with special case for simple types)
- Need to take parameter direction into account

Output Parameters

- For IDL type T, a type T_out appears on the list of formal parameters
  - For fixed-sized types T, T_out is T&
  - For variable-sized types, T_out performs memory management

Passing of Simple Types

- char, long, double, etc
- in: pass by value
- out:T_out (i.e. T&)
- inout: pass by reference
- result: return value

Passing of Fixed-Size Types

- Like simple types, but avoid copy if possible
- in: pass by const T &
- out: T_out (i.e. T&)
  - can also pass T_var
- inout: T&
- Not discussed here: Passing of array
  - Mapping defines _slice types, to allow pass-by-value of arrays

Passing Variable-Length String Types

- in: const char*
- inout: char*
- out: CORBA::String_out
  - needed to support passing of String_var as argument for out parameter (String_var needs to deallocate old value)
  - can be initialized from char*& or String_var
- Must not pass NULL pointers

Passing Variable-Length Complex Types

- in: const T&
  - Passing as reference avoids copying in parameter passing
- inout: T&
  - Allows modification of value inside implementation
- out: T_out
  - Parameter value should be T* or T_var
  - caller needs to explicitly deallocate the result using `delete`

Passing Object References

- in: I_ptr
- inout: I_ptr&
- out: I_out
  - can pass either I_ptr& or I_var
  - need to release result eventually

Using _var for Parameter Passing

IDL Type	in	inout/out	Result
`string`	`const String_var&`	`String_var&`	`String_var`
`wstring`	`const String_var&`	`WString_var&`	`WString_var`
`any`	`const Any_var&`	`Any_var&`	`Any_var`
`objref`	`const` *`objref`*`_var&`	*`objref`*`_var&`	`objref_var`
`sequence`	`const` *`sequence`*`_var&`	*`sequence`*`_var&`	*`sequence`*`_var`
`struct`	`const` *`struct`*`_var&`	*`struct`*`_var&`	*`struct`*`_var`
`union`	`const` *`union`*`_var&`	*`union`*`_var&`	*`union`*`_var`

Mapping for Exceptions

- System exceptions, user exceptions:

```
namespace CORBA{
   class Exception{
   //...
   };
   class UserException: public Exception{
   //...
   };
   class SystemException: public Exception{
   //...
   };
}
```

CORBA::Exception

```
class Exception{
public:
                    Exception(const Exception&);
    virtual         ~Exception();
    Exception&      operator=();
    virtual void    raise()=0;
protected:
                    Exception();
};
```

CORBA::SystemException

```
enum CompletionStatus{ COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE};
class SystemException: public Exception { public:
                                        SystemException();
                                        SystemException(const SystemException&);
                                        SystemException(     ULong minor,
                                                             CompletionStatus Status
                                        );
    SystemException&                    operator=(const SystemException&);
    ULong                               minor()const;
    void                                minor(ULong);
    CompletionStatus                    completed()const;
    void                                completed(CompletionStatus);
    ...
};
```

CORBA::SystemException (2)

```
static SystemException *
                _downcast(Exception*);
static const SystemException *
                _downcast(const Exception*);
virtual void    _raise()=0;
```

- Specific system exceptions inherit from CORBA::SystemException

User Exceptions

- Map to class

```
// IDL
exception DidntWork{
    long max_supported;
    string error_msg;
};
// C++
class DidntWork: public CORBA::UserException{ public:
    CORBA::Long             max_supported;
    CORBA::String_mgr       error_msg;
    ...
};
```

User Exceptions (2)

```
DidntWork
static DidntWork*
virtual void
```

```
DidntWork();
DidntWork(CORBA::Long max_supported,
          const char* error_msg);
DidntWork(const DidntWork&);
~DidnWork();
operatpor=(const DidntWork&);
_downcast(CORBA::Exception*);
_raise();
```

Exception Specifications

- Mapping IDL exception specifications to C++ exception specifications is optional.

```
// IDL
exception Failed{};
interface Foo {
    void can_fail() raises (Failed);
};
// C++
class Foo: ... {public:
    virtual void can_fail()
            throw(CORBA::SystemException, Failed) // optional
    = 0;
```

Other Aspects of the C++ Mapping

- Not discussed here:
  - Contexts
  - ORB interface
  - Dynamic Invocation Interface