Linguagem de Definição de Interface

Introdução

Os objetos CORBA podem ser implementados em diferentes linguagens de programação. Entretanto, a especificação da interface de um objeto deve ser separada da sua implementação. A implementação é feita usando a linguagem de programação selecionada enquanto a interface é especificada usando a linguagem OMG Interface Definition Language (OMG IDL ou IDL). Ela é uma linguagem declarativa para definição de interfaces de objetos CORBA. Basicamente CORBA IDL se parece com a linguagem C++ reduzida à declaração de classes e tipos, isto é, não é possível escrever a implementação dos métodos da classe utilizando IDL. IDL dá suporte a:

- Constantes - para auxiliar na declaração de tipos
- Declaração de tipos - para definir tipos dos parâmetros
- Atributos - que obtém e definem os valores de um tipo determinado
- Operações - que recebem parâmetros e retornam valores
- Interfaces - que agrupam declarações de tipos, atributos e operações
- Módulos - para separação do espaço de nomes

Interface

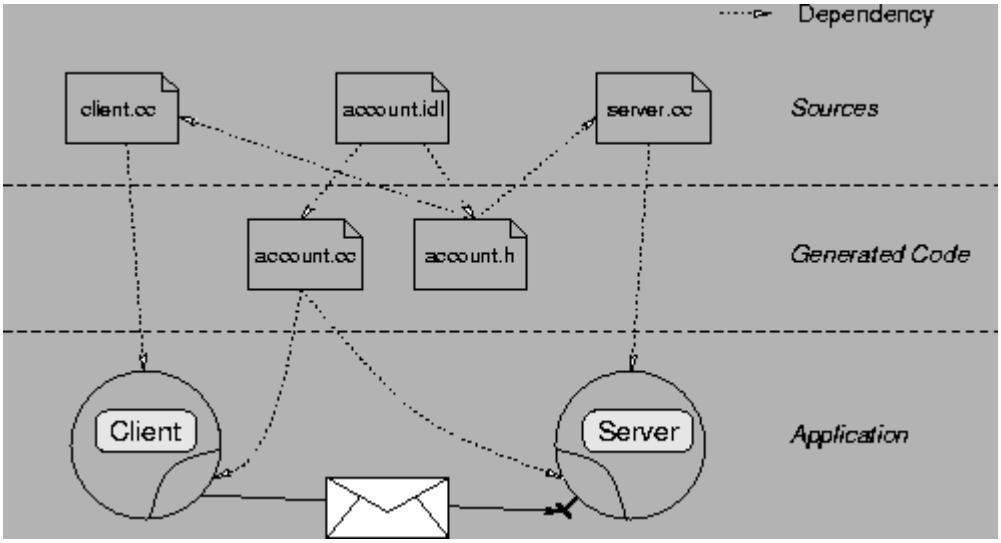
A interface é a construção mais essencial da linguagem IDL. Agora, vamos pensar em um exemplo para auxiliar na explicação, imagine um banco que mantém contas de seus clientes. Um objeto que implementa uma conta deste banco deve oferecer três operações: depósito (deposit) e saque (withdraw) de uma certa quantia de dinheiro e uma operação que retorna o saldo (balance) corrente da conta. O estado do objeto conta consiste de seu saldo.

```
interface Account {
void deposit (in unsigned long amount);
void withdraw (in unsigned long amount);
long balance ();
};
```

O declarador **in** declara amount como um parâmetro de entrada dos métodos deposit() e withdraw(). Usualmente a declaração acima seria salva em um arquivo denominado account.idl. O próximo passo é executar o compilador IDL para compilar esta interface e gerar código na linguagem de programação selecionada. No nosso exemplo, C++. O compilador IDL do MICO (uma das implementações disponíveis da especificação CORBA) é denominado idl e é usado da seguinte maneira:

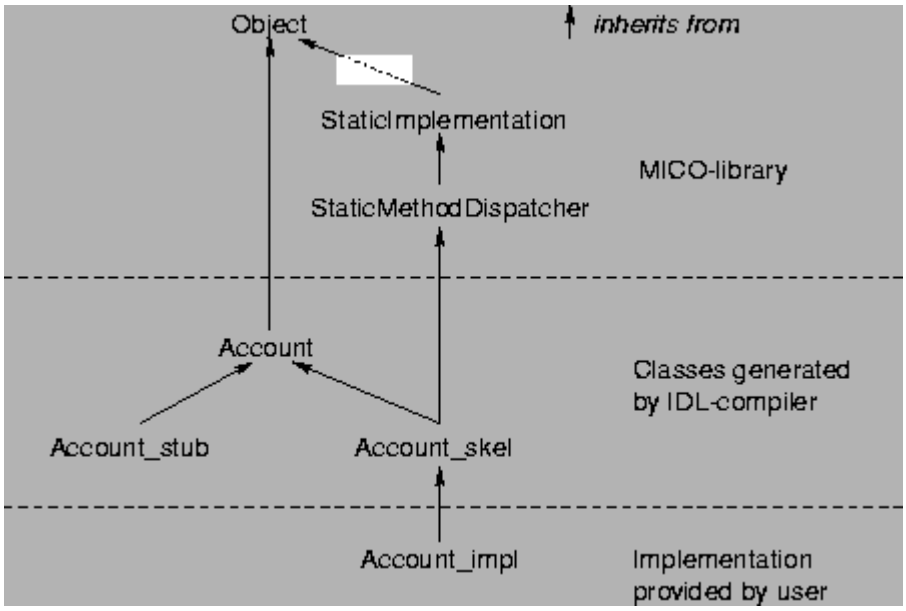
idl account.idl

O compilador IDL vai gerar dois arquivos: account.h e account.cc. O primeiro contém as declarações de classe para a classe base da implementação do objeto account e a classe stub que o cliente utiliza para invoca métodos em objetos account remotos. O segundo arquivo contém a implementação das classes e algum suporte de código.



Para cada interface declarada em um arquivo IDL, o compilador MICO IDL produzirá três classes C++. Observe a figura abaixo:

Relação de herança entre classes stub e skeleton



O arquivo account.h contém a classe *Account* que é a classe base. Ela contém todas as definições que pertencem a interface Account, como declarações locais de estruturas de dados definidas pelo programador. Esta classe também define uma função virtual para cada operação contida na interface. A seguir é apresentado um trecho de código da classe Account:

```
// Fragmento de codigo de account.h
class Account : virtual public CORBA::Object {
...
public:
...
virtual void deposit (CORBA::ULong amount) = 0;
```

```
virtual void withdraw (CORBA::ULong amount) = 0;
virtual CORBA::Long balance () = 0;
};
```

A classe Account_skel é subclasse de Account. Ela adiciona um dispatcher para as operações definidas na classe Account mas não define as funções virtuais da classe Account. As classes Account e Account_skel são, portanto, classes base abstratas na terminologia C++. Para implementar o objeto account deve-se criar uma subclasse de Account_skel fornecendo as implementações para os métodos virtuais deposit(), withdraw() e balance().

A classe Account_stub também é uma subclasse da classe Account. Mas ao contrário da classe Account_skel ela define as funções virtuais. A implementação dessas funções que são automaticamente geradas pelo compilador IDL é responsável pelo marshalling (serialização) dos parâmetros. O código de Account_stub seria mais ou menos assim:

```
// Fragmento de codigo de account.h e account.cc
class Account;
typedef Account *Account_ptr;
class Account_stub : virtual public Account {
    ...
public:
    ...
    void deposit (CORBA::ULong amount)
    {
       // Codigo de serializacao para deposit
    }
    void withdraw (CORBA::ULong amount)
    {
       // Codigo de serializacao para withdraw
    }
    CORBA::Long balance ()
    {
       // Codigo de serializacao para balance
    }
}
```

Isto torna Account_stub uma classe C++ concreta que pode ser instanciada. O programador nunca usa a classe Account_stub diretamente. O acesso é feito através da classe Account como será explicado mais tarde.

A classe Account herda da classe Object, a classe base para todos os objetos CORBA. Esta classe está localizada na biblioteca MICO. Account_skel herda de StaticMethodDispatcher que também está localizada na biblioteca MICO. Esta classe é responsável por despachar a invocação do método. A classe StaticMethodDispatcher herda de StaticImplementation que espelha o comportamento da DSI (dynamic skeleton interface) mas é projetada mais eficientemente.

Parâmetros de operação

O mapeamento dos tipos IDL nos parâmetros de operações em C++ depende de uma variedade de fatores. Faz-se distinção com respeito ao aspecto direcional do parâmetro (**in**, **out**, **inout** ou valor de retorno) e com relação ao tipo do parâmetro. A especificação CORBA diferencia entre tipos de dados de tamanho fixo ou variável. O tipo é de tamanho variável se é um dos seguintes:

1. o tipo *any*

2. uma *string* de tamanho limitado ou não-limitado
3. uma *sequence* de tamanho limitado ou não-limitado
4. uma referência a objeto
5. uma *struct* ou *union* com tipos de elementos variáveis
6. um array com elementos de tamanho variável
7. um *typedef* para tipos de tamanho variável

A razão desta diferenciação entre tipos de dados de tamanho fixo ou variável é para permitir uma maior flexibilidade no esquema de alocação de memória dos parâmetros. A tabela abaixo apresenta o mapeamento entre tipos IDL e C++.

Passagem de Argumentos e Resultado

Tipo	**In**	**Inout**	**Out**	**Retorno**
short	Short	Short&	Short&	Short
long	Long	Long&	Long&	Long
unsigned short	UShort	UShort&	UShort&	UShort
unsigned long	ULong	ULong&	ULong&	ULong
float	Float	Float&	Float&	Float
double	Double	Double&	Double&	Double
boolean	Boolean	Boolean&	Boolean&	Boolean
char	Char	Char&	Char&	Char
octet	Octet	Octet&	Octet&	Octet
enum	enum	enum&	enum&	enum

ponteiro para referência a objeto	objref_ptr	objref_ptr&	objref_ptr&	objref_ptr
struct, tamanho fixo	const struct&	struct&	struct&	struct
struct, tamanho variável	const struct&	struct&	struct*&	struct*
union, tamanho fixo	const union&	union&	union&	union
union, tamanho variável	const union&	union&	union*&	union*
string	const char*	char*&	char*&	char*
sequence	const sequence&	sequence&	sequence*&	sequence*
array, tamanho fixo	const array	array	array	array slice*
array, tamanho variável	const array	array	array slice*&	array slice*
any	const any&	any&	any*&	any*

Herança de Interface

A especificação CORBA prescreve que as interfaces IDL precisam ser mapeadas em classes C++. Quando se usa herança de interface Mico oferece duas alternativas na implementação de skeletons. Considere a seguinte definição IDL:

```
interface Base {
  void op1();
};

interface Derived : Base {
```

```
    void op2();
  };
```

`Base` é uma interface e serve de base para a interface `Derived`. Isto significa que todas as declarações feitas em `Base` são herdadas por `Derived`. O compilador idl cria as classes stub e skeleton para cada interface. As operações são mapeadas em métodos virtuais puros que devem ser implementadas pelo programador. Para a interface `Base` isto é direto:

```
  class Base_impl : virtual public Base_skel
  {
  public:
    Base_impl()
    {
    };
    void op1()
    {
      cout << "Base::op1()" << endl;
    };
  };
```

O skeleton para `Derived` permite duas formas diferentes de implementar o skeleton. A diferença entre os dois é se a implementação de `Derived` herda a implementação de `Base` ou não. Vejamos como isto é traduzido para código. A primeira alternativa é:

```
class Derived_impl :
    virtual public Base_impl,
    virtual public Derived_skel
  {
  public:
    Derived_impl()
    {
    };
    void op2()
    {
      cout << "Derived::op2()" << endl;
    };
  };
```

No fragmento de código acima a implementação de `Derived` herda da implementação `Base`. Note que `Derived_impl` herda de `Base_impl` e portanto precisa implementar somente `op2()` uma vez que `op1()` já está implementado em `Base_impl`.

Observação importante: ao implementar a classe `X_impl` que herda de múltiplas classes assegure-se que o construtor de `X_skel` é o último as ser chamado. Isto pode ser feito colocando `X_skel` como a entrada mais a direita na lista de herança:

```
  class X_impl : ..., virtual public X_skel {
    ...
  };
```

Agora a segunda alternativa (note que a classe skeleton ainda é a mesma; não existe um parâmetro do compilador `idl` para decidir entre as duas alternativas:

```
  class Derived_impl :
    virtual public Base_skel,
    virtual public Derived_skel
```

```
{
public:
  Derived_impl()
  {
  };
  void op1()
  {
    cout << "Derived::op1()" << endl;
  };
  void op2()
  {
    cout << "Derived::op2()" << endl;
  };
};
```

Observe também que: `Derived_impl` não herda de `Base_impl` mas de `Base_skel`. Por esta razão `Derived_impl` precisa implementar a operação `op2()`. A Figura abaixo apresenta a hierarquia das classes geradas pelo compilador IDL e a relação entre elas e as classes contidas na biblioteca MICO.

Hierarquia de classes C++ para herança de interface.