Introdução a Linguagem IDL (Interface Definition Language)

Sumário

1 – Introdução

- Objetos CORBA podem ser implementados em diferentes linguagens de programação, entretanto a especificação da interface deve ser separada da sua implementação.
  - ... a implementação é feita utilizando a linguagem de programação selecionada, p. ex., C++, Java, etc.
  - ... enquanto a interface é especificada usando a linguagem IDL (Interface Definition Language) da OMG.

... 1 - Introdução

- 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. Tipos suportados em IDL:

- constantes: para auxiliar a declaração de tipos;
- declaração: para definir tipos de parâmetros;
- atributos: que obtém e definem os valores de um tipo determinado;
- operações: que recebem parâmetros e retornam valores interfaces;
- interfaces: agrupam declarações de tipos, atributos e operações;
- módulos: para separação do espaço de nomes.

2 - Interface

- Construção essencial na Linguagem IDL que é responsável pelo agrupamento de tipos, atributos e operações.

- Problema: imagine um banco que mantém contas (account) de seus correntistas. Um objeto account deve oferecer três operações para o correntista:

  - saque (*withdraw*) - sacar um certo valor;
  - depósito (*deposit*) - depositar um certo valor;
  - saldo (*balance*) - consultar seu saldo;
  - saldo corrente deve ser armazenado pela classe.

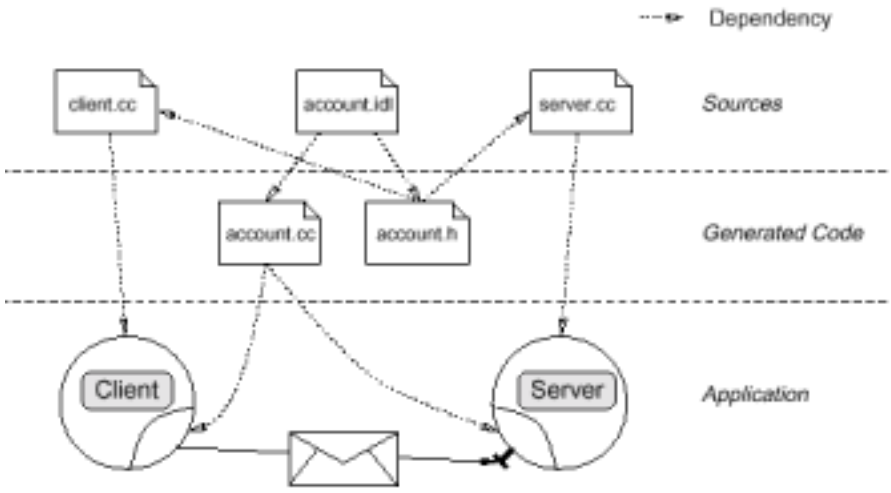
... 2 - Interface

- Exemplo de Interface IDL (arquivo account.idl):

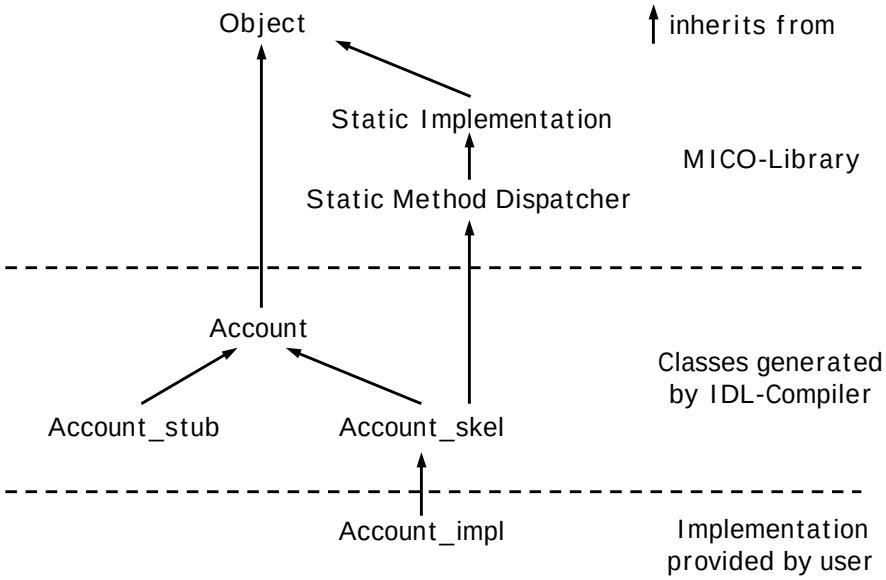
```
interface Account {
    void deposit (in unsigned long amount);
    void withdraw (in unsigned long amount);
    long balance ();
};
```

  - Novidades: palavras interface e in
  - Para compilar: idl account.idl
  - Arquivos gerados automaticamente: account.cc e account.h (skeletons + stubs)

... 2 - Interface



... 2 - Interface



... 2 - Interface

- account.h - contém a Classe “Account” que é a classe base.
  - contém todas as definições que pertencem a interface “Account”, como declarações locais de estruturas de dados definidas pelo programador.

- Fragmento da Código da Classe “Account”:

```
class Account : virtual public CORBA::Object {
public:
    virtual void deposit ( CORBA::ULon amount) = 0;
     virtual void withdraw ( CORBA::ULong amount) = 0;
     virtual CORBA::Long  balance ( ) = 0;
};
```

... 2 - Interface

- Account_sketl - 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 se criar um objeto account, deve-se criar uma subclasse de “Account_skel” fornecendo as implementações para os métodos virtuais “deposit( )”, “withdrawn( )”, e “balance( )”.

... 2 - Interface

- Account_stub - ao contrário da classe “Account_skel”, aqui são definidas as funções virtuais da classe “Account”.
- ... a implementação dessas funções geradas automaticamente pelo Compilador IDL é responsável pela serialização (marshalling) dos parâmetros;
- “Account_stub” é uma classe concreta, no entanto, o programador nunca usa a classe “Account_stub” diretamente, ou seja, o acesso é sempre feito através da classe “Account”.

... 2 - Interface

- Fragmento de código da classe “Account_stub”:

```
class Account;
typedef Account* Account_ptr;
class Account_stub : virtual public Account {
...
public:
    void deposit( CORBA::ULong amount ) {
    // código de serialização para “deposit( )”
    }
    void withdrawn( CORBA::ULong amount ) {
    // código de serialização para “deposit( )”
    }
    ... ...
};
```

... 2 - Interface

- A classe “Account” herda da classe “Object”, a classe base para todos os objetos CORBA (biblioteca MICO).
- “Account_skel” herda de “StaticMethodDispatcher” que também está localizada na biblioteca MICO:
  - responsável por despachar a invocação do método.
- “StaticMethodDispatcher” herda de “StaticImplemantion” que espelha o comportamento da DSI (Dynamic Skeleton Invocation) mas é projetada mais eficientemente.

3 - 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 (string, sequence, referência de objeto, struct, array ou typedef)

... 3 - Parâmetros de Operação

- 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.
- Mapeamento entre tipos IDL e C++:

Tipo	In	Inout	Out	Retorno
short	Short	Short&	Short&	Short
long	Long	Long&	Long&	Long
unsignet short	UShort	UShort&	UShort&	UShort

... 3 - Parâmetros de Operação

- **continuação mapeamento IDL p/ C++**

Tipo	In	Inout	Out	Retorno
unsignet 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
ptr ref obj	objref_ptr	objref_ptr&	objref_ptr&	objref_ptr

... 3 - Parâmetros de Operação

- **continuação mapeamento IDL p/ C++**

Tipo	In	Inout	Out	Retorno
struct (tam. fixo)	const struct&	struct&	struct&	struct
string	const char*	char*&	char*&	char*
sequence	const sequence&	sequence&	sequence&	sequence*
array (tam. fixo)	const array	array	array	array slice*
array (tam. variável)	const array	array	array slice*&	array slice*
anny	const any&	any&	any*&	any*

4 - 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( );
};
```

... 4 - Herança de Interface

- ... o Compilador IDL cria classes stub e skeleton para cada interface, sendo aue as operações são mapeadas em métodos virtuais puros que devem ser implementados pelo programador.

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

... 4 - Herança de Interface

- ... o skeleton para “Derived” permite duas formas diferentes, uma herda a implementação da base e a outra não.

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

... 4 - Herança de Interface

- ... no fragmento de código acima a implementação de “Derived” herda da implementação “Base”.
- ... note que ao herdar de “Base_impl”, não mais é necessário implementar “op1( )”, apenas “op2( )”, pois “op1( )” já está implementado em “Base_impl”.
- Nota: 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:

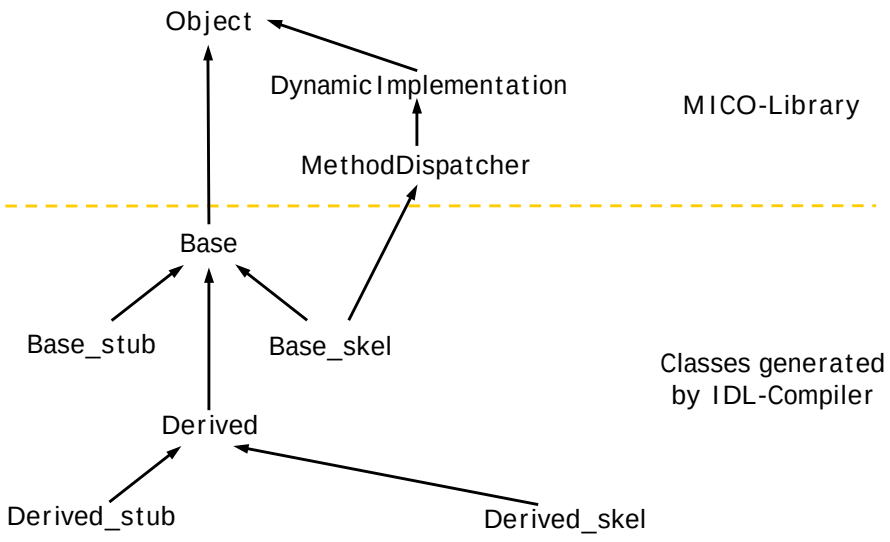
```
class X_impl : ..., virtual public X_skel {
... ... ...  };
```

... 4 - Herança de Interface

- segunda alternativa: observe que “Derived_impl” não herda de “Base_impl” mas de “Base_skel” e por isto precisa implementar a operação “op1( )”.

```
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;
  };
};
```

... 4 - Herança de Interface



... 4 - Herança de Interface