Object Adapters (OAs)

- Conectam implementações de objetos ao ORB
  - Diferentes OAs podem suportar diferentes estilos de implementação de objetos
- Ativam implementações de objetos (servidores)
- Ativam os múltiplos objetos implementados por um servidor
- Encaminhmento de requisições de serviço
  - para implementações de objetos (servidores)
  - para um dos objetos implementados pelo servidor
- Geram *object references*
  - Para ter o status de “objeto CORBA”, o objeto deve ser registrado com o adaptador

O *Basic Object Adapter* (BOA)

- Presente na especificação CORBA até a revisão 2.1
- Problemas na especificação do BOA:
  - Aspectos importantes ficaram indefinidos
  - Muita ambiguidade
- CORBA 2.2 substituiu o BOA pelo POA, um adaptador cuidadosamente especificado
  - Mas o BOA ainda é o adaptador de objetos oferecido pela maioria dos ORBs
  - Por isso os servidores atuais não são portáveis

Responsabilidades do BOA

- Ativação e desativação
  - de implementações de objeto
    - execução de processos servidores
  - de objetos individuais num servidor
    - instanciação de serventes dentro do servidor
- Geração e interpretação de *object references*
  - para tanto, colabora com o servidor
- Invocação de métodos via esqueletos (faz *upcalls* para os esqueletos)

Ativação de Implementações e de Objetos

- Ao tratar uma requisição de servico o BOA pode ter de fazer dois tipos de ativação
  - **Ativação de implementação** é necessária quando o servidor não estiver disponível
  - **Ativação de objeto** é necessária quando o servidor estiver disponível, mas não o particular objeto ao qual a requisição se destina
  - O BOA só pode chamar o um método do esqueleto se tanto o servidor como o objeto destino estiverem ativos

Políticas de Ativação de Implementação

O BOA define 4 políticas de ativação de servidores:

- Servidor **compartilhado** (*shared server*)
  - múltiplos objetos num servidor
- Servidor **não compartilhado** (*unshared)*
  - um processo servidor por objeto
- Servidor **por método**
  - *à la* CGI
- Servidor **persistente** (o nome é muito ruim)
  - servidor compartilhado, mas sem ativação automatica

A Interface do BOA

```
// pseudo IDL
module CORBA {
    interface InterfaceDef; // forward decls
    interface ImplementationDef;
    interface Object;
    typedef sequence<octet,1024> ReferenceData;
    ...
    interface BOA {
        ... // operações na próxima figura
    }; ...
}
```

Operações na Interface do BOA

```
// registro e desativação de implementações
//                       e de objetos

void impl_is_ready(in ImplementationDef impl);
void deactivate_impl(in ImplementationDef impl);
void obj_is_ready(in Object obj,
                  in ImplementationDef impl);
void deactivate_object(in Object obj);
```


Operações na Interface do BOA (cont.)

```
// geração e interpretação de object references

Object create(in ReferenceData id,
              in InterfaceDef intf,
              in ImplementationDef impl);
ReferenceData get_id(in Object obj);
void dispose(in Object obj);

// outras operações

...
```

Geração e Interpretação de *Object References*

- Um campo da *object reference* é definido pelo servidor que a criou
- Esse campo, denominado **`id`**, tem o propósito de distinguir os objetos implementados pelo servidor
- Quando o BOA faz uma *upcall* a um servidor para ativar um certo objeto, ele passa ao servidor o **`id`** do objeto

Geração e Interpretação de *Object Refs.* (cont.)

- Um **`id`** pode conter o nome de um arquivo ou uma chave de acesso a uma base de dados com os estados de muitos objetos
- É responsabilidade do servidor prover uma rotina de ativação de objetos que interpreta o **`id`**
  - O BOA faz *upcalls* a essa rotina

Problemas com o BOA

- No que diz respeito à ativação de objetos, a especificação do BOA deixou de definir muitas coisas:
  - Como o BOA conhece a rotina de ativação de objetos a ser chamada?
  - Quais os parâmetros dessa rotina?
  - Que eventos podem causar ativação de objetos?
- Cada implementador de ORB fez suas escolhas...

"BOAs Proprietários"

- No Orbix, a ativação de objetos é chamada *loading* e é feita por *loaders*
  - Para instalar um *loader*, um servidor cria uma instância de uma classe derivada da classe **`LoaderClass`**
  - A rotina de ativação de objetos é o método **`load`** da classe **`LoaderClass`**
    - Esse método pode ser redefinido no servidor (pela classe derivada da classe **`LoaderClass`**)

“BOAs Proprietários” (cont.)

- O VisiBroker adota um esquema semelhante
  - Uma objeto **`Activator`** é passado como parâmetro para **`impl_is_ready`**, juntamente com uma string (o nome do **`Activator`**)
  - A rotina de ativação de objetos é o método **`activate`** do objeto **`Activator`**
  - Para que um objeto seja ativado dinamicamente, ao se registrar o objeto (via **`obj_is_ready`**) deve-se passar o nome de um **`Activator`** e um identificador do objeto
    - Essas informações são incorporadas à *objref*

“BOAs Proprietários” (cont.)

- Tanto o Orbix como o VisiBroker permitem que um servidor instancie objetos CORBA diretamente
  - No Orbix, o construtor de um objeto CORBA registra o objeto com o ORB e cria uma *object reference;* o destrutor faz o oposto
  - No VisiBroker, o registro é feito chamando-se **`BOA::obj_is_ready`**
  - Nesses ORBs não se usa a operação **`BOA::create`**

Implementando Interfaces IDL

- Como indicar que uma classe C++ ou Java implementa uma interface IDL?
  – Através de uma conexão entre essa *implementation class* e a classe esqueleto gerada pelo compilador IDL
- CORBA prevê duas abordagens:
  – Heranca
  – Delegação

Implementando Interfaces IDL (cont.)

- Nos próximos slides vamos considerar:
  – Uma **interface IDL**
  – As correspondentes **classe interface** (*interface class)* e **classe esqueleto**, ambas geradas automaticamente pelo compilador IDL
  – A **classe implementação** (*implementation class*) fornecida pelo implementador da interface
- Nosso interesse é a conexão entre a classe esqueleto e a classe implementação

Uma Interface IDL

```
// IDL
interface A {
    short op1();
    void op2(in long l);
};
```


Sua Classe Interface

```
// C++
class A : public virtual CORBA::Object {
public:
    virtual CORBA::Short op1() = 0;
    virtual void op2(CORBA::Long l) = 0;
    ...
};
```

Sua Classe Esqueleto

```
// C++
class _sk_A : public A {
public:
    // ... detalhes específicos de ORB
    //     omitidos daqui ...
    virtual CORBA::Short op1() = 0;
    virtual void op2(CORBA::Long l) = 0;
    ...
};
```


Uma Classe Implementação

- Abordagem: herança

```
// C++
class A_impl : public _sk_A {
public:
    CORBA::Short op1();
    void op2(CORBA::Long l);
    ...
};
```

Sua *"Tie Class"*

- Classe esqueleto voltada para delegação

```
// C++
template <class T>
class _tie_A : public A {
public:
    _tie_A(T& t) : _ref(t) {}
    CORBA::Short op1() { return _ref.op1(); }
    void op2(CORBA::Long l) { _ref.op2(); }
private:
    T& _ref;
};
```


Outra Classe Implementação

- Abordagem: delegação

```
// C++
class A_impl {
public:
    CORBA::Short op1();
    void op2(CORBA::Long l);
    ...
};
```

  – Objetos de implementação são "amarrados" (*tied*) a interfaces IDL em tempo de execução:

    `_tie_A<A_impl> my_obj(my_impl);`

    ❑ aqui `my_impl` é uma instância da classe `A_impl`

O Lado do Servidor

- Vejamos um exemplo de servidor escrito em C++
- Usaremos um ORB específico, o VisiBroker para C++
  - O servidor não será portável entre ORBs
- Implementaremos a interface **`Quoter`**
  - Usando herança
  - Usando delegação

Exemplo de Servidor: os Arquivos Fonte

- Compilação IDL:
  - **`quoter.idl`** ⇒ **`quoter_c`** (**`.hh`** e **`.cc`**)
    `quoter_s` (**`.hh`** e **`.cc`**)
- Vamos escrever:
  - **`quotimpl.h`**
    - header file da classe implementação **`QuoterImpl`**
  - **`quotimpl.cc`**
    - métodos da classe **`QuoterImpl`**
  - **`quotmain.cc`**
    - mainline do servidor

Implementando a Interface **Quoter** com Herança

```
// arquivo quotimpl.h:
#include "quoter_s.hh" // gerado p/ comp. IDL

class QuoterImpl : public _sk_Stock::_sk_Quoter
  {
public:
    QuoterImpl(const char* my_name);
    char* name();
    CORBA::Long value(const char* name);
private:
    string m_name; // classe string ANSI/ISO
    map<string,CORBA::Long> m_map; // map da STL
};
```


QuoterImpl::name()

```
// em quotimpl.cc:

char*
QuoterImpl::name()
{
    char* nm = CORBA::string_dup(m_name.c_str());
    if (nm == 0)
        throw CORBA::NO_MEMORY();
    return nm;
}
```

QuoterImpl::value()

```
// em quotimpl.cc:

CORBA::Long
QuoterImpl::value(const char* stock)
{
    map<string,CORBA::Long>::iterator iter =
        m_map.find(stock);
    if (iter == m_map.end())
        throw Stock::UnknownStock(stock);
    return (*iter).second;
}
```


QuoterImpl::QuoterImpl()

```
// em quotimpl.cc:

QuoterImpl::QuoterImpl(const char* my_name)
    : m_name(my_name)
{
    // inicializa m_map com informações lidas
    // de um arquivo ou banco de dados
    // ...

    //     (num quoter real, m_map deveria
    //      ser atualizado periodicamente)
}
```

Mainline do Servidor

```
// arquivo quotmain.cc:

#include "quotimpl.h"
int main(int argc, char* argv[])
{
    CORBA::ORB_var orb_ref =
        CORBA::ORB_init(argc, argv);
    CORBA::BOA_var boa_ref =
        orb_ref->BOA_init(argc, argv);
```


Mainline do Servidor (cont.)

```
    // cria um objeto QuoterImpl
    Quoter_var my_quoter =
        new QuoterImpl("Dow Jones");
    try {
        boa_ref->obj_is_ready(my_quoter);
        boa_ref->impl_is_ready("DowJones");
    }
    catch (CORBA::SystemException& e) {
        cerr << "Exceção: " << endl << &e;
        return 1;
    }
    return 0;
}
```

Mainline do Servidor: Observações

- A string passada à função `CORBA::BOA::impl_is_ready()` é o **nome do servidor** (o nome com o qual o servidor foi registrado no repositório de implementações, via comando `oadutil reg` do VisiBroker)
- Essa função só retorna ao final da execução do servidor (o laço de processamento de eventos está dentro de `impl_is_ready()`)