Introdução à Programação
Orientada a Objetos com Java

Introdução a Objetos Distribuídos

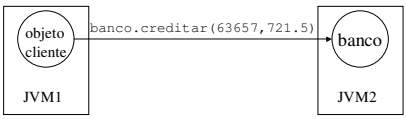
Paulo Borba
Centro de Informática
Universidade Federal de Pernambuco

Motivação

- Aplicações distribuídas
  - crescente demanda (comércio eletrônico, educação a distância, multimídia, jogos)
  - complexidade
  - integração à tecnologia de objetos
- Alternativas
  - *sockets*: baixo nível
  - RPC: não tem semântica de objetos

Definição

- RMI: *Remote Method Invocation*
- Permite invocação de método em objeto localizado em outra máquina virtual, com mesma sintaxe de invocação local



Modelo de objetos distribuídos de RMI

- Objeto remoto
  - métodos invocados de outra máquina virtual Java, possivelmente em outro *host*
  - implementa pelo menos uma interface remota
- Referência a objeto remoto
  - cliente tem que procurá-la
  - é do tipo de uma interface remota

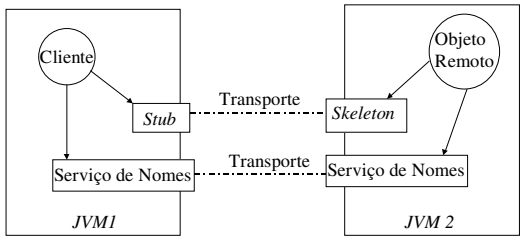
Modelo de objetos distribuídos de RMI

- Invocação de método remoto
  - invocação de método de uma interface remota implementada por objeto remoto
  - semântica estendida:
    - modos de falha mais complexos
    - exceção adicional

Modelo de objetos distribuídos de RMI

- Invocação de método remoto
  - passagem de parâmetros e retorno de método:
    - objeto remoto: por referência
    - objeto não remoto: por valor

Arquitetura




```
banco.creditar(63657,721.5)
```

Arquitetura

- *Stub*
  - referência a objeto remoto
  - implementa interface remota
  - inicia conexão com JVM remota
  - faz serialização de parâmetros para a JVM remota

Arquitetura

- espera pelo resultado da invocação do método
- faz a desserialização do valor ou exceção de retorno
- retorna o valor ou exceção ao cliente
- gerado automaticamente pelo compilador ***rmic***

Arquitetura

- *Skeleton*
  - implementa interface remota
  - desserializa os parâmetros para o método remoto
  - invoca o método no objeto remoto
  - serializa valor de retorno ou exceção
  - gerado automaticamente pelo compilador ***rmic***

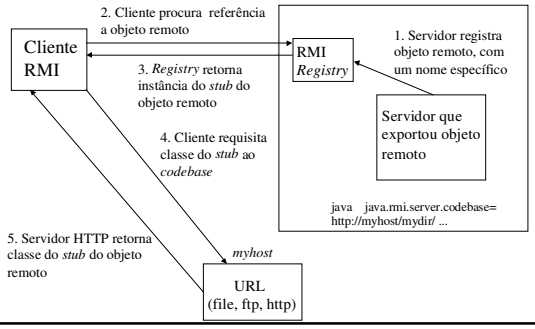
Arquitetura

- Serviço de nomes (*Registry*)
  - serviço básico para obtenção de referência a objeto remoto
  - objeto remoto é registrado e procurado pelo mesmo nome (*string*)
  - é um objeto remoto, cuja referência é acessível a qualquer JVM através de uma URL
  - deve estar na mesma máquina que os objetos remotos

Arquitetura

- Carregamento dinâmico de código
  - essencial para modelo de objetos distribuídos de RMI
  - carregamento dos *stubs* de objetos remotos
  - carregamento da definição de classes de objetos não remotos
  - mecanismo de serialização de objetos
  - segurança (*Security Manager*)

Arquitetura



Modelo de programação

- Modelo de objetos distribuídos -> API
- Servidor
  - interface remota
  - objeto remoto
  - inicializador
- Cliente
  - referência indireta a objeto remoto
  - referência direta ao *stub*

Modelo de programação

- Interface remota
  - define métodos que serão invocados remotamente
  - estende `java.rmi.Remote`
  - cada método deve levantar `RemoteException`
  - classe de objeto não remoto como parâmetro deve implementar `java.io.Serializable`

Modelo de programação

```
public interface MyBankRemote extends Remote {

   void creditar (int nConta, double valor) throws
      ContaNaoExisteException, RemoteException;

   void cadastrar (Conta c) throws ContaJaExisteException,
      ContaNaoEspecificadaException, RemoteException;
   ...
}
```

Modelo de programação

```
public class Conta implements java.io.Serializable {

...

}
```

Modelo de programação

- Objeto remoto
  - implementa interface remota
  - herda de `UnicastRemoteObject`
  - construtor
    - exporta o objeto remoto (aguarda invocação em porta)
    - levanta `RemoteException`

Modelo de programação

```
public class MyBankRemoteImplementation extends
          UnicastRemoteObject implements MyBankRemote {
...
   public MyBankRemoteImplementation() throws
            RemoteException {
       super();
       ...
    }

   public void creditar (int nConta, double valor) throws
            ContaNaoExisteException, RemoteException {
       ...
    }
    ...
}
```

Modelo de programação

- Inicializador
  - instala *Security Manager*
  - cria objeto remoto
  - registra objeto remoto no serviço de nomes (*Registry*)

```
public static void main(String[] args){

  // cria e instala o security manager
  if (System.getSecurityManager() == null)
    System.setSecurityManager( new RMISecurityManager());

 try{
   // instancia objeto remoto
   MyBankRemoteImplementation banco =
           new MyBankRemoteImplementation();

   // registra objeto remoto no serviço de nomes
  Naming.rebind("//www.di.ufpe.br:2120/BankServer",banco);
  }
  catch (Exception e) {
     System.out.println("Banco err: " + e.getMessage());
     e.printStackTrace();
  }
 }
```

Modelo de programação

- Cliente
  - interage com interface remota

```
MyBankRemote banco;
```

  - procura referência a objeto remoto

```
banco = (MyBankRemote)
   Naming.lookup("//www.di.ufpe.br:2120/BankServer");
```

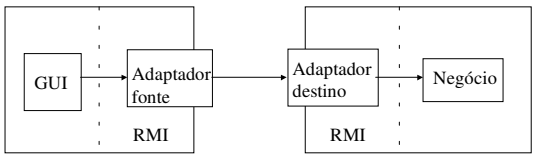
  - trata exceção remota

```
try{
       banco.creditar(nConta, valor);
  }
  catch (RemoteException e){
     ...
  }
```

Padrão de projeto

- Estrutura em camadas
- Interface representando comportamento distribuído
- Adaptadores fonte e destino isolando camada de comunicação

Padrão de projeto



Padrão de projeto

- Interface representando comportamento distribuído
  - exceções de negócio e exceção de comunicação genérica (`CommunicationException`)

Padrão de projeto

```
public interface MyBank {

    void creditar (int  nConta, double valor) throws
        CommunicationException, ContaNaoExisteException;
    ...
}
```

Padrão de projeto

- Adaptador fonte
  - isola a GUI da camada de comunicação (RMI)
    - —obtém referência ao objeto remoto
    - —delega invocação ao objeto remoto
    - —troca exceções RMI por exceção genérica

Padrão de projeto

```
public class MyBankSourceRMIAdapter implements MyBank {
   private MyBankTargetRMIAdapter  banco;

   public MyBankSourceRMIAdapter() throws
            CommunicationException{
     try{
      banco =(MyBankTargetRMIAdapter)
        Naming.lookup("//www.di.ufpe.br:2120/BankServer");
     } catch (Exception e) {
         e.printStackTrace();
         throw new CommunicationException ( );
       }
   }
   ...
```

Padrão de projeto

```
public void creditar (int nConta, double valor) throws
      CommunicationException, ContaNaoExisteException {

    try{
        banco.creditar(nConta, valor);
    }
     catch (RemoteException e){
       throw new CommunicationException ( );
     }
}
...
}
```

Padrão de projeto

```
public interface MyBankTargetRMIAdapter extends Remote {

    void creditar (int nConta, double valor) throws
          CommunicationException,ContaNaoExisteException,
           RemoteException;
    ...
}
```

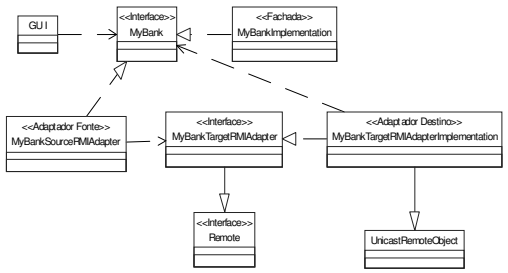
Padrão de projeto

- Adaptador destino
  - isola a camada de negócio da camada de comunicação (RMI)
    - faz papel de objeto remoto
    - faz papel de inicializador
    - delega invocação de método remoto a método de negócio

```
public class MyBankTargetRMIAdapterImplementation
        extends UnicastRemoteObject implements
            MyBankTargetRMIAdapter {

   private MyBank banco;

   public MyBankTargetRMIAdapterImplementation() throws
            RemoteException,InicializacaoBancoException {

        banco  =  new MyBankImplementation();
   }

   public void creditar (int nConta, double valor) throws
           RemoteException, CommunicationException,
             ContaNaoExisteException {

        banco.creditar(nConta, valor);
   }
   ...
```

```
...
public static void main(String[] args){

   if (System.getSecurityManager() == null)
     System.setSecurityManager (newRMISecurityManager());

     try{
        MyBankTargetRMIAdapterImplementation banco =
            new MyBankTargetRMIAdapterImplementation();

        Naming.rebind ("//www.di.ufpe.br:2120/BankServer",
                       banco);

     }catch (Exception e) {
         System.out.println("Banco err: " +
                            e.getMessage());
        e.printStackTrace();
      }
}
```

Padrão de projeto



Padrão de projeto

- Considerações
  - produtividade
    - acréscimo de classes geradas, mas ferramentas podem ajudar
  - maior modularidade
  - separação de conceitos
  - mudança de plataforma não causa impacto no sistema
  - adequado à implementação progressiva