AGNALDO KIYOSHI NODA

MECANISMO DE INVOCAÇÃO REMOTA DE MÉTODOS EM MÁQUINAS VIRTUAIS

Projeto de Dissertação de Mestrado apresentado ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática Aplicada.

Curitiba
2005

AGNALDO KIYOSHI NODA

MECANISMO DE INVOCAÇÃO REMOTA DE MÉTODOS EM MÁQUINAS VIRTUAIS

Projeto de Dissertação de Mestrado apresentado ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática Aplicada.

Área de Concentração: Computação Distribuída

Orientador: Prof. Dr. Alcides Calsavara

Curitiba
2005

Noda, Agnaldo Kiyoshi
MECANISMO DE INVOCAÇÃO REMOTA DE MÉTODOS EM MÁQUINAS VIRTUAIS. Curitiba, 2005.

Projeto de Dissertação de Mestrado - Pontifícia Universidade Católica do Paraná Programa de Pós-Graduação em Informática Aplicada.

1. RMI 2. Orientação à Objetos 3. Computação Distribuída I.Pontifícia Universidade Católica do Paraná. Centro de Ciências Exatas e Tecnologia. Programa de Pós-Graduação em Informática Aplicada II - t

Agradecimentos

Agradeço ao meu orientador Alcides Calsavara por toda a paciência, dedicação e disponibilidade para compartilhar seus conhecimentos. À minha esposa Rogeria pelo amor, dedicação, apoio e incentivo. Agradeço também aos meus pais Noda e Tiyko, minhas irmãs Beth e Edna e meu irmão Satoshi que sempre me apoiaram, vibraram e incentivaram. Aos colegas de mestrado, Kolb e Juarez, que caminharam comigo nesta jornada. Não posso deixar de agradecer aos meus amigos e sócios Cássio, Luis César, Luis Fernando, Naoki e Ortega pela compreensão e apoio. E finalmente, à Deus, que sem Ele nada seria possível.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

SCD	*Sistemas Computacionais Distribuídos*
RPC	*Remote Procedure Call*
CORBA	*Common Object Request Broker Architecture*
RMI	*Remote Method Invocation*
UDP	*User Datagram Protocol*
EBCDIC	*Extended Binary Code Decimal Interchange Code*
ASCII	*American Standard Code for Information Interchange*
DNS	*Domain Naming Service*
TCP	*Transmission Control Protocol*
IP	*Internet Protocol*
OSF	*Open Software Foundation*
DCE	*Distributed Computing Environment*
URL	*Uniform Resource Locator*
IDL	*Interface Definition Language*
ORB	*Object Request Broker*
DOSA	*Distributed Object System Architecture*
OMG	*Object Management Group*
ORB	*Object Request Broker*
OA	*Object Adapter*
DII	*Dynamic Invocation Interface*

DSI	*Dynamic Skeleton Interface*
AMI	*Asynchronous Programming Model*
SII	*Static Invocation Interface*
POA	*Portable Object Adapter*
API	*Application Programming Interface*
COSNS	*Common Object Service Naming Service*
URL	*Uniform Resource Locator*
TAD	*Tipo Abstrato de Dados*
MVV	***Máquina Virtual*** VIRTUOSI
ECL	*Entrada de Classe Local*
ECR	*Entrada de Classe Remota*
EIL	*Entrada de Invocável Local*
EIR	*Entrada de Invocável Remota*
EOL	*Entrada de Objeto Local*
EOR	*Entrada de Objeto Remoto*
IANA	*Internet Assigned Numbers Authority*

Resumo

RPC (Chamada de Procedimento Remoto) tornou-se um dos mais poderosos paradigmas de comunicação para sistemas distribuídos. Recentemente, linguagens orientadas a objetos têm causado grande impacto na semântica do RPC, com um grande número de variações, como o RMI (Invocação Remota de Método). Este trabalho propõe mais uma variação. O objetivo deste trabalho é definir um mecanismo de invocação remota de métodos para a VIRTUOSI. A VIRTUOSI é um sistema computacional distribuído orientado a objetos que atua sobre máquinas virtuais cooperantes. A arquitetura é baseada em árvores de programas e tabelas de referências para facilitar a distribuição dos objetos. Além de definir um mecanismo de invocação remota de métodos, este trabalho se propõe a definir um serviço de nomes bem como o protocolo utilizado para troca de mensagens entre máquinas virtuais e serviço de nomes.

Palavras-chave: Invocação Remota de Métodos; Máquinas Virtuais; Sistemas Distribuídos; Serviço de Nomes.

Abstract

RPC (Remote Procedure Call) has established itself as one of the more powerful communication paradigms for distributed computing. Recently, object-oriented languages have impacted RPC semantics, with a number of variants, like RMI (Remote Method Invocation). This work presents one more variant. The main goal of this work is to define a mechanism for remote method invocation. VIRTUOSI is an object-oriented computational system based on collaborative virtual machines. The architecture is based on program trees and reference tables that facilitate object distribution. Besides defining the mechanism of remote method invocation, this work considers defines a name service as well as a protocol used to exchange messages between virtual machines and the name service.

Keywords: Remote Method Invocation; Virtual Machine; Distributed System; Name Service.

Capítulo 1

Introdução

1.1 Motivação

Diversos estudos no campo da orientação a objetos e em sistemas distribuídos foram e estão sendo feitos com o objetivo de criar comunicações confiáveis e transparentes. Grande parte das propostas de sistemas distribuídos utilizam metodologias orientadas a objetos, pois, a metodologia proporciona benefícios como confiabilidade, os sistemas são projetados para executarem exatamente as tarefas para a qual ele foi projetado [Mey97], podem ser extensíveis, além de possuir algo grau de reusabilidade. A grande difusão da linguagem Java[1] deve-se ao fato dela ser orientada a objetos e principalmente devido à portabilidade de seu código que é executado em máquinas virtuais, em diversas plataformas. Um importante desafio, na área de sistemas distribuídos, que diversas tecnologias como RPC (*Remote Procedure Call*) [Bir84], CORBA (*Common Object Request Broker Architecture*) [OMG02b], Java RMI (*Remote Method Invocation*) [Inc99] dentre outras; é possibilitar que objetos, processos e máquinas virtuais distribuídas comuniquem-se de forma transparente.

A arquitetura VIRTUOSI proposta por [Cal00] também vem ao encontro desta necessidade. A VIRTUOSI corresponde a um ambiente de execução distribuída de sistemas de software orientados a objetos. Este ambiente é composto de máquinas virtuais cooperantes que são executas em computadores com sistemas operacionais e hardware padrão, embora heterogêneos, interconectados por uma rede. Cada máquina virtual hospeda uma parcela do universo de objetos e assegura que o comportamento de cada objeto esteja de acordo com a definição do correspondente tipo ou classe. Os objetos residem em um espaço único de endereçamento, como se cada sistema fosse único em execução a cada momento, sem concorrência entre sistemas ou entre partes de um mesmo sistema que executam em paralelo. A arquitetura inclui o conceito de reflexão computacional, que serve como meio para a concretização de diversos mecanismos do ambiente de execução e que permitem a aplicação redefinir o comportamento desse ambiente, assim como o seu próprio comportamento. Além disso, a arquitetura tem como característica a utilização de árvores de programa [KF99] para representação de classes e métodos, sendo que os acessos as classes, métodos e objetos são feitos de forma indireta através de tabelas de referência [Cox03]. Embora existam diversos mecanismos de comunicação entre objetos

[1] A tecnologia Java foi desenvolvida pela Sun Microsystems o seu lançamento oficial ocorreu em Maio de 1995

remotos, não existe ainda, nenhuma solução aplicada ao ambiente VIRTUOSI que forneça comunicação entre os objetos de forma simples, transparente e que trate possíveis exceções de execução.

1.2 Objetivos

RPC (*Remote Procedure Call*) tornou-se um dos mais poderosos paradigmas de comunicação para sistemas distribuídos. Recentemente, linguagens orientadas a objetos têm causado grande impacto na semântica do RPC, com um grande número de variações. Este trabalho propõe mais uma variação com o objetivo de prover um mecanismo de comunicação entre objetos distribuídos no ambiente da VIRTUOSI. O mecanismo será implementado utilizando algumas técnicas de chamada de procedimentos remota (RPC). O objetivo principal do mecanismo de comunicação está relacionado à transparência. Esta transparência deve ser implementada de tal maneira que pareça esconder do usuário (programador) a separação dos objetos distribuídos, de tal forma que o sistema seja entendido com centralizado. O enfoque principal em relação à transparência foi dado à transparência de acesso e localização. A transparência de acesso e localização são importantes para o programador, pois, as chamadas de métodos remotos devem ser realizadas como se fossem locais, proporcionando maior conforto e agilidade no desenvolvimento de sistemas.

Para possibilitar a transparência de acesso aos objetos e também para facilitar a distribuição dos objetos, empregou-se neste trabalho o conceito de tabelas de referências, conforme [Cox03]. O conceito de tabela de referências foi utilizado não somente para acesso aos objetos como também para acesso às definições de classes e invocáveis. Em relação à transparência de localização, o objetivo é projetar e implementar um serviço de nomes destinado ao registro, remoção e localização de máquinas virtuais VIRTUOSI. O serviço de nomes somente será aplicado para a localização de máquinas virtuais, uma vez que, a localização dos objetos será provida através das tabelas de referências, conforme abordado na seção 4.3.

Nota-se, atualmente, uma tendência crescente para o emprego e desenvolvimento de ferramentas de desenvolvimento, para fins comerciais, ou seja, com uma perspectiva comercial. Por outro lado, existem as perspectivas pedagógica e experimental (simulação como [LPZ+05]). A perspectiva pedagógica se opõe à perspectiva comercial no sentido de proporcionar ao usuário (programador) conceitos teóricos e qualitativos, do que simplesmente a busca de alto desempenho para execução de programas, sem nenhuma conceituação.

Assim a VIRTUOSI, cujo enfoque é pedagógico e experimental, objetiva-se a desenvolver uma ferramenta para o desenvolvimento de software, onde o usuário (programador) possa aprender computação distribuída – perspectiva pedagógica –, e, ao mesmo tempo, construir sistemas de software distribuído de qualidade através da prototipação – perspectiva experimental. Conseqüentemente, a construção do mecanismo de invocação remota de métodos, contribuirá para a fundamentação do ambiente VIRTUOSI e, servirá de subsídio para trabalhos futuros correlatos a área de distribuição e comunicação entre objetos remotos para SCD (*Sistemas Computacionais Distribuídos*).

Em síntese, pode-se dizer que o objetivo geral deste trabalho é projetar um mecanismo de invocação remota de métodos, bem como especificar o protocolo utilizado para comunicação entre objetos remotos; além disso, projetar e implementar um serviço de nomes para as máquinas virtuais e o seu protocolo de comunicação. E o objetivo específico é implementar estes mecanismos no ambiente VIRTUOSI.

1.3 Problemática

Normalmente o desenvolvimento de sistemas distribuídos torna-se muito difícil pelo fato de existir uma grande necessidade de conhecimentos técnicos por parte do desenvolvedor de sistemas. Um importante fator na computação distribuída é a transparência. Diversos mecanismos que implementam RPC buscam obter esta transparência, porém acabam falhando no aspecto da complexidade que é criada para o desenvolvedor. Esta complexidade traz consigo um aumento no tempo de desenvolvimento, assim como dificuldades de identificar falhas no sistema com um todo.

Para o desenvolvimento deste trabalho pesquisou-se alguns mecanismos de invocação remota de métodos. A arquitetura VIRTUOSI mostrou-se bastante diferente em importante aspectos em relação aos mecanismos pesquisados. Aplicação dos conceitos de árvores de programas e de tabelas de referências tanto para as próprias árvores como para os objetos instanciados, geram facilidades por serem projetados para prover uma arquitetura totalmente voltada para a distribuição de sistemas. A VIRTUOSI por possuir o enfoque pedagógico e experimental [CNdC+04] influenciou nos requisitos a serem atendidos pelo mecanismo.

1.4 Delimitação do Tema

Este trabalho propõe-se ao estudo e a implementação de um mecanismo para invocação remota de métodos que atenda a arquitetura VIRTUOSI, garantindo que o processo de invocação remota atenda os requisitos estabelecidos pelo sistema. Outro ponto a ser atendido é o estudo e implementação de um serviço de nomes para as máquinas virtuais. O trabalho não contempla definições de políticas de segurança, nem de tratamento de falhas e nem de desempenho.

1.5 Organização do Trabalho

O capítulo 2 apresenta o contexto atual sobre chamadas de procedimento remoto, como também o funcionamento dos serviços de nomes apresentando uma visão do processo de implementação e utilização dos mecanismos de chamada de procedimento remoto.

No capítulo 3 são descritos a arquitetura VIRTUOSI e seus diversos aspectos: metamodelo, árvores de programa, objetos, atividades, classes e máquina virtual.

O capítulo 4 apresenta a proposta para o mecanismo de invocação remota de métodos para a arquitetura VIRTUOSI, serviço de nomes, protocolo utilizado para comunicação e também diversos cenários de invocação remota de métodos.

São apresentados alguns cenários referentes à implementação do mecanismo de invocação remota de métodos no capítulo 5, os casos apresentados variam de acordo a invocação do método remoto, como passagem de parâmetros e retorno de valores. E, finalmente, o capítulo 6 apresenta a conclusão deste trabalho e também uma proposta para trabalhos futuros para o tema de sistemas distribuídos orientados a objetos.

Capítulo 2

Mecanismos de Comunicação Remota

Será apresentado neste capítulo alguns trabalhos relacionados com o mecanismo de invocação remota de procedimentos e métodos. Dentre estes trabalhos destacamos o RPC (*Remote Procedure Call*), CORBA (*Common Object Request Broker Architecture*) e Java RMI (*Remote Method Invocation*); e é apresentado também uma descrição funcional dos serviços de nomes destes mecanismos.

As implementações de RPC como Sun RPC, DCE RPC e MS RPC utilizam uma linguagem de definição de interface, com sintaxe semelhante a linguagem de programação C, e basicamente a divisão das aplicações eram baseadas entre módulos cliente e servidor. Com a introdução da linguagem de programação orientada a objetos, a distribuição dos programas passa a ser da granularidade de objetos.

A programação orientada a objetos é provavelmente um dos paradigmas mais aceitos para computação distribuída. A orientação a objetos foi introduzida primeiramente através da linguagem Simula 67 [DMN70]. Conforme [MMP88], Simula 67 já provia algum suporte para simular logicamente programas distribuídos. Uma outra importante linguagem orientada a objetos é a Smalltalk [GR83], Smalltalk distribuído é por proposto por [Ben87], cujo objetivo é prover comunicação e interação entre usuários remotos Smalltalk, acessar objetos remotos, e criar aplicações distribuídas em um ambiente Smalltalk. As principais características do Smalltalk distribuído são:

- Invocação remota transparente para o usuário;
- Criação automática de referências de retorno para argumentos de mensagens remotas;
- Suporte para depuração de processos remotos;
- Suporte para controle de acesso remoto;
- Objetos remotos reativos;
- Suporte para mobilidade de objetos.

Smalltalk distribuído foi construído com o objetivo de ser um mecanismo básico para comunicação entre objetos remotos, cujas mensagens são transmitidas de forma transparente para os objetos remotos.

Mais recentemente temos implementações como o padrão CORBA e Java RMI, as quais serão apresentadas nas próximas seções, provêem também mecanismos de invocação de métodos de objetos remotos que funcionam de forma transparente, no que diz respeito a localização dos objetos. No entanto, antes de abordar estes mecanismos mais recentes será apresentado o mecanismo RPC.

2.1 RPC - Remote Procedure Call

A idéia básica do RPC surgiu a partir do trabalho de Nelson na década de 80 [Bir84]. A concepção do RPC foi criada com base na observação do funcionamento do mecanismo de transferência de dados e controle dentro um programa em um simples computador. A partir desta observação procurou-se estender esta mesma funcionalidade para um ambiente distribuído, ou seja, a partir da implementação deste mecanismo seria possível chamar procedimentos localizados em outras máquinas. Em outras palavras, a idéia por trás do RPC é fazer com que uma chamada de procedimento remoto se pareça como uma chamada local, tornando-se o mais transparente possível.

Assim, pode-se dizer que o RPC tem como objetivo a invocação de procedimentos remotos evitando que os programadores de sistemas distribuídos se preocupem com detalhes de interface com a rede[SM95]. Idealmente, o mecanismo de RPC deve ser totalmente transparente para o programador, no sentido em que ele desenvolverá os módulos cliente e servidor como se fossem estar residentes no mesmo nó. O módulo cliente não é ligado diretamente ao módulo servidor, mas mais propriamente a um *stub*[1] cliente que atua com uma representação para o servidor e ativa um mecanismo genérico de RPC [LI86]. Por outro lado, o servidor representa o cliente no lado do servidor.

De forma sumária, quando uma chamada de procedimento remoto é invocada, o processo invocador é suspenso, os parâmetros são passados pela rede com destino da máquina onde o procedimento será efetivamente executado. Quando o procedimento invocado termina, o resultado é retornado para o ambiente invocador, e o processo invocador retoma sua execução normal.

Embora o conceito pareça simples, existem algumas questões que devem ser analisadas, como por exemplo, os procedimentos são executados em endereços de memória diferentes, parâmetros e retornos de funções podem ser complicados de serem tratados principalmente se a comunicação for entre máquinas diferentes [Tan95]. Outro ponto que deve ser levado em conta são os problemas que podem acontecer em ambos computadores que podem resultar em possíveis falhas causando diferentes problemas. Mas ainda assim o RPC é uma técnica amplamente usada como base de muitos sistemas distribuídos.

[1] Corresponde a uma rotina a qual ela mesma não executa nenhuma ação, ela somente declara a si mesma e os parâmetros aceitos.

2.1.1 Funcionamento

O funcionamento de um programa que utiliza RPC está baseado no conceito de *stubs*[2]. Para realizarmos uma chamada de procedimento remoto são envolvidas cinco partes de um programa: o usuário, o *user-stub*, o *server-stub*, servidor e o pacote utilizado nas comunicações[Bir84]. Este pacote utilizado nas comunicações, chamado de *RPCRuntime*[3], é responsável pela transmissão confiável dos pacotes, os quais contém informações sobre a chamada de procedimento remoto, como especificações do procedimento a ser invocado e argumentos.

Para que um cliente execute uma chamada de procedimento remoto, o *client stub* empacota o parâmetro em uma mensagem e solicita para o *kernel* enviar a mensagem para o servidor. Quando a mensagem chega ao servidor, o *kernel* passa a mensagem para o *server stub* que desempacota o parâmetro e então chama o procedimento do servidor. O servidor executa seu trabalho e então retorna o resultado para o cliente seguindo o processo inverso, ou seja, o *server stub* empacota o resultado do procedimento e solicita ao *kernel* para enviar uma mensagem de retorno para o cliente. Quando a mensagem chega no cliente, o *kernel* verifica que a mensagem é endereçado para um determinado processo do cliente, o *client stub* desempacota a mensagem e, o processo do cliente é desbloqueado, continuando sua execução normal.

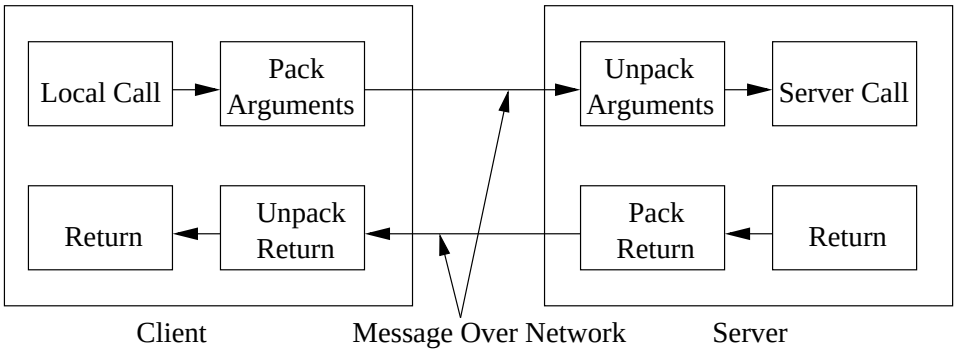


Figura 2.1: Chamadas e mensagens em um RPC.

Graficamente o processo para execução de uma chamada de procedimento remoto, ocorre como exemplificado na figura 2.1. Os passos citados a seguir resumem os acontecimentos de uma chamada de procedimento remota, basicamente ocorrem dez passos [Tan95]:

1. O cliente chama o procedimento *client stub* de maneira normal.
2. O *client stub* constrói a mensagem e encaminha para o *kernel*.
3. O *kernel* envia a mensagem para o *kernel* remoto.

[2]Parte de um software utilizada para substituir outros códigos, geralmente situados em outro computador ou em outro endereço de memória

[3]Parte padrão do sistema Cedar citado por Birrel

4. O *kernel* remoto repassa a mensagem para o *server stub*.
5. O *server stub* desempacota os parâmetros e chama o servidor.
6. O servidor executa o trabalho e retorna o resultado para o *stub*.
7. O *server stub* empacota o resultado em uma mensagem e encaminha para o *kernel*.
8. O *kernel* remoto envia a mensagem para o *kernel* do cliente.
9. O *kernel* do cliente repassa a mensagem para o *client stub*.
10. O *stub* desempacota o resultado e retorna para o cliente.

A partir da exposição dos príncipios básicos do funcionamento do RPC, será apresentado na próxima subseção, o processo de ligação entre um cliente que almeja invocar um procedimento remoto, e um servidor que disponibiliza o procedimento remoto procurado.

2.1.2 Binding

O processo de ligação[4] está relacionado com o relacionamento entre cliente e servidor. A ligação inclui informações que associam a invocação de um cliente de um RPC com a implementação de um servidor. O servidor é responsável por implementar a interface do procedimento disponibilizado. Para que este processo ocorra é necessário localizar a interface do procedimento desejado. Chama-se este processo de localização de *naming* e *location*. Existem duas partes envolvidas importantes em uma interface, o tipo e a instância. O tipo tem o objetivo de especificar um certo nível de abstração, que o cliente espera que o servidor implemente. A instância especifica a implementação da interface abstrata procurada.

Quando um importador deseja ligar com um exportador de interfaces, o código do usuário chama o *user-stub* que chama o procedimento através do *RPCRuntime* que retorna o tipo da interface desejada e sua instância. O *RPCRuntime* retorna o endereço de rede do exportador (servidor). Então executa a chamada de procedimento remoto, se a máquina remota não possui a interface procurada ocorre um erro de *binding*.

2.1.3 Protocolo de Comunicação

A comunicação em uma chamada de procedimento remoto é realizada através de pacotes, nestes pacotes contém o identificador de chamada, os dados especificando o procedimento chamado e os argumentos do procedimento. Quando o servidor recebe o pacote, o servidor executa o procedimento informado. Após a execução do procedimento, o servidor enviará um pacote com o resultado do procedimento que conterá o mesmo identificador de chamada.

A máquina que transmite o pacote é responsável por retransmitir até que um aviso de recebimento seja recebido, isto é executado para compensar a perda de pacotes.

[4]Do inglês *binding*

O identificador de chamada permite que o cliente determine se o resultado do pacote é verdadeiramente o resultado do procedimento que está sendo executado no momento e também permite ao servidor eliminar pacotes duplicados. O identificador é composto pela identificação da máquina cliente, identificador do processo e um número seqüencial. O conjunto do identificador da máquina mais o do processo é denominada atividade. Uma importante propriedade da atividade é que cada atividade possui ao menos uma chamada de procedimento remota em qualquer tempo, e não é iniciada nenhuma nova atividade até que se tenha recebido o resultado de uma chamada anterior.

Entende-se por conexão, o estado compartilhado entre uma atividade e o pacote *RPCRuntime* na máquina servidora que aceita as chamadas. Não existe necessidade de nenhum protocolo especial para estabelecer uma conexão, para receber um pacote de chamada de uma atividade desconhecida basta apenas criar uma conexão.

O cliente que transmite um pacote é também responsável pela retransmissão do pacote até que o mesmo receba uma confirmação de recebimento, para que isto ocorra, o pacote é modificado para requisição de uma explícita confirmação. Esta técnica é utilizada para manipular perdas de pacote, chamadas de longa duração e grandes intervalos entre chamadas. Após o cliente receber a notificação de recebimento, o cliente fica aguardando o retorno do resultado da chamada. Enquanto aguarda-se o resultado o cliente envia periodicamente pacotes para verificar se ocorreu algum problema com o servidor ou se ocorreu algum problema de comunicação.

Quando os argumentos de uma chamada são muito grandes para serem inseridos em um único pacote, eles são enviados em múltiplos pacotes, sendo que cada um deles necessita de uma confirmação de recebimento, cada pacote possui um número seqüencial para sua identificação.

O protocolo RPC não se preocupa como as mensagens são passadas, mas sim como as mensagens são especificadas e interpretadas [Sri95]. O protocolo RPC não implementa nenhum tipo de confiabilidade, se por exemplo o protocolo estiver sendo executado em cima do TCP, o RPC não se preocupa com garantias de confiabilidade, uma vez que o próprio TCP propricia esta confiabilidade. Por outro lado, se o protocolo RPC estiver sendo executado em cima do UDP (*User Datagram Protocol*), que é uma camada de transporte não confiável, será necessário a implementação própria de controle de tempo excedido[5], retransmissões, políticas de detecção de duplicidades que não são implementadas pelo protocolo RPC.

2.1.4 Marshalling

Marshalling corresponde ao procedimento executado para conversão dos dados para apresentação de acordo com algumas regras [DKS97]. Alguns fatores devem ser considerados na execução do *marshalling* para apresentar os dados. Diferentes plataformas utilizam seus próprios formatos de caracteres, por exemplo, mainframes IBM utilizam EBCDIC (*Extended Binary Code Decimal Interchange Code*), enquanto os computadores pessoais IBM utilizam ASCII (*American Standard Code for Information Interchange*). Um pro-

[5] Do inglês timeout

blema similar pode ocorrer com a representação dos números inteiros (complemento de 1 versus complemento de 2), e especialmente com números com ponto flutuante. Em computadores como no Intel 486 os números são representados da direita para a esquerda (*Little Endian*), ao contrário o Sun SPARC, seus números são representados da esquerda para a direita (*Big Endian*).

É inevitável que ocorra gastos com processamento das conversões dos dados em sistemas distribuídos.

O *marshalling* pode ser a parte mais custosa em uma comunicação em rede, particularmente quando o tempo necessário para passar pela rede possui um tempo menor do que o necessário para executar o *marshalling*.

Em resumo, o procedimento de *marshalling* deve ter informações sobre o formato de dados para a plataforma atual, convertê-la para algum formato padrão usado na transferência pela rede, obter os dados pela rede e ser capaz de decodificá-los de volta a partir de um formato padrão de rede para a sua plataforma. Desta forma, o procedimento de *marshalling* inclui dois processos mútuos reversos – codificar e decodificar. Conforme [Bir97], pode-se dizer que o mecanismo de *marshalling* de dados tem o propósito de representar os argumentos do cliente em uma forma que o programa servidor possa eficientemente interpretá-los.

2.2 Serviço de Nomes

Um serviço de nomes mantém um ou mais mapeamentos a partir de uma forma de nome – normalmente simbólico – para alguma forma de valor – normalmente um endereço de rede[Bir97]. O serviço de nomes pode atuar de diversas formas, por exemplo utilizando DNS (*Domain Naming Service*) para mapear o serviço de nomes para um conjunto de mapeamentos utilizando o protocolo TCP (*Transmission Control Protocol*)/IP (*Internet Protocol*), ou mapeamentos ASCII onde se mapeia o endereço IP e um número de porta, neste caso as informações de endereço e a porta devem ser exatas.

2.2.1 Serviço de Nomes no OSF DCE

O serviço de nomes no OSF (*Open Software Foundation*) DCE (*Distributed Computing Environment*) tem a finalidade de atuar como interface de ligação[6] entre o que as aplicações utilizam para exportar e importar informações de ligação. Este serviço é projetado para ser independente, ou seja, não é necessário que exista nenhuma aplicação para o seu funcionamento.

Conforme [Gro97] a interface do serviço de nomes têm duas pretensões principais:

- O serviço de nomes mantém um banco de dados de nomes, as suas entradas são acessadas via nomes com alguma sintaxe específica.
- As entradas do serviço de nomes prestam suporte a um conjunto de atributos específicos RPC, cuja interface do serviço de nomes é utilizada para exportar, pesquisar

[6] Do inglês binding

ou para importar informações de ligações.

Esta interface permite acessar e gerenciar os dados armazenados em um diretório global que suporta um espaço de nomes[7] hierárquico e atributos com multi-valores. Conforme [Die92] as entradas do servidor são classificadas no servidor, em grupos de servidores, e perfis de entrada. Os atributos mandatários de uma entrada de servidor inclui um identificador de interface único, uma versão de interface, uma representação de dados, versão do protocolo RPC e um manipulador binário para o servidor.

Um grupo de servidores representa uma coleção de servidores, considera-se que um perfil corresponde a uma referência para os servidores ou grupo de servidores. Através da especificação de um perfil para execução de operações de importação ou procura e um caminho para o banco de dados de nomes, é possível pesquisar uma entrada de servidor específica.

O serviço de nomes RPC provê alguns serviços, como:

- Operações para manipular ligações (e.g., exportar, importar, procurar). Nenhuma destas operações implica em comunicação entre cliente e servidor.
- Serviços para recuperar informações sobre entradas em geral (e.g., interfaces e objetos exportados pelo servidor).
- Operações nos grupos de servidores (e.g., criar, remover e adicionar membros).
- Operações nos perfis.
- Interfaces para gerenciamento de aplicações para criar ou remover entradas nos serviços de nomes.

2.2.2 Serviço de Nomes para o RMI

O *rmiregistry* é uma ferramenta que acompanha a máquina virtual Java, o objetivo da ferramenta é de iniciar o registro de um objeto remoto Java em uma porta específica no *host*[8] em que está sendo executada a ferramenta.

A porta de comunicação poderá ser especificada ou omitida, caso seja omitida, a porta padrão que será utilizada é a 1099. O *rmiregistry* não produz nenhuma saída[9] e, tipicamente é executado em *background*.

Um registro de um objeto remoto corresponde a um *bootstrap*[10] do serviço de nomes que é utilizado pelo servidor RMI no mesmo *host* onde ocorrerá as ligações entre os objetos remotos e os nomes.

O registro é usado tipicamente para localizar o primeiro objeto remoto a qual a aplicação necessita para invocar os métodos. Este objeto proverá o suporte específico para aplicação encontrar outros objetos.

[7] Do inglês namespace

[8] Corresponde a um computador que é acessado por um usuário trabalhando em uma localização remota.

[9] Do inglês output

[10] Auto-carregador de conjunto de instruções que serão executadas pelo computador

Os métodos da classe *java.rmi.registry.LocateRegistry* são usados para conseguir o registro no *host* local, ou no *host* local e porta.

Já os métodos da classe *java.rmi.Naming* baseadas em URL (*Uniform Resource Locator*) operam no registro e podem ser usados para procurar um objeto remoto em qualquer *host* e, um *host* local: ligação entre um simples nome e um objeto remoto, religação entre um novo nome e o objeto remoto (sobrescrevendo a ligação antiga), desligar o objeto remoto, e listar as URLs "saltadas"no registro.

Pode-se dizer que o *rmiregistry* é uma forma simplificada de serviço de nomes que permite aos clientes obter uma referência (um *stub*) para um objeto remoto. Em geral o registro é utilizado somente para localizar o primeiro objeto remoto que um cliente necessita.

```
Registry registry = LocateRegistry.getRegistry();
registry.bind("Banco", stub);
    ...
```

Figura 2.2: Trecho de código fonte relativo a obtenção de referência do registro

Conforme código mostrado na figura 2.2, o servidor obtém um *stub* para um registro no *host* local e a porta de registro padrão e, então usa o *stub* de registro para ligar com o nome "Banco"com o *stub* do objeto remoto.

O método estático *LocateRegistry.getRegistry* não possui parâmetros e retorna um *stub* que implementa a interface remota *java.rmi.registry.Registry* e envia as invocações para o registro do *host* local do servidor na porta default 1099. O método *bind* é invocado no *stub registry* com objetivo de ligar o *stub* do objeto remoto com o nome "Banco"no registro.

2.2.3 Serviço de Nomes no CORBA

O serviço de nomes no CORBA não trabalha como IDL (*Interface Definition Language*), o serviço de nomes permite aos usuários navegar e o acessar os objetos diretamente, e simultaneamente permite ao cliente e ao programador de objetos passar a referência de um objeto dentro do escopo de uma empresa. A chave para o sucesso no serviço de nomes CORBA é a interoperabilidade[Joh00], isto deve-se ao fato que objetos de um domínio poder resolver nomes de outros objetos em outros domínios, com o objetivo de invocar os seus métodos.

O serviço de nomes especifica como os nomes são armazenados dentro do programa, não provê nenhum formato padrão quando vistos de fora. Isto significa que não seria possível enviar este nome para um amigo ou colaborador e esperar que ele use o literal a menos que ambos tenham o mesmo ORB (*Object Request Broker*) e que tenham usado o mesmo código.

A unidade básica da estrutura de nomes dos objetos CORBA é contexto de no-

mes[11], pode-se entender o contexto de nomes como um subdiretório para nomes. Um contexto de nomes tem seu próprio nome e pode conter muitas entradas. Cada entrada tem um nome e pode ser um outro contexto de nomes ou um objeto CORBA.

Pode-se dizer que um contexto de nomes é um objeto que contém um conjunto de nomes de ligações a qual cada nome é único. Podem ser encontrados nomes diferentes para um objeto, no mesmo ou diferentes contextos ao mesmo tempo. Não existe requerimento, no entanto, de que todos os objetos precisem ser nomeados.

O contexto de nomes são objetos CORBA e são identificados por uma referência a um objeto CORBA. Pode-se registrar um contexto de nomes no serviço de nomes até se ele residir em outro serviço de nomes em outro computador.

Devido o contexto ser como qualquer outro objeto, é possível ligá-lo a um outro contexto de nomes. Conforme [OMG02a], o contexto de ligações em outros contextos cria um grafo de nomes – um grafo dirigido com nós e extremidades nomeadas onde os nós são contextos. Um grafo de nomes permite nomes mais complexos para referenciar um objeto. Dado um contexto em grafo de nomes, a seqüência de nomes pode referenciar um objeto. A seqüência de nomes define um caminho no grafo de nomes permitindo a navegação com o objetivo de resolver os nomes, conforme observado na figura 2.3.

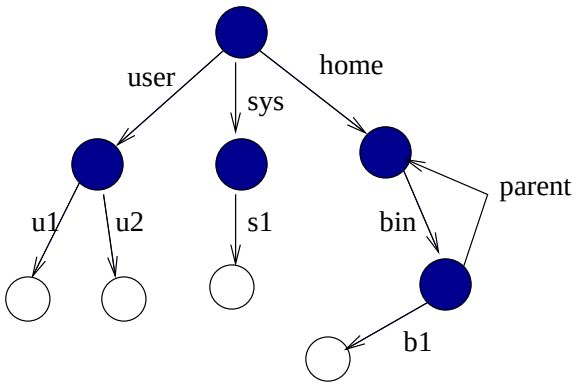


Figura 2.3: Grafo de Nomes

Um componente nome – seja ele um contexto de nomes ou o nome do objeto – consiste de um identificador ou ID e um tipo[12]. Ambos atributos ID e tipo são representados como *strings IDL*. A figura 2.4 mostra a estrutura de um componente nome.

```
struct NameComponent
{
    Istring Id;
    Istring kind;
};
```

Figura 2.4: Estrutura de um componente nome

[11] Do inglês naming context
[12] Do inglês kind

O conjunto de interfaces pertencentes ao módulo *CosNaming* define o serviço de nomes no CORBA. O módulo contém três interfaces, *NamingContext, BindingIterator e NamingContextExt* (conforme [OMG02a]).

2.3 Handle Table

A proposta relativa à utilização da *handle table* foi abordada no sistema DOSA (*Distributed Object System Architecture*). Adotou-se na VIRTUOSI apenas o conceito principal da utilização da *handle table*, uma vez que a VIRTUOSI não constitui uma arquitetura voltada para distribuição de memória compartilhada. A implementação de um sistema baseado em tabelas de referências possibilita o compartilhamento de dados de forma eficiente e transparente. Conforme [HYC+03], no sistema DOSA que utiliza tabela de referências, os objetos são re-direcionados através de um *handle* para o objeto, ou seja, não existe referência direta para o objeto. Não é permitido nenhuma referência direta para qualquer objeto, somente através de uma entrada da tabela. Cada objeto possui um identificador único que também serve como índice na tabela de referências. Quando se trabalha com a tabela de referências, obtém-se a segurança de que nenhum atributo do objeto pode ser acessado sem ter uma referência e, toda referência para o objeto deve ser obtida através da tabela. A estrutura da tabela de referência é composta pelo identificador do objeto, denominado *entry* que serve como índice da tabela de referência, outro campo denominado *local reference* que indica a referência para o objeto local, o campo *remote reference* que corresponde a um campo composto da identificação da VM remota mais o índice da handle table remota, o campo denominado *fix* que indica se um objeto pode ser migrado e, finalmente o campo *invalidade* indica se objeto pode ser acessado no momento da invocação.

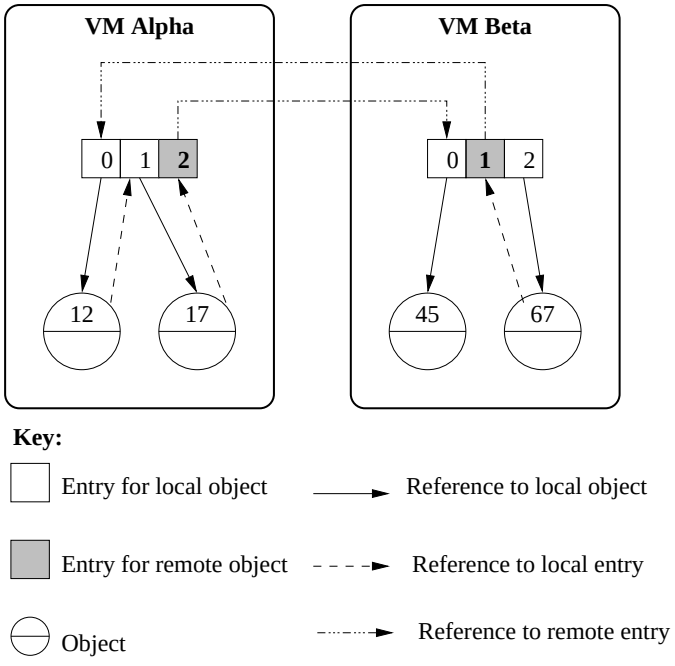


Figura 2.5: Handle Table

A figura 2.5 mostra como ocorre a referência para um objeto local ou remoto, através da tabela de referências. A *VM Alpha* apresenta uma tabela de referências com três índices. O índice 0 referencia o objeto 12, que por sua vez, possui uma relação com o objeto 17, que é referenciado pelo índice 1 da tabela de referências. O objeto 17 possui uma referência com o objeto 45, esta referência é remota e, é realizada através do índice 2 da tabela de referências. O índice 2 da tabela de referências da *VM Alpha* referencia o índice 0 da tabela de referências da *VM Beta*, pode-se notar então a ocorrência de referências remotas entre objetos.

2.4 CORBA - Common Object Request Broker Architecture

O OMG (*Object Management Group*) é um consórcio internacional que direciona as padronizações na área da computação de objetos distribuídos. Um dos padrões desenvolvidos pela OMG é o CORBA – Common Object Request Broker Architecture. O núcleo da arquitetura CORBA é o ORB (*Object Request Broker*) que atua como um barramento de objetos, estes objetos interage com outros objetos de forma transparente tanto localmente ou remotamente [CHY+97]. Um objeto CORBA é apresentado para o mundo exterior por uma interface com um conjunto de métodos. Um instância particular de um objeto é identificado por uma referência a um objeto. Um cliente de um objeto CORBA obtém a referência do objeto e utiliza para fazer as chamadas dos métodos, como se o objeto estivesse localizado no mesmo endereço do cliente. O ORB é responsável por todos os mecanismos requeridos para encontrar a implementação do objeto, prepará-lo para receber as requisições, comunicar as requisições recebidas, e transportar o retorno, caso houver, de volta para o cliente. A implementação do objeto interage com o ORB através do OA (*Object Adapter*) ou através da interface ORB.

Como todas as tecnologias, CORBA tem uma terminologia única associada a ela, a maioria dos termos e conceitos são similares as tecnologias existentes no mercado, porém também existe novos ou até mesmo termos e conceitos diferentes. Estes termos e conceitos, citados a seguir, é baseado em [HV99].

Objeto CORBA: é uma entidade virtual capaz de ser localizado por um ORB e também pode ser invocado através de requisições de clientes. É virtual no sentido em que o objeto não existe realmente, a menos que seja feita uma implementação concreta em uma linguagem de programação.

Objeto Alvo: dentro do contexto das invocações de requisições CORBA, corresponde a um objeto CORBA que é alvo de alguma requisição. O modelo de objeto CORBA é um modelo *single-dispatching* o qual o objeto alvo para requisições é determinado unicamente por uma referência a um objeto usado para invocar a requisição.

Cliente: corresponde a uma entidade que invoca uma requisição para um objeto

CORBA. Um cliente pode existir em um endereço que é completamente diferente do objeto CORBA, ou o cliente e o objeto CORBA podem existir dentro da mesma aplicação. O termo cliente tem significado somente dentro do contexto de uma requisição em particular, isto porque, a aplicação que é cliente para uma requisição pode ser servidor para outra requisição.

Servidor: é uma aplicação em que existe um ou mais objetos CORBA. Assim como dito no termo cliente, o termo servidor somente faz sentido em um contexto particular.

Requisição: corresponde a uma invocação de uma operação em um objeto CORBA efetuada por um cliente. A requisição é feita a partir do cliente para o objeto alvo no servidor e, o objeto alvo envia o resultado de volta na resposta caso a requisição requeira.

Referência de objeto: é um manipulador[13] utilizado para identificar, localizar, e endereçar um objeto CORBA. Para os clientes, a referência de objeto são entidades opacas. Os clientes usam as referências de objetos para direcionar as requisições para os objetos, mas eles não podem criar referências de objetos a partir dos seus elementos e nem podem acessar ou modificar o conteúdo de uma referência de objeto. Uma referência de objeto refere-se somente a um simples objeto.

***Servant*:** corresponde a uma entidade de linguagem de programação que implementa um ou mais objetos CORBA. *Servants*[14] são ditos como objetos CORBA encarnados porque eles encorpam, ou implementam estes objetos. Os *servants* existem dentro do contexto de uma aplicação servidor. Em C++, *servants* são instâncias de objetos de uma classe em particular.

2.4.1 ORB – Object Request Broker

O ORB é responsável por todo o mecanismo requerido para encontrar a implementação do objeto em uma requisição, preparar a implementação do objeto para receber a requisição, e comunicar os dados que fazem parte da requisição. A interface que o cliente vê é completamente independente da localização do objeto, da linguagem de programação, e qualquer outro aspecto que não é refletido na interface do objeto.

Um ORB possibilita a criação, recepção de requisições e respostas de forma transparente em ambientes distribuídos. Seu fundamento consiste em construir aplicações a partir de objetos distribuídos, e criar interoperabilidade entre aplicações em ambientes tanto homogêneos como heterogêneos.

A figura 2.6 mostra a estrutura de um ORB individual. As interfaces para o ORB são mostradas através das caixas listradas, e as setas indicam se o ORB está sendo chamado ou está executando uma chamada para o objeto de implementação através da

[13] Do inglês handle
[14] Em português empregado

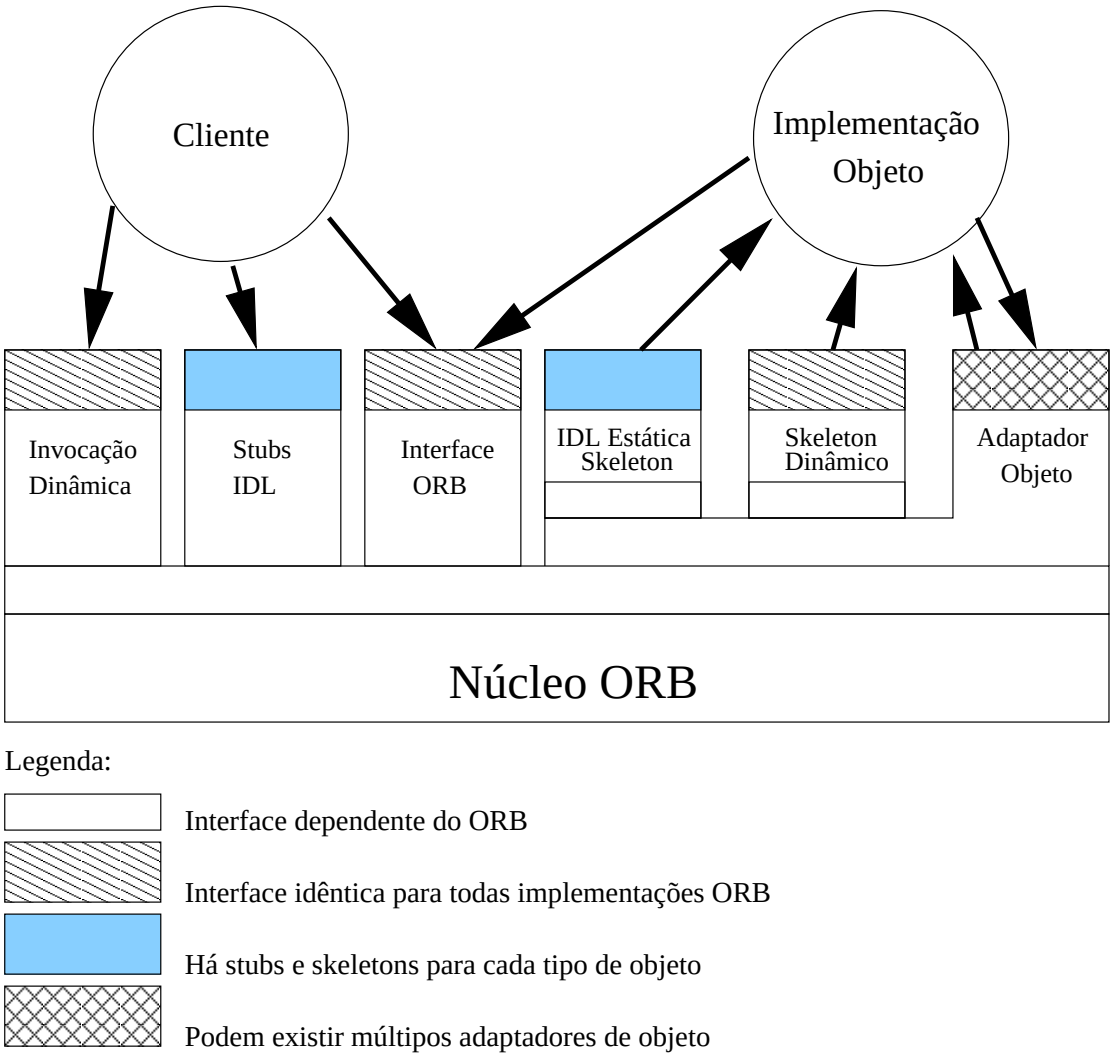


Figura 2.6: Estrutura das interfaces de requisições de objetos.

interface.

Para fazer uma requisição, o cliente usa a interface de invocação dinâmica (a mesma interface independente da interface do objeto alvo) ou um *stub*OMG IDL. O cliente também pode diretamente interagir com o ORB através de algumas funções.

A implementação do objeto recebe a requisição através do *skeleton* gerado pelo OMG IDL ou através de um *skeleton* dinâmico. A implementação do objeto pode chamar o adaptador de objeto e o ORB enquanto está processando a requisição.

As definições de interfaces dos objetos podem ser obtidas de duas maneiras: podem ser definidas estaticamente através de uma linguagem de definição de interface, chamada OMG IDL. Ou alternativamente, as interfaces podem ser adicionadas em um serviço de repositório de interfaces, o qual representa os componentes de uma interface de objetos, permitindo ter acesso a estes componentes em tempo de execução.

Não é necessário que um ORB seja implementado através de um único componente, mas de preferência ele deve ser definido através de interfaces. Qualquer implementação de ORB que forneça interfaces apropriadas é aceitável, as interfaces são organizadas em

três categorias:

1. Operações idênticas para todas as implementações ORB.
2. Operações específicas para tipos de objetos particulares.
3. Operações específicas para estilos de implementações de objetos particulares.

ORBs diferentes podem fazer escolhas de implementação totalmente diferentes, e, junto com os compiladores IDL, repositórios, e vários adaptadores de objetos, proverem um conjunto de serviços para os clientes e implementações de objetos que possuam diferentes propriedades e qualidades.

Podem existir múltiplas implementações de ORBs, que possuam diferentes representações para referências de objetos e significados diferentes para invocação de execução. É possível um cliente acessar duas referências de objetos gerenciadas por diferentes implementações de ORB.

O núcleo ORB é o local do ORB que provê a representação básica das requisições dos objetos e comunicação. O CORBA é projetado para dar suporte a diferentes mecanismos de objetos, e estruturar o ORB com os componentes que ficam acima do núcleo ORB, os quais provêem interface que podem mascarar as diferenças entre os núcleos ORBs.

2.4.2 Características

A palavra chave no CORBA é a transparência [Rui00]. Existem dois tipos de transparência: uma referente a localização e outra referente a linguagem de programação. A transparência de localização ocorre quando um cliente efetua uma chamada de método remoto; e o objeto remoto chamado parece estar localizado no mesmo espaço de endereçamento do cliente. A transparência em relação a linguagem de programação indica que o objeto chamado pode ser implementado em qualquer linguagem sem que o cliente tenha o menor conhecimento.

A transparência de localização é implementada via *stub*, o qual é produzido através de um compilador IDL. O *stub* é um representante[15] do cliente, e é responsabilidade do cliente encontrar a localização do objeto chamado e passar os parâmetros e os resultados para o ORB.

A transparência de linguagem é obtida através do uso de mapeamento de linguagem, o qual também é realizado pelo compilador IDL que gera os tipos de dados correspondentes de programação.

Similarmente, um *skeleton* e outro mapeamento de linguagem são produzidos no lado do objeto, por um compilador totalmente diferente da IDL. Este *skeleton* faz justamente o oposto do que o código *stub* faz, como por exemplo, desmontar[16] os parâmetros.

Outra característica importante em CORBA se refere ao fluxo de requisições. Na figura 2.7, a aplicação cliente faz uma requisição e, a aplicação servidora recebe a requisição e atua na requisição. O fluxo de requisição a partir da aplicação cliente, através do ORB, para a aplicação servidor ocorre conforme exposto abaixo:

[15] Do inglês delegate
[16] Do inglês disassembling

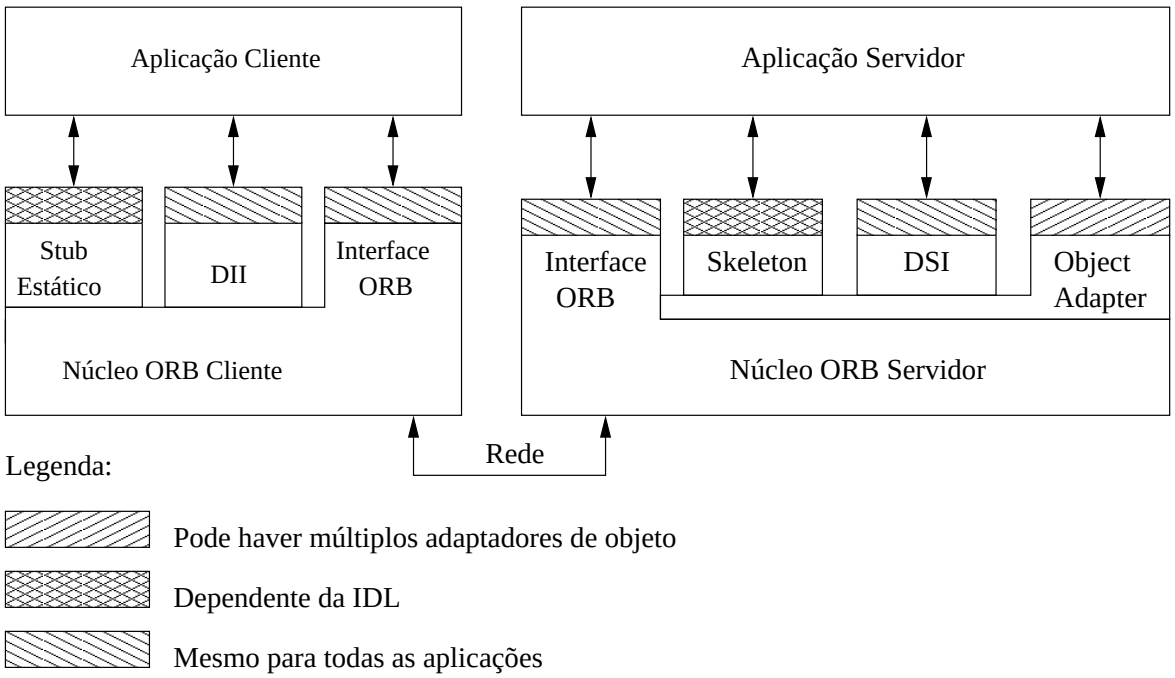


Figura 2.7: Arquitetura CORBA

- O cliente pode escolher fazer a requisição ou usando *stubs* estáticos a partir da definição de interface do objeto ou usando o DII (*Dynamic Invocation Interface*)[17]. De qualquer maneira o cliente direciona a requisição para o núcleo ORB que é ligado no seu processo.

- O núcleo ORB do cliente transmite a requisição para o núcleo ORB ligado com a aplicação servidor.

- O núcleo ORB do servidor despacha[18] a requisição para o adaptador do objeto que cria o objeto alvo.

- O adaptador do objeto promove o despacho da requisição para o *servant* que está implementando o objeto alvo. Assim como o cliente, o servidor pode escolher entre os mecanismos de despacho dinâmico ou estático para os seus *servants*. Podem ser *skeletons* estáticos originados a partir da definição de interface do objeto, ou os *servants* podem usar o DSI (*Dynamic Skeleton Interface*).

- Depois o *servant* transporta a requisição e, retorna a resposta para a aplicação cliente.

Quando o cliente invoca uma requisição síncrona, o cliente é bloqueado e fica aguardando a resposta. Estas requisições são idênticas as chamadas de procedimento remoto.

[17] Desenvolvedores de aplicações clientes que usam DII não precisam saber o nome da operação ou qualquer tipo de dado dos parâmetros ou retorno quando está escrevendo o programa

[18] Do inglês dispatches

Um cliente que invoca uma requisição síncrona adiada[19] envia a requisição, continua processando, e então depois processa a resposta. Este tipo de requisição pode ser invocado somente utilizando o DII.

O CORBA também provê uma requisição unilateral, que é uma requisição de melhor esforço que pode não ser realmente entregue para o objeto alvo e, também não é permitido haver respostas. Os ORBs permitem remover silenciosamente as requisições unilaterais se ocorrer congestionamento da rede ou faltas de recursos que causem um bloqueio no cliente enquanto a requisição não é entregue.

A partir do CORBA 3 através do AMI (*Asynchronous Programming Model*) é possível ter acesso ao modo de invocação assíncrona, embora seja necessário usar o SII (*Static Invocation Interface*)[20] ou DII [Joh00]. A invocação assíncrona é dividida em *callback* e *polling*. *Callback* é uma programação *dispare-e-esqueça*[21], para quando o cliente está fazendo a invocação. Sua vantagem é que não é necessário fazer nada, recebe-se o resultado assim que estiver pronto. No entanto, se o resultado chegar quando o programa cliente estiver ocupado fazendo outra atividade, o programa irá parar seu processamento e atenderá o retorno da chamada. O *polling*, por outro lado, pode-se escolher através do programa, o que fazer quando o resultado chegar.

2.4.3 IDL - Interface Definition Language

Para invocar operações de objetos distribuídos, o cliente deve conhecer a interface oferecida pelo objeto. Uma interface do objeto é composta de operações e de tipos de dados que podem ser passados para as operações.

No CORBA, as interfaces dos objetos são definidos pela OMG Interface Definition Language. IDL não é uma linguagem de programação como C++ e Java, assim os objetos e aplicações não podem ser implementados através da IDL. O propósito da IDL é permitir que as interfaces do objeto sejam definidas de maneira que sejam independentes de qualquer linguagem de programação. Assim as aplicações podem ser implementadas em diferentes linguagens de programação podendo interoperar entre si. A independência de linguagem da IDL é uma meta crítica do CORBA para dar suporte a sistemas heterogêneos e integrar aplicações que foram desenvolvidas separadamente.

OMG IDL suporta tipos simples, como tipos inteiros com sinal e sem sinal, caracteres, boolean, strings, tipos enumerados, estruturas, vetores (unidimensionais) e exceções. Estes tipos são usados para definir os tipos de dados dos parâmetros e tipos de dados para operações com retorno.

A figura 2.8 mostra a definição de uma IDL simples. Neste exemplo é definida uma interface chamada de Pessoa que contém uma operação chamada *getIdade()*. Esta operação não possui parâmetros e retorna um *long*. Para que um objeto CORBA suporte a interface Pessoa é necessário que ele implemente o operação *getIdade()* e retorne um número que indique a idade da pessoa o qual é representado pelo objeto.

[19] Do inglês deferred

[20] Os desenvolvedores de aplicações cliente que usam SII precisam saber antecipadamente o nome da operação, e todos os tipos de dados de parâmetros e retorno anteriormente a compilação

[21] Do inglês fire-and-forget

```
interface Pessoa {
    long getIdade();
};
```

Figura 2.8: Exemplo de definição de IDL simples

Referência para objeto é indicada na IDL pelo uso do nome da interface com um tipo. Conforme mostrado na figura 2.9.

```
interface RegistroPessoa {
    Pessoa pesquisar(in long numero_rg);
};
```

Figura 2.9: Exemplo de definição de IDL com referência de objeto

A operação *pesquisar* mostrado na figura 2.9, da interface RegistroPessoa, possui um número como parâmetro, o qual indica número de RG (Registro Geral) da pessoa e tem como retorno da operação uma referência para o objeto do tipo Pessoa. Uma aplicação poderia utilizar esta operação para recuperar uma Pessoa e então usar a referência do objeto para invocar as operações do objeto Pessoa.

Os parâmetros utilizados nas operações devem declarar as direções para que o ORB saiba se os valores devem ser enviados do cliente para o objeto alvo, vice-versa, ou ambas as direções. Na operação pesquisar foi utilizada a palavra-chave *in*, que significa que o número do RG é passado do cliente para o objeto alvo. Os parâmetros declarados como *out*, indica retorno de valor, são passados do objeto alvo para o cliente. E a palavra chave *inout* indica que o parâmetro é inicializado pelo cliente e então é enviado de volta pelo objeto alvo, sendo que o objeto alvo pode alterar o valor do parâmetro e retorná-lo modificado.

Um ponto importante das interfaces IDL é que elas podem ser herdadas de uma ou mais interfaces. Um exemplo de herança de interface é mostrado na figura 2.10.

```
interface Impressora {
    void imprimir();
};

interface ImpressoraColorida : Impressora {
    enum ModoCor {PretoBranco, Colorido};
    void aplicarCor(in ModoCor modo);
};
```

Figura 2.10: Exemplo de herança de IDL

A interface *ImpressoraColorida* é derivada da interface *Impressora*. Se uma aplicação cliente é escrita com os objetos do tipo *Impressora*, ela pode também usar um objeto que suporte a interface *ImpressoraColorida*, pois os objetos da interface *ImpressoraColorida* também fornecem total suporte a interface *Impressora*.

2.4.4 Adaptador de Objeto

Os adaptadores de objetos servem como ligação entre os *servants* e o ORB. Um adaptador de objeto corresponde a um objeto que se integra a interface de um objeto para uma interface diferente, que é esperada pelo invocador. Pode-se dizer que um adaptador de objeto é um objeto que se interpõe e delega para permitir que o invocador requisite operações de um objeto sem conhecer a sua verdadeira interface.

Para [HV99], um adaptador de objeto CORBA deve possuir três requisitos chaves:

- Criar referências para os objetos, permitindo os clientes endereçarem estes objetos.
- Assegurar que cada objeto alvo é encorpado[22] por um *servant*.
- Fazer os pedidos despachados pelo lado do servidor ORB e ajudar diretamente os *servants* a encorpar[23] cada objeto alvo.

Sem os adaptadores de objeto, o ORB teria quer fornecer diretamente estas características sem contar todas as outras responsabilidades. Como resultado, teria que ter uma interface muito complexa que seria muito difícil para a OMG gerenciar, e o número de estilos de implementações de *servants* seria limitado.

Em C++, os *servants* são instâncias de objetos C++. Eles são tipicamente definidos a partir das classes *skeleton* produzidas pela compilação das definições de interface IDL. Para implementar as operações, deve-se sobrescrever as funções virtuais da classe base *skeleton*. Devem-se também registrar os *servants* C++ com o adaptador de objeto para que seja possível despachar as requisições para os *servants* quando os clientes requisitarem os objetos encorpados pelos *servants*.

Segundo [OMG02b], os adaptadores de objeto são responsáveis pelas seguintes funções:

- Gerar e interpretar as referências de objetos.
- Invocar métodos.
- Segurança das interações.
- Ativar e desativar objetos.
- Mapear referências de objetos para as implementações correspondentes do objeto.
- Registrar as implementações.

[22] Do inglês incarnated
[23] Do inglês incarnating

Adaptador Portável de Objeto: Um adaptador de objeto é um mecanismo que conecta uma requisição usando uma referência de um objeto com um código próprio para o serviço. O Adaptador Portável de Objeto[24] é um tipo particular de adaptador de objeto que é definido pela especificação CORBA. O POA (*Portable Object Adapter*) têm os seguintes objetivos:

- Permitir aos programadores construirem implementações de objetos que sejam portáveis entre diferentes produtos ORB.
- Prover suporte para objetos com identidades persistentes.
- Prover suporte para ativação transparente de objetos.
- Permitir a um único *servant* dar suporte a múltiplas identidades de objetos simultaneamente.

2.4.5 Referência de Objeto

Uma referência de objeto é a informação necessária para especificar um objeto dentro de um ORB. Ambos, cliente e implementação do objeto têm uma noção opaca da referência do objeto de acordo com o mapeamento da linguagem, são isolados da representação atual deles. Duas implementações de ORB podem diferir na escolha da representação da referência do objeto.

A representação da referência de um objeto manipulada por um cliente somente é válida durante o ciclo de vida do cliente.

Todos os ORBs devem prover o mesmo mapeamento de linguagem para uma referência de objeto para uma linguagem de programação particular. Isto permite a um programa escrito em uma linguagem em particular acessar referências de objeto independente do ORB particular. O mapeamento da linguagem pode fornecer também caminhos adicionais para acessar referências de objetos de uma forma que seja conveniente para o programador.

A única forma que um cliente pode acessar o objeto alvo é através da referência para o objeto [HV99]. Para que o cliente possa obter a referência é necessário que o servidor publique a referência de alguma forma. Por exemplo, o servidor pode:

- Retornar uma referência como resultado de uma operação.
- Publicar a referência através de algum serviço, como por exemplo no serviço de nomes.
- Publicar uma referência de objeto através da convenção dele para um conjunto de caracteres e depois escrever estes caracteres para um arquivo.
- Transmitir uma referência de objeto através de e-mail ou publicação na Web.

[24] Em inglês POA (Portable Object Adapter)

Mas a forma mais comum do cliente adquirir a referência para um objeto é receber através da resposta de uma invocação de operação. Neste caso, as referências de objetos são valores de parâmetros e não diferentes de qualquer outro tipo de valor, como um inteiro. O cliente simplesmente contata um objeto e, o objeto retorna uma ou mais referências.

2.4.6 Implementação

A implementação apresentada abaixo está baseada na Java IDL [Mic04]. Java IDL é uma tecnologia para objetos distribuídos, ou seja, os objetos se comunicam em diferentes plataformas através de uma rede. Através da Java IDL os objetos se interagem embora os mesmos possam ter sido escritos em linguagem de programação Java ou outra linguagem como C, C++, Cobol, e outros.

Os passos a seguir provêem um guia geral para projetar e desenvolver uma aplicação distribuída com Java IDL.

Definir a interface remota: Pode-se definir a interface para o objeto remoto usando OMG IDL. Pode-se utilizar IDL, ao invés da linguagem Java, porque o compilador *idlj* faz o mapeamento a partir da IDL, gerando todos os *stubs* e *skeletons* da linguagem Java a partir dos arquivos fonte juntamente com os códigos básicos para conexão com o ORB. Através do uso da IDL é possível para os desenvolvedores implementar clientes e servidores em qualquer outra linguagem que esteja de acordo com o CORBA.

Para desenvolver uma aplicação utilizando o Java IDL é necessário ter instalado J2SE versão 1.4 ou superior, esta versão contém a API (*Application Programming Interface*) e o ORB necessário para desenvolver uma aplicação distribuída baseada em CORBA, além disso, esta versão contém o compilador *idlj*, que é o compilador usado para mapear IDL para Java.

```
module ContaApp
{
 interface Conta
 {
    float getSaldo();
    oneway void desligar();
 };
};
```

Figura 2.11: Exemplo Java IDL

Conforme mostrado na figura 2.11 para definir um arquivo Java IDL é necessário declarar um módulo CORBA que constitui um *namespace*[25] que atua como um container para interfaces e declarações. Dentro do módulo são definidas as interfaces, as interfaces CORBA são como as interfaces Java, cada interface na IDL é mapeada para uma

[25]Grupo lógico de nomes usados dentro de um programa.

interface Java. E finalmente, dentro das interfaces são definidas as operações. As operações correspondem aos comportamentos que servidor deverá executar quando os clientes invocá-los.

Compilar a interface remota: A compilação da IDL é feita através da ferramenta *idlj*. Este comando gera uma interface Java, como também os arquivos fonte da classe para os *stubs* e *skeletons* que habilitam a comunicação com o ORB.

Utilizando a opção *-fall* no comando *idlj*, o compilador gera as ligações para o lado do cliente e os *skeletons* para o lado do servidor. Pois, sem esta opção o compilador iria somente gerar as ligações para o lado cliente. Após a compilação, neste exemplo, o próprio compilador cria um diretório chamado *ContaApp* contendo seis arquivos: *_ContaStub.java, Conta.java, ContaHelper.java, ContaHolder.java, ContaOperations.java e ContaPOA.java*. Os ítens abaixo descrevem cada um desses arquivos:

1. Conta.java – Este arquivo contém a versão Java da interface IDL. A interface Conta.java é estendida da org.omg.CORBA.Object provendo as funcionalidades padrões do objeto CORBA.

2. ContaOperations.java – Neste arquivo são inseridas todas as operações contidas na interface, as quais são compartilhadas pelos *stubs* e *skeletons*.

3. HelloPOA.java – Esta é uma classe abstrata, baseada em dados seriais[26], *skeleton* do servidor, e provê as funcionalidades básicas CORBA para o servidor. É estendida da org.omg.PortableServer.Servant, e implementa a interface InvokeHandler e a interface ContaOperations.

4. _Conta.java – Nesta classe contém o *stub* cliente provendo as funcionalidades CORBA para o cliente. É estendida da org.omg.CORBA.portable.ObjectImpl e implementa a interface Conta.java.

5. ContaHelper.java – Esta classe provê funcionalidades auxiliares, como o método *narrow()* que é requerido para fazer conversões de referências de objetos CORBA para seus próprios tipos. A classe ContaHelper é responsável por ler e escrever os tipos de dados para dados seriais CORBA, e inserir e extrair tipos de dados.

6. ContaHolder.java – Esta classe final é uma instância pública do tipo Conta. Sempre que um tipo IDL é um parâmetro *out* ou um *inout*, a classe ContaHolder é usada. A classe ContaHolder delega métodos para a classe ContaHelper para ler e escrever.

Implementando o Servidor: Após ter sido executado o compilador *idlj*, pode-se utilizar os *skeletons* colocando junto com a aplicação servidora. O servidor além de implementar a interface dos métodos remotos, o código servidor inclui o mecanismo para iniciar o ORB e aguardar a invocações do cliente remoto.

[26] Do inglês stream-based

O servidor é composto por duas classes: o *servant* e a própria classe servidor. O *servant*, ContaImpl, é a implementação da interface Conta IDL; cada instância de Conta é implementada por uma instância ContaImpl. O *servant* é uma subclasse de ContaPOA.

O *servant* contém um método para cada operação IDL, neste exemplo, *getSaldo()*. Métodos do *servant* são como os métodos Java, os códigos para comunicação com o ORB, *marshalling* de argumentos e resultados são fornecidos pelo *skeleton*.

Pode-se observar na figura 2.12 que na primeira parte do código é definido o *servant* ContaImpl. Através do método *setORB()* pode-se atribuir o ORB para o *servant*. O próximo método corresponde a implementação do método *getSaldo()*, o qual retorna o valor da variável *saldo* que é do tipo de dado float.

Na classe ContaSevidor, conforme figura 2.12, um dos primeiros passos que ocorre é a inicialização do ORB. Um servidor CORBA necessita de um objeto ORB local. Toda vez em que o servidor é inicializado, o ORB é registrado, desta forma o ORB pode encontrar o servidor quando ele receber uma invocação.

Em seguida ocorre a obtenção da referência da raiz do POA e o POAManager é ativado. A operação de ativação muda o estado do gerenciador POA para ativo, fazendo com que os POAs associados comecem a processar as invocações.

O próximo passo é criar uma instância do *servant* ContaImpl e invocar o método *setORB(orb)*, o qual é definido no *servant*, e depois invocar o método *desligar()* que invoca o término das operações do ORB.

O código *org.omg.CORBA.Object ref = rootpoa.servant_to_reference(contaImpl)* é usado para obter a referência do objeto associado com o *servant*. O método *narrow* é utilizado para converter uma referência de objeto CORBA para um tipo próprio, neste caso para o objeto Conta.

O ContaServidor utiliza o serviço de nomes, COSNS (*Common Object Service Naming Service*), para tornar as operações do objeto *servant* disponíveis para o cliente. O servidor necessita de uma referência do objeto para o serviço de nomes, que então publica as referências para os objetos implementarem as diversas interfaces. A chamada *orb.resolve_initial_references("NameService")* é invocada com objetivo de obter a referência do objeto para o servidor de nomes. Esta descrição *"NameService"* é definida para todos os ORBs CORBA, quando é passado esta descrição, o ORB retorna o contexto de nomes que é uma referência do objeto para o serviço de nomes.

Como toda referência de objeto CORBA, o *objRef* é um objeto genérico CORBA, então para que seja possível acessar o serviço de nomes é necessário utilizar o objeto NamingContextExt, para isto é necessário converter *objRef* para o tipo específico. Através do objeto *ncRef* é possível acessar o serviço de nomes e registrar o servidor.

O caminho do objeto Conta é obtido através do método *ncRef.to_name(name)*, e então é passado o caminho e o objeto *servant* para o serviço de nomes, ligando o objeto *servant* com o identificador Conta. Assim quando o cliente invocar o método *resolve("Conta)*, o serviço de nomes retornará uma referência para o *servant* Conta.

E, finalmente, o método *orb.run()* é invocado com o objetivo de aguardar as requisições vindas do ORB.

```
class ContaImpl extends ContaPOA {
  private ORB orb;
  private float saldo = 100;
  public void setORB(ORB orb_val) {
    orb = orb_val;
  }
  public void desligar(){
    orb.shutdown(false);
  }
  public float getSaldo() {
    return saldo;
  }
}
public class ContaServidor {
  public static void main(String args[]) {
    try{
      ORB orb = ORB.init(args, null);
      // obtêm a referência para rootpoa & ativa o POAManager
      POA rootpoa =
            POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
      rootpoa.the_POAManager().activate();
      // cria o servant e registra-o com o ORB
      ContaImpl contaImpl = new ContaImpl();
      contaImpl.setORB(orb);
      // obtém a referencia do objeto a partir do servant
      org.omg.CORBA.Object ref = rootpoa.servant_to_reference(contaImpl);
      Conta href = ContaHelper.narrow(ref);
      // obtém o root naming context
      org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
      // Usar o NamingContextExt -- Serviço de Nomes
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
      // ligar a referência do objeto com serviço nomes
      String name = "Conta";
      NameComponent path[] = ncRef.to_name( name );
      ncRef.rebind(path, href);
      System.out.println("ContaServidor aguardando ...");
      // aguarda invocações do cliente
      orb.run();
    }
    catch (Exception e) {
        System.err.println("ERRO: " + e);
        e.printStackTrace(System.out);
    }
  }
}
```

Figura 2.12: Código fonte do *servant* e servidor.

2.5 RMI - Remote Method Invocation

Pode-se dizer que o RMI, em um nível mais baixo, consiste em um mecanismo de RPC, que permite o desenvolvimento de objetos distribuídos utilizando a tecnologia Java. O RMI provê um modelo para computação distribuída utilizando objetos Java.

O RMI é projetado para simplificar a comunicação entre dois objetos em máquinas virtuais diferentes [BDV+98], permitindo um objeto invocar métodos de um objeto em outra máquina virtual, do mesmo modo que os métodos locais são invocados.

Aplicações RMI são freqüentemente separadas [Inc99] em dois programas: um servidor e um cliente. Um servidor típico cria um conjunto de objetos remotos, tornando acessíveis as referências para os objetos remotos, e espera as invocações dos clientes. Uma aplicação cliente típica, obtém a referência do servidor de um ou mais objetos remotos para então invocar os seus métodos.

Existe uma grande dependência do RMI em relação ao *Java Object Serialization* [Rui00], que é responsável pelo processo de *marshalling* e *unmarshalling*. Cada objeto do servidor define uma interface que expõe um conjunto de métodos para os clientes que não pertencem ao mesmo endereço de rede, para que os mesmos possam acessar e utilizar os serviços disponíveis. O esquema de nomes é implementado através do *rmiregistry*, conforme descrito na subseção 2.2.2. Quando um cliente faz uma chamada para referenciar um objeto, ele usa uma URL (*Uniform Resource Locator*) para referenciar o objeto servidor, assim como se usa para referenciar uma URL de uma página Web.

Quando a referência para o objeto é obtida, o objeto não é enviado para o cliente que o solicitou. Em seu lugar existe um objeto *proxy* ou *stub* para o objeto remoto. Todas as interações entre o objeto local e o objeto remoto acontecem via *stub*. Um objeto remoto pode ter diversas chamadas de clientes, mas cada cliente possui seu próprio *stub*. Um objeto remoto não pode ser replicado porque existem muitos *stubs* trabalhando no objeto.

No lado servidor, o objeto *proxy* é chamado *skeleton* o qual gerencia as chamadas e os dados que serão exportados.

A arquitetura do RMI consiste de quatro camadas [Rui00], conforme apresentada na figura 2.13:

- A primeira camada é a camada de aplicação, onde se concentram as implementações das aplicações clientes e servidores. Para que o servidor possa disponibilizar os métodos para as aplicações clientes terem acesso, é necessário que os métodos estejam contidos em uma ou mais interfaces, as quais são estendidas da interface *java.rmi.Remote*. As interfaces implementadas desta forma são chamadas de interfaces remotas. A única diferença entre interfaces remotas e interfaces ordinárias é que as remotas são adicionadas uma exceção estendida da *RemoteException*. Depois da implementação dos métodos remotos, eles devem ser exportados de forma implícita pela extensão da classe *UnicastRemoteObject*, ou explicitamente pela chamada do método *exportObject*.

- A camada dois é a camada *proxy*. Todas as chamadas para o objeto remoto, empacotamento dos parâmetros (*marshalling*) e retorno do objeto são realizados através

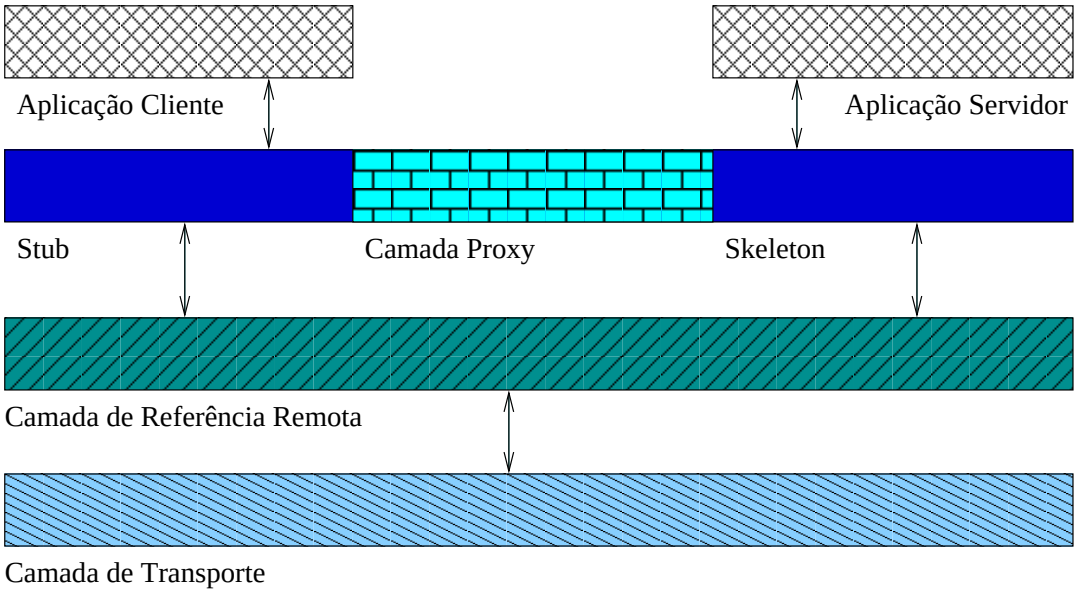


Figura 2.13: Quatro Camadas do RMI

desta camada. O *stub* e o *skeleton*, representam o lado cliente e o lado servidor, respectivamente, são arquivos de classes compilados através do comando *rmic* – compilador RMI. O *stub* corresponde ao *proxy* do lado do cliente para o objeto remoto e a interface de aplicação para o objeto remoto. Ele tem a responsabilidade de iniciar a chamada para o objeto remoto através da camada de referência remota. Os objetos são transmitidos de forma serializada, tanto do lado cliente como servidor. O *skeleton* é o proxy do lado servidor para os objetos remotos e têm o comportamento de uma classe *stub*.

- A camada três é a camada de referência remota. Nesta camada, são observados os detalhes de semântica da chamada do objeto remoto e tarefas de implementações específicas. A camada de referência remota corresponde a uma abstração entre a camada *proxy* e a camada de transporte. Sua expectativa de conseguir obter *stream-connection* a partir da camada de transporte, embora o protocolo não baseado em conexão seja usado na camada de transporte. Além disso, a camada pode executar as tarefas de replicar objetos ou de recuperar conexões perdidas.

- A quarta e última camada é a de transporte. Nela ocorre a configuração da conexão e o transporte dos dados de uma máquina para outra. Por padrão, o TCP/IP é o protocolo de comunicação. A camada também monitora a conexão e fica aguardando novas conexões de outras máquinas. Uma vez que esta camada é independente das demais camadas, as mudanças ocorridas nesta camada são transparentes para as camadas superiores.Pode-se fazer por exemplo mudanças de criptografia dos dados transmitidos, aplicação de compressão dos dados e outras otimizações de segurança e desempenho de forma que fique transparente para as demais camadas.

O RMI trata diferentemente um objeto remoto de um objeto não remoto, quando um objeto passado de uma máquina virtual para outra. Antes de fazer uma cópia da

implementação do objeto na recepção da máquina virtual, o RMI passa o *stub* remoto para o objeto remoto. O *stub* atua como uma representação local, ou *proxy*, para o objeto remoto e basicamente é, para o invocador, uma referência remota. O invocador invoca um método no *stub* local o qual é responsável por transportar a chamada do método para o objeto remoto.

O *stub* do objeto remoto implementa o mesmo conjunto de interfaces remotas que os objetos remotos implementam. Isto permite ao *stub* ser um molde[27] para qualquer interface que o objeto remoto implemente. Entretanto, isto significa que somente os métodos definidos na interface é estarão disponíveis para ser invocados por um cliente remoto.

2.5.1 Implementação

A implementação dos exemplos citados a seguir foran baseadas em [WRW96] e [WW04]. Para desenvolver uma aplicação usando RMI é necessário seguir os passos abaixo:

- Projetar e implementar os componentes da aplicação distribuída. Ou seja, definir as interfaces remotas, implementar as interfaces remotas e implementar os clientes.
- Compilar os arquivos fontes e gerar os *stubs*. Compilar os fontes via comando *javac* e gerar os *stubs* via comando *rmic*.
- Iniciar a aplicação. Neste passo deve-se executar o comando *rmiregistry* e em seguida executar o servidor e o cliente.

Para definirmos uma interface remota é necessário que ela seja estendida da interface *java.rmi.Remote*, através da definição desta interface é que os métodos podem ser acessados a partir de qualquer máquina virtual. E qualquer objeto que venha a implementar esta interface se torna um objeto remoto.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface IConta extends Remote {
    double getSaldo() throws RemoteException;
}
```

Figura 2.14: Código fonte da interface IConta

Definição de Interface Remota: Pode-se observar na figura 2.14 a definição da interface *IConta* a qual possui um único membro, o método remoto *getSaldo()*. Este

[27] Do inglês cast

método deve ser capaz de lançar[28] uma *RemoteException*. Esta exceção é lançada pelo RMI durante uma chamada de método remota quando ocorrer uma falha de comunicação ou um erro de protocolo.

Implementação do Servidor: De uma maneira geral a implementação da classe de interface remota deve ao menos:

- Declarar as interfaces remotas que serão implementadas.
- Definir o construtor para o objeto remoto.
- Prover a implementação de cada método remoto contido nas interfaces remotas.

Observa-se na figura 2.15, que a classe Conta é estendida da classe *UnicastRemoteObject*. A super-classe *UnicastRemoteObject* provê implementações para um grande número de métodos da classe *java.lang.Object* como – *equals, hashCode, toString* – que são apropriados para objetos remotos. *UnicastRemoteObject* inclui construtores e métodos estáticos usados para exportar um objeto remoto, assim é possível tornar o objeto remoto disponível para receber chamadas dos clientes.

Um objeto remoto não necessariamente precisa ser estendido da classe *UnicastRemoteObject*, mas no entanto, a implementação não fornecerá as implementações apropriadas do método da classe *java.lang.Object*. Além disso, a implementação do objeto remoto deve conter uma chamada explícita para o método *exportObject* da classe *UnicastRemoteObject* que faz com o objeto possa receber chamadas de entrada. O fato da classe Conta ser estendida da classe *UnicastRemoteObject* faz com ela possa ser usada para criar um objeto remoto simples que suporta uma comunicação remota ponto-a-ponto[29] e também usar o transporte padrão do RMI para comunicação que é baseado em soquetes[30][31].

A classe Conta utiliza o gerenciador de segurança fornecido como parte do sistema RMI, o *RMISecurityManager*. Este gerenciador de segurança obriga o uso de uma política de segurança similar a que é utilizada pelos *applets*[32], pode-se dizer que o acesso é bastante conservador.

Para que o cliente possa invocar um método remoto, o sistema provê o *RMI Registry* que tem a finalidade de auxiliar os clientes a encontrarem as referências para os objetos remotos. O *RMI Registry* é tipicamente usado para encontrar somente o primeiro objeto remoto que um cliente necessite. O primeiro objeto então provê o suporte para encontrar os demais objetos.

Conforme observado na figura 2.15, a classe Conta cria um nome para o objeto *String name = "//host/Conta";*, este nome inclui o nome do *host*, o qual o *registry* e o objeto remoto estão sendo executados e o nome – Conta – que identifica o objeto

[28] Do inglês throw
[29] Do inglês point-to-point
[30] Do inglês sockets
[31] Um socket corresponde a um ponto de uma comunicação de duas vias que liga dois programas em uma mesma rede de computadores
[32] Uma aplicação projetada para ser executada dentro de uma outra aplicação, como os Web browsers. São executados pela Java Virtual Machine, que deve estar presente no cliente.

```
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Conta extends UnicastRemoteObject implements IConta {

    private double saldo=0;
    protected Conta(double _saldo) throws RemoteException {
        super();
        this.saldo = _saldo;
    }

    public double getSaldo() throws RemoteException {
        return saldo;
    }

    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        String name = "//host/Conta";
        try {
            Conta conta = new Conta(100.00);
            Naming.rebind(name, conta);
        } catch (Exception e) {
            System.err.println("Erro ocorrido em Conta: " +
                    e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Figura 2.15: Código fonte da classe Conta que implementa a interface IConta

remoto no *registry*. A partir desta especificação é necessário adicionar ao código o nome para *RMI Registry* que está sendo executado no servidor. Isto é feito através do código: *Naming.rebind(name, conta);*.

Uma vez que o servidor foi registrado no *RMI Registry* local, o servidor está pronto para manipular as chamadas dos clientes. Não é necessário manter uma *thread* esperando para manter o servidor ativo. O objeto Conta estará disponível para aceitar chamadas e não será desativado até que ele seja desligado do *registry*, e que nenhum cliente esteja mantendo uma referência remota para com ele.

Implementação do Cliente: A implementação do código cliente é um pouco mais simples. O cliente invoca as rotinas para localizar o objeto remoto e então invoca o método remoto e aguarda seu retorno.

```
import java.rmi.Naming;
import java.rmi.RMISecurityManager;

public class PessoaFisica {

    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            String name = "//host/Conta";
            IConta conta = (IConta) Naming.lookup(name);
            double saldo = conta.getSaldo();
            System.out.println(saldo);
        } catch (Exception e) {
            System.err.println("PessoaFisica exceção: " +
                               e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Figura 2.16: Código fonte da classe PessoFisica

Pode-se observar na figura 2.16, que assim como no código Conta do servidor, o cliente começa instalando o gerenciador de segurança. Isto é necessário devido o código RMI ser carregado para o cliente.

Depois de instalar o gerenciador de segurança, o cliente constrói o nome que será utilizado para localizar o objeto remoto Conta. O cliente utiliza o método *Naming.lookup* para procurar o objeto remoto pelo nome no *registry* do *host* remoto. Enquanto ocorre a procura, o código cria uma URL que especifica o *host* onde o servidor esta sendo executado.

Após o método *Naming.lookup* localizar o objeto remoto é necessário convertê-lo para o tipo IConta, pois o método *lookup* retorna uma referência, um *stub* para objeto remoto genérico.

Através da variável *conta* é possível então invocar o método *getSaldo*. Quando é executada a chamada do método *getSaldo*, o cliente invoca o método do objeto remoto Conta, o qual é executado no lado servidor e retorna o resultado do método para o cliente que mostra na tela o resultado do método.

Execução: Vale a pena observar que para executar os programas cliente e servidor é necessário que código fonte esteja compilado utilizando o comando *javac* e que tenha sido gerado os *stubs e skeletons* via comando *rmic*. Outro ponto a ser ressaltado é que o modelo de segurança adotado pelo JDK 1.2 é mais sofisticado do que o utilizado pelo JDK 1.1, o JDK 1.2 possui algumas otimizações, por isso é necessário conceder permissões de acesso para executar certos códigos.

No JDK 1.2 é necessário que exista um arquivo de política conforme mostrado na figura 2.17.

```
grant {
   permission java.net.SocketPermission "*:1024-65535", "connect,accept";
   permission java.net.SocketPermission "*:80", "connect";
};
```

Figura 2.17: Arquivo exemplo de política de acesso

Após a definição do arquivo de política de acesso deve-se então executar o comando *rmiregistry*.

Para executar o código servidor é necessário executar o comando conforme figura 2.18.

```
java -Djava.rmi.server.codebase=file:/c:\home\classes/
     -Djava.rmi.server.hostname=homehost
     -Djava.security.policy=java.policy
        Conta
```

Figura 2.18: Comando de execução do servidor

E, finalmente, para executar o código cliente é necessário executar o comando mostrado na figura 2.19.

```
java -Djava.rmi.server.codebase=file:/c:\home\classes/
     -Djava.security.policy=java.policy
        PessoaFisica
```

Figura 2.19: Comando de execução do cliente

2.6 Considerações

A arquitetura CORBA, apresentada na seção 2.4, foi desenvolvida com o objetivo principal de prover interoperabilidade entre diversos tipos de linguagens de programação, diversas plataformas de hardware e sistemas operacionais. Justamente esta necessidade de interoperabilidade traz consigo **a complexidade** para o desenvolvimento de sistemas distribuídos, o que acarreta a necessidade de programadores experientes. A complexidade é refletida pela observação de que para se invocar um simples método *getSaldo()*, conforme apresentado na figura 2.12, o compilador *idlj* gera 6 (seis) arquivos diferentes.

Outro ponto a ressaltar sobre a arquitetura CORBA é a ausência de **transparência de acesso**. Ou seja, se comparássemos o código apresentado pela figura 2.12 com uma invocação de um método local da mesma classe, notaríamos que a invocação é totalmente diferente. No entanto, há de se salientar que CORBA possui transparência de localização.

Por outro lado, o Java RMI que é uma tecnologia para invocação de objetos remotos, executada exclusivamente sob a plataforma Java, possui uma codificação mais limpa e direta se comparada com a codificação CORBA (observe figura 2.15). O Java RMI da mesma forma que CORBA não possui transparência de acesso se comparado com a invocação de um método local da mesma classe. Por outro lado, apresenta transparência de localização.

Os trabalhos apresentados neste capítulo foram de grande importância para o desenvolvimento deste trabalho. Por exemplo, pode-se dizer que os conceitos de ORB da arquitetura CORBA, apresentada na subseção 2.4.1, teve grande influência para a criação do mecanismo de manipulação de mensagens entre máquinas virtuais. O serviço de nomes Java Rmiregistry, apresentado na subseção 2.2.2 foi de grande contribuição para a construção do serviço de nomes da VIRTUOSI. E principalmente a tabela de referência (*Handle Table*), apresentada na seção 2.3, contribuiu para a definição do mecanismo de invocação de métodos para a VIRTUOSI. Pois, através da tabela de referência é que o mecanismo de invocação remota de métodos da VIRTUOSI provê transparência de acesso e localização.

Capítulo 3

Arquitetura da VIRTUOSI

Este capítulo especifica os conceitos de orientação a objeto suportados pela VIRTUOSI através da formalização do metamodelo da VIRTUOSI. Em seguida, especifica um novo formato de representação intermediária na forma de árvore de programa interpretada pela máquina virtual VIRTUOSI. Por último, especifica a arquitetura da máquina virtual VIRTUOSI.

Para auxiliar o entendimento dos conceitos explicados ao longo desse Capítulo, são utilizados trechos de código fonte de uma aplicação de software orientado a objeto escritos na linguagem Aram[1].

Também neste capítulo são ressaltados os principais elementos do metamodelo VIRTUOSI que importam para o entendimento do mecanismo de invocação remota de métodos, o conteúdo completo do metamodelo VIRTUOSI pode ser visto no apêndice A.

3.1 Projeto VIRTUOSI

O projeto VIRTUOSI visa o desenvolvimento de uma ferramenta com perspectivas pedagógica e experimental para construção (edição, compilação e depuração) e a execução de sistemas de software orientado a objetos distribuído. Observando a figura 3.1, pode-se dizer que o projeto VIRTUOSI é apoiado sobre dois grandes pilares: o metamodelo VIRTUOSI e a arquitetura de máquinas virtuais distribuídas.

Desde a criação do projeto VIRTUOSI, no ano de 2000, produziu-se quatro dissertações de mestrado:

- Uma abordagem contínua para sistemas de software baseados em estados e eventos [Jun02].
- Persistência de Objetos Baseada em Reflexão Computacional [dCM03].
- Um Sistema de Execução para Software Orientado a Objeto Baseado em Árvores de Programa [Kol04].
- Mecanismo de Mobilidade de Objetos para a Virtuosi [dCCF04].

[1] Aram é uma linguagem de programação que dá suporte aos conceitos de orientação a objeto definidos pelo metamodelo da VIRTUOSI

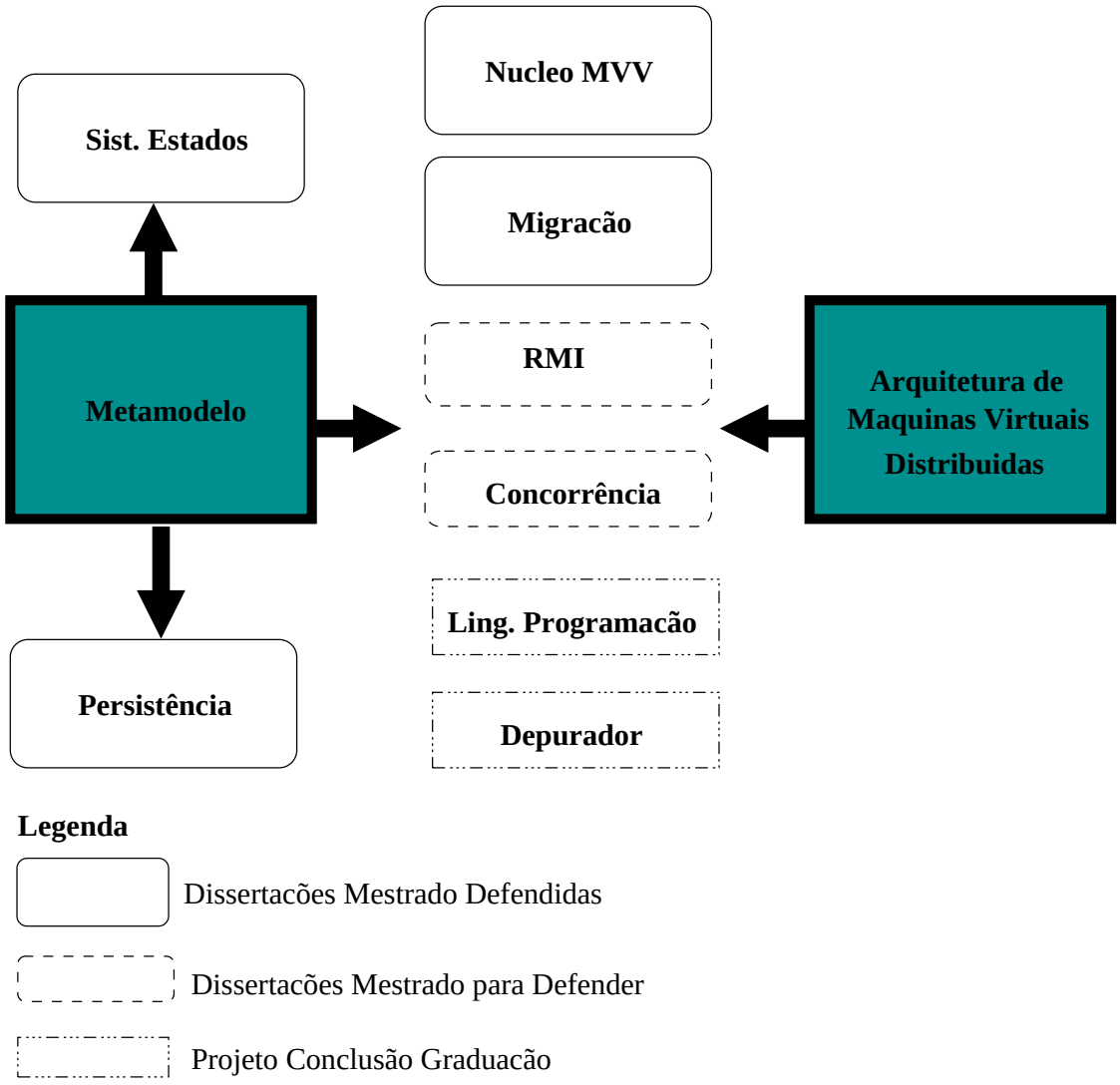


Figura 3.1: Projeto Virtuosi.

Além das dissertações de mestrado já apresentadas, serão defendidas no período de agosto de 2005 mais duas dissertações: esta própria dissertação e outra abordando controle de concorrência. Também foram apresentados dois trabalhos de conclusão de curso de graduação: um dos trabalhos foi a especificação da linguagem Aram [Aro04] e construção do respectivo compilador, e o outro trabalho foi o de criar um depurador para a linguagem Aram.

3.2 Metamodelo da VIRTUOSI

Os conceitos de orientação a objeto suportados pela VIRTUOSI são formalizados através de um diagrama de classes da UML chamado de metamodelo da VIRTUOSI. O metamodelo da VIRTUOSI possui classes e associações que representam os elementos encontrados em uma linguagem de programação orientada a objeto que de suporte aos conceitos suportados pela VIRTUOSI. Por isso, as classes do metamodelo podem ser cha-

madas de meta-classes. Por exemplo, uma classe possui atributos; o metamodelo da VIRTUOSI, portanto, possui uma meta-classe para representar uma classe de aplicação, uma meta-classe para representar um atributo e através de uma associação, explicita que uma classe possui zero ou muitos atributos. O metamodelo da VIRTUOSI pode ser entendido como um diagrama de classes que descreve conceitos de orientação a objeto e como tais conceitos se relacionam entre si. Um diagrama com uma visão unificada das principais meta-classes do metamodelo da VIRTUOSI é apresentado no Apêndice A.

3.2.1 Literal e Bloco de Dados

3.2.1.1 Valor Literal

Um valor literal é uma seqüência de caracteres sem semântica definida. A Figura 3.2 mostra as situações possíveis para o uso de valores literais.

```
class Principal
{
   constructor iniciar( ) exports all {
   ...
   Integer massa = Integer.make( 70 );// 70 é um valor literal
   }
   ...
}
...
class Boolean {
   enum { true, false } value = false;
   ...
   method void flip( ) exports all
   {
       if ( value == true )
           value = false;
       else
           value = true;
   }
   ...
}
```

Figura 3.2: Código fonte em Aram mostrando os possíveis uso de um valor literal

3.2.1.2 Referência a Literal

Uma meta-classe que representa uma **referência a literal**. Uma referência para um literal não pode sofrer atribuição. E também não existe a possibilidade de declarar uma variável local do tipo referência a literal.

3.2.1.3 Referência a Bloco de dados

Uma **referência a bloco de dados** consiste em uma referência para uma seqüência contígua de dados binários em memória. Esse tipo de referência é geralmente utilizado para a construção de classes pré-definidas (*Integer*, *Real*, *Boolean*, *Character* e *String*) utilizadas para compor outras classes.

3.2.2 Classes

A meta-classe que representa uma **classe** se relaciona através de associações com outros componentes do metamodelo da VIRTUOSI nas seguintes situações:

1. como possuidora de atributos referência a objeto em um relacionamento associação;
2. como possuidora de atributos referência a objeto em um relacionamento composição exclusiva;
3. como possuidora de atributos referência a bloco de dados em um relacionamento composição exclusiva;
4. como possuidora de atributos de variáveis enumeradas em um relacionamento composição exclusiva;
5. como tipo de uma referência a objeto;
6. como possuidora de invocáveis;
7. como cliente da lista de exportação de um invocável;

A Figura 3.3 mostra o relacionamento da meta-classe que representa uma classe de aplicação com outros componentes do metamodelo da VIRTUOSI, excetuando-se os relacionamentos com meta-classes descendentes.

3.2.3 Herança

Duas classes podem estabelecer um relacionamento de herança entre si, tal que os invocáveis da classe herdeira podem acessar tanto o estado quanto o comportamento (métodos e ações) definidos pela classe ancestral, sem qualquer restrição. Em outras palavras, todas as definições de estado e comportamento existentes na classe ancestral são válidas para a classe herdeira. Segundo o metamodelo da VIRTUOSI, uma classe pode ter apenas uma classe ancestral direta[2], mas pode ter muitas classes herdeiras recursivamente. Entretanto, uma classe não pode ser direta ou indiretamente ancestral de si própria. Assim, um conjunto de classes pode ser organizado como um grafo acíclico dirigido no qual a propriedade de herança é transitiva, isto é, uma classe herdeira assimila as definições de estado e de métodos de ancestrais diretas ou indiretas.

[2]Essa propriedade é normalmente denominada herança simples, em contra-partida à herança múltipla, quando uma classe pode ter muitas ancestrais diretas.

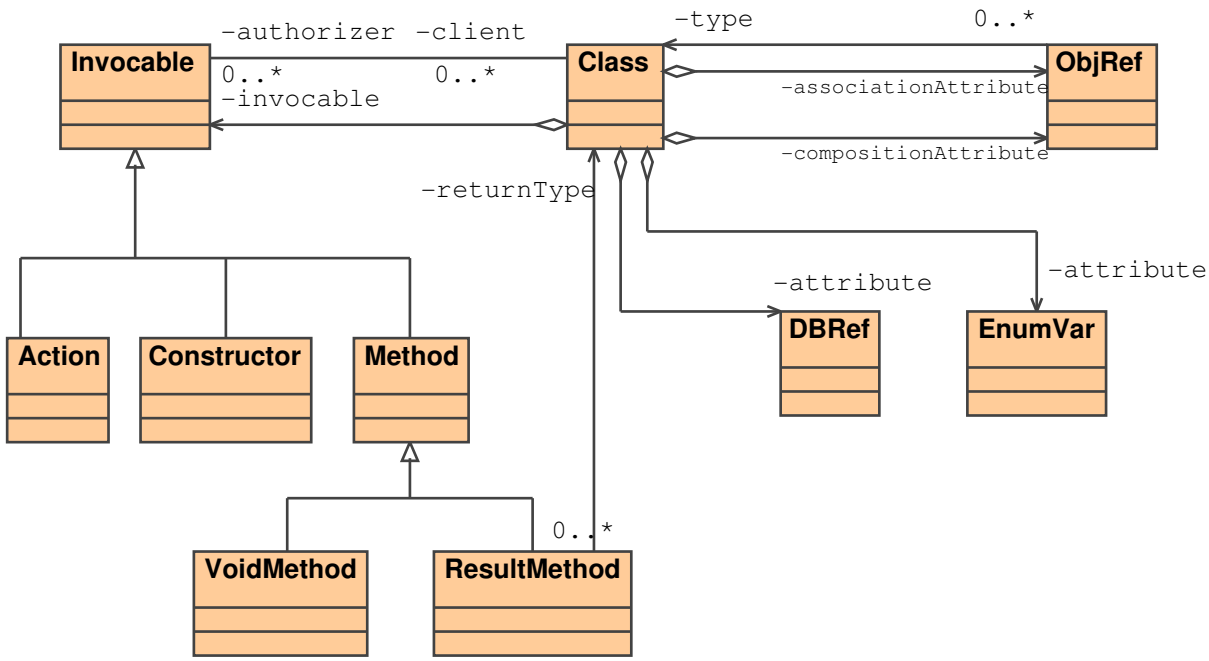


Figura 3.3: Relacionamento da meta-classe que representa uma classe de aplicação com outros componentes do metamodelo da VIRTUOSI

Uma conseqüência da transitividade da propriedade de herança é que uma referência de uma certa classe pode ter como alvo instâncias de distintas classes, desde que estas sejam herdeiras (diretas ou indiretas) da classe que define o tipo da referência, caracterizando assim a propriedade de polimorfismo.

Existem dois tipos de classe, a **classe de aplicação** e a **classe raiz**. Toda classe da aplicação possui uma classe ancestral, sendo que esta pode ser uma outra classe de aplicação ou a classe raiz.

3.2.4 Atributos

O ambiente VIRTUOSI disponibiliza uma biblioteca de classes pré-definidas (*Integer*, *Real*, *Boolean*, *Character* e *String*) utilizadas para compor novas classes, as classes de aplicação.

Os atributos de uma classe segundo o metamodelo da VIRTUOSI podem ser de três tipos, a saber:

- referência a objeto;
- referência a bloco de dados;
- variável enumerada.

O metadomelo da VIRTUOSI formaliza os três tipos de atributo que uma classe conforme mostra a Figura 3.4.

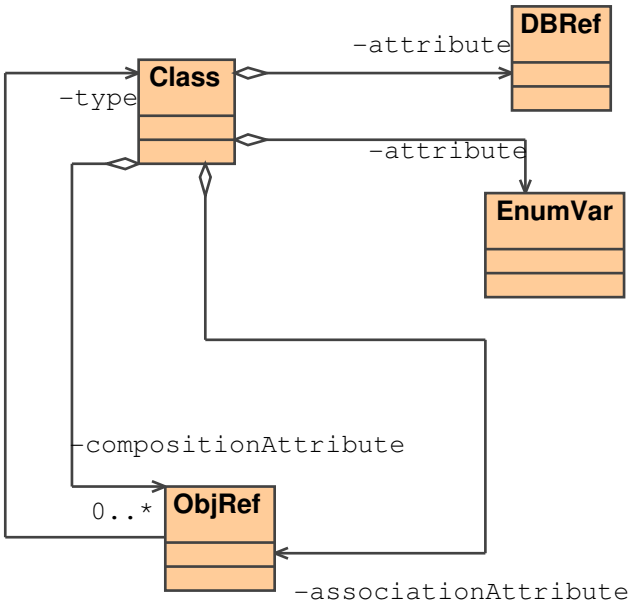


Figura 3.4: Os três tipos de atributos possíveis em uma classe segundo o metamodelo da VIRTUOSI

3.2.4.1 Referência a Objeto

A meta-classe que representa uma **referência a objeto** se relaciona através de associações com outros componentes do metamodelo da VIRTUOSI nas seguintes situações:

1. como parâmetro;
2. como atributo;
3. como atributo em um acesso a atributo objeto;
4. como objeto em um acesso a atributo bloco de dados;
5. como referência retornada por um invocável;
6. como variável local;
7. como alvo de invocação de método;
8. como alvo de invocação de uma ação.

A Figura 3.5 mostra o relacionamento da meta-classe que representa a referência a objeto com outros componentes do metamodelo da VIRTUOSI.

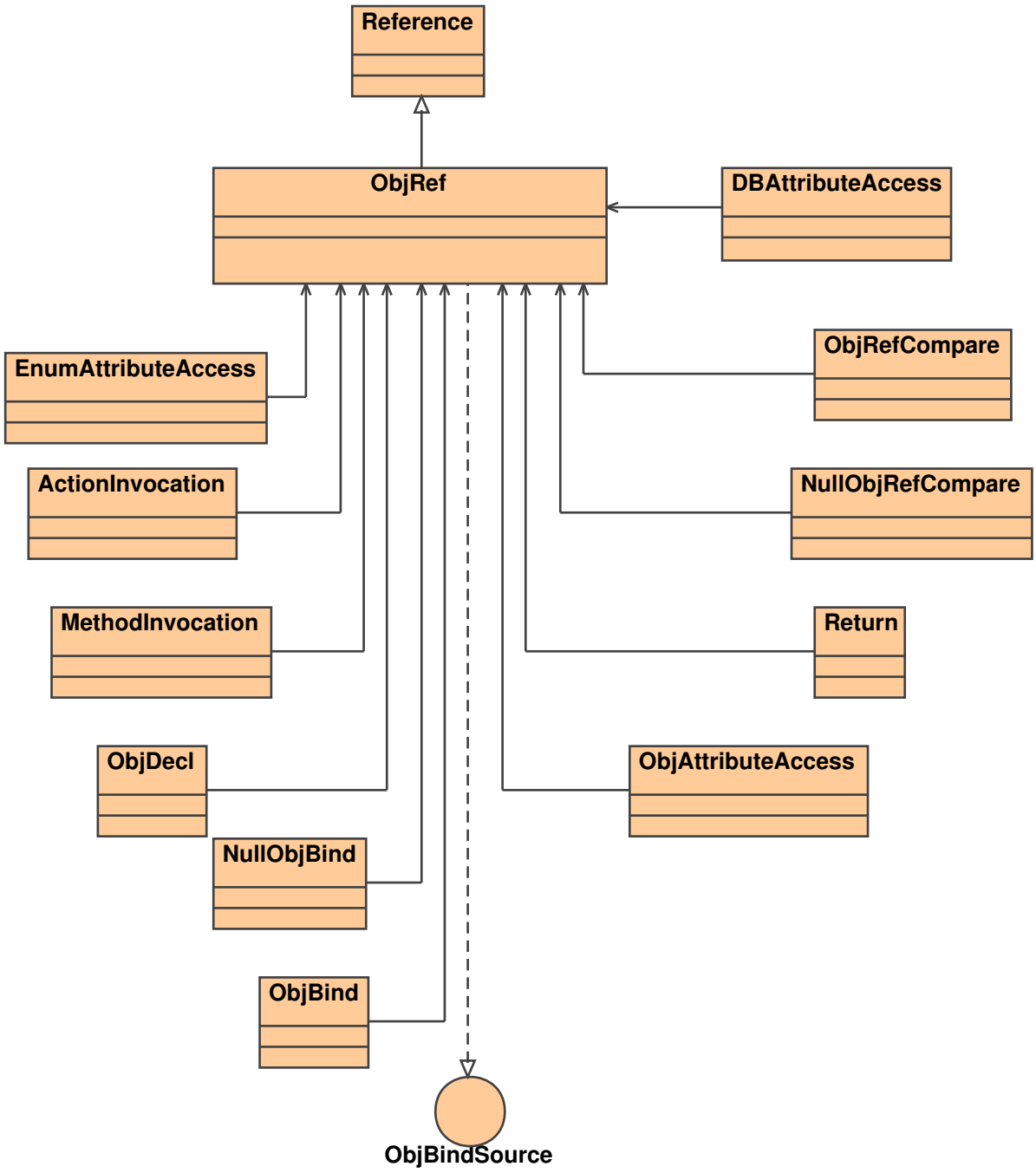


Figura 3.5: Relacionamento da meta-classe que representa a referência a objeto com outros componentes do metamodelo da VIRTUOSI

3.2.4.2 Referência a Bloco de Dados

Segundo o metamodelo da VIRTUOSI, um programador tem a possibilidade de criar novas classes que não dependam de nenhuma outra classe pré-existente. Para tanto, um atributo pode referenciar um bloco de dados. Esse tipo de referência é chamado de referência a bloco de dados. Uma referência a bloco de dados consiste em uma referência

para uma seqüência contígua de dados binários em memória. Um atributo do tipo referência a bloco de dados obrigatoriamente tem uma relação de composição exclusiva com a classe que o possui. Esse tipo de referência é geralmente utilizado para a construção de classes pré-definidas. Cada classe pré-definida é responsável por dar o significado de sua seqüência de dados binários através de suas operações. Por exemplo, um objeto do tipo básico *Integer* pode armazenar um valor inteiro utilizando um bloco de dados de qualquer tamanho, nesse caso uma operação para adicionar um outro valor inteiro (armazenado em outro objeto do tipo *Integer* também utilizando um bloco de dados) ao valor inteiro deste objeto, deve conhecer a convenção utilizada na representação binária de ambos as seqüências. A Figura 3.6 mostra o código fonte de uma classe que possui um atributo do tipo bloco de dado e uma ilustração de uma instância desta classe contendo o bloco de dados.

```
class Inteiro{
    datablock valor;

    constructor make(literal pValor) exports all
    {
        valor = datablock.make(32);
        valor.storeInteger(pValor);
    }
    constructor make(Inteiro pValor) exports all
    {
        this.set(pValor);
    }
    method void set(Inteiro i) exports all
    {
        datablock k = i.valor;
        valor = k.clone();
    }
    ...
}
```

Figura 3.6: Um exemplo de atributo do tipo bloco de dados e uma representação de um objeto correspondente – código fonte em Aram

3.2.4.3 Variável Enumerada

Uma classe implementada segundo o metamodelo da VIRTUOSI, pode ainda, ter um atributo do tipo variável enumerada. Um atributo do tipo variável enumerada possui um conjunto de valores possíveis definidos na construção da classe. Os valores possíveis de uma variável enumerada não são objetos de nenhuma outra classe, são simples valores literais. Esse conjunto de valores possíveis de uma variável enumerada chama-se **enumerado**. Um atributo do tipo variável enumerada recebe um valor inicial durante sua declaração.

3.2.5 Referências

Existem quatro tipos de referências suportadas pelo metamodelo da VIRTUOSI, a saber:

- referência a objeto;
- referência bloco de dados;
- referência a índice;
- referência a literal.

A Figura A.16 mostra o relacionamento de herança entre as referências na VIRTUOSI e mostra também que qualquer referência pode ser passada como parâmetro real e fazer parte dos parâmetros formais de um invocável. Deve-se notar que a meta-classe referência é abstrata, e é concretizada pelos quatro tipos de referência.

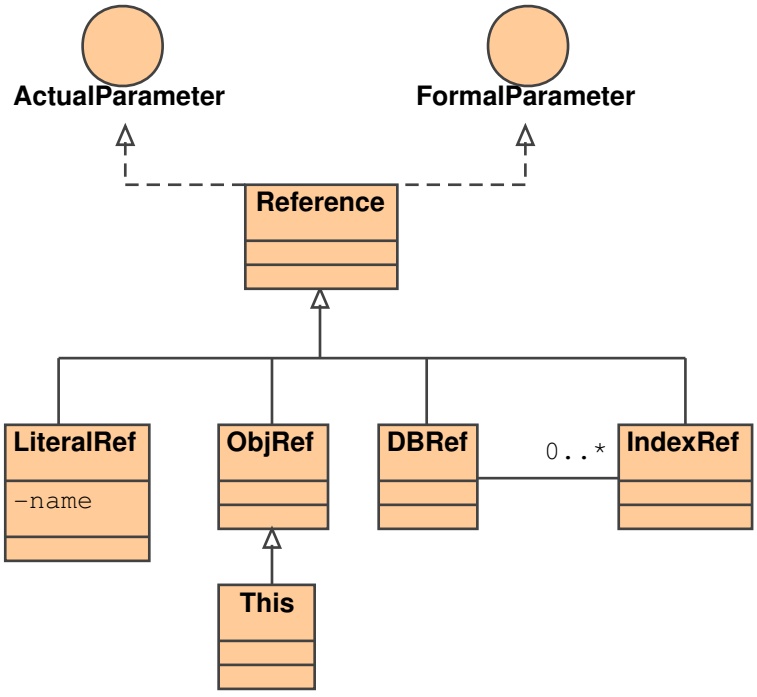


Figura 3.7: Relacionamento entre as meta-classe que definem os tipos de referência na VIRTUOSI

Observando-se a Figura 3.7 nota-se que existe uma meta-classe chamada `This` herdeira da meta-classe que representa uma referência a objeto. Essa meta-classe representa uma referência para o objeto corrente durante a interpretação de um método. Um método sempre é invocado através de uma referência a objeto sobre o objeto o qual ela aponta. Uma referência do tipo `This` é utilizada quando dentro de um método deseja-se invocar um método da própria classe e sobre o próprio objeto corrente.

3.2.6 Invocáveis

Segundo o metamodelo da VIRTUOSI, uma operação de uma classe pode ser implementada de duas maneiras, a saber:

- como um **método**;
- como uma **ação**;

Essas duas implementações descrevem o conjunto dos serviços que uma classe disponibiliza. Além das operações, uma classe precisa implementar um método especial utilizado para criar novas instâncias da classe, esse tipo de método especial é chamado de **construtor**.

Um método, um construtor ou uma ação pode sofrer invocação e, por isso, o metamodelo da VIRTUOSI os formaliza como **invocáveis**[3].

3.2.6.1 Construtor

Um construtor não faz parte das operações definidas por um TAD (*Tipo Abstrato de Dados*), pois não interfere no comportamento dos objetos representados pelo TAD. Porém, têm fundamental importância na implementação de uma classe, visto que, um objeto sempre é criado através de sua interpretação. O retorno da interpretação de um construtor é sempre um novo objeto, uma nova instância de uma classe.

3.2.6.2 Método

Um método é a maneira mais comum de implementar uma operação definida por um TAD. Existem dois tipos de métodos: método sem retorno, muitas vezes chamado de procedimento; e método com retorno, muitas vezes chamado de função. A diferenciação entre métodos com e sem retorno se dá em parte pelo uso da palavra que fica entre a palavra method e o nome do método. Caso essa palavra seja void trata-se de um método sem retorno. Caso a palavra seja o nome de uma classe trata-se de um método com retorno. Outra diferença consiste no fato de um método com retorno sempre possuir ao menos um comando retorno.

3.2.6.3 Ação

A segunda maneira de implementar uma operação definida por um TAD é através de uma ação. Uma ação pode ser vista como uma operação, cujo retorno é utilizado para a tomada de decisão referente a um comando de desvio. Em outras palavras, o retorno de uma ação permite ao comando de desvio decidir qual dentre duas seqüências de comandos deve ser interpretada. Uma ação não retorna uma referência a objeto. Diferente de um método com retorno ou um construtor, onde uma referência é retornada, o retorno de uma ação é um comando simples chamado: comando resultado de teste.

[3]Do inglês ***invocable*** (o termo **invocável** ainda não está registrado nos dicionários da língua portuguesa).

3.2.6.4 Parâmetros

Um invocável pode receber parâmetros. Por isso um invocável define uma seqüência de parâmetros que deve ser provida durante sua invocação (ou chamada).

Um parâmetro pode ser visto sob duas perspectivas diferentes. A primeira ocorre durante a construção de um invocável onde o parâmetro tem o papel de **parâmetro formal**[4], ou seja, ele define o nome, o tipo e a posição que determinado parâmetro possuirá na seqüência dos parâmetros formais daquela invocação. A segunda ocorre no momento da chamada do invocável, onde um parâmetro é uma referência a um objeto já existente, tendo assim, o papel de **parâmetro real**[5].

O conjunto de parâmetros formais de um invocável pode ser vazio ou composto de referências. Qualquer tipo de referência pode ser um parâmetro formal, inclusive uma referência a literal[6] utilizada para receber os parâmetros reais do tipo valor literal.

3.2.7 Comandos

No ambiente VIRTUOSI, toda computação é realizada através da interpretação dos comandos que compõem um invocável. Esses comandos podem ser invocações de outros invocáveis, comandos responsáveis por controlar o fluxo da interpretação, comandos para manipular referências a objetos ou ainda comandos de sistema para a manipulação de referência a bloco de dados e manipulação de referência a índice. Esses comandos são interpretados a partir do momento que um invocável é invocado.

3.2.7.1 Composição de Comandos

Um invocável define uma seqüência de comandos. Comandos podem ser simples ou compostos. Um **comando simples**[7] pode ser uma atribuição, uma invocação de operação, um desvio condicional, conforme detalhado no restante dessa Seção. Um **comando composto**[8] é uma seqüência de comandos simples ou compostos, recursivamente.

3.2.7.2 Declaração de Variáveis

Para se invocar um invocável é preciso possuir uma referência para um objeto da classe que o define(com exceção de construtores que são invocados a partir do nome da classe). Segundo o metamodelo da VIRTUOSI, uma referência existe sob três formas, a saber:

1. atributo;

[4]Do inglês ***formal parameter***.
[5]Do inglês ***actual parameter***.
[6]Embora um parâmetro seja utilizado pelo invocável da mesma forma que uma variável local, no caso de parâmetros do tipo referência a literal, o parâmetro não pode sofrer atribuição, visto que, não existe um comando de atribuição para referência a literal. Portanto, uma referência a literal sempre tem seu valor literal atribuído por um comando de invocação de invocável
[7]Do inglês ***simple statement***.
[8]Do inglês ***compound statement***.

2. parâmetro;
3. **variável local**.

Ao contrário dos atributos e parâmetros que são definidos durante a construção da classe e seus respectivos invocáveis, uma variável local não existe até que seja declarada. Portanto, existem instruções para a declaração de alguns tipos de referência utilizadas como variáveis locais.

3.2.7.3 Atribuição

Uma referência nula não permite uma invocação a partir dela. Isso é verdade tanto para variáveis locais quanto para atributos e parâmetros. Portanto, é preciso fazer com que uma referência aponte para um alvo, ou seja, uma referência a objeto precisa apontar para um objeto em memória, uma referência a bloco de dados precisa apontar para uma seqüência de dados binários em memória e uma referência a índice precisa apontar para uma posição na seqüência de dados binários em memória. Isso ocorre através da interpretação de um comando de atribuição. Esse comando é identificado no código fonte pelo uso do caracter '=' chamado de caracter de atribuição. O que estiver a esquerda do caracter de atribuição é chamado **alvo da atribuição**, e o que estiver a direita do caracter de atribuição é chamado **origem da atribuição**.

Segundo o metamodelo da VIRTUOSI, os comandos de atribuição existentes são os seguintes:

- atribuição de referência a objeto;
- atribuição de referência nula a objeto;
- atribuição de referência a bloco de dados;
- atribuição de referência nula a bloco de dados;
- atribuição de referência a índice;
- atribuição de variável enumerada.

3.2.7.4 Retorno de Método

O comando de retorno utilizado por um método é chamado simplesmente de **retorno**, sendo que sempre retorna uma referência a objeto do mesmo tipo definido pelo tipo de retorno do método.

O metamodelo da VIRTUOSI formaliza o relacionamento entre um comando de retorno e uma referência a objeto, conforme mostra a Figura 3.8.

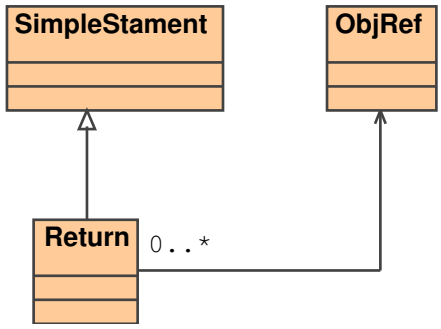


Figura 3.8: Relacionamento entre um comando de retorno e uma referência a objeto

3.2.7.5 Invocação de Invocáveis

O metamodelo da VIRTUOSI define os comandos para realizar a invocação de construtores, métodos com retorno e métodos sem retorno. A invocação de um construtor é realizada a partir do nome da classe que o possui. Já a invocação de um método, com ou sem retorno, é realizada a partir de uma referência a objeto. Tanto o comando de invocação de método com retorno quanto o comando de invocação de método sem retorno podem ser generalizados como comandos de invocações de método. O comando de invocação de método e o comando de invocação de construtor podem ser generalizados como comandos de invocação.

Deve-se notar que uma ação, embora seja um componente invocável, não possui um comando para invocação correspondente. A invocação de uma ação é abordada em detalhe na discussão sobre componentes testáveis.

3.2.7.6 Chamadas de Sistema

Conforme descrito na subseção 3.2.4, o ambiente Virtuosi disponibiliza uma biblioteca de classes pré-definidas (Integer, Real, Boolean, Character e String) para a construção de novas classes chamadas de classes de aplicação. Contudo, o ambiente Virtuosi também disponibiliza comandos simples e testáveis de sistema para a construção de classes que utilizem referências a bloco de dados. As próprias classes pré-definidas disponibilizadas pelo ambiente Virtuosi são construídas com estes comandos e testáveis de sistema. Para cada uma das classes pré-definidas existe um conjunto definido de comandos de sistema e um conjunto definido de testáveis de sistema.

3.3 Árvore de Programa

3.3.1 Justificativa

Desde o final da década de 70, observam-se estudos preocupados em prover portabilidade para aplicações computacionais através do uso de máquinas virtuais. Essa

tendência aumentou com o avanço das tecnologias de rede e a necessidade de integração entre computadores de plataformas heterogêneas.

Uma máquina virtual provê uma camada de abstração sobre o *hardware* e sistema operacional de cada uma das plataformas onde é implementada. Essa estratégia permite que uma mesma aplicação seja interpretada em diferentes plataformas. Uma máquina virtual, como o próprio nome diz, é um programa computacional capaz de interpretar outros programas. Para tanto, os programas devem ser escritos em termos do conjunto de operações e dos tipos de dado suportados pela máquina virtual.

Além da portabilidade, outro benefício da utilização da estratégia de interpretação de software através de máquinas virtuais é a segurança. Uma máquina virtual é geralmente um simples processo – dentre os muitos – em um sistema operacional da arquitetura alvo. Dessa forma, é possível restringir um programa – que é interpretado em uma máquina virtual – de ter acesso de forma direta aos recursos oferecidos pelo sistema operacional nativo.

A tecnologia Java é o principal exemplo de utilização da estratégia de máquinas virtuais. Java tornou-se uma plataforma padrão de desenvolvimento e execução de aplicações portáveis, principalmente em sistemas embarcados[9], em *applets* na Internet ou ainda em aplicações multi-plataforma. A portabilidade de Java é realizada através da compilação de um programa fonte escrito em Java para uma seqüência de instruções da máquina virtual Java. Essa seqüência de instruções é uma representação intermediária do programa fonte, e no caso de Java, é chamada de *bytecodes* do Java. Os bytecodes do Java são independentes da arquitetura computacional onde o programa é interpretado.

Quando um compilador traduz o código fonte em Java para a representação intermediária *bytecode*, as informações estruturais de alto nível presentes no código fonte de um programa são eliminadas. Isto não impede, no entanto, a reconstrução do código fonte de um programa a partir de seus *bytecodes*.

Uma outra representação intermediária chamada de *Slim Binaries* é descrita em [KF99]. Ao invés de *bytecodes*, o código fonte é compilado para uma forma de representação intermediária que preserva as informações estruturais de alto nível, permitindo que otimizações sejam realizadas sem a reconstrução do código fonte. Essa representação intermediária é baseada em árvores sintáticas abstratas, também chamada de árvores de programa.

No ambiente VIRTUOSI, utilizou-se a idéia de árvores sintáticas abstratas encontradas em [KF99] para criar uma representação intermediária própria no formato de **árvore de programa**. Uma árvore de programa na VIRTUOSI pode ser vista com um grafo cujos nós são objetos que representam os elementos encontrados em uma linguagem de programação orientada a objeto, ou seja, uma árvore de programa é um grafo cujos nós são instâncias das meta-classes definidas pelo metamodelo da VIRTUOSI e as ligações entre os nós são as relações entre tais meta-classes.

[9] Do inglês, **embedded systems**

3.3.2 Exemplos de árvores de programa

Cada um dos objetos que compõem uma árvore de programa é uma instância de uma das meta-classes definidas pelo metamodelo da VIRTUOSI. Deve-se notar também que as associações entre os objetos da árvore são as associações definidas segundo o metamodelo da VIRTUOSI.

No contexto desse trabalho, uma **aplicação** VIRTUOSI pode ser vista como uma grande árvore de programa formada por um conjunto de árvores de programa menores ligadas entre si. Cada uma das classes de uma aplicação é traduzida em uma árvore de programa. Por exemplo, na aplicação cujo código fonte é transcrito na Seção A.1 do Apêndice A, existe uma árvore para cada uma das classes da aplicação: *Principal*, *Taxi*, *Pessoa* e ainda existem as árvores referentes as classes pré-definidas: *Integer*, *String* e *Boolean*.

A Figura 3.9 ilustra o relacionamento entre as diversas árvores de programa que compõem uma aplicação.

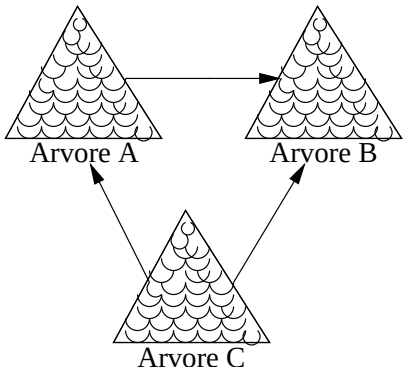


Figura 3.9: Conjunto de árvores de programa que compõem uma aplicação de software

A seguir são mostrados alguns exemplos de fragmentos de código fonte e os respectivos fragmentos de árvore de programa.

3.3.2.1 Exemplo Básico

A Figura 3.10 apresenta o fragmento de código fonte da classe `Pessoa`.

A Figura 3.11 mostra a árvore de programa correspondente ao código fonte da Figura 3.10 através de um diagrama de colaboração UML.

Com base na Figura 3.11, deve-se notar que:

1. Existe um objeto chamado `pessoa` que é uma instância da meta-classe utilizada para representar uma classe de aplicação. Este objeto representa a classe de aplicação `Pessoa`;

2. Associado ao objeto chamado `pessoa` existe um objeto instância da meta-classe utilizada para representar uma referência a objeto, chamado `posicao`. Este objeto está ligado ao objeto chamado `pessoa` por uma associação e realiza o papel de atributo por composição;

```
class Pessoa
{
   composition Integer posicao;
   ...

   method void setPosicao(Integer p) exports all
   {
       posicao = p;
   }
   ...
}
```

Figura 3.10: Fragmento de código fonte da classe `Pessoa`

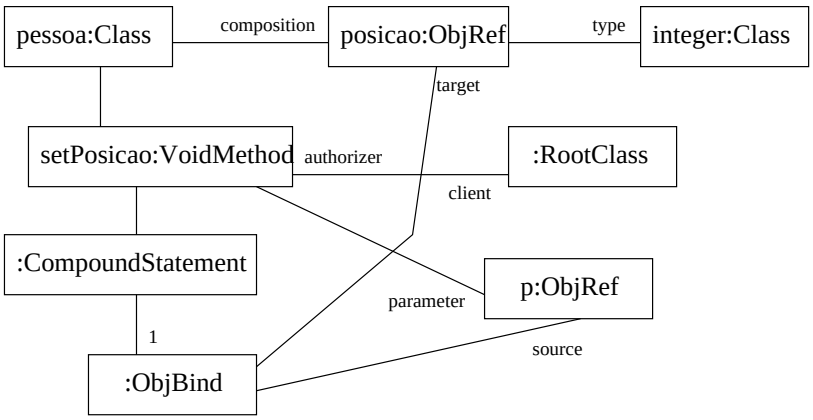


Figura 3.11: Árvore de programa parcial referente à classe Pessoa

3. O objeto chamado `pessoa` também possui uma associação com um objeto chamado `setPosicao` que é instância da meta-classe que representa um método sem retorno. Esse objeto – que representa um método sem retorno da classe de aplicação `Pessoa` – por sua vez, possui uma associação com um objeto instância da meta-classe que representa uma referência a objeto chamado `p`. Este objeto representa o parâmetro da classe de aplicação `Pessoa`;

4. O objeto chamado `setPosicao` possui uma associação com um objeto chamado `raiz` que é instância (única em todo o sistema) da meta-classe que representa a classe raiz. Essa associação é responsável por definir a lista de exportação do método em questão, ou seja, quais as outras classes de aplicação que podem invocar o método em questão. Neste caso especifico, qualquer classe poderá invocar o construtor;

5. Observa-se que o objeto chamado `setPosicao` possui uma associação com um objeto não nomeado instância da meta-classe que representa um comando composto, e este, por sua vez, possui associação com um objeto instâncias da meta-classe que representa um comando simples de atribuição de referência a objeto. O objeto que

representa um comando de atribuição possui duas associações com objetos instância da meta-classe que representa referência a objeto, um representando o alvo da atribuição e o outro representando a origem da atribuição, neste caso específico o par: `posicao` – `p`.

Deve-se notar que os dois objetos instância da meta-classe que representa uma referência a objeto – `posicao` e `p` – possuem uma associação com um objeto instância da meta-classe que representa uma classe de aplicação. Esta associação indica a classe de aplicação da qual a referência a objeto em questão é instância, neste caso específico a classe pré-definida *Integer*. Nota-se portanto, que a árvore de programa em questão não está completa, uma vez que, a classe pré-definida *Integer* não consta no diagrama. Isto ocorre porque cada classe de aplicação ou classe pré-definida define uma árvore de programa particular.

3.3.2.2 Exemplo Avançado

A Figura 3.12 apresenta o fragmento de código fonte da classe `Pessoa`.

```
class Pessoa
{
   composition String nome;
   composition Integer massa;
   ...
   enum {masculino, feminino } sexo = masculino;

   constructor instanciar( String pNome, Integer pMassa, Integer pAltura,
                           literal pSexo) exports all
   {
       nome = pNome;
       massa = pMassa;
       ...
       sexo = pSexo;
       ...
   }
   ...
}
```

Figura 3.12: Fragmento de código fonte da classe `Pessoa`

A Figura 3.13 mostra a árvore de programa correspondente ao código fonte da Figura 3.12 através de um diagrama de colaboração UML.

Com base na Figura 3.13, deve-se notar que:

1. Existe um objeto chamado `pessoa` que é uma instância da meta-classe utilizada para representar uma classe de aplicação. Este objeto representa a classe de aplicação `Pessoa`;

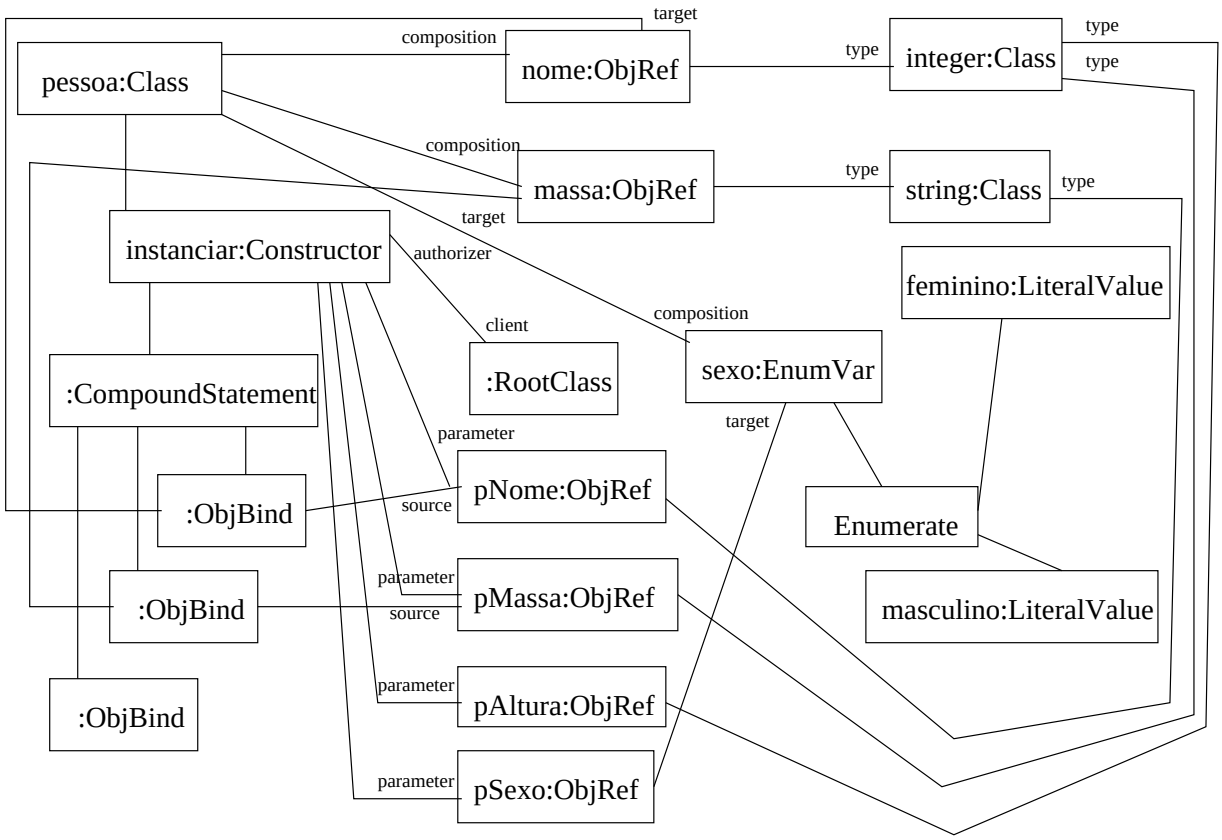


Figura 3.13: Árvore de programa parcial referente à classe Pessoa

2. Associados ao objeto chamado `pessoa` existem dois objetos instâncias da meta-classe utilizada para representar uma referência a objeto, o primeiro deles é chamado `nome` e o segundo é chamado `massa`. Ambos estão ligados ao objeto chamado `pessoa` por uma associação e realizam o papel de atributo por composição;

3. O objeto chamado `pessoa` possui uma associação de atributo por composição com um objeto instância da meta-classe que representa uma variável enumerada chamado `sexo` que, por sua vez, está associado a um objeto instância da meta-classe que representa um enumerado, que, por sua vez, está associado a dois objetos instâncias da meta-classe que representa valores literais, no caso `masculino` e `feminino`;

4. O objeto chamado `pessoa` também possui uma associação com um objeto chamado `instanciar` que é instância da meta-classe que representa um construtor. Esse objeto – que representa um construtor da classe de aplicação `Pessoa` – por sua vez, possui três associações com objetos instâncias da meta-classe que representa uma referência a objeto (chamados respectivamente de `pNome`, `pMassa` e `pAltura`) e uma associação com um objeto instância da meta-classe que representa uma referência a literal chamado `pSexo`. Estes quatro objetos compõem a lista de parâmetros do construtor;

5. O objeto chamado `instanciar` possui uma associação com um objeto chamado `raiz`

que é instância (única em todo o sistema) da meta-classe que representa a classe raiz. Essa associação é responsável por definir a lista de exportação do construtor em questão, ou seja, quais as outras classes de aplicação que podem invocar o construtor em questão. Neste caso especificamente, qualquer classe poderá invocar o construtor;

6. Observa-se que o objeto chamado `instanciar` possui uma associação com um objeto não nomeado instância da meta-classe que representa um comando composto, e este, por sua vez, possui associação com três objetos instâncias da meta-classe que representa um comando simples de atribuição de referência a objeto. Cada um dos objetos que representam os comandos de atribuição possui duas associações com objetos instância da meta-classe que representa referência a objeto, um representando o alvo da atribuição e o outro representando a origem da atribuição (no caso os pares: `nome` – `pNome`, `massa` – `pMassa` e `sexo` – `pSexo`).

Deve-se notar que todos os objetos instâncias da meta-classe que representa uma referência a objeto possuem uma associação com um objeto instância da meta-classe que representa uma classe de aplicação. Esta associação indica a classe de aplicação à qual a referência a objeto em questão é instância. Nota-se portanto, que a árvore em questão não está completa, uma vez que, cada uma das outras classes de aplicação mostradas no diagrama – as classes pré-definidas *String* e *Integer* – não constam no diagrama.

As árvores de programa de uma aplicação geralmente são armazenadas separadamente em um meio físico. Isso faz com que um objeto instância da meta-classe que representa referência a objeto, não possa apontar diretamente para o objeto instância da meta-classe que representa a correspondente classe de aplicação, caso esta pertença a outra árvore de programa. Dessa forma, uma árvore de programa isolada mantém em seus objetos instâncias da meta-classe referência a objeto uma referência simbólica para o objeto instância da meta-classe que representa a classe de aplicação. O mesmo acontece com os objetos instância da meta-classe que representam uma invocação de invocável (construtor, método ou ação) localizado em outra árvore de programa. A Figura 3.14 ilustra a ocorrência de referências simbólicas entre árvores de programa.

Para facilitar o trabalho do carregador de árvores (detalhado na Seção 3.4.2.3) na tarefa de resolver as referências simbólicas entre árvores, cada árvore de programa possui duas listas, a saber:

1. lista de referência a classes;

2. lista de referência a invocáveis.

3.3.3 Lista de Referências a Classe

A lista de referência a classes armazena um indicador para cada um dos objetos instâncias da meta-classe que representa um referência a objeto, existentes na árvore.

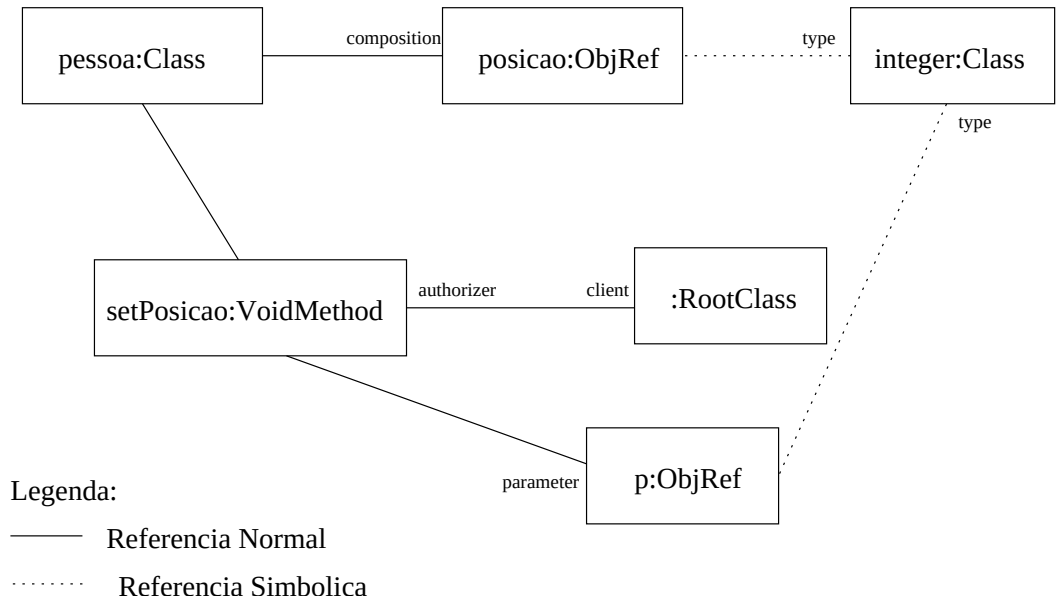


Figura 3.14: Referências simbólicas entre árvores de programa

3.3.4 Lista de Referências a Invocáveis

A lista de referência a invocáveis armazena um indicador para cada um dos objetos instâncias da meta-classe que representa uma invocação de invocável, existentes na árvore.

3.4 Máquina Virtual VIRTUOSI

O ambiente VIRTUOSI tem como um de seus princípios fundamentais o uso de máquinas virtuais distribuídas. O ambiente VIRTUOSI, portanto, define uma máquina virtual – a MVV (***Máquina Virtual*** VIRTUOSI).

A tarefa básica de uma instância da MVV é interpretar os comandos definidos por invocáveis pertencentes a uma árvore de programa. O restante dessa Seção tem como objetivo definir os principais elementos da Máquina Virtual VIRTUOSI e o como estes interagem.

3.4.1 Ciclo de Vida de Uma Aplicação

Uma instância da MVV pode interpretar mais de uma aplicação VIRTUOSI ao mesmo tempo.

Para dar início à interpretação de uma aplicação, deve-se informar à máquina virtual o nome de uma classe de aplicação – classe inicial – e o nome de um construtor desta classe – construtor inicial. A partir dessas duas informações o ciclo de vida de uma aplicação na MVV segue uma seqüência de eventos bem definida:

1. A máquina virtual utiliza o subsistema de carga de árvore para carregar as árvores de programa que compõem a aplicação para a **área de classes**;

2. A máquina virtual cria um novo objeto instância da classe inicial – objeto inicial – e adiciona-o na **área de objetos**;

3. A máquina virtual cria uma nova **atividade** – atividade inicial – associada ao construtor inicial e ao objeto inicial;

4. A máquina virtual empilha a atividade inicial na **pilha de atividades**, o que faz com que a atividade inicial seja interpretada. Novas atividades são empilhadas – recursivamente – na pilha de atividades em resposta a interpretação de comandos de invocação de método e construtor ou em resposta a um comando de desvio condicional cujo teste seja uma ação;

5. Após o término da interpretação da atividade inicial a máquina virtual a retira do topo de pilha de atividades e caso não hajam outras pilhas de atividades com atividades empilhadas, a aplicação é encerrada. A existência de outras pilhas de atividade é discutida na Seção 3.4.4.

3.4.2 Área de Classes

A área de classes é a região de memória onde as árvores de programa de cada uma das classes de aplicação são armazenadas.

Dentro da MVV as referências entre classes sempre são feitas de forma indireta, ou seja, os **pontos de ligação** entre duas árvores de programa – os objetos instâncias da meta-classe que representa uma referência a objeto e os os objetos instâncias da meta-classe que representa uma invocação de invocável – nunca apontam diretamente para seus respectivos alvos – um objeto da meta-classe classe e um objeto da meta-classe invocável. Essas duas ligações na perspectiva da aplicação significam, respectivamente, o tipo de um objeto e o método de uma invocação.

No caso dos objetos instâncias da meta-classe que representa uma referência a objeto, a ligação com o seu respectivo tipo é feita de forma indireta através de uma tabela chamada **tabela de classes**. Essa estratégia é utilizada devido a natureza distribuída do ambiente VIRTUOSI, onde uma referência a objeto pode ser de um tipo (uma classe de aplicação) cuja árvore pode estar localizada na própria instância da MVV ou localizada remotamente em outra instância da MVV.

3.4.2.1 Tabela de Classes

Cada uma das classes de aplicação carregadas na MVV tem uma entrada correspondente na tabela de classes. Cada entrada possui o nome da classe de aplicação para a qual aponta e um indicador para a área de memória onde a classe está localizada. Uma classe pode referenciar uma outra classe localizada em outra máquina virtual. Portanto, existem dois tipos de entrada na tabela de classes, a saber:

- Entrada de Classe Local (ECL) – ver subseção 4.3;

- Entrada de Classe Remota (ECR) – neste caso a entrada também armazena o nome da classe de aplicação, mas ao invés de possuir um indicador para uma área de memória, a entrada possui a identificação da MVV remota e a posição da entrada da tabela de classes remota – ver subseção 4.3.

A Figura 3.15 ilustra a tabela de classes contendo entradas locais e entradas remotas.

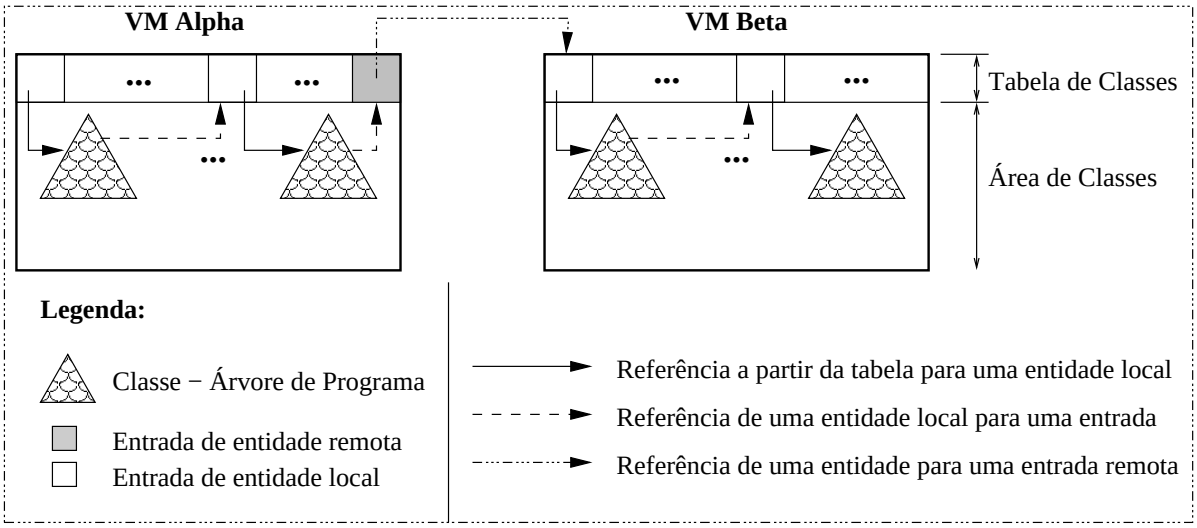


Figura 3.15: Tabelas de classes com referências locais e remotas

No caso dos objetos instâncias da meta-classe que representa uma invocação de invocável a ligação também é feita de forma indireta através de uma tabela chamada **tabela de invocáveis**.

3.4.2.2 Tabela de Invocáveis

Cada um dos invocáveis de cada uma das classes de aplicação carregadas na MVV tem uma entrada correspondente na tabela de invocáveis. Cada entrada possui o nome do invocável apontado, o nome da classe de aplicação que possui o invocável e um indicador para a área de memória onde o invocável está localizado. Pode-se invocar um invocável de um objeto local ou de um objeto remoto. Portanto, existem dois tipos de entrada na tabela de invocáveis, a saber:

- Entrada de Invocável Local (EIL) – ver subseção 4.3;
- Entrada de Invocável Remoto (EIR) – neste caso, a entrada armazena a identificação da MVV remota e a posição da entrada da tabela de invocáveis remota – ver subseção 4.3.

A Figura 3.16 ilustra a tabela de invocáveis contendo entradas locais e entradas remotas.

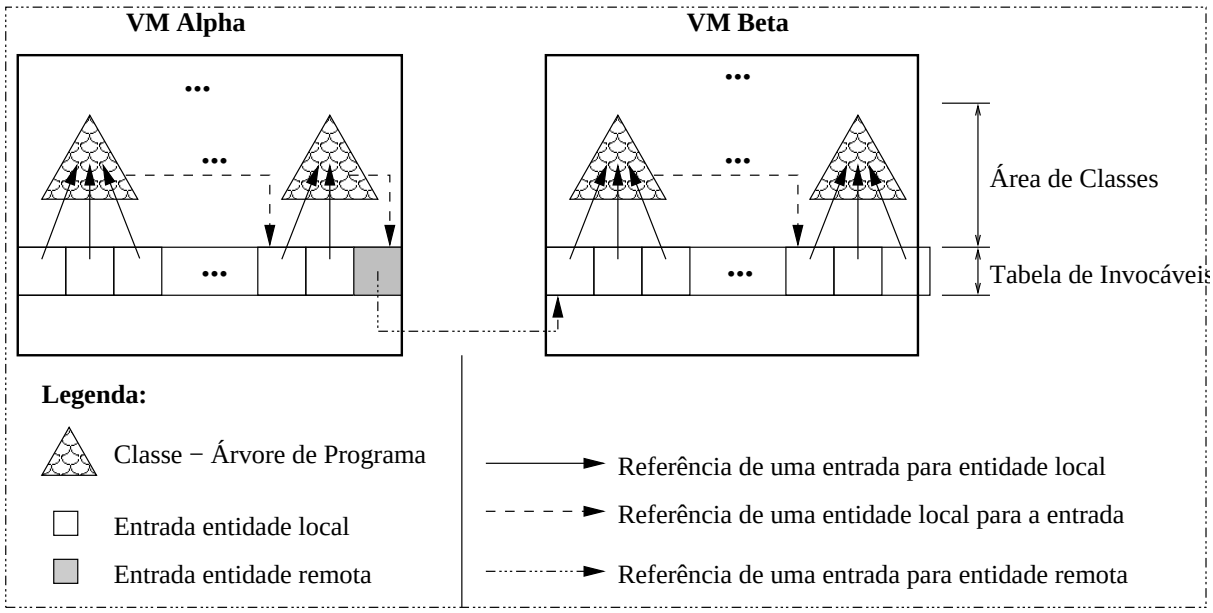


Figura 3.16: Tabelas de invocáveis com referências locais e remotas

3.4.2.3 Carregador de Árvores

Antes da MVV começar a interpretar as árvores de programa que compõem a aplicação, é preciso carregá-las na área de classes da MVV.

O subsistema de carga de árvores de programa da MVV realiza a seguinte seqüência de ações:

1. localizar e carregar uma árvore de programa.

2. para cada objeto instância da meta-classe que representa uma referência a objeto, transformar a referência simbólica em uma referência real que aponte para uma entrada na tabela de classes;

3. para cada objeto instância da meta-classe que representa uma invocação de invocável, transformar a referência simbólica em uma referência real que aponte para uma entrada na tabela de invocáveis.

4. ligar a própria classe de aplicação cuja árvore foi carregada a uma entrada da tabela de árvores; caso a entrada não exista, deve-se criar uma nova.

5. ligar os invocáveis da própria classe de aplicação cuja árvore foi carregada às entradas na tabela de invocáveis; caso a entrada para o invocável não exista, deve-se criar uma nova.

6. refazer essa seqüência de forma recursiva para todas as classes de aplicação referenciadas na árvore corrente.

A Figura 3.17 ilustra (de forma simplificada) a carga das árvores de programa de duas classes de aplicação. No caso, a classe `Pessoa` possui um atributo (uma referência a

objeto) da classe `Integer` e um método da classe `Pessoa` faz uma invocação de um método do atributo da classe `Integer`. Observa-se na Figura 3.17.a a situação da classe `Pessoa` ainda não carregada e com os pontos de ligação ainda como referências simbólicas. A Figura 3.17.b ilustra a situação da classe `Pessoa` ainda não carregada mas com os pontos de ligação já apontados para entradas na tabela de árvore e tabela de invocável. A Figura 3.17.c ilustra a situação da classe `Pessoa` já carregada e com os pontos de ligação já apontados para entradas na tabela de árvore e tabela de invocável. E a Figura 3.17.d ilustra a situação da classe `Pessoa` já carregada e com os pontos de ligação já apontados para entradas na tabela de árvore e tabela de invocável e as entradas apontando para a árvore da classe `Integer`.

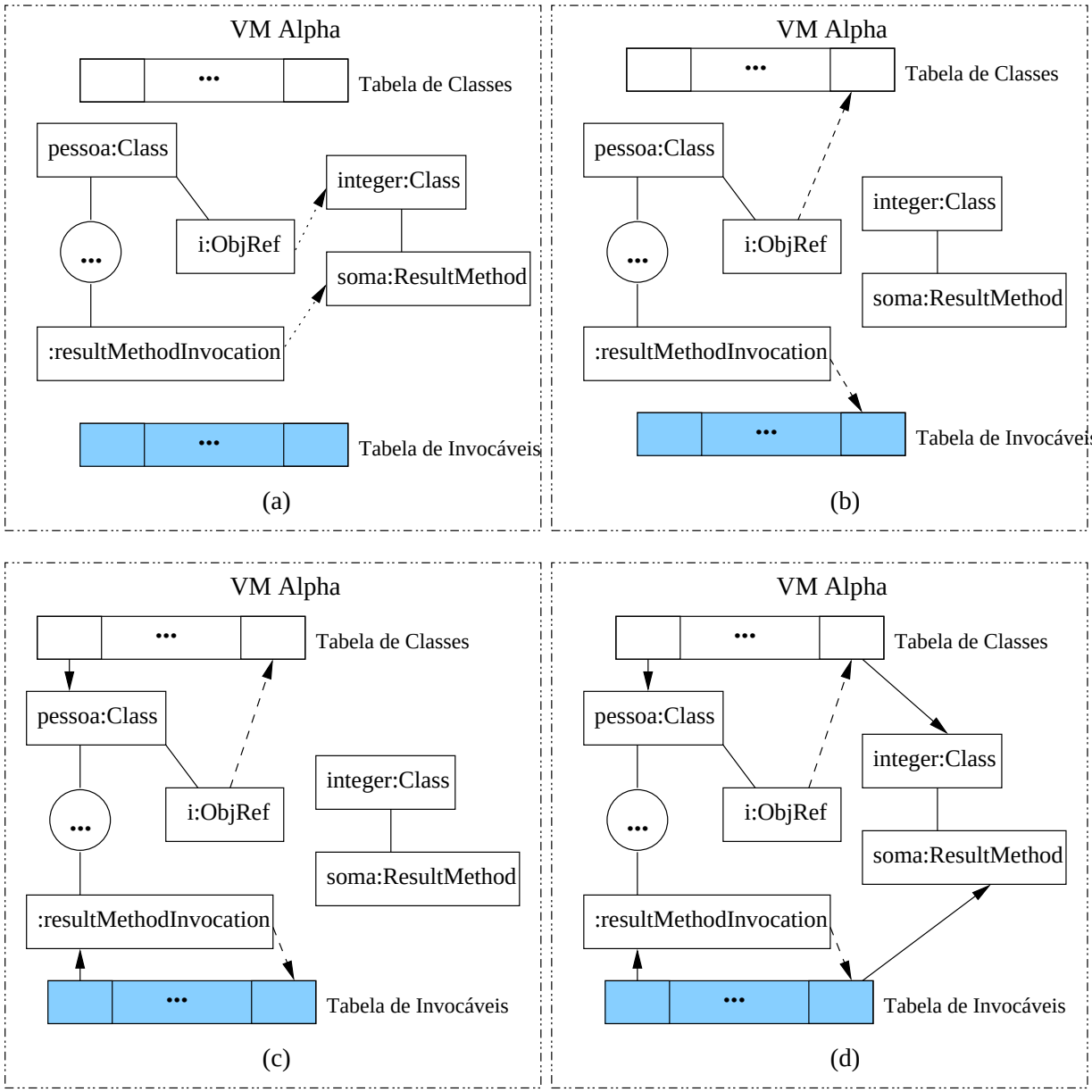


Figura 3.17: Ilustração da carga de duas árvores de programa ligadas entre si

3.4.3 Área de Objetos

A área de objetos é a região de memória onde objetos instâncias de classes de aplicação são armazenados.

Dentro da MVV as referências entre objetos sempre são indiretas, ou seja, um objeto nunca referencia diretamente um outro objeto[10]. A ligação entre os objetos é feita sempre através da **tabela de objetos**. Essa estratégia é utilizada devido a natureza distribuída do ambiente VIRTUOSI, onde objeto estar localizado na própria instância da MVV ou remotamente em outra instância da MVV.

3.4.3.1 Tabela de Objetos

Para cada um dos objetos que a MVV instância, há uma entrada correspondente na tabela de objetos. Cada entrada possui o nome do objeto o qual aponta, o nome do objeto contentor (no caso do objeto estar contido em outro objeto) e um indicador para a área de memória onde o objeto está localizado.

Um objeto pode referenciar um objeto localizado em outra máquina virtual. Portanto, existem dois tipos de entrada na tabela de objeto, a saber:

- Entrada de Objeto Local (EOL);
- Entrada de Objeto Remoto (EOR) – neste caso a entrada também armazena o nome do objeto e nome do objeto contentor (se for o caso), mas ao invés de possuir um indicador para uma área de memória, a entrada possui a identificação da MVV remota e a posição da entrada da tabela de objetos remota.

A Figura 3.18 ilustra a tabela de objetos contendo entradas locais e entradas remotas entre duas MVV.

3.4.3.2 Estrutura de um Objeto

A estrutura interna de um objeto na MVV é composta por:

1. um nome (identificador único em todo o ambiente distribuído);
2. um conjunto de atributos formado por indicadores para a Tabela de Objetos;
3. um conjunto de blocos de dados;
4. um conjunto de variáveis enumeradas;
5. um indicador direto para árvore de programa correspondente à classe de aplicação da qual o objeto é instância.

O estado de um objeto é composto por seus conjuntos de bloco de dados, variáveis enumeradas e atributos.

[10] A mobilidade de objetos não faz parte do escopo deste trabalho e é objeto de estudo de outra dissertação em andamento.

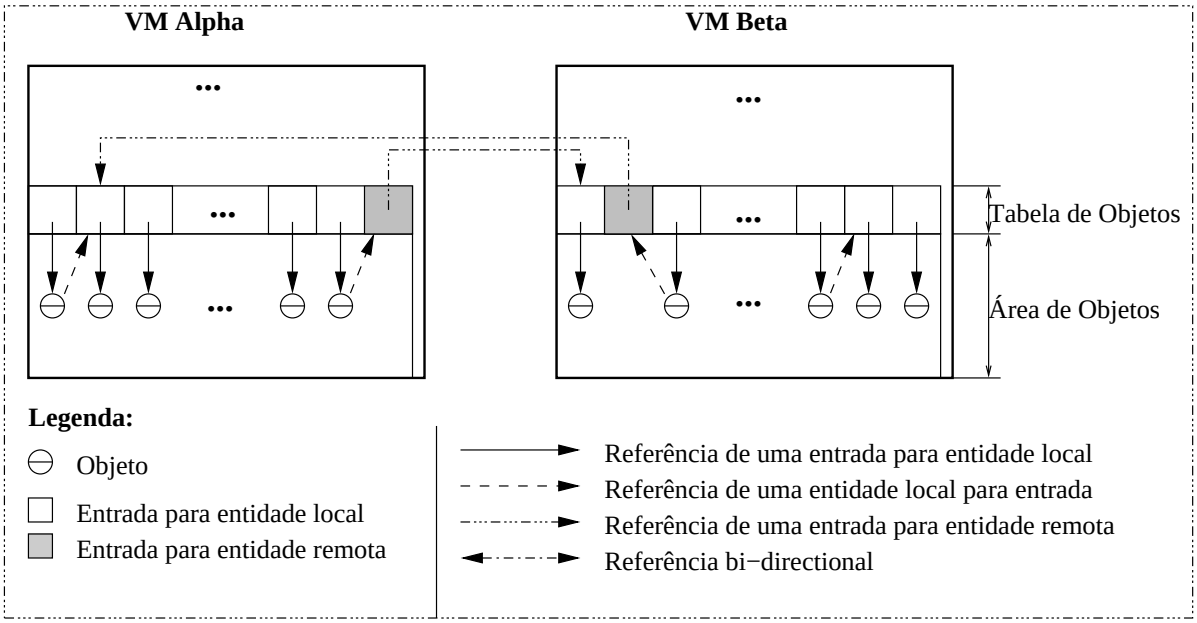


Figura 3.18: Tabelas de objetos com referências locais e remotas

3.4.4 Área de Atividades

Conforme descrito em [Cal00], a forma mais básica de comunicação entre objetos é a invocação de uma atividade de um objeto por outro objeto. Esse mecanismo possibilita a concepção de uma aplicação baseada no encadeamento de atividades de um conjunto de objetos.

A área de atividades é a região de memória da MVV onde as pilhas de atividade são armazenadas. Antes de definir uma pilha de atividades é preciso definir o que é uma atividade.

3.4.4.1 Atividade de Objeto

Uma **atividade** de um objeto corresponde à interpretação de um de seus invocáveis (construtor, método ou ação), ou seja, cada invocação de invocável de um certo objeto dá início a uma nova atividade deste objeto. Toda a computação realizada pela MVV é obtida através da interpretação dos comandos definidos por um invocável. Uma atividade termina quando a interpretação do invocável termina, seja normal ou anormalmente (quando ocorre uma exceção[11]). Duas invocações de dois métodos distintos dão início a duas atividades independentes. Da mesma forma, duas invocações do mesmo invocável também dão início a duas atividades independentes. Assim, para cada instante no tempo, cada objeto na MVV pode ter zero ou mais atividades, dependendo de como são utilizados pelas aplicações. Um mesmo objeto pode, inclusive, ter duas atividades simultâneas pertencentes a aplicações distintas. Nesse modelo de execução, uma aplicação consiste em em um encadeamento de atividades, envolvendo um conjunto de objetos rela-

[11] O mecanismo de tratamento de exceções não faz parte do escopo desse trabalho.

cionados. Cada aplicação determina quais objetos são relacionados, que invocável invoca qual invocável, em que ordem e sob quais condições ocorrem as invocações e ainda, para cada par de **atividade invocadora** e **atividade invocada** o modo de invocação com relação ao sincronismo.

Em relação ao sincronismo, uma atividade invocada pode ser classificada como **síncrona** ou **assíncrona**.

Atividade Síncrona: A atividade invocadora fica bloqueada até que a atividade termine. Necessariamente, então, a atividade invocadora tem que ser notificada do término da atividade invocada, seja ele normal (com um possível retorno) ou anormal (com geração de exceção).

Atividade Assíncrona: A atividade invocadora não fica bloqueada devido à invocação e nem recebe qualquer notificação de término. Assim, a atividade invocadora pode encerrar-se independentemente do que aconteça com a atividade invocada. Nesse caso, se a atividade tiver um retorno, este será ignorado. Igualmente, se gerar alguma exceção, esta não será notificada na atividade invocadora.

Deve-se observar que o modo de sincronismo para um certo invocável não precisa ser fixo, isto é, pode ser escolhido em tempo de execução pela atividade invocadora. Assim, é possível que um mesmo invocável seja interpretado como atividade síncrona em uma situação e como atividade assíncrona em outra, dentro da mesma aplicação ou não.

Estrutura de uma Atividade: Conforme supracitado, uma atividade é referente à um objeto e a um dos invocáveis definidos pela classe de aplicação da qual o objeto é instância. Portanto, a estrutura interna de uma atividade na MVV é composta por:

1. um indicador para o objeto dono da atividade localizado na área de objetos;
2. um indicador para o invocável sendo interpretado;
3. um conjunto de parâmetros;
4. um conjunto de variáveis locais.

Deve-se observar que tanto o conjunto de parâmetros quanto o conjunto de variáveis locais têm como seus elementos referências a objeto, e cada uma dessas referências a objeto possui um indicador para uma entrada na tabela de objetos.

A Figura 3.19 mostra a notação gráfica para a representação de uma atividade de objeto.

3.4.4.2 Pilha de Atividades

Quando uma atividade interpreta um comando de invocação de construtor, método ou ação, isto faz com que uma nova atividade seja criada para a interpretação do invocável em questão.

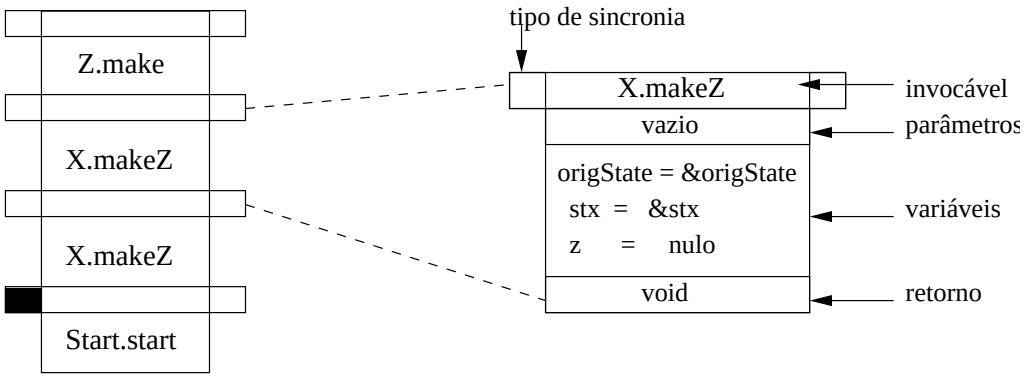


Figura 3.19: Exemplo de uma pilha de execução e o detalhe de uma atividade.

Quando a nova atividade – atividade invocada – é uma atividade síncrona, ela é então empilhada sobre a atividade invocadora. Ao final da interpretação da atividade invocada ela é desempilhada e a atividade invocadora deve continuar a interpretar o próximo comando após o comando de invocação que causou a criação da atividade invocada. Esse processo é recursivo, fazendo que todas as atividades síncronas sejam empilhadas na mesma pilha de atividades onde a atividade invocadora existe.

Quando a nova atividade – atividade invocada – é uma atividade assíncrona, uma nova pilha de atividades é criada e a atividade invocada é empilhada como base da pilha.

Uma vez que a invocação de uma atividade assíncrona cria uma nova pilha de atividades, a área de atividades pode ter mais de uma pilha de atividades sendo interpretada simultaneamente.

Uma nova atividade síncrona ou assíncrona pode ser criada em uma outra instância da MVV. No caso de uma atividade assíncrona uma nova pilha de atividades é criada remotamente e a nova atividade é empilhada como primeiro elemento da pilha. No caso de uma atividade síncrona a nova atividade também é colocada na base de uma nova pilha de atividades localizada na MVV remota, porém ao término da **atividade invocada remota**, a **atividade invocadora local** é desbloqueada e continua sua interpretação normalmente[12].

A Figura 3.20 ilustra uma situação de invocação de atividade assíncrona remota.

Existem dois momentos onde pode haver comunicação entre duas atividades em uma pilha de atividades.

- na passagem de parâmetros;
- no retorno de construtores, métodos com retorno e ações.

No caso dos construtores, após o término da atividade uma referência para o novo objeto criado é adicionado no topo da pilha de atividades. No caso de um método com retorno uma referência a objeto é adicionada no topo da pilha em resultado da interpretação de um comando de retorno. No caso de uma ação, o que é adicionado no

[12] A invocação de invocáveis de forma remota não faz parte do escopo deste trabalho e é objeto de estudo de outra dissertação em andamento.

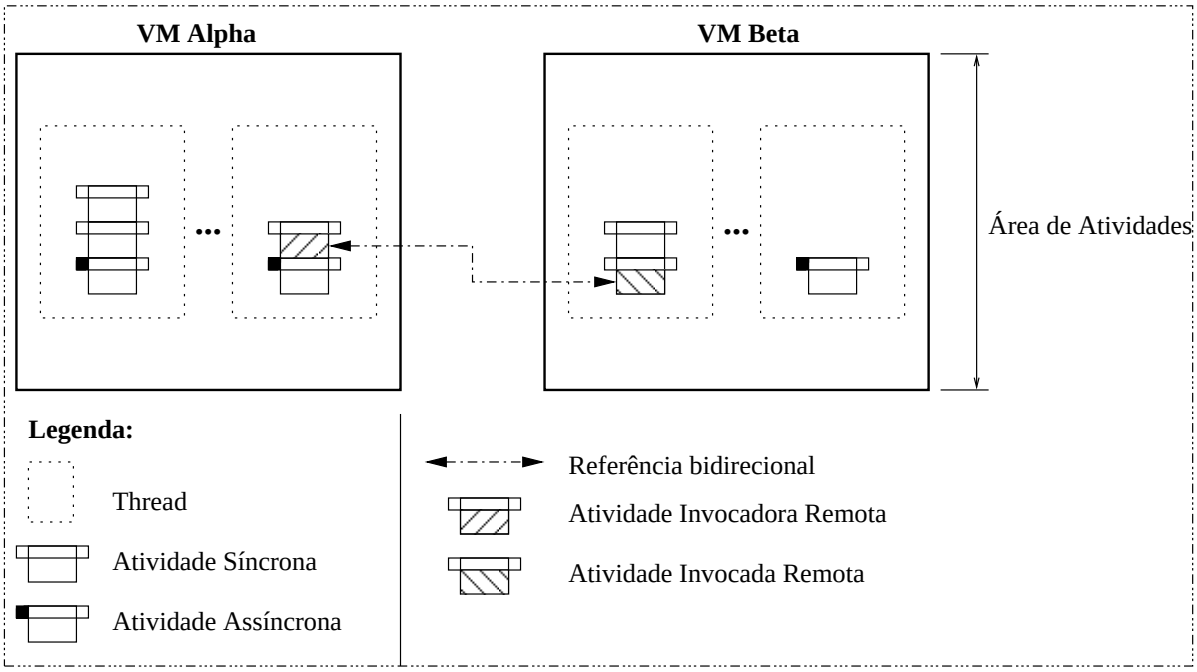


Figura 3.20: Uma atividade assíncrona remota

topo da pilha de atividades é um dos comandos resultado de teste. Portanto, uma pilha de atividades pode empilhar elementos que são atividades, referências a objeto ou até mesmo comandos resultado de teste.

3.4.5 Resumo da Arquitetura da Máquina Virtual VIRTUOSI

Uma visão geral de todas as áreas que compõem a arquitetura da MVV é fornecida na Figura 3.21.

3.4.6 Funcionamento da Máquina Virtual VIRTUOSI

Além das estruturas e áreas de memória da MVV, esse trabalho define as funções principais realizadas pela da MVV a fim de interpretar uma aplicação já carregada na área de árvores.

Em suma, a MVV cria uma atividade relacionada a um determinado objeto e relacionada a um determinado invocável definido na árvore de programa da classe de aplicação cujo o objeto é instância. Então a MVV interpreta os comandos definidos pelo invocável definido pela árvore de programa.

3.4.6.1 Criação de um Objeto

Para a criar um objeto – uma instância de uma classe de aplicação – na MVV é preciso primeiramente obter um nome único para o objeto. De posse do nome do objeto e um indicador para a árvore de programa correspondente à classe de aplicação do objeto, deve-se realizar a seguinte seqüência de ações:

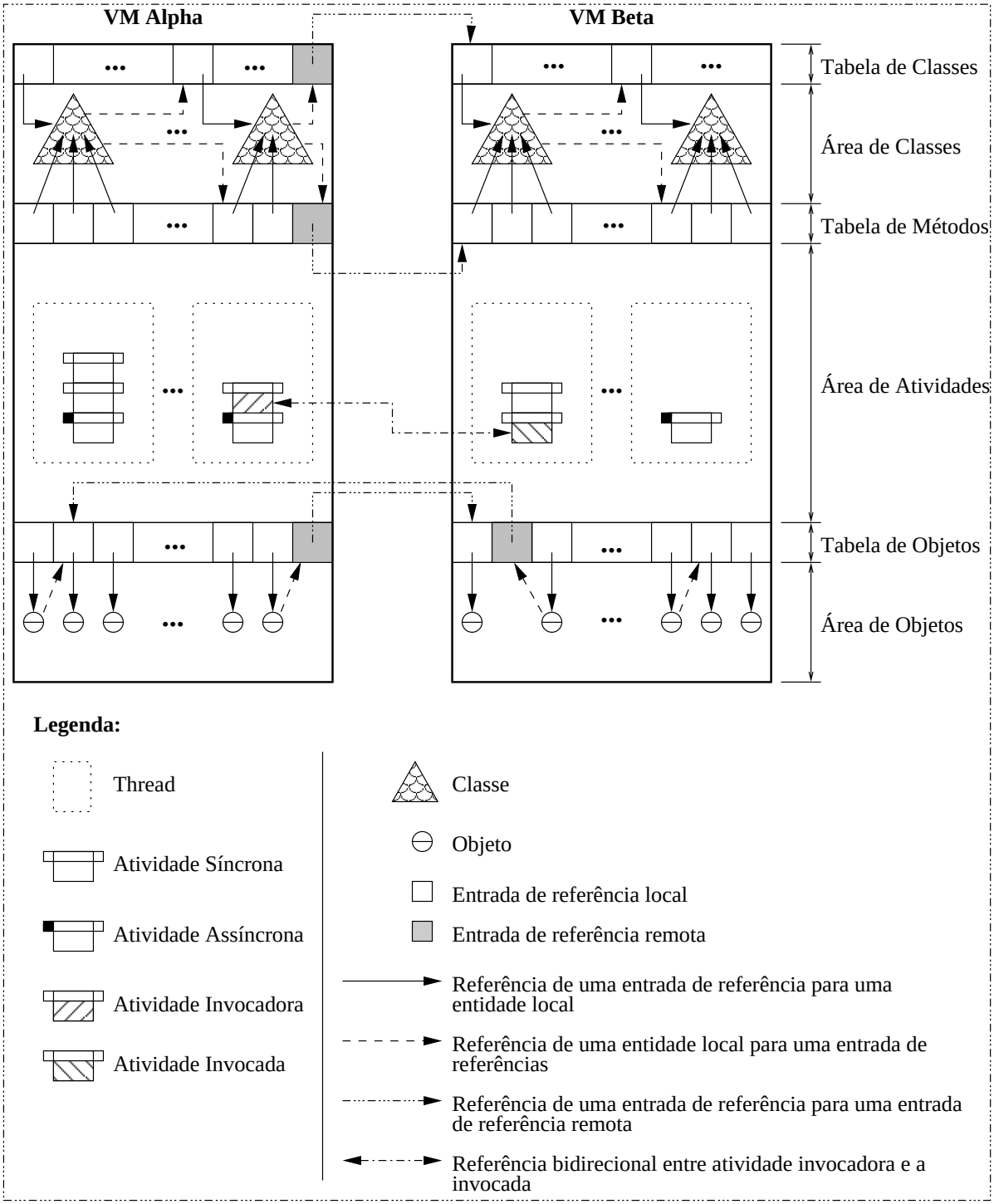


Figura 3.21: Arquitetura da Máquina Virtual VIRTUOSI

1. atribuir o nome ao objeto;

2. atribuir ao indicador do objeto a referência para a árvore de programa correspondente à sua classe de aplicação;

3. para cada um dos atributos referência a objeto definidos na árvore de programa correspondente à classe de aplicação a qual o objeto é instância, criar uma entrada

na Tabela de Objetos com o indicador nulo;

4. para cada um dos atributos blocos de dados definidos na árvore de programa correspondente à classe de aplicação que o objeto é instância, declarar um indicador para uma sequência de dados binários em memória (área de objetos);

5. para cada um dos atributos variável enumerada definidos na árvore de programa correspondente à classe de aplicação que o objeto é instância, declarar uma variável enumerada e atribuir-lhe o valor inicial definido pelo enumerado associado.

Nota-se que o segundo passo supracitado somente cria entradas na Tabela de Objetos para os atributos do objeto que são outros objetos. Isto ocorre porque a alocação de memória destes objetos é realizada através da interpretação dos comandos apropriados (comandos de sistema) durante a interpretação da atividade criada pela invocação do construtor do objeto.

Nota-se também que os atributos que são referências a bloco de dados são somente declarados, a alocação de memória se dá através da interpretação de um comando de sistema apropriado durante a interpretação da atividade criada pela invocação do construtor do objeto. Já os atributos que são variáveis enumeradas tem um espaço de memória alocado dentro do próprio objeto e um valor inicial atribuído, sendo que, este valor pode ser alterado pela interpretação de um comando de atribuição de variável enumerada.

3.4.6.2 Criação de uma Atividade

A criação de uma atividade é um processo simples. Para criar uma atividade na MVV é fornecer um indicador para o objeto sobre o qual a atividade será criada, um indicador para o invocável em questão definido em uma árvore de programa e um conjunto de parâmetros.

3.4.6.3 Interpretação de uma Atividade

Um invocável está associado a um comando composto, e este por sua vez, está associado a uma série de comandos (simples ou compostos, recursivamente) que quando interpretados realizam a computação desejada. Cada comando é um objeto instância de uma meta-classe do metamodelo da VIRTUOSI e portanto possui a informação necessária para ser interpretado – através de suas associações.

Após a criação de uma atividade, a MVV a adiciona no topo da pilha de atividades e ordena a atividade que comece a se interpretar. A atividade então é passada ao comando composto definido pelo invocável para que o comando seja interpretado. A interpretação do comando composto por sua vez, é interpretar cada um dos comandos que ele possui. A atividade é então passada para cada um dos comandos na seqüência em que são interpretados, isto permite ao comando corrente ter acesso às variáveis locais e aos parâmetros da atividade.

Cada comando "sabe" o que deve ser feito, por exemplo, um comando de atribuição de referência a objeto tem associado a ele uma origem e um alvo, e o resultado de sua

interpretação é que a referência alvo passe a apontar para o mesmo objeto apontado pela referência origem. Quando um comando de invocação é interpretado, isto faz com que uma nova atividade seja criada e comece a ser interpretada em um processo recursivo. Quando uma atividade termina de interpretar seu comando composto, isto faz com que a atividade seja retirada da pilha de atividades.

Pode ocorrer também a interpretação de um comando de retorno – no caso de uma atividade criada para responder à invocação de um método com retorno – o que faz com que a atividade corrente seja retirada da pilha de atividades e uma referência a objeto apontando para o objeto resultado da atividade é adicionado no topo da pilha de atividades, permitindo assim, que um comando de atribuição pertencente à atividade invocadora utilize a referência a objeto no topo da pilha como a origem da atribuição. No caso de uma atividade de invocação de construtor o processo é o mesmo, embora não haja um comando de retorno explícito. Quando a atividade é criada em resposta a uma invocação de ação, a diferença é que, neste caso, o que é adicionado à pilha de atividades após a retirada da atividade invocada é um comando resultado de teste que é utilizado pelo comando de desvio pertencente à atividade invocadora.

Cada um dos comandos definidos pelo metamodelo da VIRTUOSI – já descritos na Seção A.1.7 – define as ações realizadas em decorrência de sua interpretação. Abaixo segue uma lista contendo o nome e uma descrição sucinta do resultado da interpretação de cada um dos comandos:

- **Comando Composto** – sua interpretação implica na interpretação de cada um dos comandos que o compõem, de forma recursiva;
- **Comando de Declaração de Variáveis do Tipo Referência a Objeto** – sua interpretação adiciona à atividade invocada uma variável local que referencia um objeto;
- **Comando de Declaração de Variáveis do Tipo Referência a Bloco de Dados** – sua interpretação adiciona à atividade invocada uma variável local que referencia um bloco de dados;
- **Comando de Declaração de Variáveis do Tipo Referência a Índice** – sua interpretação adiciona à atividade invocada uma variável local que referencia um índice;
- **Comando de Atribuição de Referência a Objeto** – sua interpretação faz com que a referência a objeto alvo passe a apontar para o mesmo objeto em memória apontado pela origem da atribuição;
- **Comando de Atribuição de Referência Nula a Objeto** – sua interpretação faz com que a referência a objeto alvo passe a não apontar para nenhum objeto em memória;
- **Comando de Atribuição de Referência a Bloco de Dados** – sua interpretação faz com que a referência a bloco de dados alvo passe a apontar para a mesma

seqüência contígua de dados binários em memória apontada pela origem da atribuição;

- **Comando de Atribuição de Referência Nula a Bloco de Dados** – sua interpretação faz com que a referência a bloco de dados alvo passe a não apontar para nenhuma seqüência contígua de dados binários em memória;
- **Comando de Atribuição de Referência a Índice** – sua interpretação faz com que a referência a índice alvo passe a apontar para a mesma posição de uma seqüência contígua de dados binários em memória apontada pela origem da atribuição;
- **Comando de Atribuição de Variável Enumerada** – sua interpretação faz com que a variável enumerada receba o valor enumerado existente na origem da atribuição;
- **Comando de Retorno de Método** – sua interpretação faz com que a atividade invocada seja retirada do topo da pilha de atividades e em seguida faz com que a referência a objeto – o alvo – seja colocada no topo da pilha de atividades. Dessa forma a referência a objeto retornada poderá ser a origem de uma atribuição na atividade invocadora;
- **Comando de Invocação de Construtor** – sua interpretação faz com que um novo objeto seja criado (um objeto instância da classe de aplicação dona do construtor associado) e uma nova atividade – atividade invocada – seja criada, adicionada no topo da pilha de atividades e por fim interpretada (recursivamente). É possível que existam parâmetros associados ao comando de invocação do construtor, estes parâmetros então são adicionados à atividade invocada;
- **Comando de Invocação de Método sem Retorno** – sua interpretação faz com que uma nova atividade – atividade invocada – seja criada, empilhada no topo da pilha de atividades e por fim interpretada (recursivamente). Ao final da interpretação, a atividade invocada é retirada do topo da pilha de atividades e o próximo comando pertencente à atividade invocadora é interpretado;
- **Comando de Invocação de Método com Retorno** – sua interpretação faz com que uma nova atividade – atividade invocada – seja criada, empilhada no topo da pilha de atividades e por fim interpretada (recursivamente). O fim da interpretação de um método com retorno ocorre pela interpretação de um comando de retorno de método. Dessa forma, o próximo comando da atividade invocadora, que é obrigatoriamente um comando de atribuição de referência a objeto, utiliza a referência a objeto empilhada como a origem da atribuição;
- **Comando Execute** – sua interpretação faz com que a atividade invocada pelo comando de desvio seja retirada do topo da pilha de atividades e em seguida faz com que o próprio comando *execute* seja empilhado no topo da pilha de atividades;

- **Comando Desvie** – sua interpretação faz com que a atividade invocada pelo comando de desvio seja retirada do topo da pilha de atividades e em seguida faz com que o próprio comando *desvie* seja empilhado no topo da pilha de atividades;

- **Comando de Desvio Condicional** – sua interpretação faz com que uma nova atividade seja invocada. Após a atividade ser executada o comando de desvio condicional desempilha o resultado de teste empilhado no topo da pilha de atividades. Caso o resultado de teste seja um comando de *execute*, então, o caminho destino (que por sua vez é um comando simples ou composto) é interpretado recursivamente. Caso o resultado de teste seja um comando *desvie*, então, o caminho alternativo (que por sua vez é um comando simples ou composto) é interpretado recursivamente;

- **Comando de Desvio Incondicional** – sua interpretação faz com que o caminho destino (que por sua vez é um comando simples ou composto) seja interpretado recursivamente;

- **Comando de Vazio** – sua interpretação não possui nenhum efeito, seu uso está geralmente associado à simulação de um comando de repetição que não realiza nenhuma computação;

- **Comandos de Sistema** – são os comandos disponibilizados pela MVV para realizar as operações de baixo nível, ou seja, operações que manipulam blocos de dados, ou que manipulam índices de bloco de dados. O Apêndice A mostra uma lista com os comandos de sistema disponibilizados pelo ambiente VIRTUOSI.

Capítulo 4

Mecanismo de Invocação Remota de Métodos

4.1 Arquitetura

Uma das principais características da arquitetura VIRTUOSI é a utilização de tabelas de referências (ver seção 4.3), a utilização dessas tabelas proporciona a facilidade para controlar, manter e distribuir os objetos entre máquinas virtuais. As tabelas de referências constituem um importante fator na distribuição dos objetos, pois através delas é que o mecanismo de invocação remota de métodos se torna transparente.

Outro ponto importante da arquitetura é a utilização de um serviço de nomes. O serviço de nomes tem o objetivo de registrar, remover e recuperar dos endereços de outras máquinas virtuais para que as mesmas possam se comunicar, conforme descrito na seção 4.2.

Nas seções que seguem serão abordados os itens citados acima com mais detalhes, no entanto, antes de abordar estes detalhes é importante descrever as premissas que foram seguidas para implementar o mecanismo de invocação remota de métodos.

4.1.1 Premissas

É importante salientar que foram seguidas algumas premissas que proporcionam uma melhor compreensão sobre a abordagem que foi utilizada para desenvolvimento do mecanismo de comunicação entre objetos remotos.

- Inicialmente todos os objetos são locais, ou seja, nenhum objeto é criado remotamente.
- Um objeto somente passa a ser remoto caso ocorra a migração do objeto, seja via programação ou via ferramenta externa. O mecanismo de migração de objetos da VIRTUOSI pode ser observado em [dCCF04].
- Não existe retorno de método por valor, o conteúdo que será retornado é a referência para o objeto.

4.1.2 Tipos de Atividades

Conforme citado na subseção 3.4.4.1 para cada vez que ocorre uma invocação de um invocável (seja construtor, método ou ação) , presente em uma máquina virtual, é criada uma atividade. Conforme apresentado na figura 4.1, as classes *Caller*, *Callee* e *LocalActivity* foram derivadas da classe *Activity*. Assim é possível identificar se uma atividade da máquina virtual é do tipo remota ou local.

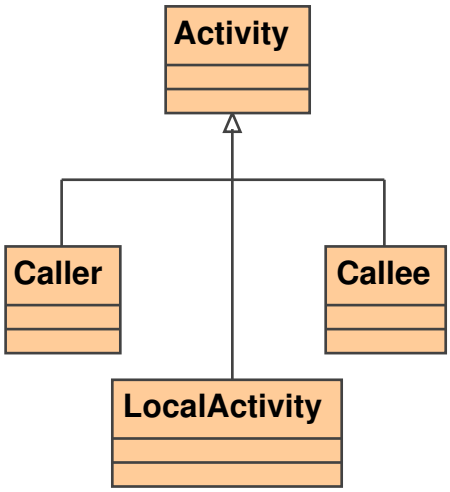


Figura 4.1: Classes relacionadas com atividade remota

Para que ocorra a invocação de um método remoto, é necessária a criação de uma atividade que controle a invocação deste método, tanto do lado que invoca o método, quanto do lado que executa o método remotamente.

Para designar a atividade que invoca um método remoto, a máquina virtual instância a classe chamada *Caller* que é estendida da classe *Activity*. No lado remoto, quando a máquina virtual recebe uma mensagem de execução de método remoto, a máquina virtual cria uma atividade – denominada *Callee* – para executar o método invocado.

Em se tratando de uma atividade local a máquina virtual cria uma instância da classe *LocalActivity*.

4.1.3 Invocáveis

Conforme descrito na subseção 3.2.6, um invocável pode ser um método, construtor ou ação.

Pode-se observar na na figura 4.2 que para cada tipo de invocação existe um relacionamento com o invocável correspondente. Por exemplo, uma invocação de um método com retorno (*ResultMethodInvocation*) é relacionado com a classe *ResultMethod*. Pode-se notar também na figura 4.2 que as classes *ConstructorInvocation* e *MethodInvocation* são derivadas da classe *Invocation* com a exceção da classe *ActionInvocation*. Isto se deve ao

fato que a classe *Invocation* corresponde a um comando simples[1], enquanto que a classe *ActionInvocation* corresponde a uma instrução que pode ser testada[2]. No caso de uma invocação remota de uma ação o retorno será *skip* ou *execute*, *skip* indica que a condição de teste é falsa e *execute* indica que a condição de teste é verdadeira.

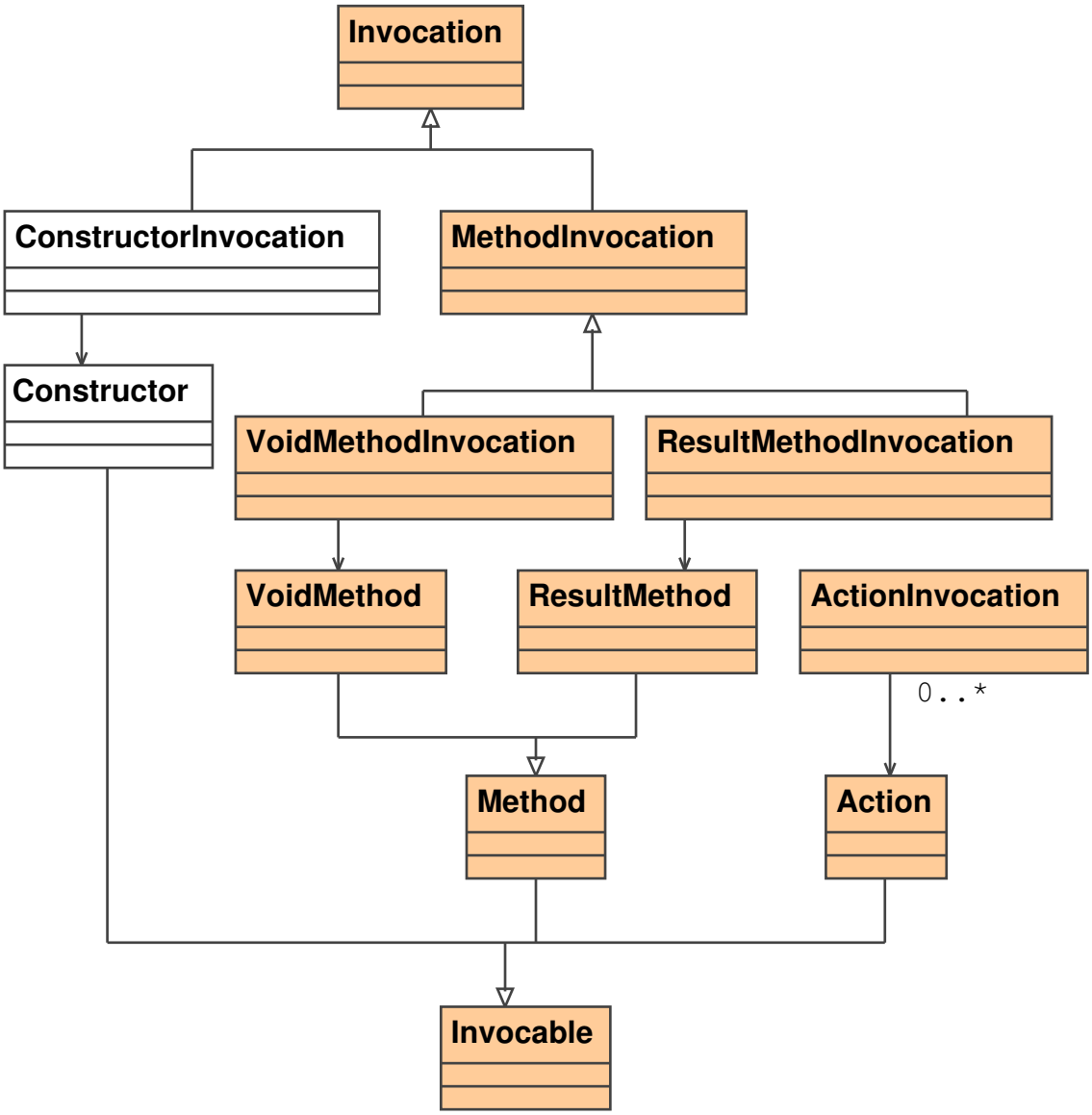


Figura 4.2: Classes Invocávies

Uma invocação de um invocável remoto ocorre quase que analogamente a uma invocação de um invocável local, a invocação remota é totalmente transparente do ponto de vista do usuário (programador), devido a utilização das tabelas de referências, conforme explicado na seção 4.3. A única diferença a ser ressaltada é que não é permitido invocar o construtor de um objeto remotamente, conforme citado na subseção 4.1.1.

[1]Do inglês Simple Statement
[2]Do inglês Testable

Assim, a invocação de um invocável remoto somente pode ser do tipo *MethodInvocation* ou *ActionInvocation*, pois como citado anteriormente, não é utilizada a invocação *ConstructorInvocation* por não haver invocação remota de construtores. Existem duas classes derivadas de *MethodInvocation*, *VoidMethodInvocation* e *ResultMethodInvocation*.

4.1.3.1 Parâmetros

Em relação aos parâmetros dos métodos remotos, pode-se dizer que existem dois modelos de transmissão de parâmetros:

Por Valor: Um único parâmetro que é passado por valor é : *Literal*.

Por Referência: Os demais tipos de objetos passados com parâmetro são transmitidos por referência.

4.1.4 Linguagem de Programação

A linguagem Aram [Aro04] é uma linguagem de programação desenvolvida com o objetivo de facilitar o aprendizado do paradigma da orientação a objetos. A linguagem segue as definições do metamodelo VIRTUOSI e obriga a utilização rigorosa do paradigma da orientação a objetos não permitindo o emprego de recursos da programação imperativa.

Por ser a primeira versão da linguagem a mesma possui algumas restrições como:

- vetores;
- classes aninhadas[3];
- pacotes;
- polimorfismo;
- *for*[4];
- variáveis globais;
- tratamento de exceções;
- interfaces e classes abstratas;
- e principalmente a ausência de herança entre objetos.

Um ponto importante da linguagem é o fato de que a mesma é voltada para o aprendizado do paradigma da orientação a objetos, assim ela obriga o uso de termos como *composition*, *association*, *export*, *method* e *constructor*. Estes elementos se comparado com a linguagem de programação Java proporcionam ao aprendiz um conhecimento específico sobre o funcionamento do seu algoritmo em relação a orientação a objetos.

[3]Do inglês inner class

[4]Laço de estrutura de controle

O produto da compilação de uma classe escrita em Aram gera uma árvore de programa, a partir da qual a *Virtuosi* executa o código gerado.

Um ponto importante a ressaltar é que a arquitetura VIRTUOSI é independente da linguagem de programação Aram. O elemento chave da arquitetura é metamodelo, assim se por exemplo, se houvesse a necessidade da linguagem de programação Java gerar programas para serem executados no ambiente VIRTUOSI, seria possível desde que ela seguisse o metamodelo VIRTUOSI.

Os algoritmos descritos nesta dissertação utilizam a sintaxe da linguagem Aram, embora seja utilizada a linguagem de programação Java para o desenvolvimento dos estudos de casos.

4.1.5 Estrutura da Arquitetura

Do ponto de vista da invocação de métodos remotos a estrutura da arquitetura VIRTUOSI pode ser dividida em quatro componentes: Serviço de Nomes, Tabelas de Referências, Meta-Classes e Protocolo das Mensagens.

O serviço de nomes juntamente com as tabelas de referências auxilia na localização de classes, métodos e objetos.

As tabelas de referências é um mecanismo que permite controlar os acessos às definições das classes, métodos e objetos. Ou seja, não é permitido que ocorra nenhum acesso a métodos, classes ou objetos, todas as referências são feitas pelas tabelas de referências.

Através do meta-modelo e das meta-classes VIRTUOSI é possível a execução de um programa suportado pela VIRTUOSI e através destas meta-classes que é possível identificar os métodos com ou sem retorno, construtores, ações, referências de objetos, parâmetros e assim por diante.

Basicamente o conjunto dos grupos citados acima, tabelas de referências, serviço de nomes, meta-classes e o protocolo de mensagens constituem a arquitetura utilizada para a invocação de métodos remotos.

Visando uma melhor especificação da arquitetura serão apresentados nas seções seguintes os itens: serviço de nomes, as tabelas de referências, protocolo das mensagens e o próprio procedimento para invocação remota de métodos.

4.2 Serviço de Nomes

O serviço de nomes da VIRTUOSI tem o objetivo de registrar, remover e recuperar os endereços das máquinas virtuais que estão sendo executadas em uma determinada rede de computadores. Cada serviço de nomes corresponde a uma comunidade de máquinas virtuais. O enfoque principal do serviço de nomes se refere somente a localização das máquinas virtuais e não a localização de objetos, uma vez que a localização dos objetos é obtida através da tabela de referências de objetos conforme explicado na seção 4.3.

Após o serviço ser iniciado, ele entra em estado de espera aguardando por requisições de registro ou remoção de registro de máquinas virtuais, ou para informar os

endereços das máquinas virtuais. O serviço de nomes utiliza a porta 8874 para aguardar requisições, quando chega uma requisição, a classe *NameService* cria uma *thread*[5], representada pela classe *ThreadNameService* (ver figura 4.3), com o objetivo de realizar todo o processo de troca de mensagens e efetivação do serviço solicitado. Através desta abordagem é possível atender diversas solicitações simultâneas evitando que o serviço fique dedicado somente a uma máquina virtual enquanto outras máquinas fiquem aguardando para serem atendidas.

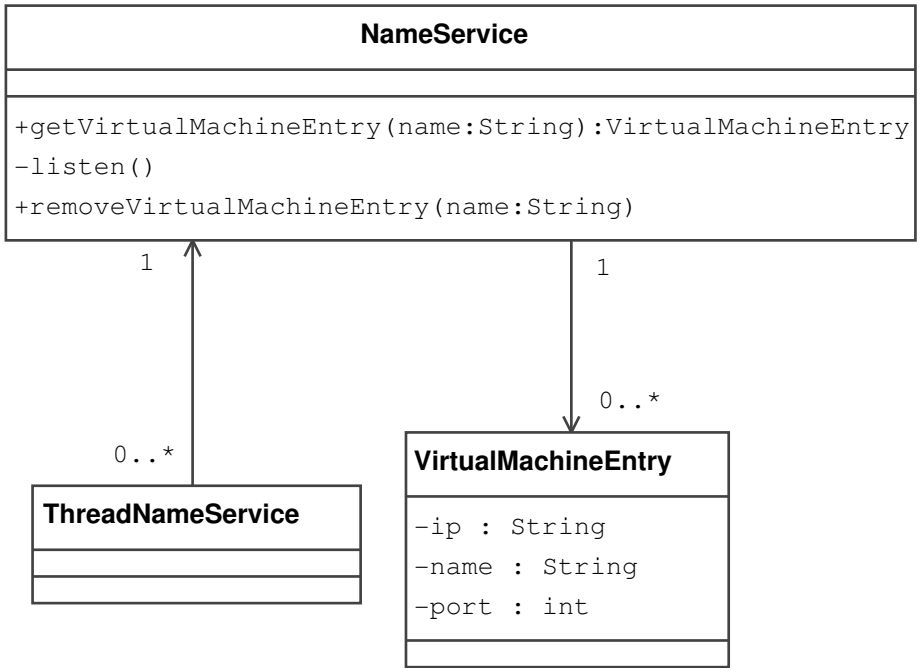


Figura 4.3: Modelo do serviço de nomes

Conforme o modelo apresentado na figura 4.3, para cada requisição de registro no serviço de nomes é criada uma instância da classe *VirtualMachineEntry*, assim a classe *NameService* mantém uma coleção de entradas de máquinas virtuais com os devidos endereços e portas, que podem ser recuperados assim que for necessário.

4.2.1 Mensagens do Serviço de Nomes

As solicitações de serviços para o Serviço de Nomes são efetuadas através de troca de mensagens.

Para efetuar o registro da máquina virtual no serviço de nomes são realizados os seguintes passos, os quais podem ser visualizados através da figura 4.4:

1. Máquina virtual envia mensagem REGVM para o servidor de nomes;

2. Envia o nome da máquina virtual;

[5]Parte de um programa que pode ser executado independentemente de outras partes

3. Envia o endereço IP;
4. Envia o número da porta da máquina virtual e finalmente;
5. O servidor de nomes envia uma mensagem de confirmação.

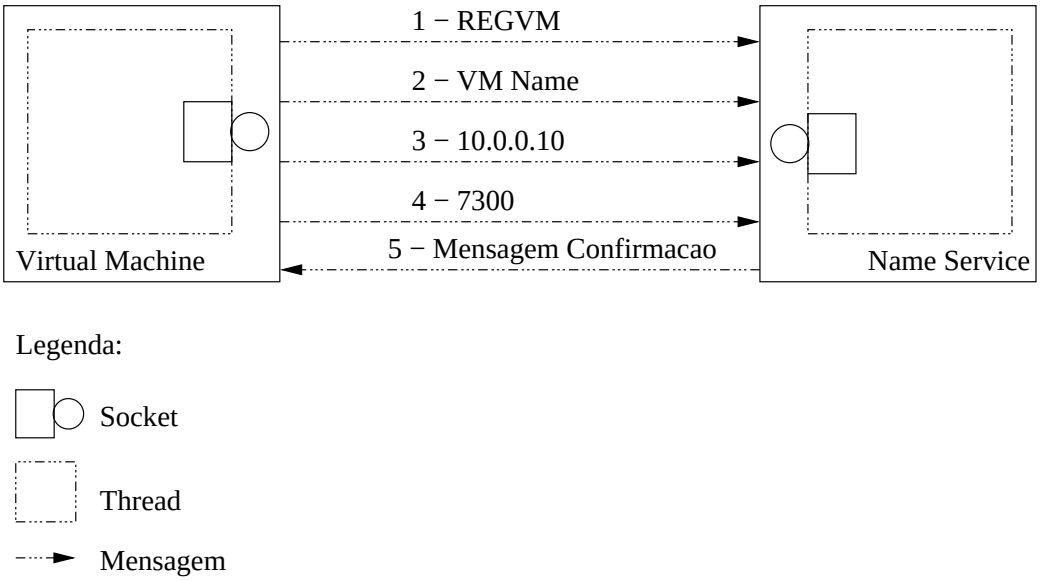


Figura 4.4: Mensagens para registrar máquina virtual

A remoção do registro da máquina virtual do serviço de nomes é realizada pelos passos descritos abaixo assim como mostrados na figura 4.5.

1. Máquina virtual envia mensagem UNREGVM;
2. Envia o nome da máquina virtual e;
3. Servidor de nomes envia mensagem de confirmação.

Para recuperar o endereço de uma máquina virtual são seguidos os passos abaixo e também representados pela figura 4.6.

1. Máquina virtual envia mensagem GETVM;
2. Envia o nome da máquina virtual e;
3. Se o servidor de nomes não encontrar a máquina virtual então é enviada a mensagem VMNOTFOUND;
4. Caso a máquina virtual for encontrada então é enviado o endereço IP;
5. Envia a porta da máquina virtual e finalmente;
6. O servidor de nomes envia uma mensagem de confirmação.

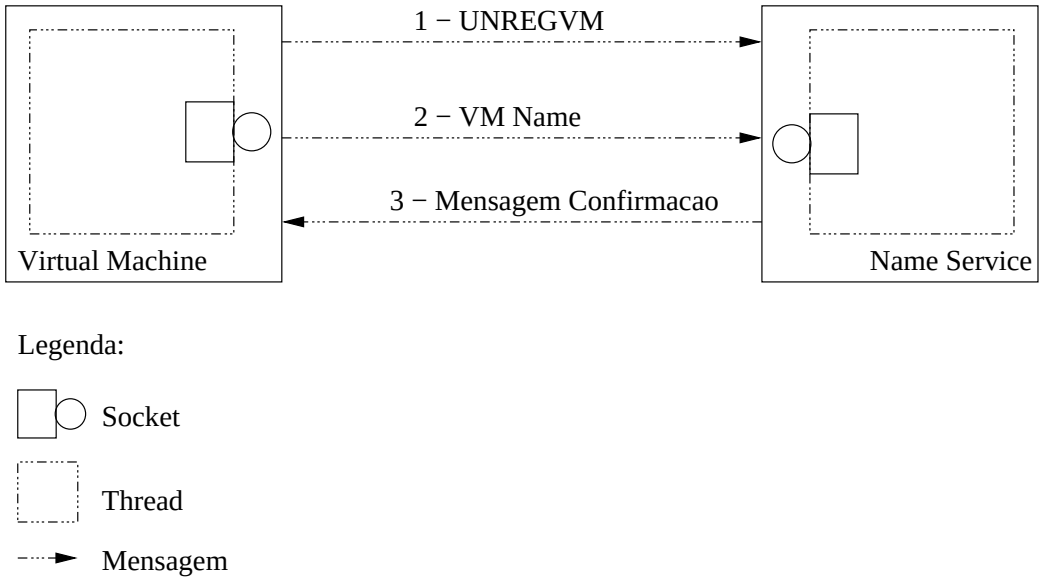


Figura 4.5: Mensagens para desregistrar máquina virtual

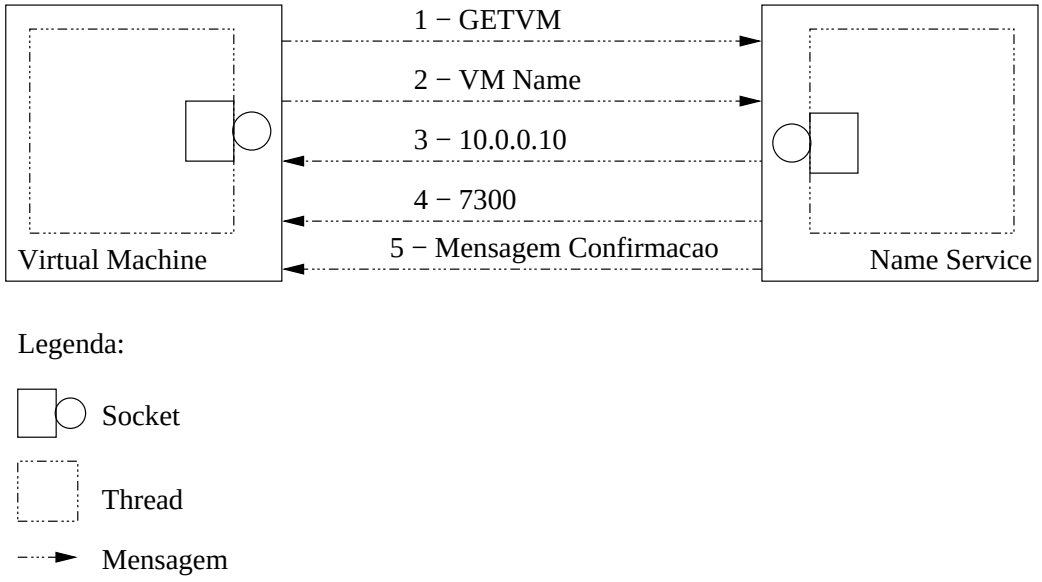


Figura 4.6: Mensagens para obter endereço da máquina virtual

4.3 Tabelas de Referências

Um dos pontos chaves no ambiente VIRTUOSI é a utilização de tabelas de referências. Quando uma árvore de programa é carregada pela máquina virtual VIRTUOSI ela traz consigo as definições da classe como seu nome, atributos, ações, métodos, etc. Após a carga da árvore de programa é que ocorre a inicialização das tabelas de referências de classes, invocáveis e objetos.

Pode-se observar através da figura 4.7 que existem três tabelas de referência associadas com a máquina virtual, uma tabela para referenciar as classes, outra tabela para referenciar os invocáveis e uma última tabela para referenciar os objetos.

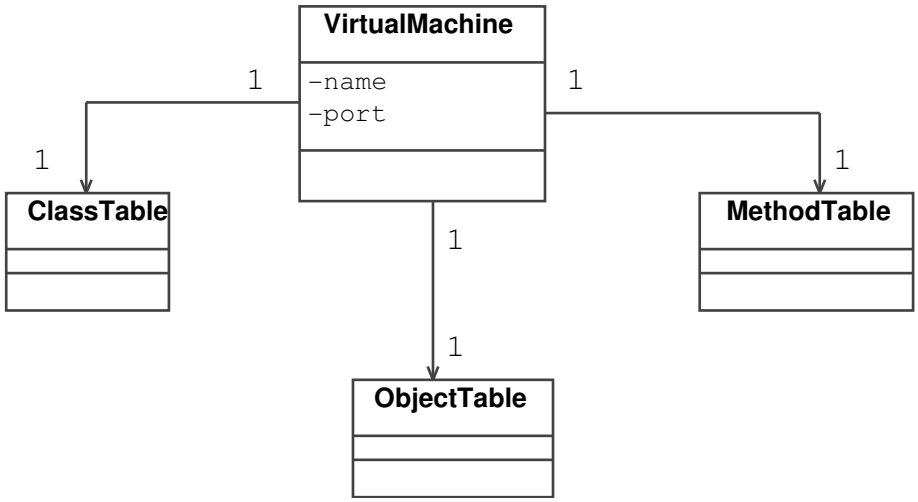


Figura 4.7: Relacionamento entre máquina virtual e as tabelas de referências

Para cada instância da máquina virtual existe também somente uma instância de cada tabela de referência. Utilizou-se no desenvolvimento destas classes o padrão de projeto *Singleton* conforme descrito em [Eri95], desta forma garantiu-se que uma única instância da classe fosse criada para cada tabela.

Nas próximas subseções serão apresentados com mais detalhes as tabelas de referências de classes, invocáveis e objetos.

4.3.1 Tabela de Classes

A Tabela de Classes é um mecanismo que permite à máquina virtual VIRTUOSI referenciar as definições de todas as classes localizadas na máquina virtual. Estas referências para as classes podem ser locais ou remotas.

Para cada classe carregada é criada uma referência na tabela de classes, inicialmente as referências são todas locais, se ocorrer a migração de algum objeto, o proceso de migração pode trazer a referência da classe do objeto migratório e atualizar a tabela de classes como referências locais ou simplesmente criar uma referência na tabela de classes como sendo remota, caso não ocorra a migração da árvore de programa. Assim, se a máquina virtual necessitar de alguma informação sobre uma classe remota, será necessário abrir uma conexão com a máquina virtual remota para obtenção da informação.

As entradas de classes conforme figura 4.8 estão representadas pelas classes *ClassTableEntry*, *LocalClassTableEntry* e *RemoteClassTableEntry*.

A classe *ClassTableEntry* possui duas derivações: *LocalClassTableEntry* - ECL (*Entrada de Classe Local*) e *RemoteClassTableEntry* - ECR (*Entrada de Classe Remota*).

A classe *LocalClassTableEntry* é utilizada para referenciar as classes que possuem suas definições localizadas na mesma máquina virtual onde está sendo executado o programa. Através da associação com a tabela *Classe* é que é possível obter as definições contidas na classe.

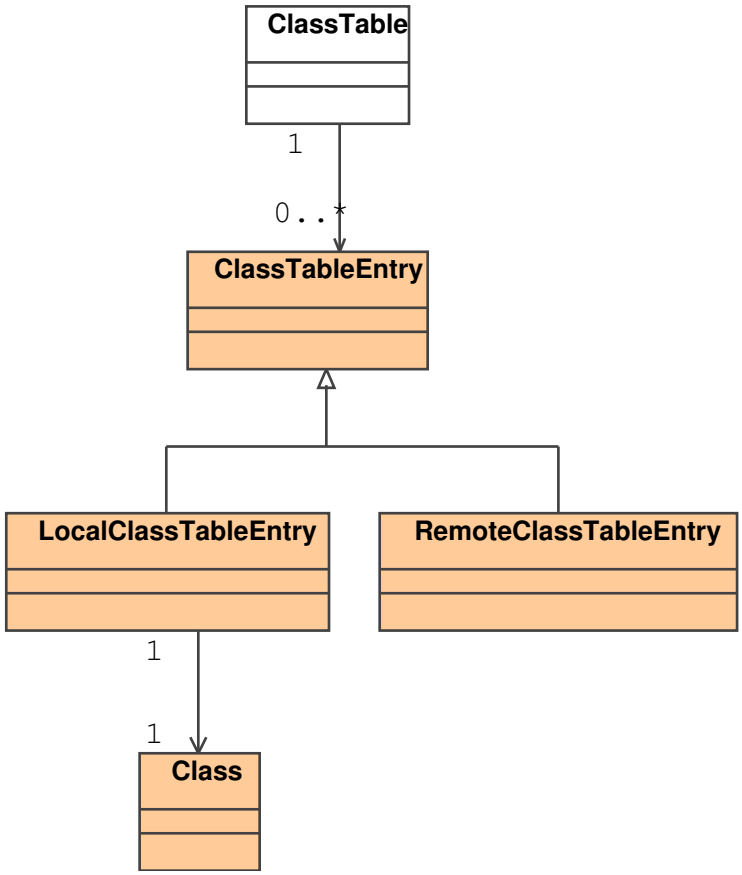


Figura 4.8: Entrada de Classes

A classe *RemoteClassTableEntry* é utilizada para referenciar as classes que possuem suas definições localizadas remotamente, ou seja, as definições estão em uma máquina virtual diferente de onde está sendo executado o programa.

Pode-se observar que na figura 4.9 que o índice 1 da tabela de classes da *VM Alpha* se refere a uma *LocalClassTableEntry*, enquanto que o índice 9 da tabela de classes da *VM Alpha* corresponde a uma *RemoteClassTableEntry* que referencia o índice 1 da tabela de classes da *VM Beta*.

Outro ponto importante a salientar é fato das localização da árvore de programa ser ortogonal à localização do objeto invocado. Ou seja, a localização das árvores de programas são independentes da localização dos objetos, pois podem ocorrer casos, conforme apresentado na figura 4.10, em que um objeto invocador se encontra na VM Alpha, o objeto invocado na VM Beta e têm-se uma entrada de referência remota de classe (ECR) na VM Alpha referenciando a árvore de programa do objeto invocado, porém, situada na VM Gama.

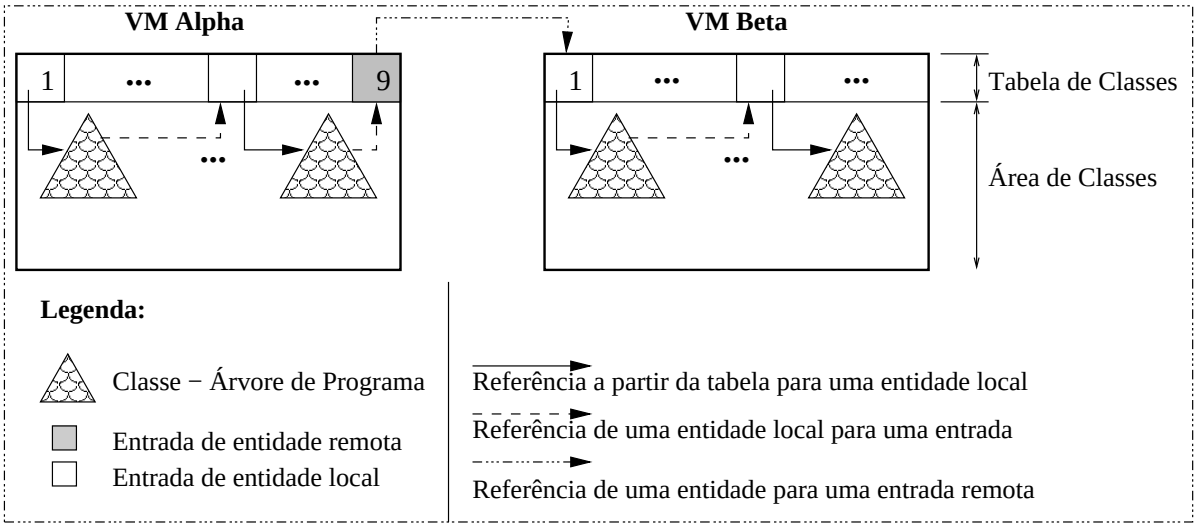


Figura 4.9: Exemplo de Entrada de Classes

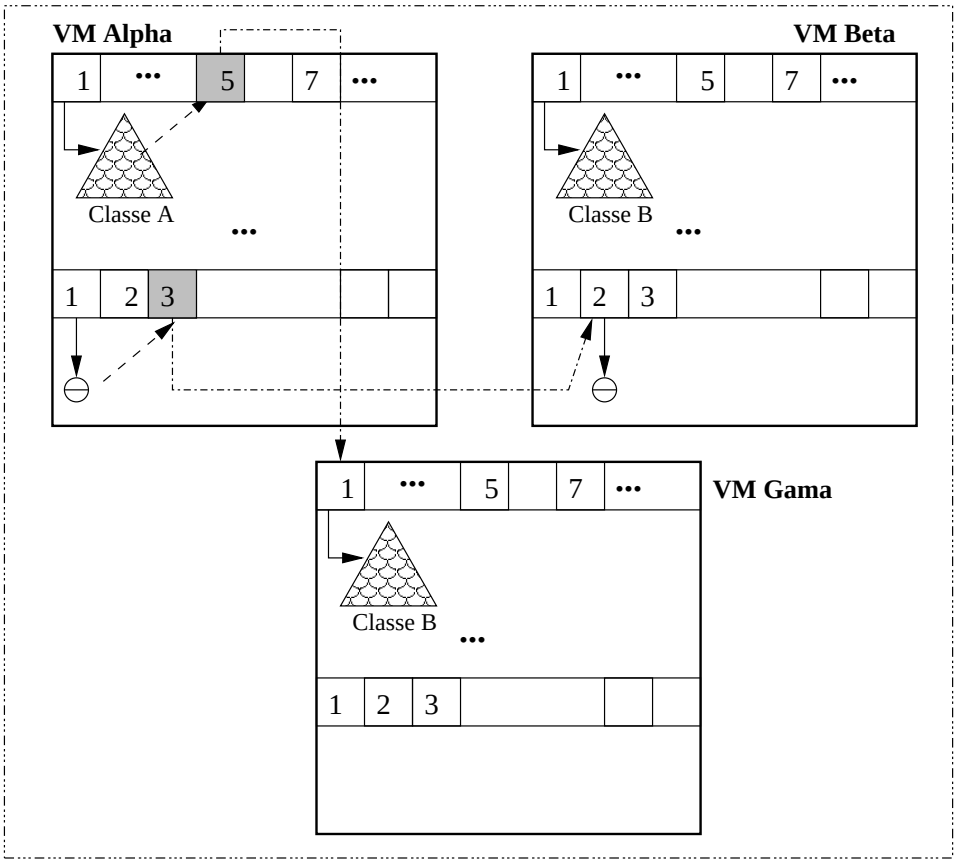


Figura 4.10: Máquinas virtuais com independência de localização entre objeto e classe.

4.3.2 Tabela de Invocáveis

A tabela de invocáveis é preenchida da mesma forma que a tabela a classes, no entanto ao invés de criar referências para as classes são criadas referências para os invocáveis (construtores, métodos e ações).

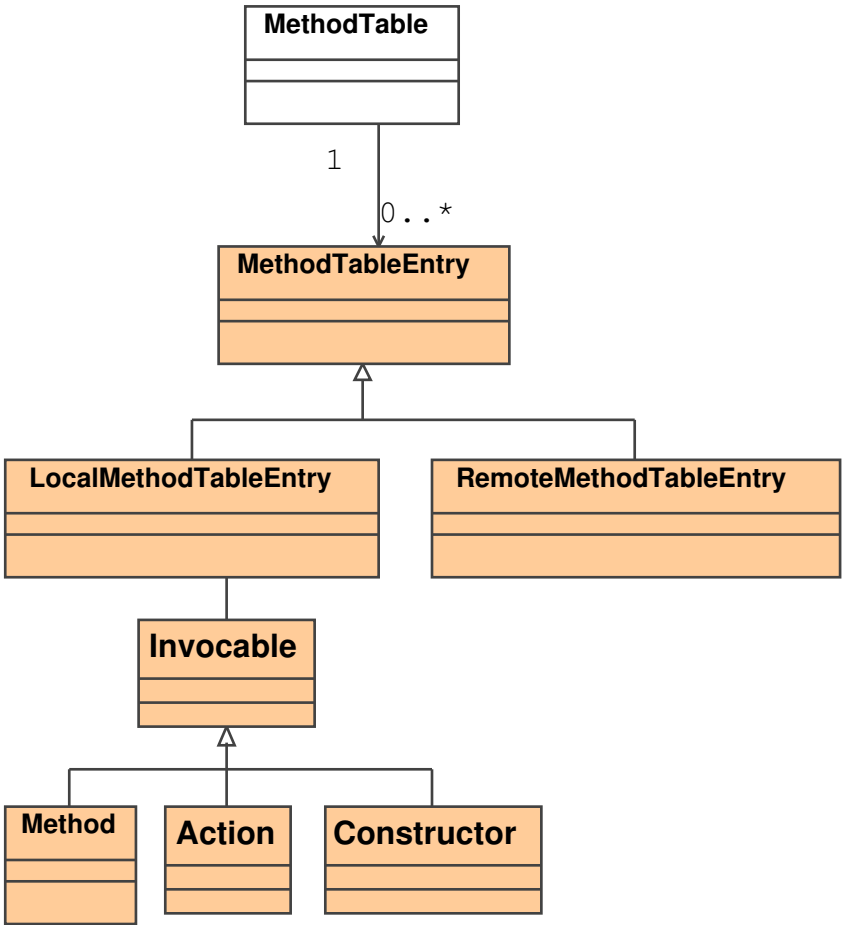


Figura 4.11: Entrada de Invocáveis

As entradas de invocáveis, conforme figura 4.11, estão representadas pelas classes *MethodTableEntry*, *LocalMethodTableEntry* e *RemoteMethodTableEntry*.

Existem duas classes derivadas da classe *MethodTableEntry*, a classe *LocalMethodTableEntry* - EIL (*Entrada de Invocável Local*) e a classe *RemoteMethodTableEntry* - EIR (*Entrada de Invocável Remota*).

A classe *LocalMethodTableEntry* é utilizada para referenciar os métodos, ações ou construtores que possuem suas definições localmente em relação à máquina virtual onde está sendo executado o programa. Através da associação com a tabela *Invocable* é possível obter as informações sobre o invocável.

A classe *RemoteMethodTableEntry* é utilizada para referenciar os métodos ou ações que possuem suas definições situadas remotamente, ou seja, as definições estão situadas em uma máquina remota.

Na figura 4.12 é observado que os índices 1, 2, 3, 6 e 7 da *VM Alpha* e os índices 1, 2, 3, 7, 8 e 9 da *VM Beta* correspondem a entradas do tipo *LocalMethodTableEntry*, enquanto que o índice 8 da *VM Alpha* corresponde a uma entrada do tipo *RemoteMethodTableEntry* que referencia o índice 1 da *VM Beta*.

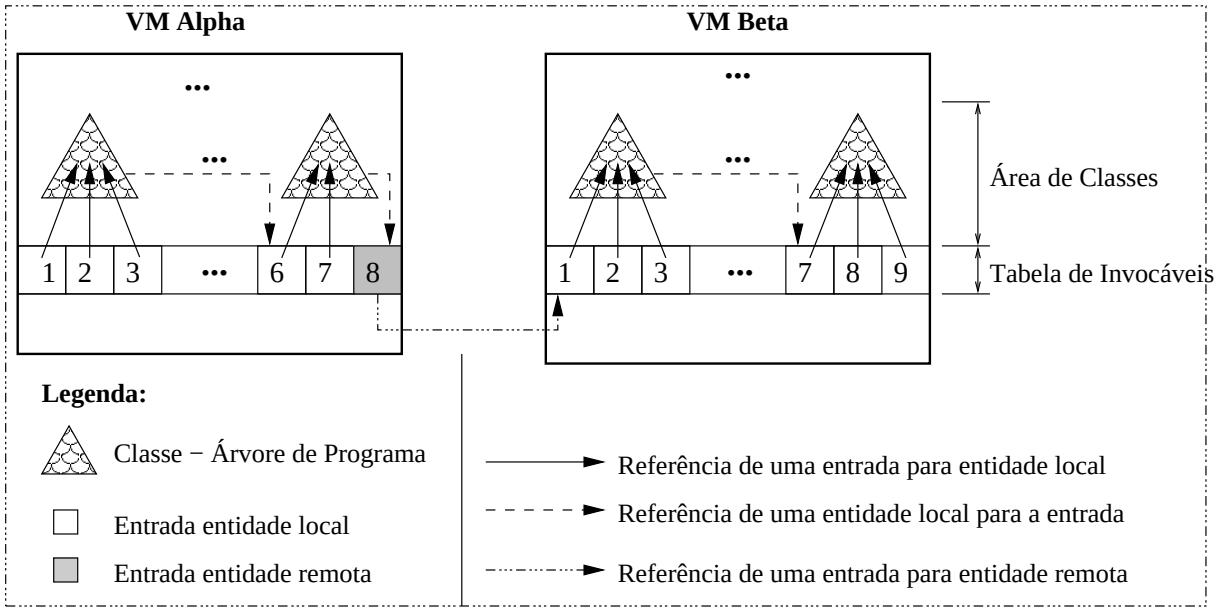


Figura 4.12: Exemplo de Entrada de Invocáveis

4.3.3 Tabela de Objetos

Para cada objeto instanciado pela máquina virtual é criada uma referência na tabela de objetos, desta forma não ocorre nenhum acesso direto ao objeto todo o acesso ocorre através da tabela de objetos.

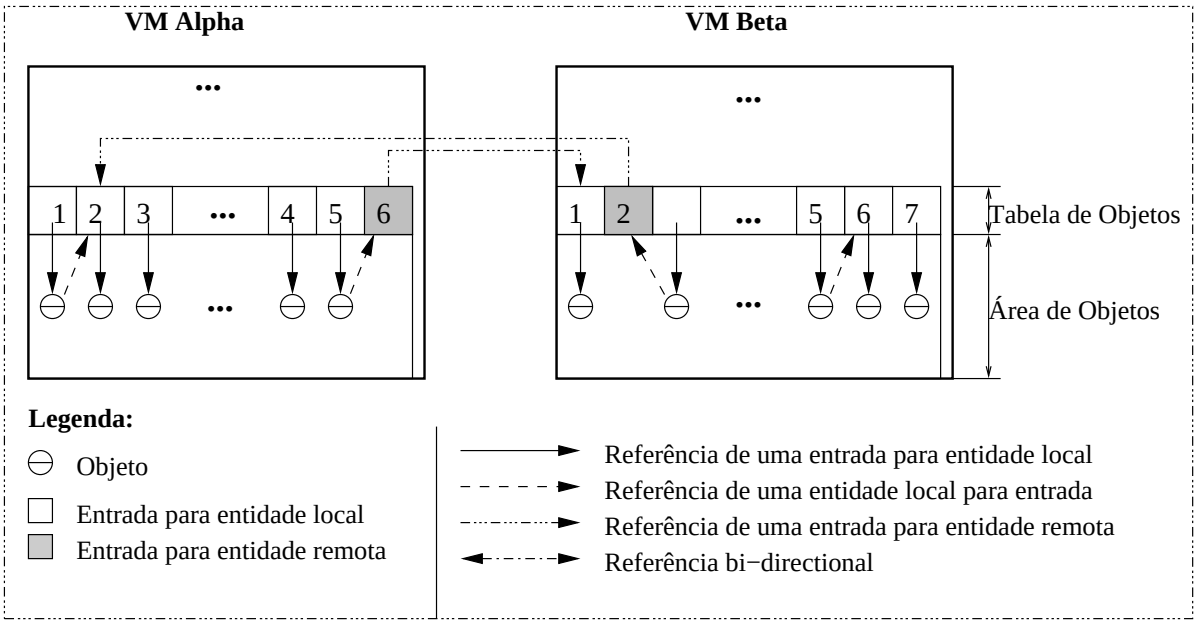


Figura 4.13: Exemplo de Entrada de Objetos

Por exemplo, considerando a figura 4.13, se o objeto que está referenciado pelo índice 1 da *VM Alpha* pretendesse acessar o objeto que está referenciado pelo índice 2 da *VM Alpha*, ele somente conseguiria acessar o objeto através da tabela de objetos. Outro

ponto a ser observado na figura é que índice 6 da *VM Alpha* se refere a uma entrada de objeto remoto, que por sua vez está referenciado pelo índice 2 da *VM Beta*.

Igualmente as classes *ClassTableEntry e MethodTableEntry* a classe *ObjectTableEntry* possui duas derivações conforme figura 4.14: *LocalObjectTableEntry* - EOL (*Entrada de Objeto Local*) e *RemoteObjectTableEntry* - EOR (*Entrada de Objeto Remoto*).

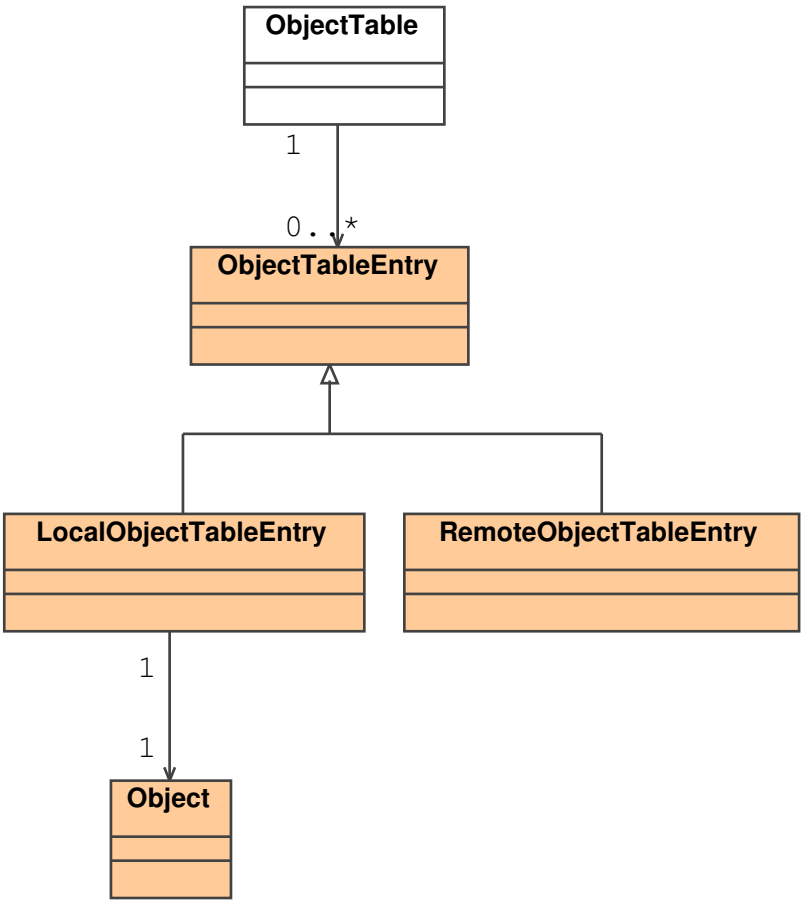


Figura 4.14: Entrada de Objetos

A classe *LocalObjectTableEntry* é utilizada para referenciar os objetos que possuem suas definições localizadas na mesma máquina virtual onde está sendo executado o programa. A classe *Object* que está associada a *LocalObjectTableEntry* constitui uma instância do objeto em execução.

Enquanto que a classe *RemoteObjectTableEntry* é utilizada para referenciar os objetos que possuem suas definições localizados remotamente em relação à máquina virtual de onde está sendo executado o programa.

4.4 Protocolo

O mecanismo de comunicação entre as máquinas virtuais e da máquina virtual com o serviço de nomes está baseado em soquetes[6] sob o protocolo de rede TCP. O fato da máquina virtual VIRTUOSI utilizar soquete permite a execução de várias máquinas virtuais em um mesmo computador utilizando apenas portas diferentes.

Embora tenha-se utilizado o protocolo TCP para o desenvolvimento deste trabalho, seria possível a substituição por outro protocolo, dado o fato de que a camada de protocolo é independente do mecanismo de invocação remota de métodos.

A solicitação de execução de procedimentos é realizado através do envio de mensagens. As mensagens podem ser trocadas entre máquinas virtuais e entre máquinas virtuais e serviço de nomes.

4.4.1 Mensagens entre Máquina Virtual e Serviço de Nomes

REGVM Solicitação de registro da máquina virtual no serviço de nomes;

GETVM Solicitação para o serviço de nomes do endereço IP e porta de uma máquina virtual;

VMNOTFOUND Esta mensagem indica que a máquina virtual solicitada para o serviço de nomes não foi encontrada;

UNREGVM Solicitação para remover o registro da máquina virtual no serviço de nomes;

ENDSN Solicitação para finalizar comunicação com o serviço de nomes;

4.4.2 Mensagens entre Máquinas Virtuais

EXEC_METHOD Solicitação para execução de método remoto;

GET_DATA_METHOD_ENTRY Solicitação para obter nome de método remoto;

GET_DATA_CLASS_ENTRY Solicitação para obter nome de classe remota;

GET_RESULT_TYPE_METHOD Solicitação para obter o tipo de retorno do invocável. Este retorno pode ser: **VOID** para método sem retorno, **RESULT** para método com retorno ou **ACTION** em caso de ação.

ENDEXEC Solicitação para encerrar comunicação de execução de método remoto;

NUM_PARAM Indica o número de parâmetros que o método possui;

ERR_PARAM Indica que não foram recebidos os parâmetros do método esperado.

PARAM_TYPE_LITERAL Indica que o tipo de parâmetro a ser passado, tipo *literal*.

[6] Do inglês socket

PARAM_TYPE_OBJREF Indica que o tipo de parâmetro a ser passado, tipo de referência para objeto.

ADT_ENTRY_NOTFOUND Entrada da classe não encontrada.

METHOD_ENTRY_NOTFOUND Entrada do invocável não encontrado.

RESULT_TYPE_NOTFOUND Entrada da classe referente ao tipo de retorno do método não encontrado.

A partir deste protocolo é que as máquinas virtuais e o serviço de nomes conseguem se comunicar. Nas próximas seções serão apresentados com mais detalhes os processos para invocação de métodos locais e remotos.

4.5 Comparativo Entre Invocação de Métodos Locais e Remotos

Nesta seção serão abordadas as principais diferenças que envolvem a invocação de métodos locais e remotos. Os mecanismos para a invocação de métodos locais e remotos são semelhantes, evidentemente, foram incluídos alguns procedimentos na MVV (Máquina Virtual Virtuosi) para contemplar a comunicação com máquinas virtuais remotas.

Os exemplos demonstrados na invocação de métodos locais remotos são baseados nas classes apresentados nas figuras 4.15 e 4.16.

```
class Client {

   association Product product;

   construtor make() exports  all {
        product = Product.make();
        sendOrder();
   }

   method void sendOrder() {
      p.order();
   }
   ...
}
```

Figura 4.15: Código fonte da classe Client

4.5.1 Invocação de Métodos Locais

Para que a MVV possa dar início à interpretação de uma aplicação VIRTUOSI é necessário que sejam informados o nome classe inicial e o construtor desta classe, isto por-

```
class Product {
   composition Integer supply;

   construtor make() exports all {
       this.supply = Integer.make(100);
   }

   method void order() {
      this.supply.subtract(1);
   }
   ...
}
```

Figura 4.16: Código fonte da classe Product

que na VIRTUOSI não existe um ponto pré-definido de início da aplicação, diferentemente de outras linguagens como Java e C++, as quais possuem em seu código o método *main* para indicar o início da aplicação.

Em um cenário, conforme apresentado na figura 4.17, a classe inicial é a classe *Client*, identificada a classe inicial o próximo passo a ser executado pela máquina virtual é carregar as árvores de programas *Client* e *Product* para a Área de Classes. A classe *Product* é também carregada devido a MVV identificar o seu relacionamento com a classe *Client*. Neste processo de carga da árvore de programa é que acontece a criação das entradas de classes para a Tabela de Classes, nestes casos as entradas são do tipo ECL (Entrada de Classe Local). Além das entradas da Tabela de Classes, são criadas também as entradas para a Tabela de Invocáveis:

- Client: Entrada para o construtor *make*;
- Client: Entrada para o método *sendOrder*;
- Product: Entrada para o construtor *make*;
- Product: Entrada para o método *order*.

Estas entradas para a Tabela de Invocáveis são do tipo EIL (Entrada de Invocável Local).

A partir da classe inicial *Client* a máquina virtual cria um objeto da classe *Client* e outro objeto da classe *Product* e os adiciona na Área de Objetos, criando também uma entrada para cada objeto na Tabela de Objetos, sendo tipo EOL (Entrada de Objeto Local). Os procedimentos que ocorrem normalmente para criação de um objeto podem ser vistos na subseção 3.4.6.1.

Com a criação do objeto e atualização das tabelas de referências, a máquina virtual cria uma atividade do tipo *LocalActivity* (não mostrado na figura) – correspondente ao construtor *make de Client* –, justamente por se tratar de uma execução de um objeto

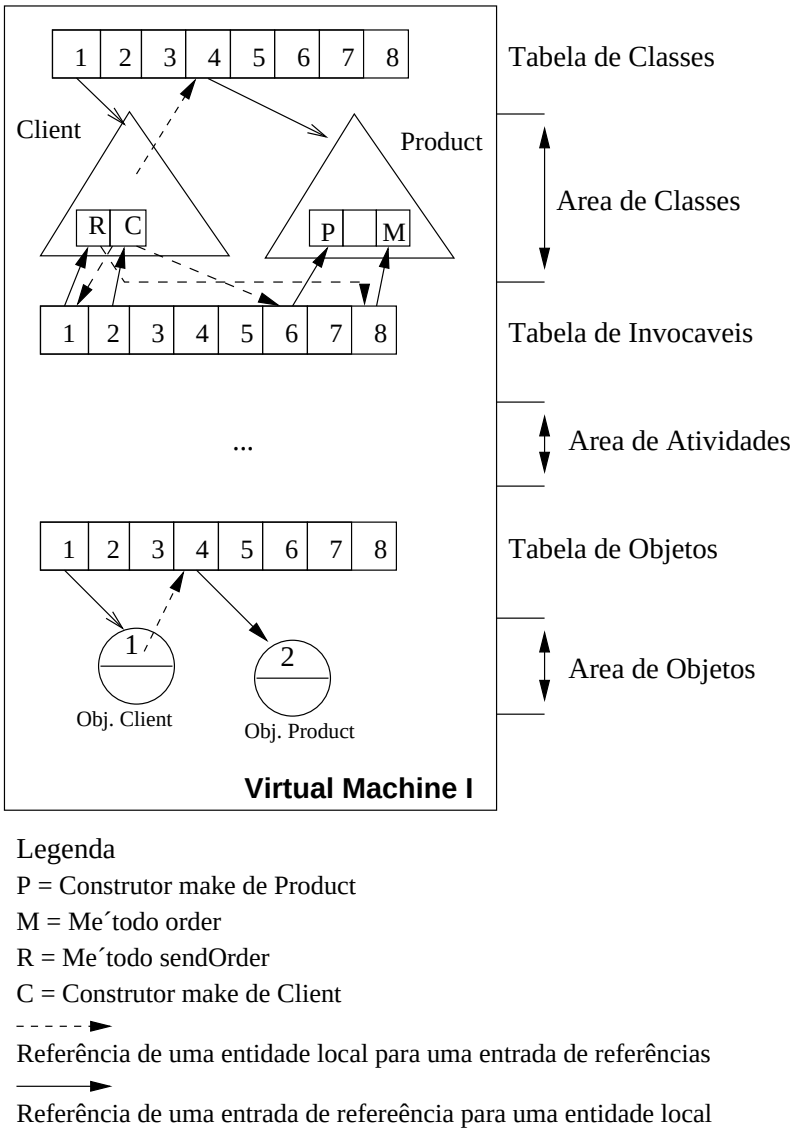


Figura 4.17: Cenário de Invocação de Método Local

local. Esta atividade que foi criada é empilhada na Pilha de Atividades, da mesma forma quando a máquina virtual identifica comandos de invocação dos invocáveis *Product.make* e *sendOrder* cria-se novas atividades e as empilha na Pilha de Atividades. A medida que estas atividades são finalizadas, elas são removidas da Pilha de Atividades.

Por exemplo, de uma forma sintética, para se executar o método *sendOrder*, a máquina virtual identifica se a atividade se trata de uma atividade local, obtém a referência do objeto pela Tabela de Objetos e referência do método pela Tabela de Invocáveis e executa o método. E, finalmente, a atividade é removida da pilha de atividades.

4.5.2 Invocação de Métodos Remotos

Nesta subseção serão abordados os procedimentos que foram contemplados na MVV, para dar suporte a invocação de métodos remotos.

Inicialização: O procedimento de inicialização consiste na criação de uma instância da máquina virtual. Quando uma instância da máquina virtual é iniciada é atribuída para esta instância o nome da máquina virtual. Em seguida ocorre obtenção de um número livre para porta do soquete a qual será aberta para escutar as requisições de solicitação de invocação de métodos remotos[7].

Registro no Servidor de Nomes: Para que uma máquina virtual possa se comunicar com outra máquina virtual é necessário que ocorram os registros das máquinas virtuais no servidor de nomes. Evidentemente, é necessário que o serviço de nomes esteja sendo executado em um endereço conhecido pela máquina virtual e que esteja também na mesma rede física onde a instância da máquina virtual será executada.

Após a obtenção da porta de soquete, ocorrido na fase de inicialização, uma máquina virtual abre uma conexão com o servidor de nomes para que ela possa efetuar o seu registro, permitindo assim que outras máquinas virtuais possam migrar objetos e invocar métodos remotos. Para que ocorra o registro da máquina virtual no servidor de nomes é necessário que ela informe o seu nome, IP e porta. As mensagens trocadas entre a máquina virtual e o servidor de nomes podem ser observadas na seção 4.2.

Preparação para as Requisições: A próximo procedimento da máquina virtual é criar uma instância a classe *ThreadVirtualMachine*. A finalidade desta classe é de recepcionar as requisições de comunicação (RPC) vindas de outras máquinas virtuais. Assim que a comunicação é estabelecida com uma máquina virtual solicitante, a classe *ThreadVirtualMachine* cria uma instância da classe *ThreadCommands*. A classe *ThreadCommands* é responsável enfim por manipular as mensagens – requisições – enviadas pela máquina virtual solicitante[8]. O relacionamento da classe *VirtualMachine* com as classes *ThreadVirtualMachine* e *ThreadCommands* pode ser observado na figura 4.18.

Por exemplo, no cenário como apresentado na figura 4.19, o objeto da classe *Client* deseja invocar um método remoto da classe *Product* – considerando que a definição da classe *Product* também se encontra remotamente, ou seja, a árvore de programa de *Product* somente existe na *Virtual Machine II*. Com relação as *threads* da máquina virtual, pode-se dizer que funcionam da seguinte maneira:

- Máquina virtual *Virtual Machine I*, após iniciada, cria uma instância da classe *ThreadVirtualMachine*;
- Por outro lado, a máquina virtual *Virtual Machine II*, após iniciada, também cria uma instância da classe *ThreadVirtualMachine*;

[7] O intervalo de portas utilizadas pela máquina virtual varia de 49200 a 50000, que conforme [Int04] corresponde a portas privadas e/ou dinâmicas – conforme IANA (*Internet Assigned Numbers Authority*) as portas dinâmicas e/ou privadas possuem os valores de 49152 até 65535 – as quais não correspondem a "portas bem conhecidas-- Do inglês Well Known Ports – ou portas registradas.

[8] Tanto a classe *ThreadVirtualMachine* como a classe *ThreadCommands* são implementações da interface *Java Runnable*. A interface Runnable deve ser implementada por qualquer classe cujas instâncias são planejadas para serem executadas por uma thread

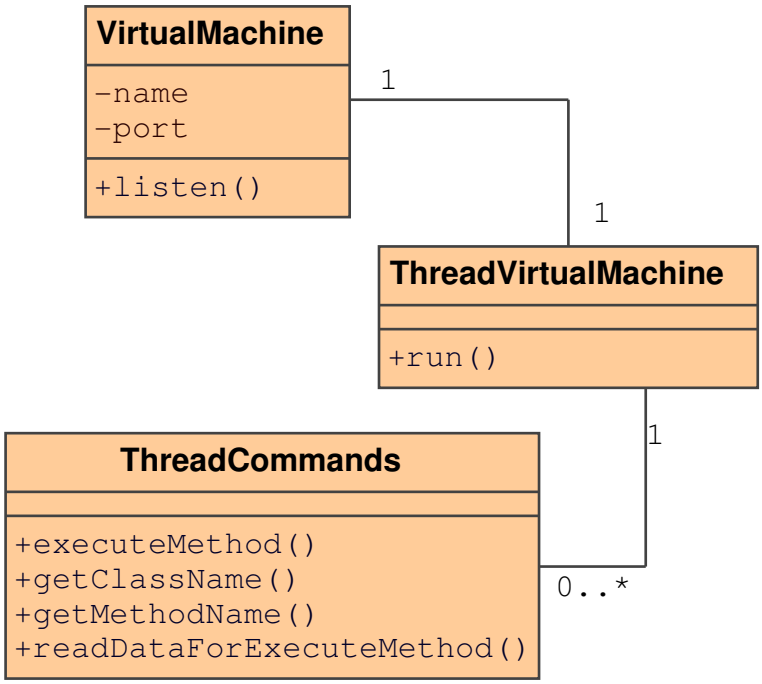
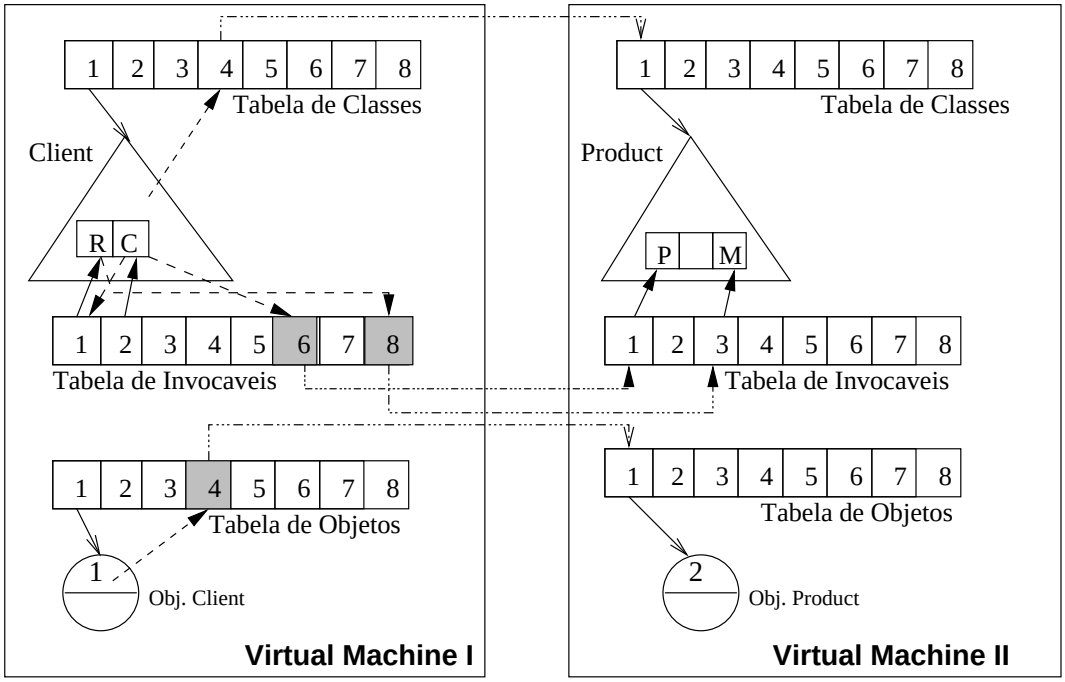


Figura 4.18: Threads de comunicação

- Quando a *Virtual Machine I* identifica que o objeto a ser executado é remoto, a *Virtual Machine I* cria uma conexão com a *Virtual Machine II*;
- A *Virtual Machine II*, após receber a conexão, cria uma instância da classe *ThreadCommands*.
- A partir deste momento a classe *ThreadCommands* fica responsável por gerenciar a comunicação com a máquina virtual *Virtual Machine I*;
- Desta forma se alguma outra máquina virtual desejar se comunicar com *Virtual Machine II*, ela estará disponível para aceitar outras conexões independentemente se ela esteja conectada a uma outra máquina virtual;

Na figura 4.20 é mostrado um exemplo de comunicação entre máquinas virtuais. Neste exemplo, as máquinas virtuais Beta e Gama estabelecem conexão com a máquina virtual Alpha. A conexão é feita por uma atividade A1 de Beta que se conecta com a ThreadVirtualMachine da VM Alpha, após a conexão a ThreadVirtualMachine repassa o soquete para ThreadCommands que fica responsável pela troca de mensagens com a VM Beta. O mesmo acontece com a VM Gama, ou seja, a VM Alpha pode receber n conexões e manipulá-las independentemente uma das outras.

Em síntese, pode-se dizer que a classe *ThreadVirtualMachine* fica responsável por aguardar novas requisições de comunicação, assim que se estabelece uma comunicação, a *ThreadVirtualMachine* repassa o controle de troca de mensagens para a classe *ThreadCommands*, e fica aguardando por novas requisições.



Legenda:
R = Me´todo sendOrder
C = Construtor make de Client
M = Me´todo order
P = Construtor make de Product

Referência de uma entrada de referencia para uma entidade local

Referência de uma entrada de referência para uma entrada de referência remota

Referência de uma entidade local para uma entrda de referências

Figura 4.19: Cenário de invocação remota de método com a definição de classe remota.

Execução:

Após carregar a árvore de programa e atualizar os índices das tabelas de referências, a máquina virtual começa a executar a aplicação.

O processo para chamada de um método remoto funciona de forma síncrona, ou seja, a máquina virtual invocadora fica aguardando o término da execução do método remoto para continuar sua execução.

Durante a execução do programa a máquina virtual identifica as sentenças[9] que devem ser executadas; caso seja identificado que deva ser executada uma instância da classe *MethodInvocation*, independentemente se for uma instância da classe *VoidMethodInvocation* ou *ResultMethodInvocation*, a máquina virtual buscará na Tabela de Objetos a referência para a entrada da tabela do objeto alvo. Através da entrada da Tabela de Objetos, a máquina virtual identifica se a entrada se trata de um EOL (Entrada de Objeto Local) ou um EOR (Entrada de Objeto Remoto). Se a entrada da Tabela de Objetos se

[9]Do inglês statement

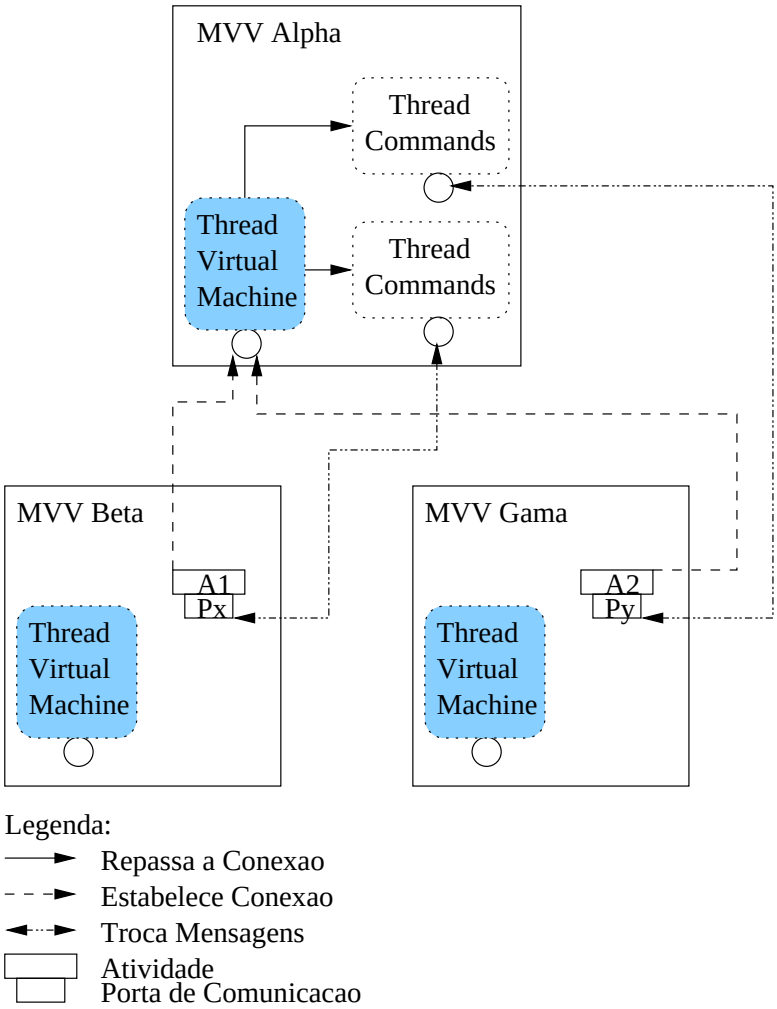


Figura 4.20: Exemplo comunicação entre máquinas virtuais utilizando as ThreadVirtual-Machine e ThreadCommands.

tratar de um EOR então a máquina virtual cria uma instância da classe *Caller*, a qual é derivada da classe *Activity*. O diagrama de colaboração pelo lado do invocador pode ser observada conforme figura 4.21.

Quando a atividade *Caller* está sendo executada ocorre a obtenção do endereço IP e porta do objeto remoto a partir da EOR. O endereço IP e a porta são obtidos a partir do servidor de nomes conforme apresentado na seção 4.2.

Em seguida ocorre a verificação se a entrada da Tabela de Invocáveis é do tipo local (EIL) ou remoto (EIR). Se for local, obtém-se o nome da classe e o nome do método (conforme figura 4.22). No entanto, se a entrada da Tabela de Invocáveis for remota, assim como na figura 4.19, a máquina virtual obtém o endereço IP e porta da localização da definição da classe, e inicia a comunicação para obtenção do nome da classe e do método que será executado.

Obtido o nome da classe e método, independentemente se local ou remotamente, a máquina virtual inicia o processo de comunicação com a máquina virtual remota para que seja executado o método desejado.

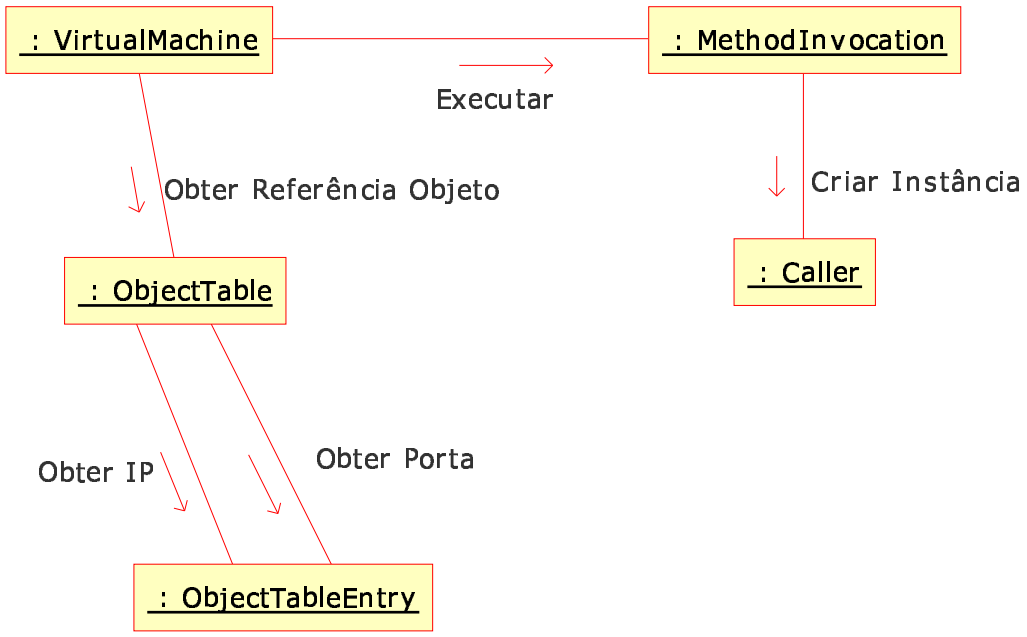


Figura 4.21: Diagrama de colaboração de método remoto, lado invocador

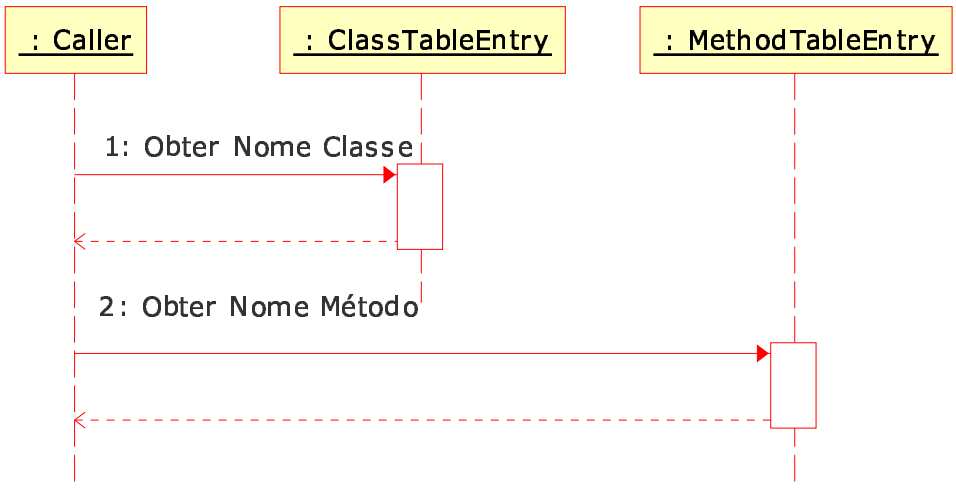


Figura 4.22: Diagrama de seqüência referente a obtenção do nome da classe e método.

Com relação aos parâmetros dos métodos, eles podem ser transmitidos de duas maneiras:

- Por referência (nome da máquina virtual e índice da Tabela de Objetos da MVV origem), em caso de serem objetos ou,
- Por valor, para parâmetros do tipo *literal*.

Quando a máquina virtual remota receber as referências dos objetos passados como parâmetros ela criará as entradas do tipo EOR (Entrada de Objeto Remoto) na Tabela de Objetos.

No caso de execução de métodos com retorno, na verdade, o resultado retornado será o índice da entrada da Tabela de Objetos onde está referenciado o objeto de retorno. Ao receber o índice, a máquina virtual cria uma entrada do tipo EOR (Entrada de Objeto Remoto) de Tabela Objetos para referenciar o objeto de retorno.

Se a invocação remota for correspondente a uma ação, o valor retornado será *skip* (0 – zero) quando for para a atividade invocadora ignorar a sentença e *execute* (1 – um) para ser executada a sentença.

No lado da máquina virtual destino, quando recebe a solicitação de conexão, ela cria uma instância da classe *ThreadCommands* a qual será responsável por gerenciar a comunicação, ver figura 4.23. Quando a classe *ThreadCommands* identifica que a solicitação enviada corresponde a execução de método ela cria e empilha uma atividade *Callee*.

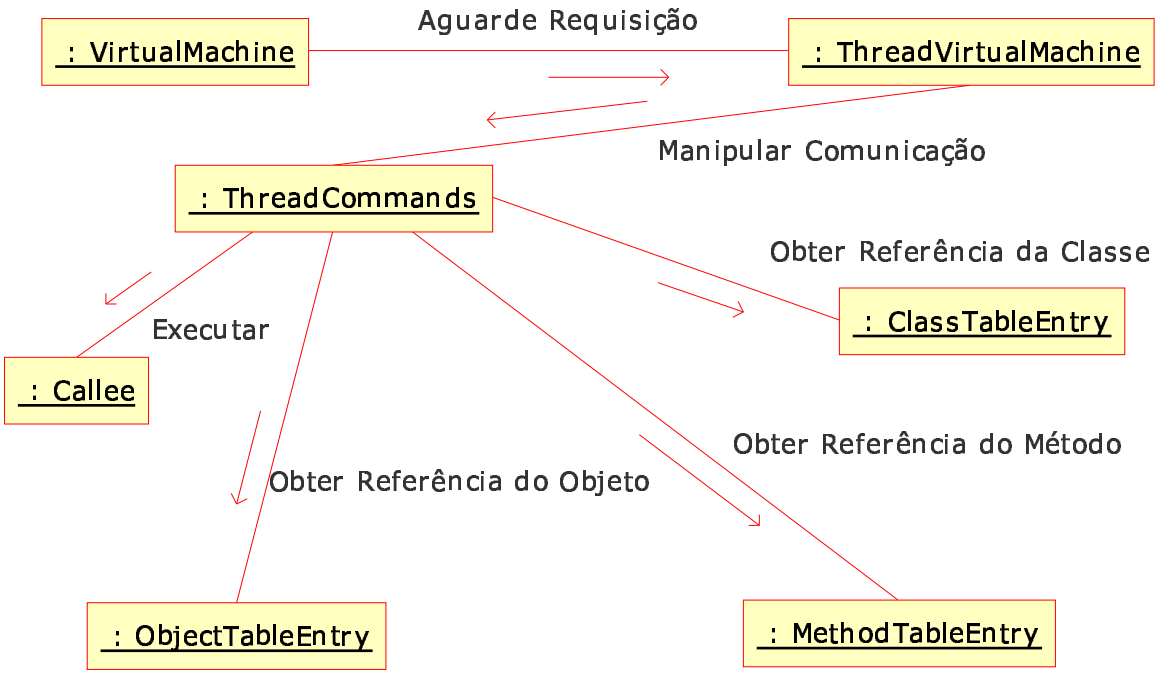


Figura 4.23: Diagrama de colaboração referente a invocação de método remoto, lado invocado

A partir do índice da tabela de objetos, enviado pela máquina virtual origem, a máquina virtual destino consegue identificar qual objeto deverá ser executado. O procedimento para obtenção das informações do método a ser executado ocorre de forma semelhante ao do objeto.

A partir de então, a máquina virtual de posse de todas as informações necessárias para a execução do método como: nome da classe, nome do método, índice do objeto na Tabela de Objetos, indicação do número de parâmetros, lê as informações (nome da MVV e índice da Tabela de Objetos) dos parâmetros caso houver, e executa o método solicitado.

4.6 Cenários Referentes à Invocação Remota de Métodos

O objetivo desta seção é demonstrar cenários referentes à invocação remota de métodos e, conforme citado na subseção 4.1.1, inicialmente todos os objetos são locais, em relação a uma máquina virtual VIRTUOSI, eles somente passam a ser remoto através de três maneiras:

- **Por migração de objetos**;
- **Invocação de método remoto com retorno**. Considerando duas máquinas virtuais Alpha e Beta, e três objetos A, B e C, sendo que A está localizado na máquina virtual Alpha e, B e C estão localizados na máquina virtual Beta. Assim supondo que A invoque um método do objeto B, o qual é um método com retorno para o objeto C, a máquina virtual Alpha ao receber o retorno do método cria uma EOR (Entrada de Objeto Remoto), desta forma é criada então uma referência remota para o objeto C que está situado na máquina virtual Beta;
- **Invocação de método remoto com parâmetros**. Neste caso uma referência remota poderia ser criada a partir de um cenário que apresentasse duas máquinas virtuais Alpha e Beta e, três objetos A, B e C. Os objetos A e B estão localizados na máquina virtual Alpha e C na máquina virtual Beta. Supondo que A invoque um método remoto com parâmetro do objeto C, passando como parâmetro B, a máquina virtual Beta ao receber a referência para o objeto B, cria uma EOR na tabela de objetos da máquina virtual Beta, a partir de então é criada uma referência remota para o objeto B situado na máquina virtual Alpha.

Levando-se em conta que um objeto somente pode se tornar remoto a partir das situações citadas acima, torna-se claro também que as entradas de referências remotas das tabelas de referências somente podem ser criadas quando ocorre um dos procedimentos citados, estes procedimentos serão detalhados nas subseções 4.6.1, 4.6.2 e 4.6.3.

Os cenários que serão apresentados nas subseções seguintes são baseados nas classes apresentados nas figuras 4.24, 4.25 e 4.26, sendo que a classe inicial para a execução é a classe Banco.

4.6.1 Invocação de Método Remoto a partir da Migração de Objetos

Ao observarmos o código da classe Banco apresentado na figura 4.27, nota-se que as classes Banco, Conta e Cliente são instanciadas localmente em relação a máquina virtual (instante 0). O construtor de Cliente recebe como parâmetros uma Conta e um número que corresponde a um identificador.

Em termos gráficos, resumidamente, teríamos uma situação conforme apresentada na figura 4.28; neste instante de tempo os objetos ainda estão todos locais, situados na máquina virtual Alpha.

```
class Banco
{
  association Cliente cliente;

  constructor make(  )
  {
    //Instante 0: Criação da Conta e Cliente
    Conta conta = Conta.make( 1234 );
    cliente = Cliente.make(conta, 888);
    //Instante 1: Cliente migra para máquina virtual Beta
    cliente.migrate("Beta");
    //Instante 2: Invoca método printIdentificador
    cliente.printIdentificador();
    //Instante 3: Invoca método remoto com retorno, porém
    //objeto Conta ainda se encontra na VM Alpha
    Conta conta1 = cliente.obterConta();
    //Instante 4: Conta migra para máquina virtual Beta
    conta.migrate("Beta");
    //Instante 5: Invoca método remotos com retorno,
    //agora objeto Conta se encontra na VM Beta
    Conta conta2 = cliente.obterConta();
    Conta contaNova = Conta.make( 5678 );
    //Instante 6: Substitui nova conta, invocação método remoto com parâmetro
    cliente.atualizarConta(contaNova);
    //Instante 7: Invoca método com parâmetro, o qual é
    //passado por valor
    conta.deposito(100);
    //Instante 8: Invoca ação remota
    if(conta.saldoNegativo())
       ...
    else
       ...
  }

   ...
}
```

Figura 4.24: Código fonte das classes utilizadas no cenário: Banco

Juntamente com a migração do objeto Cliente para a máquina virtual Beta, conforme código mostrado na figura 4.29 (instante 1), ocorre também a criação das entradas remotas das tabelas de referências. Conforme descrito em [dCCF04], para cada objeto migrado para a máquina virtual destino, é criada uma entrada de objeto remoto (EOR).

Conforme indicado na figura 4.30, após migração do objeto e da classe Cliente, são criadas as entradas de referências remotas:

- VM Alpha: Índice 5 da Tabela de Classes referencia o índice 1 da Tabela de Classes da VM Beta – classe Cliente.

```
class Cliente
{
  association Conta conta;
  composition Integer identificador;

  constructor make(Conta _conta, Integer _identificador ) exports  Banco
  {
     this.conta = _conta;
     this.identificador = _identificador;
  }

  method Conta obterConta() exports  Banco
  {
    return this.conta;
  }

  method void atualizarConta(Conta _conta) exports Banco
  {
    this.conta = _conta;
  }

  method void printIdentificador() exports Banco
  {
     System.out.println(this.identificador);
  }
  ...
}
```

Figura 4.25: Código fonte das classes utilizadas no cenário: Cliente

- VM Alpha: Índice 3 da Tabela de Invocáveis referencia o índice 1 da Tabela de Invocáveis da VM Beta – construtor make de Cliente.
- VM Alpha: Índice 4 da Tabela de Invocáveis referencia o índice 2 da Tabela de Invocáveis da VM Beta – método obterConta.
- VM Alpha: Índice 5 da Tabela de Invocáveis referencia o índice 3 da Tabela de Invocáveis da VM Beta – método atualizarConta.
- VM Alpha: Índice 6 da Tabela de Invocáveis referencia o índice 4 da Tabela de Invocáveis da VM Beta – método printIdentificador.
- VM Alpha: Índice 3 da Tabela de Objetos referencia o índice 2 da Tabela de Objetos da VM Beta – objeto Cliente .
- VM Beta: Índice 5 da Tabela de Classes referencia o índice 7 da Tabela de Classes da VM Alpha – árvore de programa Conta.

```
class Conta
{
  composition Integer numero;
  composition Float saldo;

  constructor make( Integer numConta ) exports  Banco, Cliente
  {
    numero = numConta;
    saldo = Float.make();
  }

  method void deposito(literal valor) exports Banco
  {
     saldo.adiciona(valor);
  }

  action saldoNegativo() exports Banco
  {
     if(saldo.lt(0)) //saldo menor do que zero
        skip;
     else
        execute;
  }
  ...
}
```

Figura 4.26: Código fonte das classes utilizadas no cenário:Conta

```
...
//Instante 0: Criação da Conta e Cliente
Conta conta = Conta.make( 1234 );
cliente = Cliente.make(conta, 888);
...
```

Figura 4.27: Trecho do código fonte da Classe Banco – Instante 0

- VM Beta: Índice 3 da Tabela de Objetos referencia o índice 7 da Tabela de Objetos da VM Alpha – objeto Conta.

4.6.1.1 Localização da Árvore de Programa

Um ponto importante a considerar é o fato de que ao migrar o objeto, o mecanismo de migração pode deixar uma cópia da árvore de programa na máquina virtual antecedente com o objetivo de facilitar a obtenção de informações da classe e invocáveis. Neste caso, conforme observado na figura 4.31 não teríamos entradas de referências remotas para na

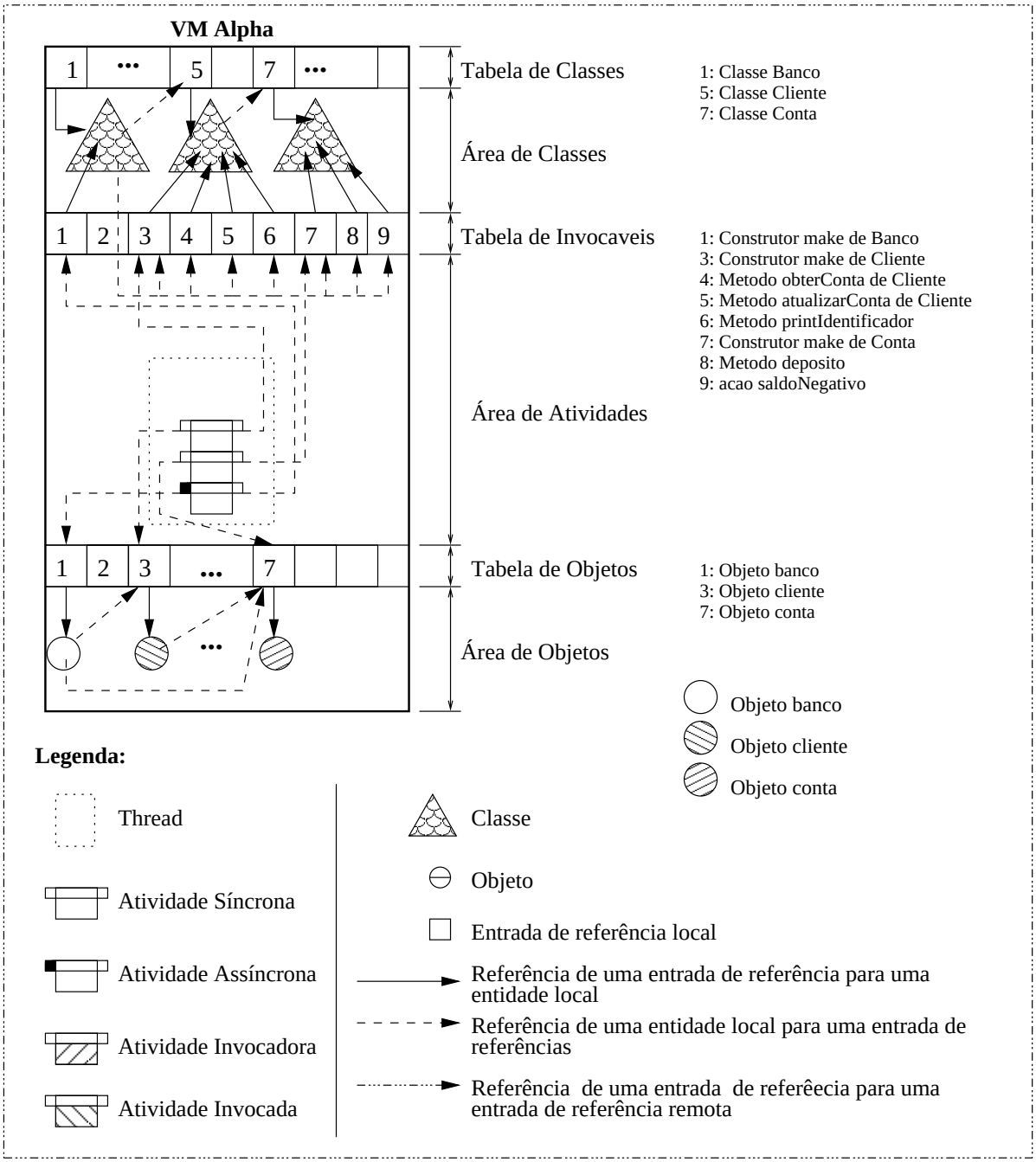


Figura 4.28: Máquina virtual com objetos construídos e referenciados localmente.

```
...
//Instante 1: Cliente migra para máquina virtual Beta
cliente.migrate("Beta");
...
```

Figura 4.29: Trecho do código fonte da classe Banco – Instante 1

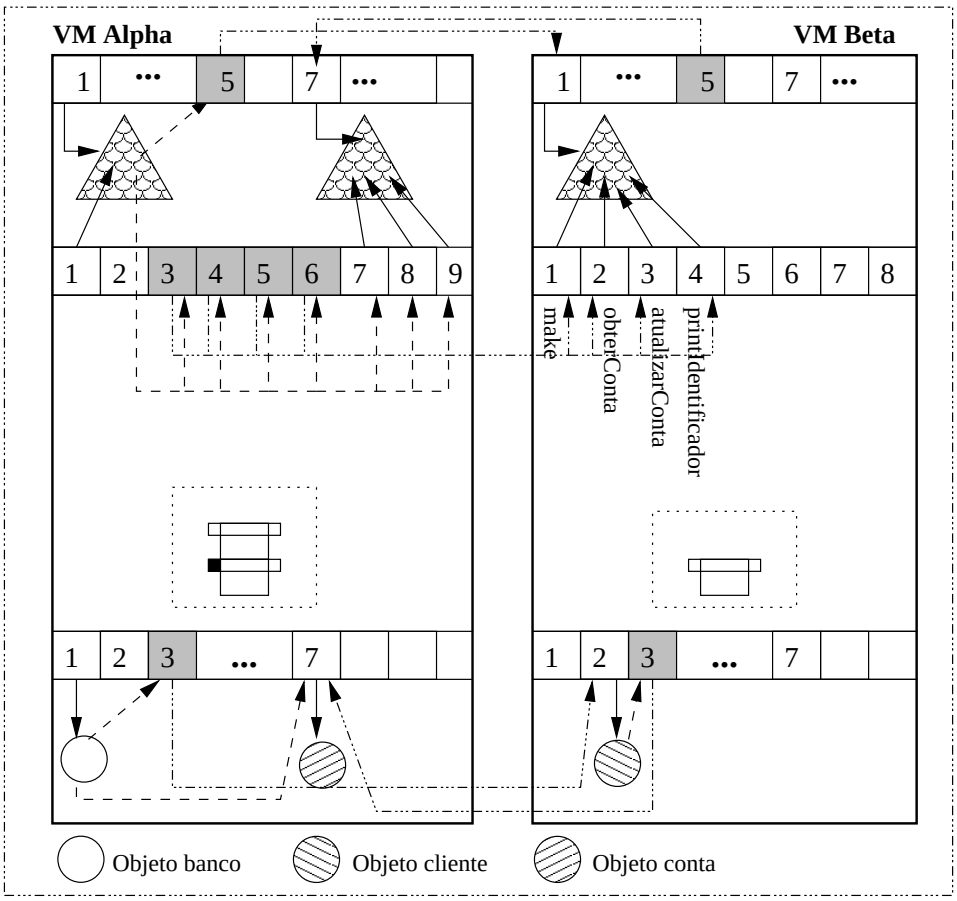


Figura 4.30: Máquinas virtuais após migração de objeto e de classe Cliente.

Tabela de Classes e Invocáveis da VM Alpha.

Pode-se dizer então com relação à localização da árvore de programa, após a migração do objeto:

- A migração pode deixar uma cópia da árvore de programa do objeto migrado;
- A migração pode não deixar nenhuma cópia da árvore de programa e somente deixar as entradas de referências remotas;
- Ou, ainda, pode existir uma entrada de referência remota de classe apontando para uma determinada máquina virtual, enquanto que o objeto migrado se encontra em outra máquina virtual.

4.6.1.2 Execução Remota de Método Referenciado Localmente

O objetivo desta subseção é demonstrar as etapas que ocorrem para a execução do método *printIdentificador* (ver figura 4.32), considerando que o objeto Cliente já tenha migrado para a VM Beta e que exista uma cópia da árvore de programa de Cliente na máquina Alpha, conforme apresentado na figura 4.31.

Para que o método *printIdentificador* do objeto *Cliente* seja executado na VM Beta é necessário que aconteçam os passos a seguir na **VM Alpha**:

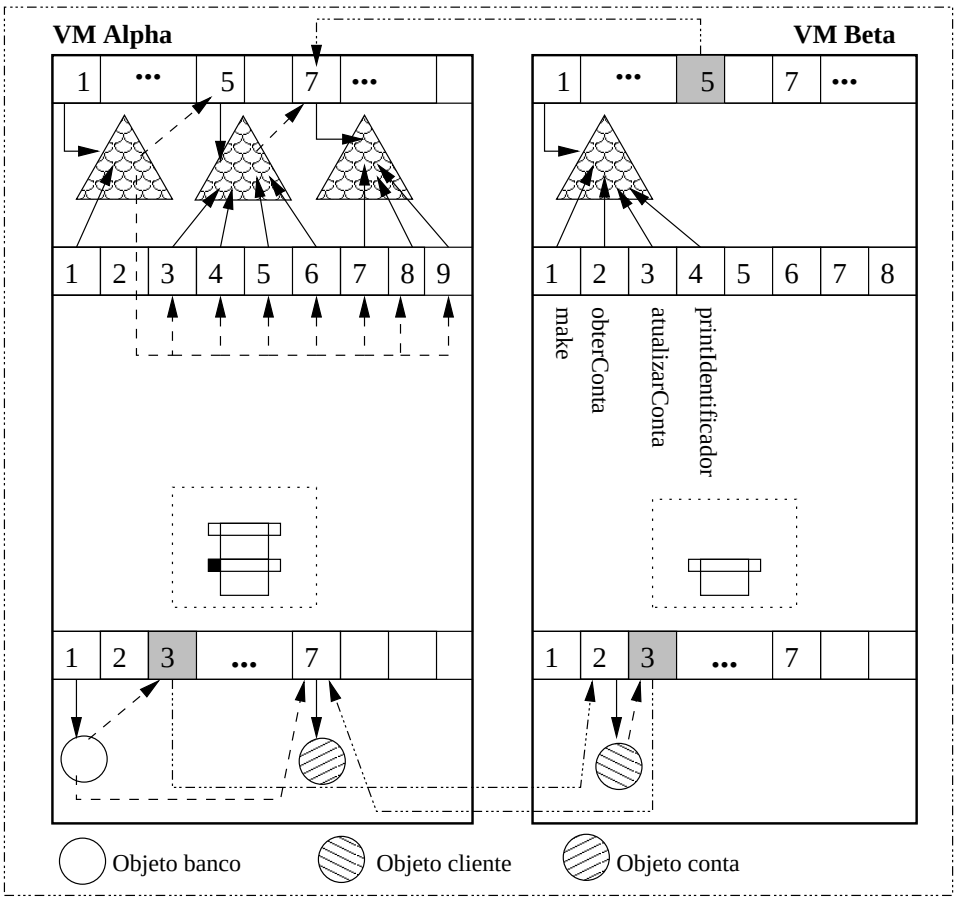


Figura 4.31: Máquinas virtuais após migração de objeto e cópia de classe.

```
//Instante 2: Invoca método printIdentificador
cliente.printIdentificador();
...
```

Figura 4.32: Trecho do código fonte da classe Banco – Instante 2

1. Obter EOR (Entrada de Objeto Remoto) do objeto *Cliente* na Tabela de Objetos – índice 3;

2. Obter a entrada do método *printIdentificador* na Tabela de Invocáveis – índice 6;

3. Obter a entrada da classe (árvore de programa) Cliente na Tabela de Classes – índice 5;

4. Criar uma instância da atividade Caller[10], passando como parâmetros a máquina virtual e o EOR;

[10]É uma classe estendida da classe Atividade, a qual representa a atividade criadora da invocação remota de um método

5. Obter o endereço IP e porta do EOR, ou seja, obter o IP e porta da VM Beta no servidor de nomes;
6. Obter o nome da classe – *Cliente*;
7. Obter o nome do método – *printIdentificador*;
8. Obter o tipo de retorno do método – *VOID*;
9. Obter número de parâmetros do método – 0 (zero);
10. E finalmente a VM Alpha estabelece conexão com a VM Beta para começar a trocar mensagens.

A partir dos passos citados acima, a máquina virtual VM Alpha começa a enviar mensagens para a execução do método invocado *printIdentificador*. Assim que as mensagens começam a ser enviadas, o sistema entra em estado de espera[11], aguardando que as mensagens sejam enviadas e que o método seja executado pela VM Beta para então continuar sua execução normal. As mensagens enviadas são listadas abaixo:

1. Envia o comando EXEC_METHOD;
2. Envia o nome da classe a qual pertence o método invocado, neste caso *Cliente*;
3. Envia o nome do método a ser invocado, neste caso *printIdentificador*;
4. Envia o número do índice onde se encontra o objeto *Cliente* na tabela de referências de objetos;
5. Envia a o comando NUM_PARAM, informando o número de parâmetros que o método possui, neste caso do método *printIdentificador* – 0 (zero).

Após o envio das mensagens, a **VM Beta** executa os passos a seguir:

1. Obter na Tabela de Objetos, o EOL de acordo com índice informado – índice 2;
2. Criar uma instância da classe Callee[12], passando como o EOL anteriormente obtido;
3. Obter na Tabela de Invocáveis, o EIL de acordo com o nome da classe e do método informados – índice 4;
4. Obter efetivamente o objeto *Cliente* através do EOL;
5. Executar o método *printIdentificador* do objeto *Cliente*.

Na a figura 4.33 é possível observar, na Pilha de Atividades, a atividade *Caller* pelo lado da VM Alpha, a atividade *Callee* pelo lado da VM Beta e o relacionamento indireto entre elas.

Terminada a execução do método, a VM Alpha continua a executar suas operações normalmente.

[11] Isto porque o método é chamado de forma síncrona

[12] É uma classe estendida da classe Atividade a qual representa a atividade executora de uma invocação remota

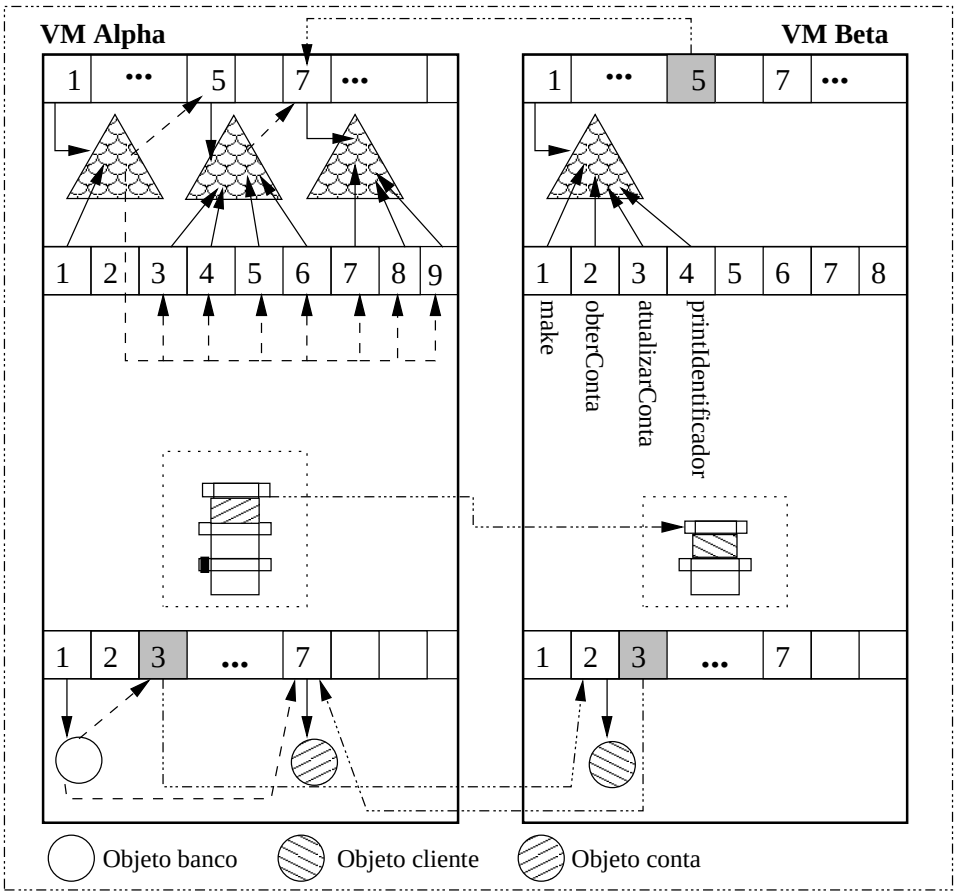


Figura 4.33: Máquinas virtuais com invocação de método remoto, mostrando pilha.

4.6.1.3 Execução Remota de Método Referenciado Remotamente

Nesta subseção será apresentado também a invocação do método *printIdentificador*, a diferença ocorre na localização da classe e do método invocado (árvore de programa), neste caso, a definição classe e dos métodos do objeto *Cliente* não se encontram na máquina virtual onde ocorre a invocação do método – VM Alpha, existe somente uma referência remota da classe e dos métodos (ver figura 4.30). Assim, para que ocorra a chamada para invocação do método remoto é necessário que ocorra antes a obtenção do nome da classe e do método que se deseja invocar.

A maioria dos passos que se seguem são os mesmos que são executados na subseção 4.6.1.2, porém para conveniência de leitura são repetidos novamente. Os passos necessários para a execução do do método *printIdentificador* do objeto *Cliente*, pelo lado da **VM Alpha**, são os seguintes:

1. Obter o EOR do objeto *Cliente* na Tabela de Objetos – índice 3;
2. Obter o EIR do método *printIdentificador* na Tabela de Invocáveis – índice 6;
3. Obter ECR do classe *Cliente* na tabela de referências de invocáveis – índice 5;

4. Criar uma instância da classe Caller, passando como parâmetros a máquina virtual e o EOL;
5. Obter o endereço IP e porta do EOR, ou seja, obter IP e porta da VM Beta no servidor de nomes;
6. Obter o endereço IP e porta do ECR, neste caso o IP e porta da VM Beta no servidor de nomes;
7. Obter remotamente o nome da classe – Cliente;
8. Obter o endereço IP e porta do EIR, neste caso o IP e porta da VM Beta no servidor de nomes;;
9. Obter remotamente nome do método – printIdentificador;
10. Obter o tipo de retorno do método – VOID;
11. Obter número de parâmetros do método – 0 (zero);
12. E finalmente a VM Alpha estabelece conexão com a VM Beta para começar a troca de mensagens.

Assim como no caso anterior, logo após o envio da mensagens, o sistema entra em estado de espera até que o método seja completamente executado.

Antes de passar as mensagens para execução do método é necessário obter o nome da classe remota e o nome do método remoto. Para isso são enviadas as seguintes mensagens:

1. Envia a seguinte mensagem para o obter o nome da classe:
   GET_DATA_ADT_ENTRY;
2. Envia o índice onde se encontra a classe na tabela de classes remota – índice 1 da Tabela de Classes da VM Beta;
3. Recebe o nome da classe remota – Cliente;
4. Envia a seguinte mensagem para o obter o nome do método:
   GET_DATA_METHOD_ENTRY;
5. Envia o índice onde se encontra método na tabela de referências de invocáveis – índice 4 da Tabela de Invocáveis da VM Beta;
6. Recebe o nome do método – printIdentificador.

A partir das informações da classes e do método são enviadas as mensagens para a execução do método.

1. Envia o comando EXEC_METHOD;

2. Envia o nome da classe a qual pertence o método invocado – Cliente;
3. Envia o nome do método a ser invocado – printIdentificador;
4. Envia o número do índice onde se encontra o objeto *Cliente* na tabela de referências de objetos – índice 2;
5. Envia a o comando NUM_PARAM, informando o número de parâmetros – neste caso 0 (zero).

Para executar o método *printIdenficador* a **VM Beta** executa os passos abaixo:

1. Obter o EOL de acordo com índice informado – índice 2 da Tabela de Objetos;
2. Criar uma instância da classe Callee, passando como o EOL anteriormente obtido;
3. Obter a referência do EIL de acordo com o nome da classe e do método informados – índice 4 da Tabela de Invocáveis;
4. Obter efetivamente o objeto *Cliente* através do EOL;
5. Executar o método *printIdenficador* do objeto *Cliente*.

Após o método terminar sua execução, a VM Alpha recebe da VM Beta a mensagem informando que terminou a execução do método e a mesma retorna a execução da atividade Caller.

4.6.2 Invocação de Método Remoto com Retorno

Em uma invocação de método remoto com retorno, o valor retornado do método remoto contém apenas informações de localização do objeto[13] que é retornado. Assim que a máquina virtual recebe estas informações, ela cria uma entrada de objeto remoto (EOR) na Tabela Objetos.

Considerando a figura 4.34 (instante 3), continuação do código do construtor make da classe Banco –, pode-se notar que após a migração do objeto Cliente ocorre a invocação do método *obterConta*, que retorna a referência para o objeto Conta situado na VM Beta.

```
//Instante 3: Invoca método remoto com retorno
Conta conta1 = cliente.obterConta();
```

Figura 4.34: Trecho do código fonte da classe Banco – Instante 3

Conforme citado anteriormente, o retorno do método não é a referência física do objeto Conta, mas apenas o índice da Tabela de Objetos da VM Beta e o nome da máquina virtual onde se encontra (VM Beta). Neste instante 3, acontece algo curioso, pois neste

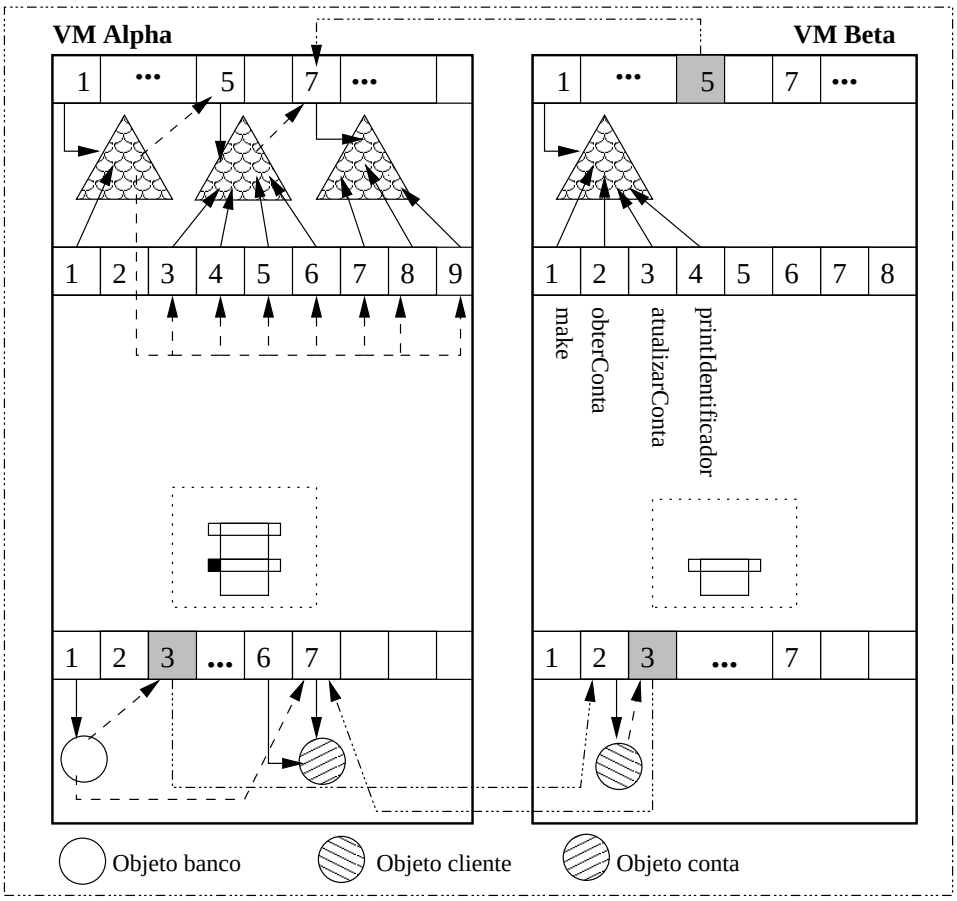


Figura 4.35: Máquinas virtuais após migração de objetos e invocação método com retorno com referência local.

momento o objeto Conta ainda se encontra na própria VM Alpha, na Tabela de Objetos da VM Beta encontra-se uma EOR que aponta para o objeto Conta situado na VM Alpha.

Em uma situação, por exemplo, que o objeto Conta estivesse na VM Beta o retorno seria o índice da Tabela de Objetos (índice 3) e no nome da máquina virtual (VM Beta). No entanto, neste caso a entrada da tabela de referências de objetos se refere a uma entrada remota (EOR). Em se tratando de uma referência remota, a máquina virtual interpreta as informações contidas na EOR e repassa como retorno do método. Após a interpretação da EOR, o valor retornado é o índice 7 da Tabela de Objetos e nome da máquina virtual (VM Alpha). Ou seja, neste ponto ocorreu uma interpretação que possibilitou uma otimização, no sentido de se evitar o uso de uma referência remota por uma referência local. Observe que na figura 4.35 que foi adicionado uma EOL (Entrada de Objeto Local) na Tabela de Objetos da VM Alpha – índice 6 – que corresponde a variável *conta1*.

No instante 4 apresentado pela figura 4.36 o objeto Conta migra para a VM Beta. Ocorrendo a migração do objeto as entradas das tabelas de referências ficam dispostas, como mostra a figura 4.37, da seguinte maneira:

[13] Estas informações são: índice da tabela de objetos e o nome da máquina virtual.

```
//Instante 4: Conta migra para máquina virtual Beta
conta.migrate("Beta");
```

Figura 4.36: Trecho do código fonte da classe Banco – Instante 4

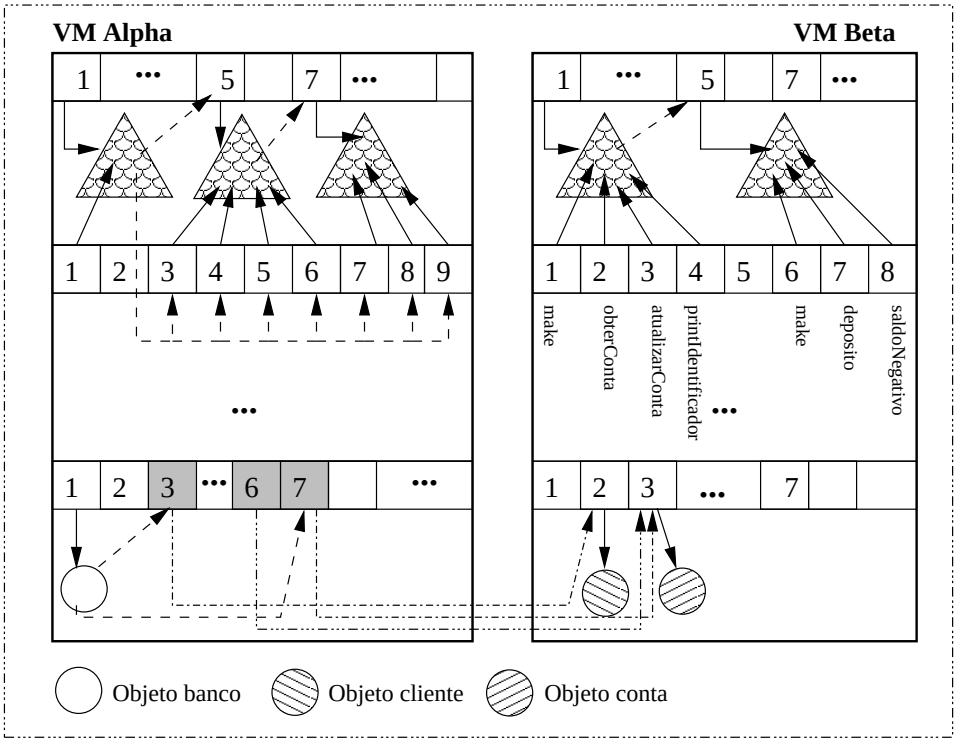


Figura 4.37: Máquinas virtuais após migração de objetos que se referenciam.

- VM Alpha: Índice 6 da Tabela de Objetos passa a ser remoto EOR e referencia o índice 3 da Tabela de Objetos da VM Beta – objeto conta1.
- VM Alpha: Índice 7 da Tabela de Objetos passa a ser remoto EOR e referencia o índice 3 da Tabela de Objetos da VM Beta – objeto conta.
- VM Beta: O índice 5 anteriormente do tipo ECR (Entrada de Classe Remota) passa a ser ECL (Entrada de Classe Local), referenciando a classe Conta.
- VM Beta: Índice 3 anteriormente do tipo EOR (Entrada de Objeto Remoto) passa a ser também local EOL (Entrada de Objeto Local) fazendo referência para o objeto conta.

Nota-se também, na figura 4.37, que foi mantido uma cópia da árvore de programa de Conta que é referenciado pelo índice 7 da Tabela de Classes da VM Alpha, e também a cópia da árvore de programa de Cliente referenciada pelo índice 5 da Tabela de Classes da VM Alpha.

```
//Instante 5: Invoca método remotos com retorno,
//agora objeto Conta se encontra na VM Beta
Conta conta2 = cliente.obterConta();
```

Figura 4.38: Trecho do código fonte da classe Banco – Instante 5

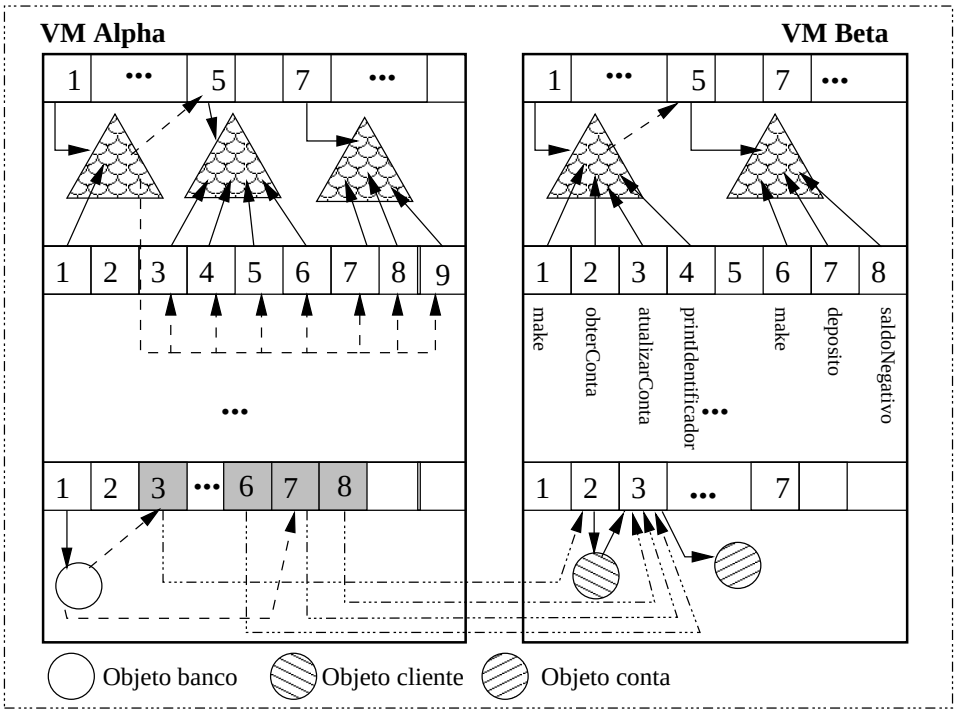


Figura 4.39: Máquinas virtuais após migração de objetos e invocação de método com retorno.

No instante 5, apresentado pela figura 4.38, ocorre a invocação do método remoto *obterConta* propriamente dito.

A máquina virtual VM Alpha, ao receber o retorno do método *obterConta*, cria uma entrada de referência remota (EOR) na Tabela de Objetos (índice 8). A figura 4.39, mostra o índice 8 da Tabela de Objetos da VM Alpha, o qual referencia o índice 3 da Tabela de Objetos da VM Beta. O índice 8 da Tabela de Objetos da VM Alpha, corresponde à variável *conta2*.

4.6.2.1 Execução do Método

Nesta subseção são descritas as etapas que ocorrem para a execução do método *obterConta*, que são similares aos casos citados anteriormente. Estas etapas seguintes ocorrem na **VM Alpha**:

1. Obter o EOR do objeto *Cliente* na tabela de referências de objetos – índice 3;

2. Obter a entrada do método *obterConta* na Tabela de Invocáveis – índice 4;

3. Obter a entrada da árvore Cliente na tabela de referências de classes – índice 5;
4. Criar uma instância da classe Caller, passando como parâmetros a máquina virtual e o EOL;
5. Obter o endereço IP e porta do EOR, ou seja, obter o IP e porta da VM Beta no servidor de nomes;
6. Obter o nome da classe – Cliente;
7. Obter o nome do método – obterConta;
8. Obter o tipo de retorno do método – RESULT;
9. VM Alpha estabelece conexão com a VM Beta e executa a troca de mensagens;
10. Aguarda o retorno do método;
11. Cria uma entrada de referência remota (EOR) na Tabela de Objetos obtida com o retorno do método – índice 8.

Para que o método possa ser executado e possa retornar a referência para o objeto de retorno na VM Beta, é necessário que ocorra a seguinte troca de mensagens:

1. Envia o comando EXEC_METHOD;
2. Informa o nome da classe a qual pertence o método invocado, neste caso *Cliente*;
3. Informa o nome do método a ser invocado, *obterConta*;
4. Informa o número do índice onde se encontra o objeto *Cliente* na Tabela de Objetos – índice 2;
5. VM Beta envia o nome da máquina virtual e índice correspondente ao objeto de retorno – VM Beta e índice 3.

Pela lado da **VM Beta**, para ocorrer a execução do método *obterConta*, é necessário que ocorram os passos abaixo:

1. Obter o EOL de acordo com índice informado – índice 2;
2. Criar uma instância da classe Callee, passando o EOL anteriormente obtido;
3. Obter a referência do EIL, de acordo com o nome da classe e do método informados – índice 2;
4. Obter efetivamente o objeto *Cliente* através do EOL;
5. Executar o método *obterConta* do objeto *Cliente*;

6. Enviar o nome da máquina virtual e o índice da Tabela de Objetos onde se encontra o objeto de retorno – VM Beta e índice 3.

Após o método ser executado e o processo de obtenção e registro da referência do objeto de retorno ser efetuado, o processamento das atividades executadas pela máquina virtual segue normalmente.

4.6.3 Invocação de Método Remoto com Parâmetro Passado por Referência

Em uma invocação de método remoto que possua parâmetro, o que é passado como parâmetro na verdade são informações de localização (índice na Tabela de Objetos e nome da máquina virtual), a menos que os parâmetros sejam do tipo *literal*, pois neste caso é passado como valor.

```
//Instante 6: Substitui nova conta, invocação método remoto com parâmetro
Conta contaNova = Conta.make( 5678 );
cliente.atualizarConta(contaNova);
```

Figura 4.40: Trecho do código fonte da classe Banco – Instante 6

Continuando a seqüência de execução do construtor make de Banco, nota-se no instante 6 – conforme figura 4.40 – que é criado uma nova instância da classe Conta (contaNova) e é passado como parâmetro para o método remoto *atualizarConta* de Cliente.

Pode-se observar na figura 4.41 que ocorreram algumas alterações na Tabela de Objetos da VM Alpha e Beta:

- VM Alpha: Criado índice 9 do tipo EOL, referente ao objeto *contaNova*;
- VM Beta: Criado índice 7 EOR a qual se refere ao parâmetro do método *atualizarConta*, e referencia o índice 9 da Tabela de Objetos da VM Alpha.
- VM Beta: Índice 2 (objeto cliente) da Tabela de Objetos, passa a referenciar o índice 7 da Tabela de Objetos;

4.6.3.1 Etapas para Execução do Método

A diferença principal em relação aos cenários anteriores, é a constatação de que método possui parâmetros. Desta forma, o mecanismo de invocação remota de métodos procura obter todas as referências destes parâmetros, para então enviá-los para a VM Beta.

Para que o método remoto *atualizarConta* do objeto *Cliente* seja executado é necessário que ocorram os passos abaixo na **VM Alpha**.

1. Obter o EOR do objeto *Cliente* na tabela de referências de objetos – índice 3;

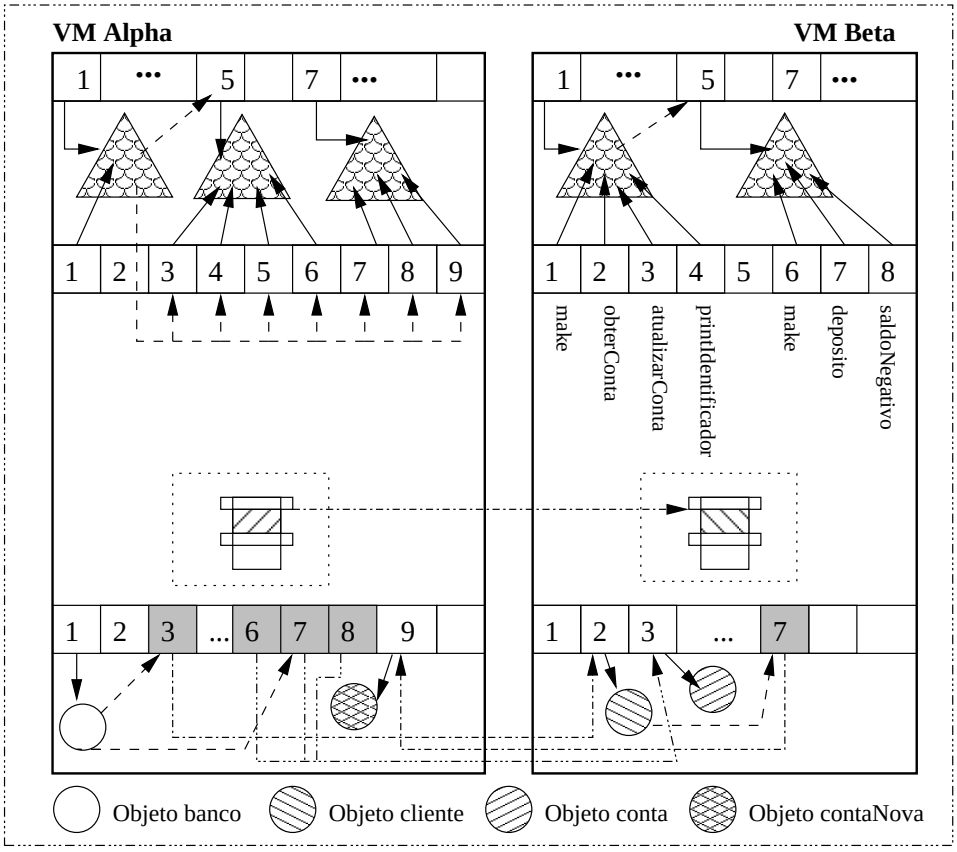


Figura 4.41: Máquinas virtuais após migração de objetos e atualização de referências.

2. Obter a entrada do método *atualizarConta* na tabela de referências de invocáveis – índice 5;
3. Obter a entrada da árvore de programa Cliente na tabela de referências de classes – índice 5;
4. Criar uma instância da classe Caller, passando como parâmetros a máquina virtual e o EOL;
5. Obter o endereço IP e porta do EOR – IP e porta da VM Beta;
6. Obter o nome da classe – Cliente;
7. Obter o nome do método – atualizarConta;
8. Obter o tipo de retorno do método – VOID;
9. Obter número de parâmetros do método – 1 parâmetro;
10. Obter a referência do parâmetro do método – VM Alpha e índice 9 da Tabela de Objetos;
11. VM Alpha estabelece conexão com a VM Beta e executa a troca de mensagens.

Assim como nos casos anteriores, após a obtenção de todos os dados para efetuar a chamada do método, o sistema começa a enviar as mensagens com os dados para que método possa ser executado.

1. Envia o comando EXEC_METHOD;
2. Informa o nome da classe a qual pertence o método invocado, neste caso *Cliente*;
3. Informa o nome do método a ser invocado, *atualizarConta*;
4. Informa o número do índice onde se encontra o objeto *Cliente* na tabela de referências de objetos – índice 2;
5. Envia a o comando NUM_PARAM, indicando número de parâmetros – 1 parâmetro;
6. Informa o tipo do parâmetro – PARAM_TYPE_OBJREF;
7. Informa a referência do parâmetro Conta – neste caso *contaNova* – , passa-se o nome da máquina virtual em que se encontra e também o índice na Tabela de Objetos.

Para efetuar a execução do método *atualizarConta* na **VM Beta**, é necessário que ocorram os passos abaixo:

1. Obter o EOL de acordo com índice informado – índice 2;
2. Criar uma instância da classe Callee, passando como o EOL anteriormente obtido;
3. Obter a referência do EIL de acordo com o nome da classe e do método informados – índice 3;
4. Obter efetivamente o objeto *Cliente* através do EOL;
5. Atualizar a Tabela de Objetos de acordo com as referências recebidas como parâmetros – criado índice 7 do tipo EOR na Tabela de Objetos;
6. Executar o método *atualizarConta* do objeto *Cliente*.

É importante ressaltar que, apenas são incluídas as referências para os parâmetros na execução do método. Quando houver a necessidade de acessar um valor do parâmetro, a *Virtuosi* identificará que se trata de um objeto remoto e criará uma conexão remota para acessar o valor do parâmetro.

4.6.4 Invocação de Método Remoto Com Parâmetro Passado por Valor

Na VIRTUOSI, *literal* é o único parâmetro que é passado por valor. A figura 4.42 mostra o *instante 7*, onde a variável *valor*, do tipo literal, é passada como parâmetro do método (deposito) do objeto remoto *conta*.

```
//Instante 7: Invoca método com parâmetro, o qual é
//passado por valor
conta.deposito(100);
```

Figura 4.42: Trecho do código fonte da classe Banco – Instante 7

4.6.4.1 Etapas para Execução do Método

Para que o método *deposito* do objeto remoto *conta* seja executado é necessário que ocorram os passos abaixo na **VM Alpha**.

1. Obter o EOR do objeto *conta* na tabela de referências de objetos – índice 7;
2. Obter a entrada do método *deposito* na tabela de referências de invocáveis – índice 8;
3. Obter a entrada da árvore de programa Conta na tabela de referências de classes – índice 7;
4. Criar uma instância da classe Caller, passando como parâmetros a máquina virtual e o EOL;
5. Obter o endereço IP e porta do EOR – IP e porta da VM Beta;
6. Obter o nome da classe – Conta;
7. Obter o nome do método – deposito;
8. Obter o tipo de retorno do método – VOID;
9. Obter número de parâmetros do método – 1 parâmetro;
10. Obter o valor do parâmetro – "100";
11. VM Alpha estabelece conexão com a VM Beta e executa a troca de mensagens.

Após a obtenção de todos os dados para efetuar a chamada do método, o sistema começa a enviar as mensagens com os dados para que método possa ser executado.

1. Envia o comando EXEC_METHOD;
2. Informa o nome da classe a qual pertence o método invocado, neste caso *Conta*;
3. Informa o nome do método a ser invocado, *deposito*;
4. Informa o número do índice onde se encontra o objeto *conta* na tabela de referências de objetos – índice 3;
5. Envia a o comando NUM_PARAM, indicando número de parâmetros – 1 parâmetro;

6. Informa o tipo do parâmetro – PARAM_TYPE_LITERAL;

7. Informa a o valor do parâmetro – "100".

Para ocorrer a execução do método *deposito* na **VM Beta**, é necessário que aconteçam os passos abaixo:

1. Obter o EOL de acordo com índice informado – índice 3;

2. Criar uma instância da classe Callee, passando como o EOL anteriormente obtido;

3. Obter a referência do EIL de acordo com o nome da classe e do método informados – índice 7;

4. Obter efetivamente o objeto *conta* através do EOL;

5. Criar uma variável local do tipo *literal* para ser utilizada pelo método;

6. Executar o método *deposito* do objeto *conta*.

4.6.5 Invocação de Ação Remota

Além da invocação remota de métodos, o mecanismo de invocação remota da VIRTUOSI permite a invocação remota de ações. Uma ação é um tipo de invocável, conforme descrito na subseção 3.2.6.3, cujo retorno é utilizado para a tomada de decisão referente a um comando de desvio.

```
//Instante 8: Invoca ação remota
if(conta.saldoNegativo())
   ...
else
   ...
```

Figura 4.43: Trecho do código fonte da classe Banco – Instante 8

O instante 8, apresentado pelo figura 4.43, mostra a invocação da ação *saldoNegativo* do objeto remoto *conta*.

4.6.5.1 Etapas para Execução da Ação

Para que a ação *saldoNegativo* do objeto remoto *conta* seja executada é necessário que ocorram os passos abaixo na **VM Alpha**.

1. Obter o EOR do objeto *conta* na tabela de referências de objetos – índice 7;

2. Obter a entrada da ação *saldoNegativo* na Tabela de Invocáveis – índice 9;

3. Obter a entrada da árvore de programa Conta na tabela de referências de classes – índice 7;
4. Criar uma instância da classe Caller, passando como parâmetros a máquina virtual e o EOL;
5. Obter o endereço IP e porta do EOR – IP e porta da VM Beta;
6. Obter o nome da classe – Conta;
7. Obter o nome da ação – saldoNegativo;
8. Obter o tipo de retorno do método – ACTION;
9. Obter número de parâmetros do método – 0 (zero);
10. VM Alpha estabelece conexão com a VM Beta e executa a troca de mensagens.

Em seguida o sistema começa a enviar as mensagens com os dados para que a ação possa ser executada.

1. Envia o comando EXEC_METHOD;
2. Informa o nome da classe a qual pertence a ação invocada, neste caso *Conta*;
3. Informa o nome da ação a ser invocada, *saldoNegativo*;
4. Informa o número do índice onde se encontra o objeto *conta* na tabela de referências de objetos – índice 3;
5. Envia a o comando NUM_PARAM, indicando número de parâmetros – 0 (zero);

Para ocorrer a execução da ação *saldoNegativo* na **VM Beta**, é necessário que aconteçam os passos abaixo:

1. Obter o EOL de acordo com índice informado – índice 3;
2. Criar uma instância da classe Callee, passando como o EOL anteriormente obtido;
3. Obter a referência do EIL de acordo com o nome da classe e do método informados – índice 8;
4. Obter efetivamente o objeto *conta* através do EOL;
5. Executar a ação *saldoNegativo* do objeto *conta*;
6. Retornar 0 (zero), que em termos de implementação equivale ao *skip* da ação.

4.7 Tratamento de Exceções

Neste trabalho, o tratamento de exceções, no ambiente VIRTUOSI, somente abrange aspectos de implementação. Ou seja, o metamodelo VIRTUOSI não foi modificado para contemplar o tratando de exceções.

O tratamento das exceções aplicado na implementação do protótipo é simples e, somente abrange o contexto da invocação remota de métodos; e também não contempla conceitos transacionais.

Em relação às exceções tratadas na implementação, basedas em [Tan95], dentre elas podemos destacar:

- O cliente não é capaz de localizar o servidor;
- A mensagem de requisição do cliente para o servidor é perdida;
- A mensagem de confirmação do servidor para o cliente é perdida;
- Acontece algum problema no servidor após receber a requisição e;
- Ocorre algum problema com o cliente após enviar a requisição.

As exceções de tempo excedido[14] são levantadas pelo próprio mecanismo de soquete da linguagem de programação Java. Quando ocorre uma exceção de tempo excedido, a partir de máquina virtual origem para a máquina virtual destino, o mecanismo de invocação remota de métodos remove a atividade *Caller* da pilha de atividades. Do mesmo modo, se o tempo excedido ocorrer da máquina virtual destino para a máquina virtual origem, o mecanismo de invocação remota de métodos remove a atividade *Callee* da pilha de atividades.

As exceções citadas acima, de uma forma geral, foram incorporadas com as exceções relacionadas ao servidor de nomes e a própria invocação de método remoto. Estas exceções serão discutidas nas próximas subseções.

4.7.1 Exceções Referentes ao Servidor de Nomes

Servidor de Nomes Não Encontrado: Este caso ocorre quando endereço IP ou porta do servidor de nomes é informado incorretamente.

Máquina Virtual Não Encontrada: Esta situação ocorre quando uma máquina virtual deseja estabelecer comunicação com uma outra máquina virtual e não encontra a entrada no servidor de nomes.

Máquina Virtual Já Existe no Servidor Nomes: O servidor de nomes realiza uma verificação antes de efetuar o registro da máquina virtual no servidor de nomes.

[14] Do inglês timeout

Caso encontre uma máquina virtual com o mesmo nome informado o servidor de nomes gera uma exceção negando o registro.

Remover Registro do Servidor de Nomes: Esta situação ocorre quando uma máquina virtual envia uma solicitação de remoção para o servidor de nomes e, o servidor de nomes não encontra o registro da máquina virtual.

4.7.2 Exceções Referentes à Invocação Remota de Métodos

Entrada de Classe Não Encontrada: Esta exceção ocorre quando uma máquina virtual deseja obter remotamente a informação do nome da classe e, a entrada da classe da Tabela de Classes não é encontrada a partir do índice informado.

Entrada de Invocável Não Encontrado: Esta exceção ocorre quando uma máquina virtual deseja obter remotamente a informação do nome do método e, a entrada do invocável da Tabela de Invocáveis não é encontrada a partir do índice informado.

Erro ao Estabelecer Conexão com Máquina Virtual Remota: Este caso ocorre quando não é possível estabelecer conexão entre a máquina virtual origem e a máquina virtual destino, ou vice-versa.

Erro de Parâmetros: Esta situação ocorre quando a máquina virtual origem não informa o número de parâmetros, ou quando, as informações (índice da Tabela de Objetos e nome da máquina virtual) passadas para a máquina virtual destino estão nulas.

Capítulo 5

Implementação do Mecanismo de Invocação Remota de Métodos

Para a validação do mecanismo de invocação remota de métodos foi desenvolvido um protótipo onde é possível constatar as principais características conforme apresentadas no capítulo 4. O objetivo deste capítulo é de demonstrar como foi implementado o servidor de nomes, invocação de métodos remotos simples[1], métodos remotos com parâmetros e métodos remotos com retorno. Os testes aplicados ao protótipo somente são testes de unidade aplicados ao próprio mecanismo de invocação remota de métodos, não se aplicam a VIRTUOSI como um todo.

5.1 Ambiente de Desenvolvimento

Utilizou-se para a construção do protótipo a linguagem de programação Java. A escolha da linguagem Java deve-se ao fato de ser uma boa linguagem, moderna, bem aceita no meio acadêmico e industrial, com conceitos de orientação a objetos implementados de forma adequada para as necessidades do protótipo e não introduz qualquer restrição na implementação do servidor de nomes e nem para a implementação do mecanismo de invocação remota de métodos para a MVV (Máquina Virtual VIRTUOSI).

5.2 Limitações

O protótipo implementado possui algumas limitações que não foram incorporadas por não fazer parte do escopo mecanismo de invocação remota de métodos propriamente dito, tais como:

1. O protótipo não se preocupou com detalhes referentes à própria execução da MVV, estes detalhes podem ser vistos em [Kol04].

2. Não possui mecanismo para mobilidade de objetos, partiu-se do príncipio que os objetos já estavam distribuídos.

3. Não possui mecanismo para criar ou carregar árvores de programas.

[1] Métodos sem retornos e sem parâmetros

5.3 Principais Classes Envolvidas no Mecanismo de Invocação Remota de Métodos

As classes listadas a seguir são contribuições para a arquitetura VIRTUOSI e são as principais envolvidas com o mecanismo de invocação remota de métodos.

VirtualMachine Classe principal da MVV (Máquina Virtual VIRTUOSI).

VirtualMachineForm Classe que apresenta uma interface para o usuário selecionar os cenários e invocar os métodos remotos.

NameService Classe responsável pelo servidor de nomes, sendo este, simplificado e centralizado.

NameServiceForm Classe que apresenta uma interface para o usuário visualizar os registros do servidor de nomes.

ThreadNameService Classe responsável por gerenciar as solicitações de registro, remoção ou pesquisa no servidor de nomes.

ThreadVirtualMachine Classe responsável por gerenciar as requisições de solicitações de serviços vindos de outras máquina virtuais.

ThreadCommands Classes responsável por gerenciar o protocolo de comunicação com outras máquinas virtuais.

Caller Atividade invocadora; estabele uma atividade para ser executada remotamente.

Callee Atividade invocada; executa uma atividade invocada remotamente.

RemoteClassTableEntry Classe de entrada da Tabela de Classes que armazena informações de referência remota.

RemoteMethodTableEntry Classe de entrada da Tabela de Invocáveis que armazena informações de referência remota.

RemoteObjectTableEntry Classe de entrada da Tabela de Objetos que armazena informações de referência remota.

As principais classes da VIRTUOSI relacionadas com a invocação remota de métodos, pode ser observada no diagrama de classes apresentado na figura 5.1.

A figura 5.2 mostra o diagrama de classes relacionadas com o serviço de nomes da VIRTUOSI.

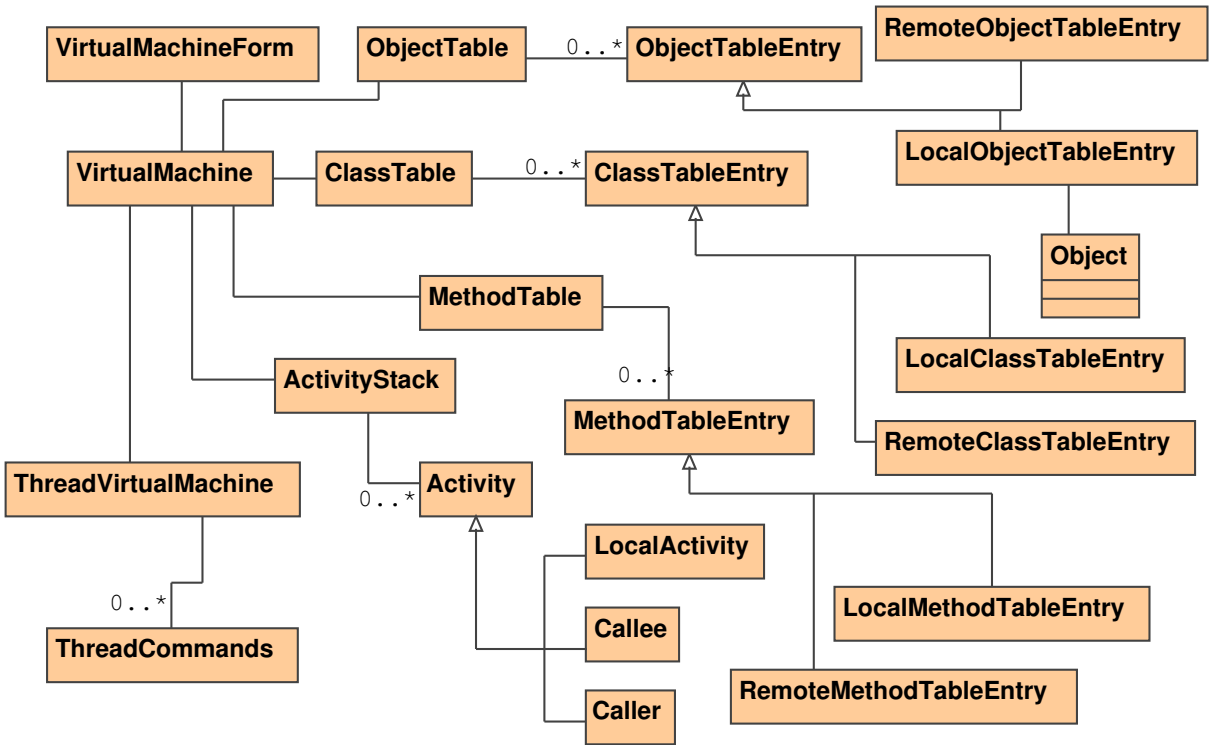


Figura 5.1: Diagrama de classes relacionadas com invocação remota de métodos

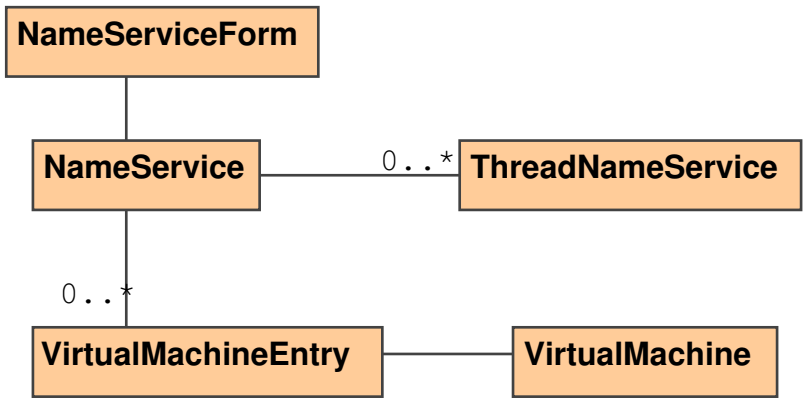


Figura 5.2: Diagrama de classes relacionadas com o serviço de nomes.

5.4 Ciclo de Vida da MVV em Relação ao Protótipo Implementado

Esta seção apresenta o ciclo da vida da máquina virtual VIRTUOSI em termos de classes e em relação ao protótipo implementado.

Em uma execução normal, a partir da seleção de um cenário por um usuário, considerando os cenários pré-definidos, seria:

1. Executar a classe *NameServiceForm*.
2. A classe *NameServiceForm* instancia a classe *NameService*.
3. O serviço de nomes através da classe *NameService* aguarda as solicitações de serviços.
4. Executar a classe *VirtualMachineForm*.
5. O usuário deve informar o nome da máquina virtual.
6. É criada a instância da classe *VirtualMachine* e, é atribuído o nome informado anteriormente.
7. A classe *VirtualMachine* obtém o endereço IP da máquina virtual.
8. A classe *VirtualMachine* gera um número aleatório (entre 49200 à 50000) e realiza um teste para ver se a porta já não está ocupada. Este número é para ser atribuído como porta da máquina virtual.
9. A partir de então a *VirtualMachine* fica no aguardo de requisições.
10. O usuário clica no botão para registrar a máquina virtual no serviço de nomes.
11. O usuário seleciona o cenário que deseja executar. Os cenários que o usuário pode selecionar podem ser vistos na seção 5.5.
12. O usuário clica no botão *Executar*.
13. A máquina virtual invoca o execução do cenário selecionado.

Os passos de 4 a 14 devem ser executados em um outro computador ou até mesmo no mesmo computador para simular a outra máquina virtual invocada.

5.5 Cenários de Testes

Foram elaborados alguns cenários com o objetivo de validar o mecanismo de invocação de métodos remotos. A validação dos resultados dos cenários foram obtidos através de logs que são apresentadas através da MVV. Nestes logs são apresentados os passos, comandos e protocolos utilizados na invocação do método remoto.

A figura 5.3 mostra a interface do protótipo do servidor de nomes, onde é possível identificar duas máquinas virtuais registradas e também o log de operações ocorridas.

Pode-se observar na figura 5.4, que corresponde ao protótipo da VM Alpha, que é necessário informar o nome da máquina virtual remota antes de clicar no botão Executar.

Cenário 1	
VM Alpha	**VM Beta**
Objeto banco	Objeto cliente
Execução: Objeto Banco invoca método *printIdentificador* do objeto Cliente	
Observação: Existe uma cópia da árvore de programa de Cliente na VM Alpha	

Tabela 5.1: Quadro resumo referente ao cenário de invocação remota de método simples.

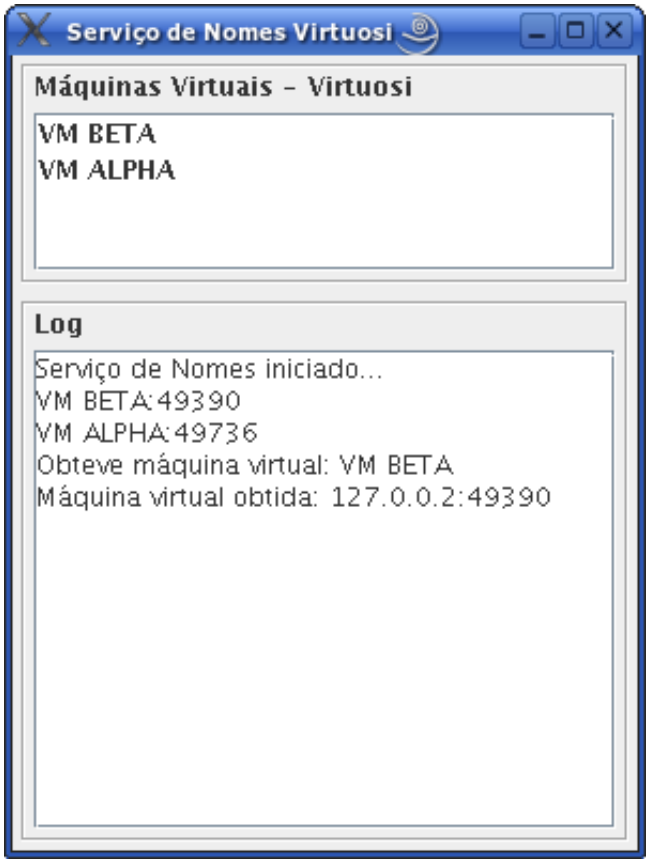


Figura 5.3: Interface do serviço de nomes

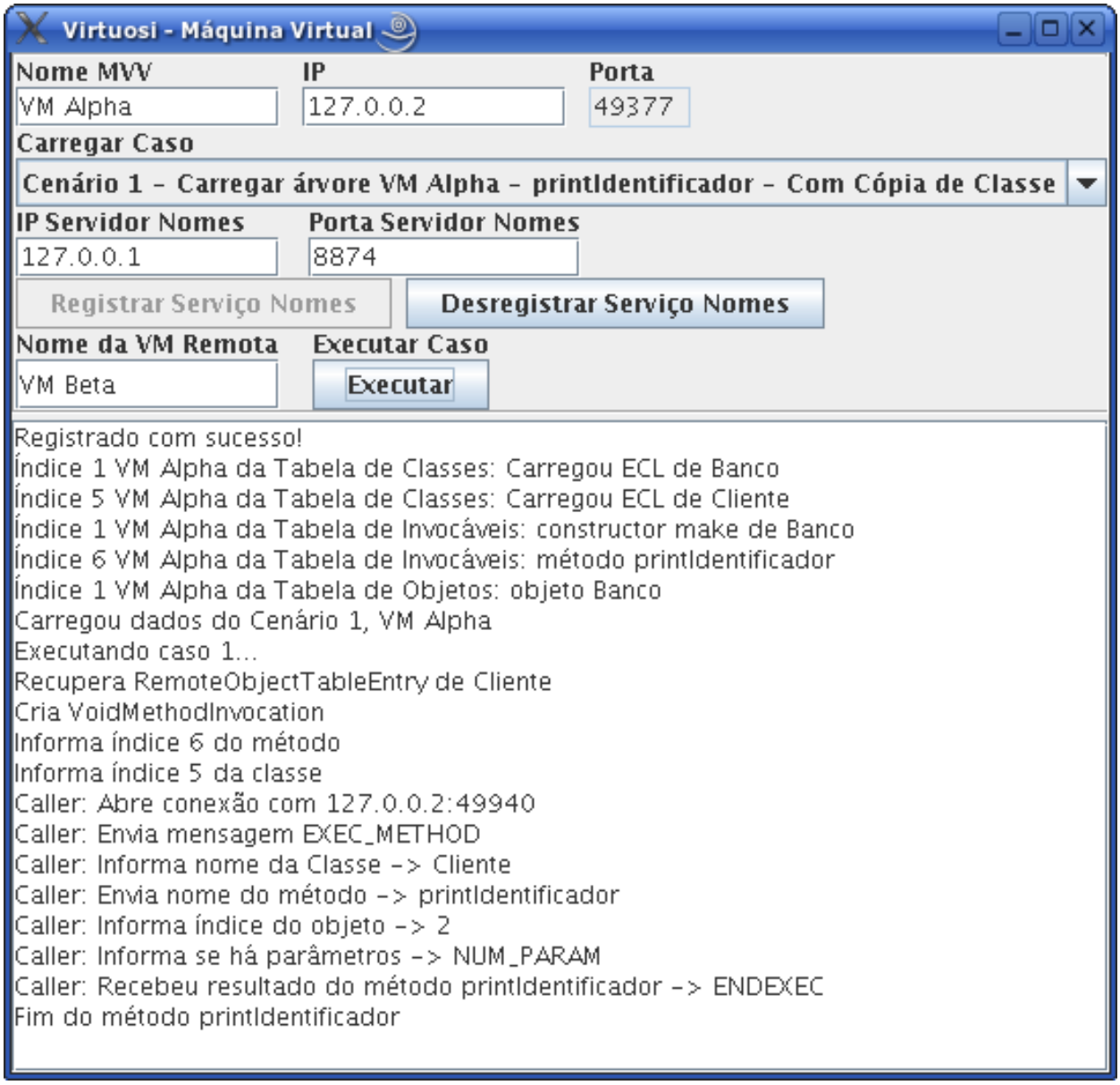


Figura 5.4: Interface da VM origem referente a invocação remota de método simples.

A figura 5.5 corresponde a VM Beta, que aguarda as requisições vindas da VM Alpha.

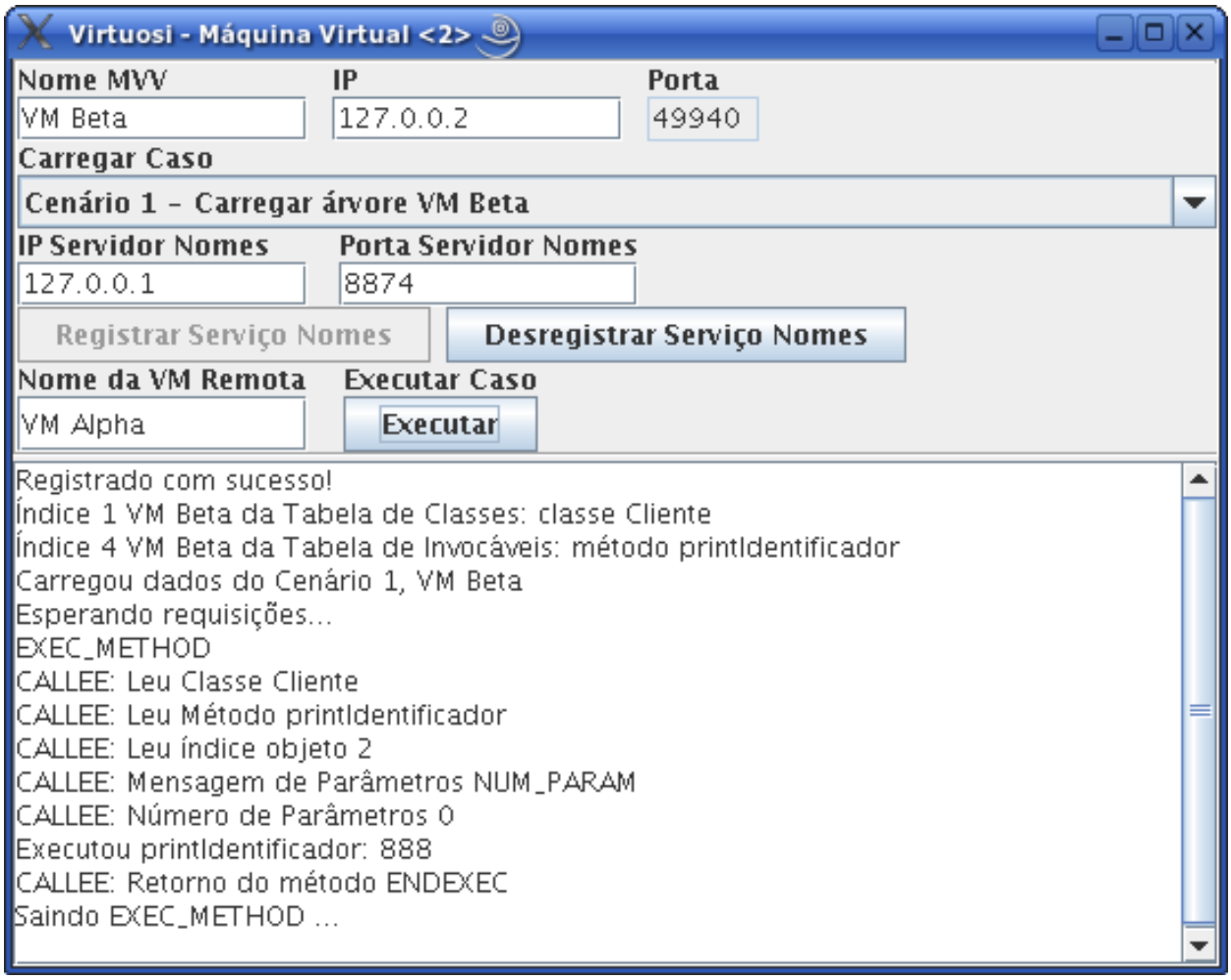


Figura 5.5: Interface da VM destino referente a invocação remota de método simples.

Cenário 2	
VM Alpha	**VM Beta**
Objeto banco	Objeto cliente
Execução: Objeto Banco invoca método *printIdentificador* do objeto Cliente	
Observação: Não existe uma cópia da árvore de programa de Cliente na VM Alpha. As informações do nome da classe e nome do método são obtidas na VM Beta	

Tabela 5.2: Quadro resumo referente ao cenário de invocação remota de método simples com árvore de programa remota.

A figura 5.6 mostra a VM Alpha no Cenário 2.

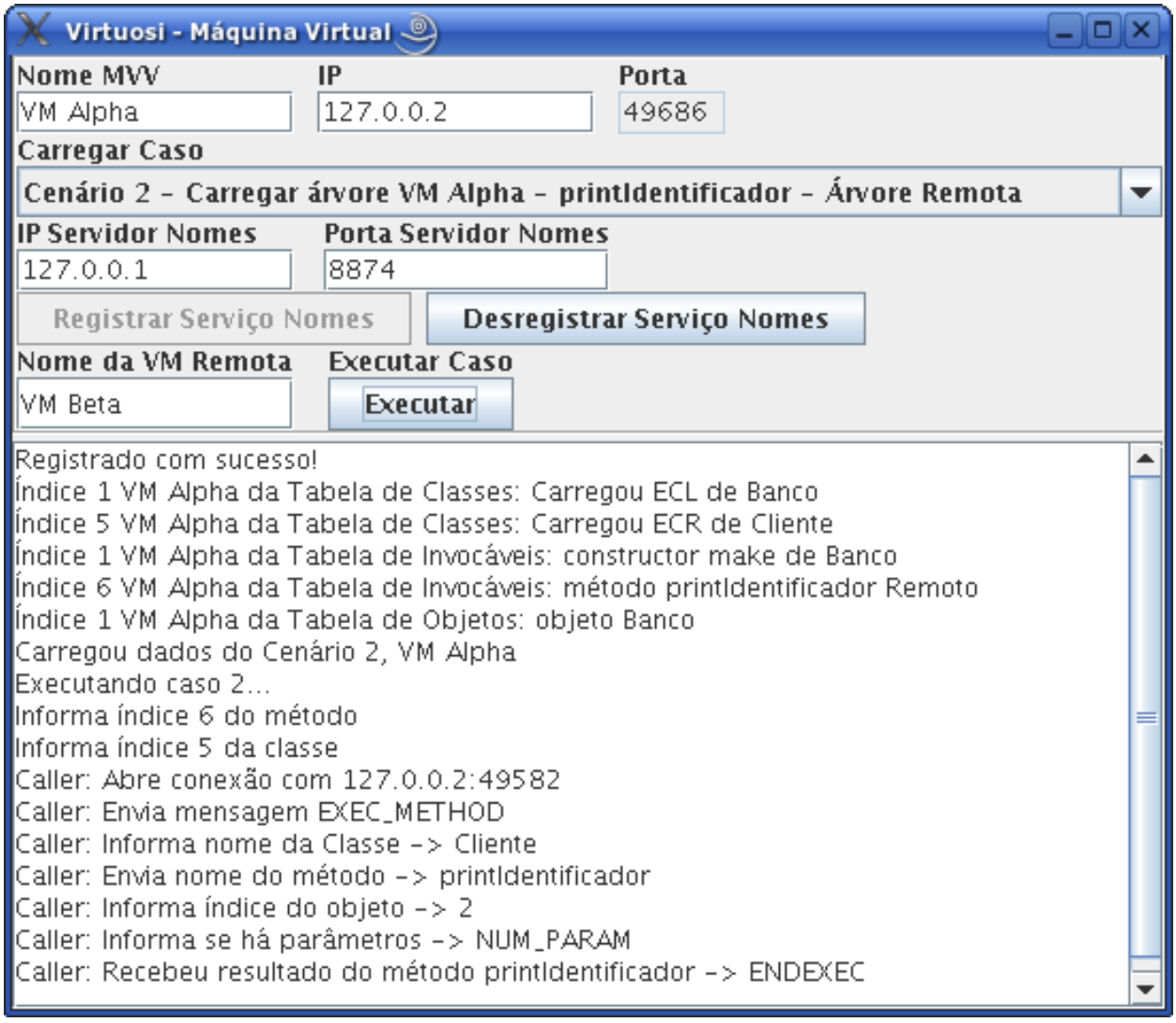


Figura 5.6: Interface da VM origem referente a invocação remota de método simples com árvore de programa remota.

Pode-se observar na figura 5.7 a VM Beta.

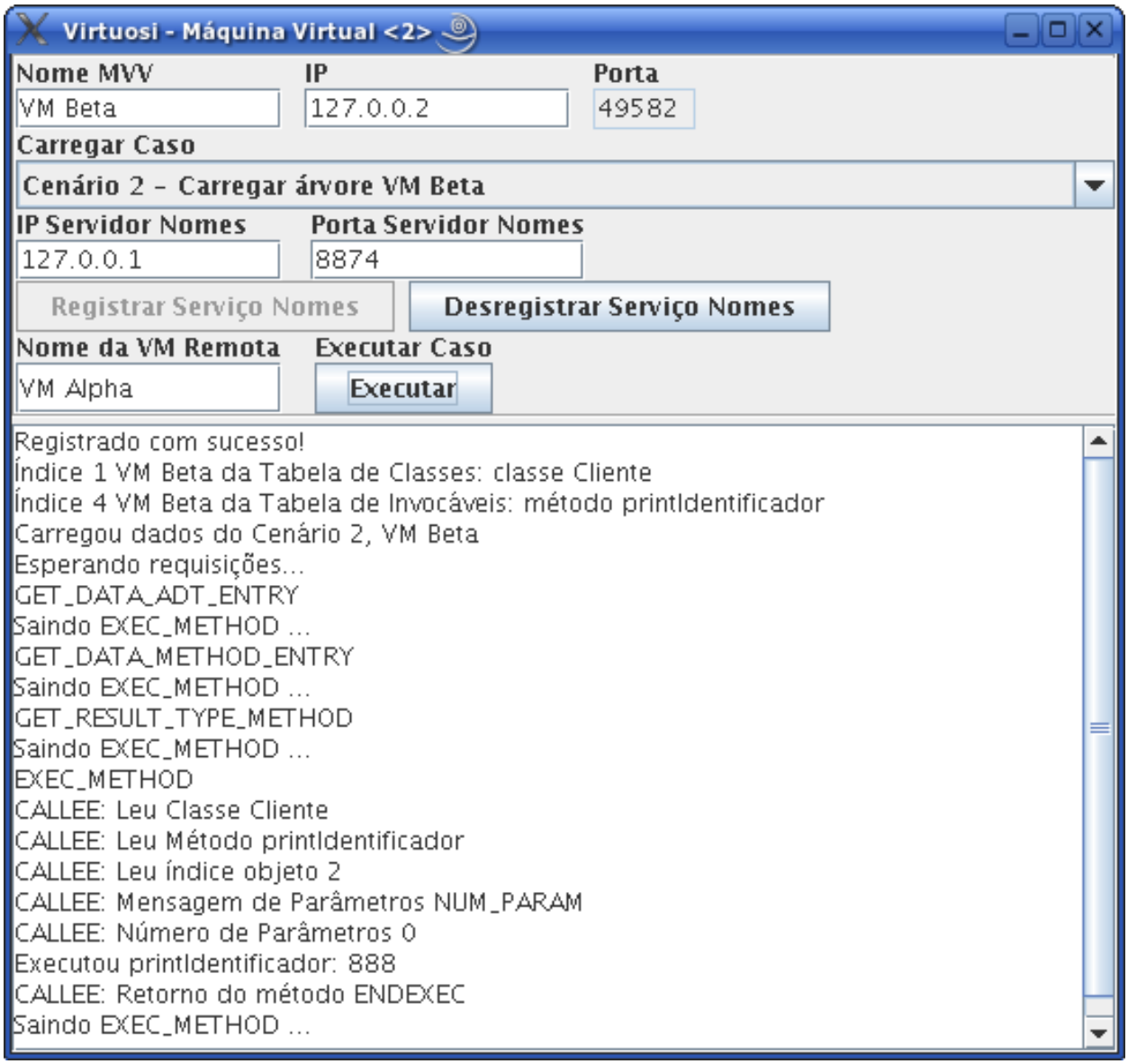


Figura 5.7: Interface da VM destino referente a invocação remota de método simples com árvore de programa remota.

Cenário 3	
VM Alpha	**VM Beta**
Objeto banco Objeto conta	Objeto cliente
Execução: Objeto Banco invoca método *obterConta* do objeto Cliente	
Observação: É chamado o método *obterConta*, porém o objeto Conta ainda se encontra na própria VM Alpha.	

Tabela 5.3: Quadro resumo referente a invocação remota de método com retorno, porém objeto de retorno se encontra na VM origem.

A figura 5.8 mostra a interface da VM Alpha durante a execução do cenário 3. Pode-se observar na figura 5.9 a VM Beta no momento da execução do cenário 3.

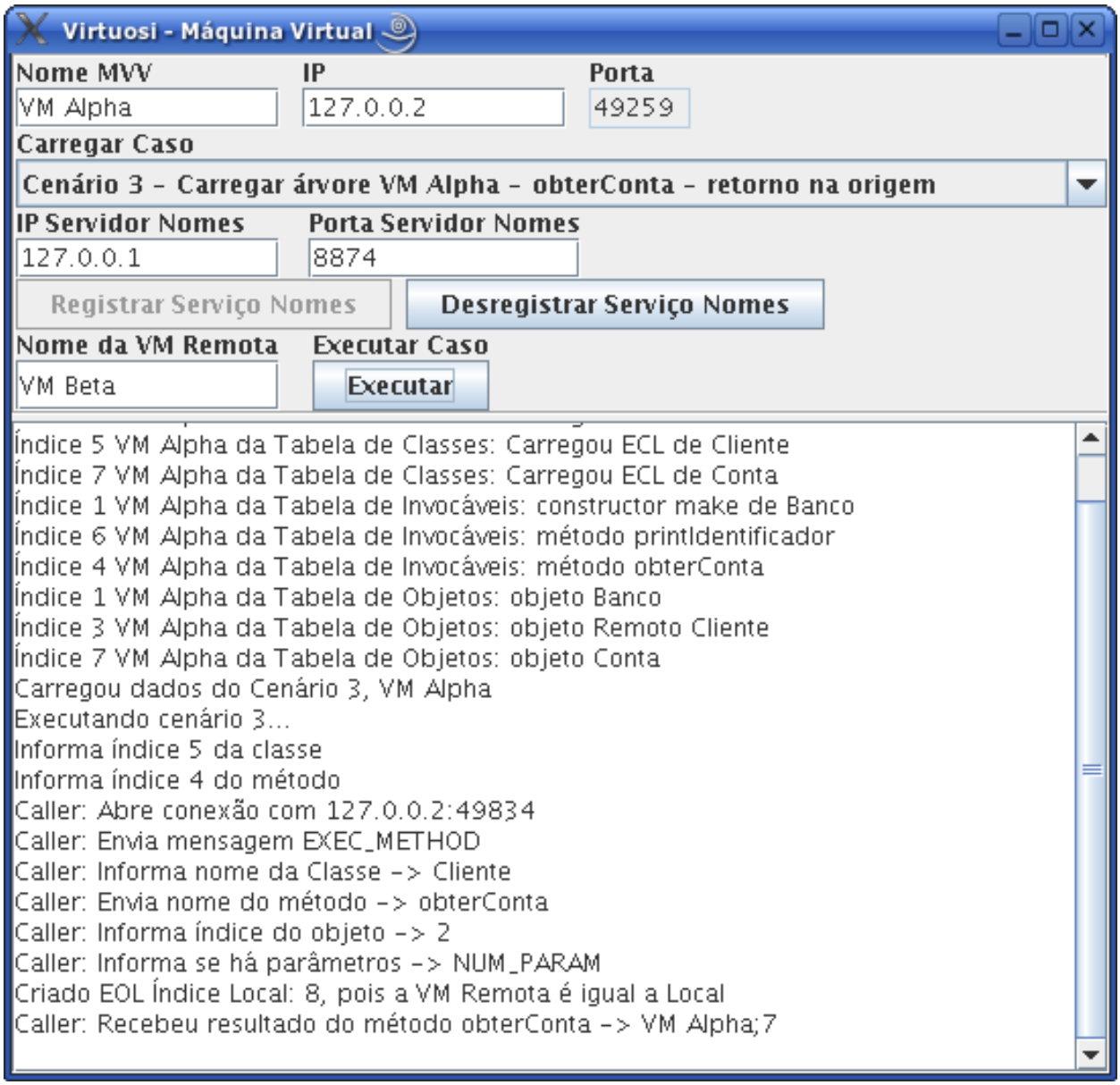


Figura 5.8: Interface da VM origem referente a invocação remota de método com retorno, estando o retorno na VM origem.

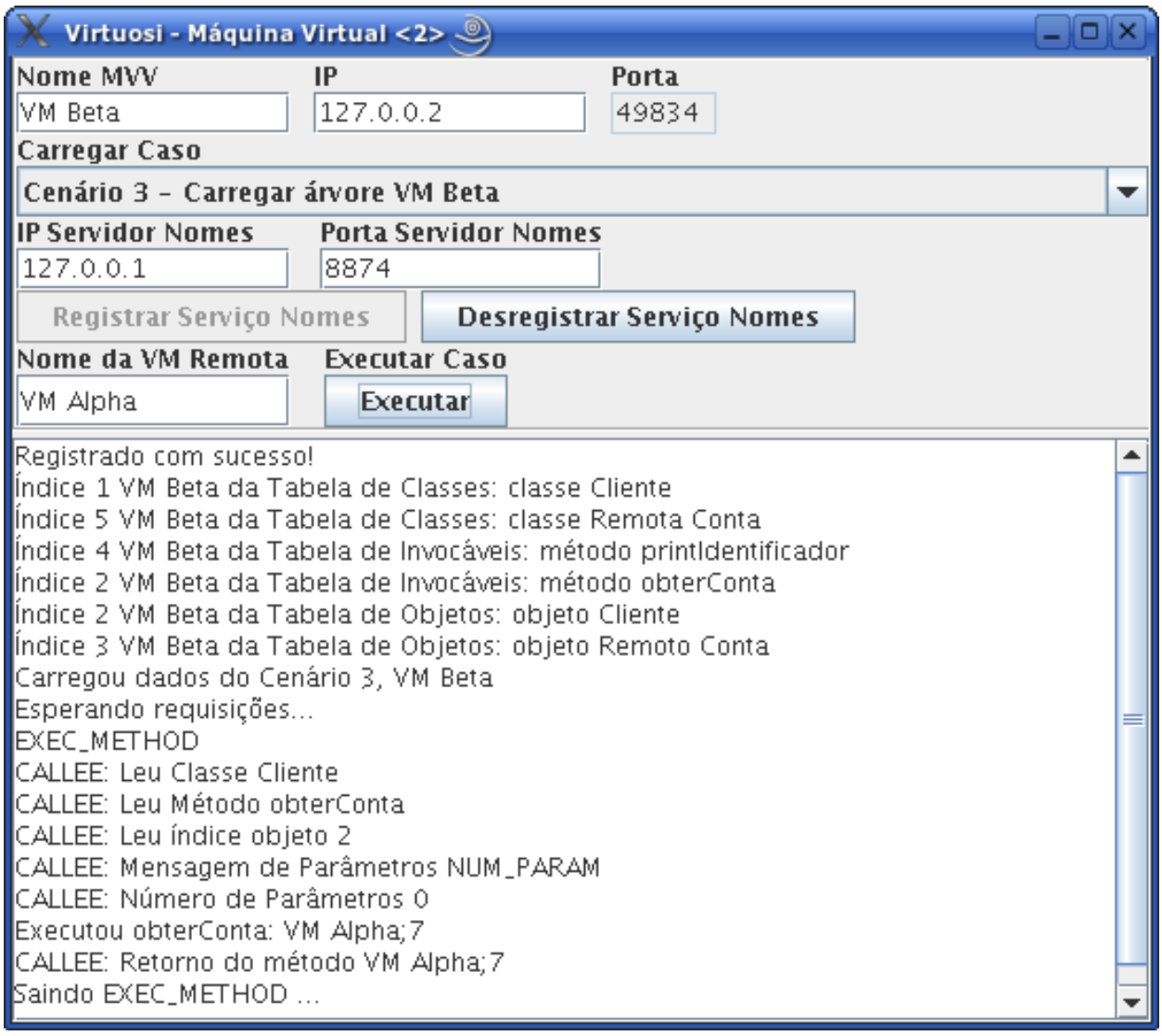


Figura 5.9: Interface da VM destino referente a invocação remota de método com retorno, estando o retorno na VM origem.

Cenário 4	
VM Alpha	**VM Beta**
Objeto banco	Objeto cliente Objeto conta
Execução: Objeto Banco invoca método *obterConta* do objeto Cliente	
Observação: É chamado o método *obterConta*, neste caso o objeto conta se encontra na VM Beta.	

Tabela 5.4: Quadro resumo referente a invocação remota de método com retorno.

A figura 5.10 mostra a interface da VM Alpha durante a execução do cenário 4. Observe na figura 5.11, a VM Beta no momento da execução do cenário 4.

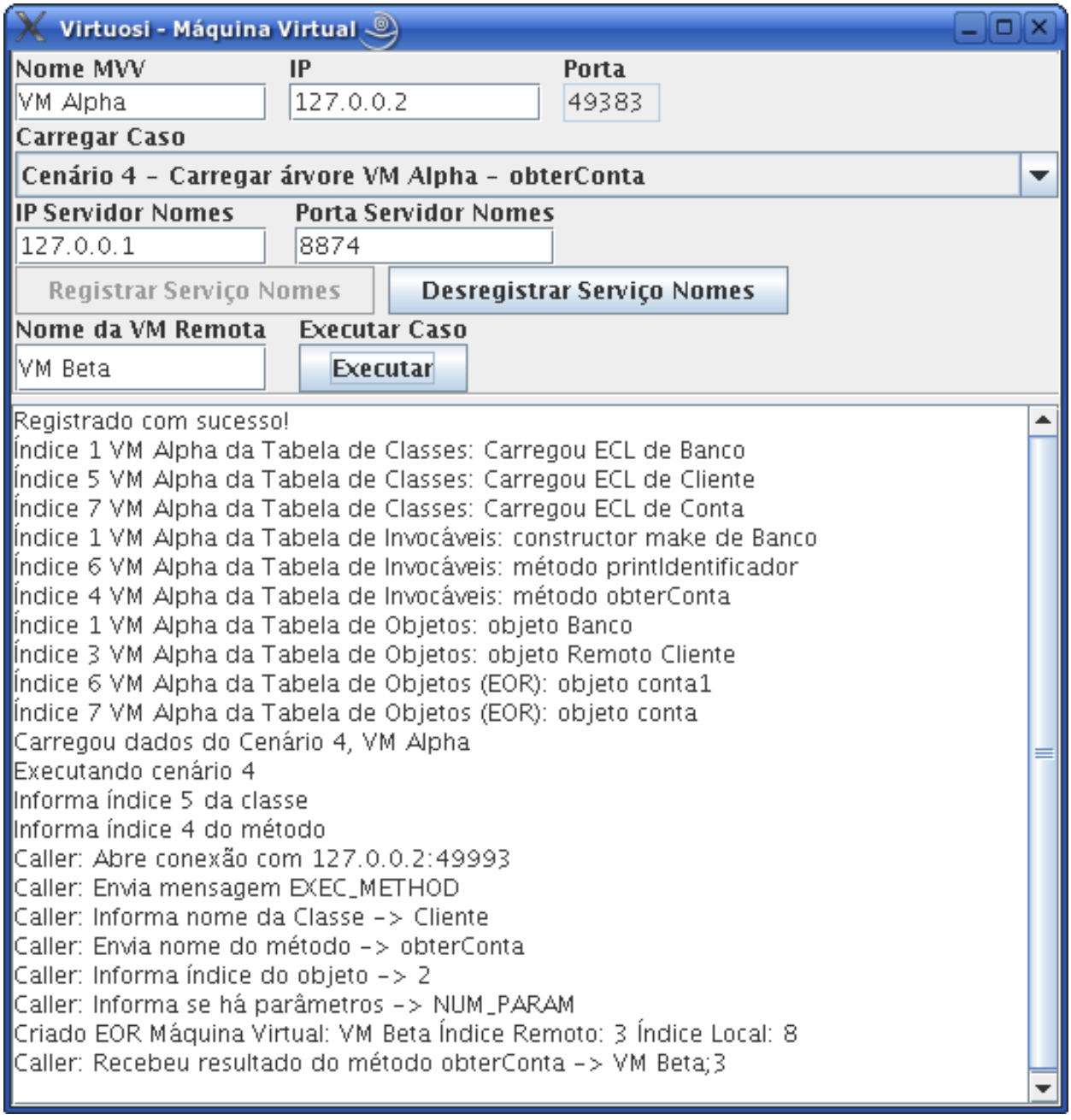


Figura 5.10: Interface da VM origem referente a invocação remota de método com retorno.

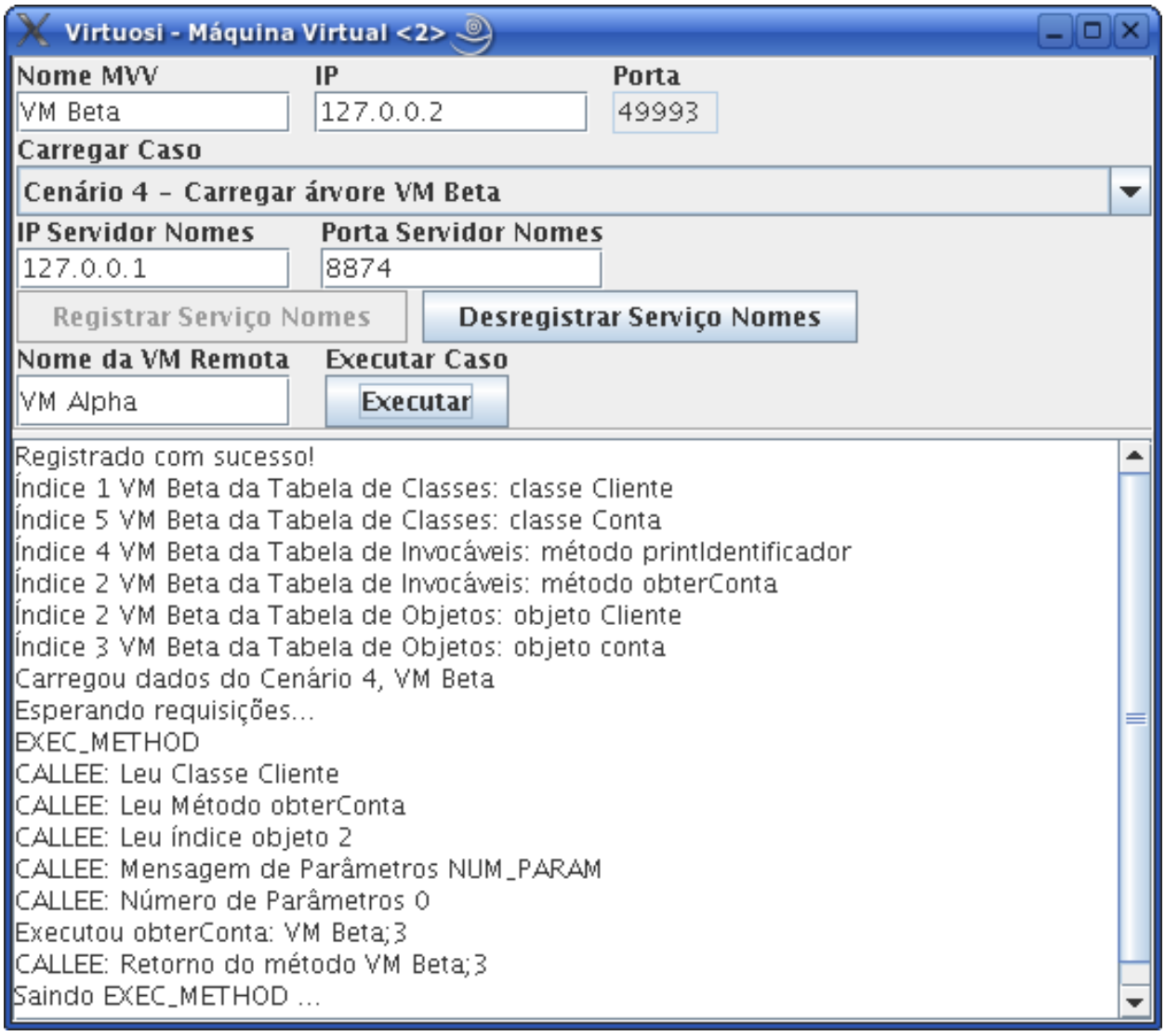


Figura 5.11: Interface da VM destino referente a invocação remota de método com retorno.

Cenário 5	
VM Alpha	**VM Beta**
Objeto banco Objeto contaNova	Objeto cliente Objeto conta
Execução: Objeto Banco invoca método *atualizarConta* do objeto Cliente	
Observação: O método *atualizarConta*, possui um parâmetro chamado *contaNova*, que o objeto está localizado na VM Alpha.	

Tabela 5.5: Quadro resumo referente a invocação remota de método com parâmetro (referência para objeto).

O cenário 5, em relação à VM Alpha pode ser observado na figura 5.12.
A figura 5.13 mostra a VM Beta no momento da execução do cenário 5.

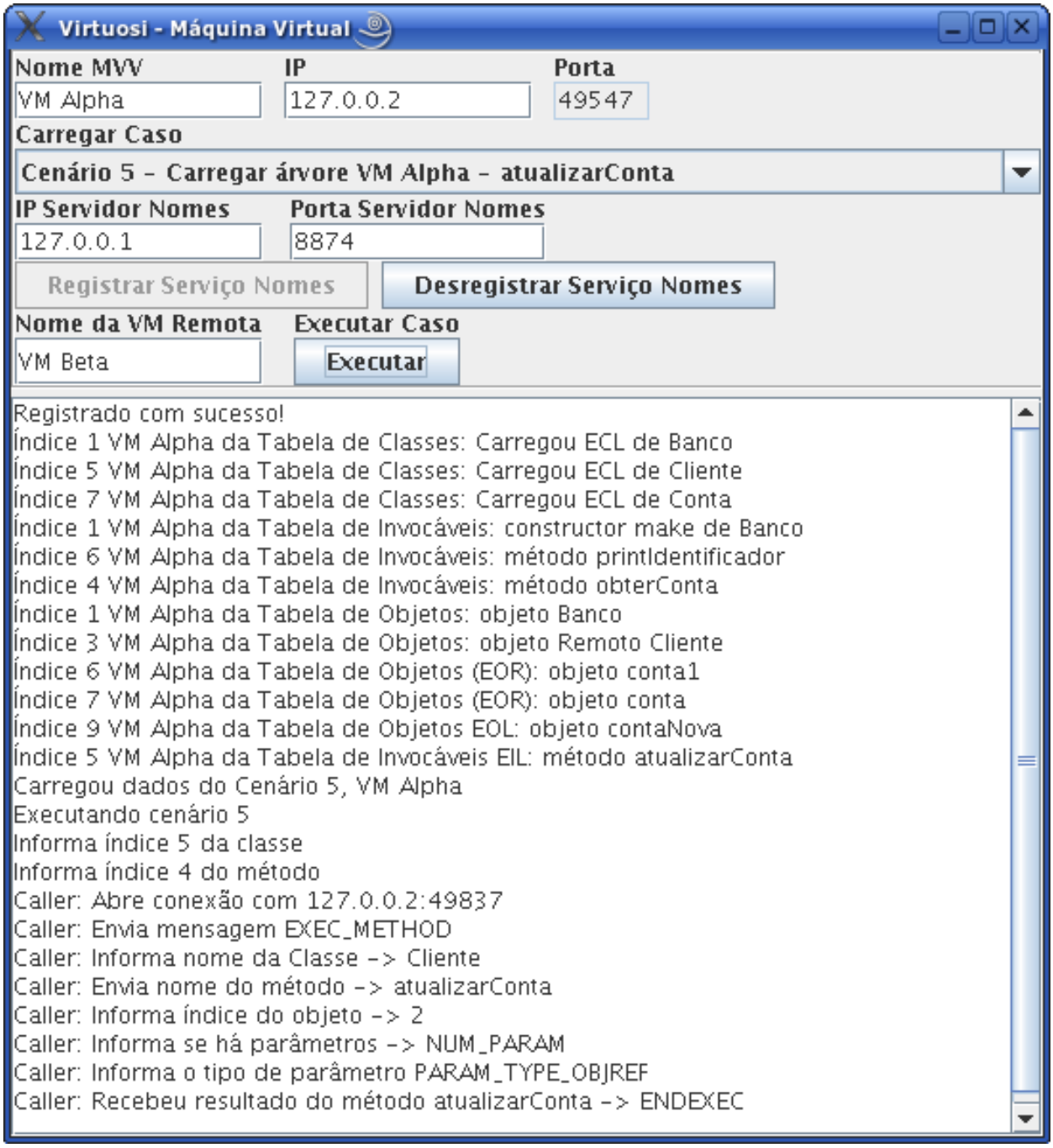


Figura 5.12: Interface da VM origem referente a invocação remota de método com parâmetro.

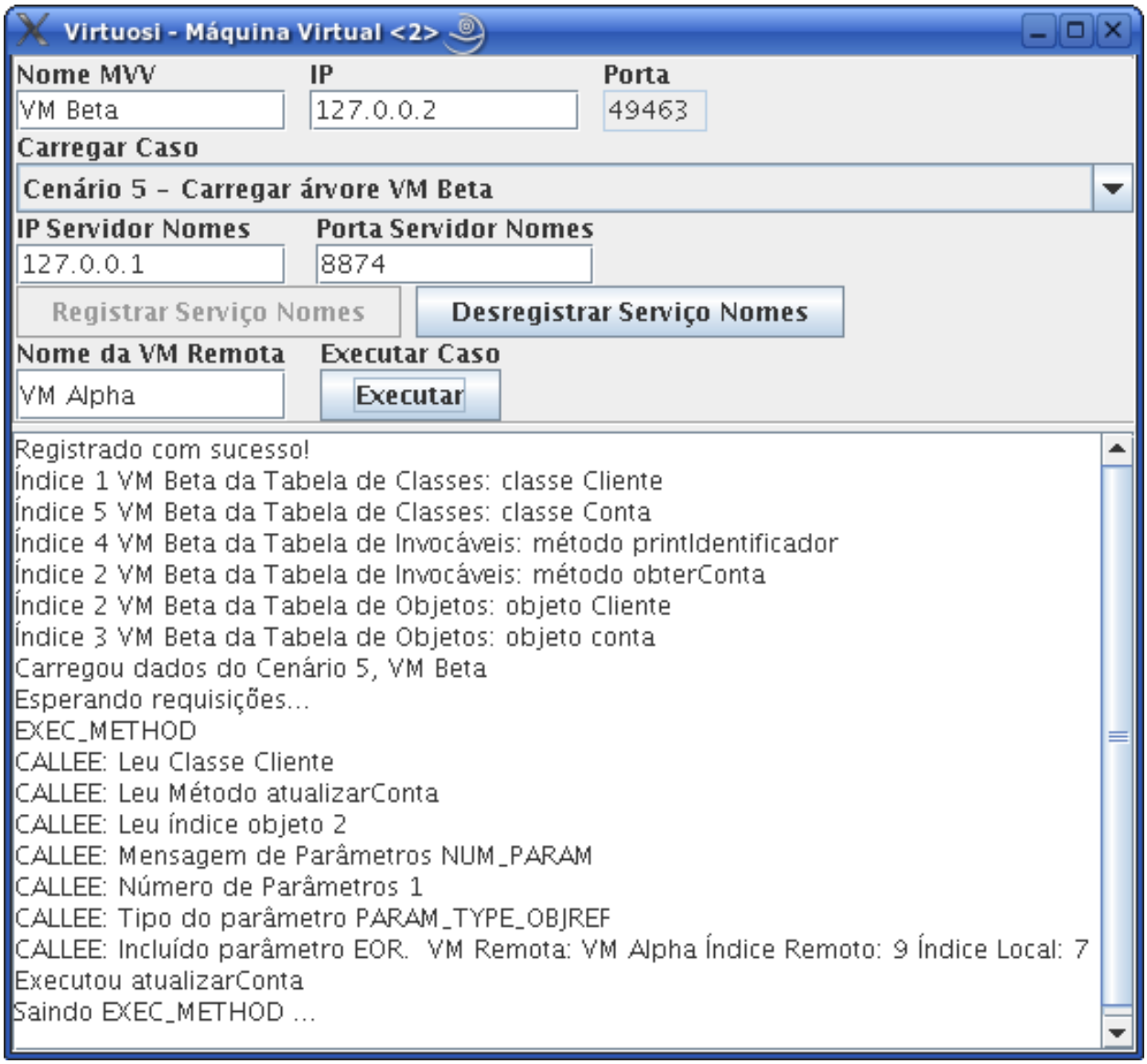


Figura 5.13: Interface da VM destino referente a invocação remota de método com parâmetro.

Cenário 6	
VM Alpha	**VM Beta**
Objeto banco Objeto contaNova	Objeto cliente Objeto conta
Execução: Objeto Banco invoca método *deposito* do objeto conta	
Observação: O método *deposito*, possui um parâmetro do tipo literal que transmitido por valor.	

Tabela 5.6: Quadro resumo referente a invocação remota de método com parâmetro passado por valor.

Pode-se observar na figura 5.14, o cenário 6, em relação à VM Alpha.
A VM Beta, do cenário 6, é apresentado através da figura 5.15.

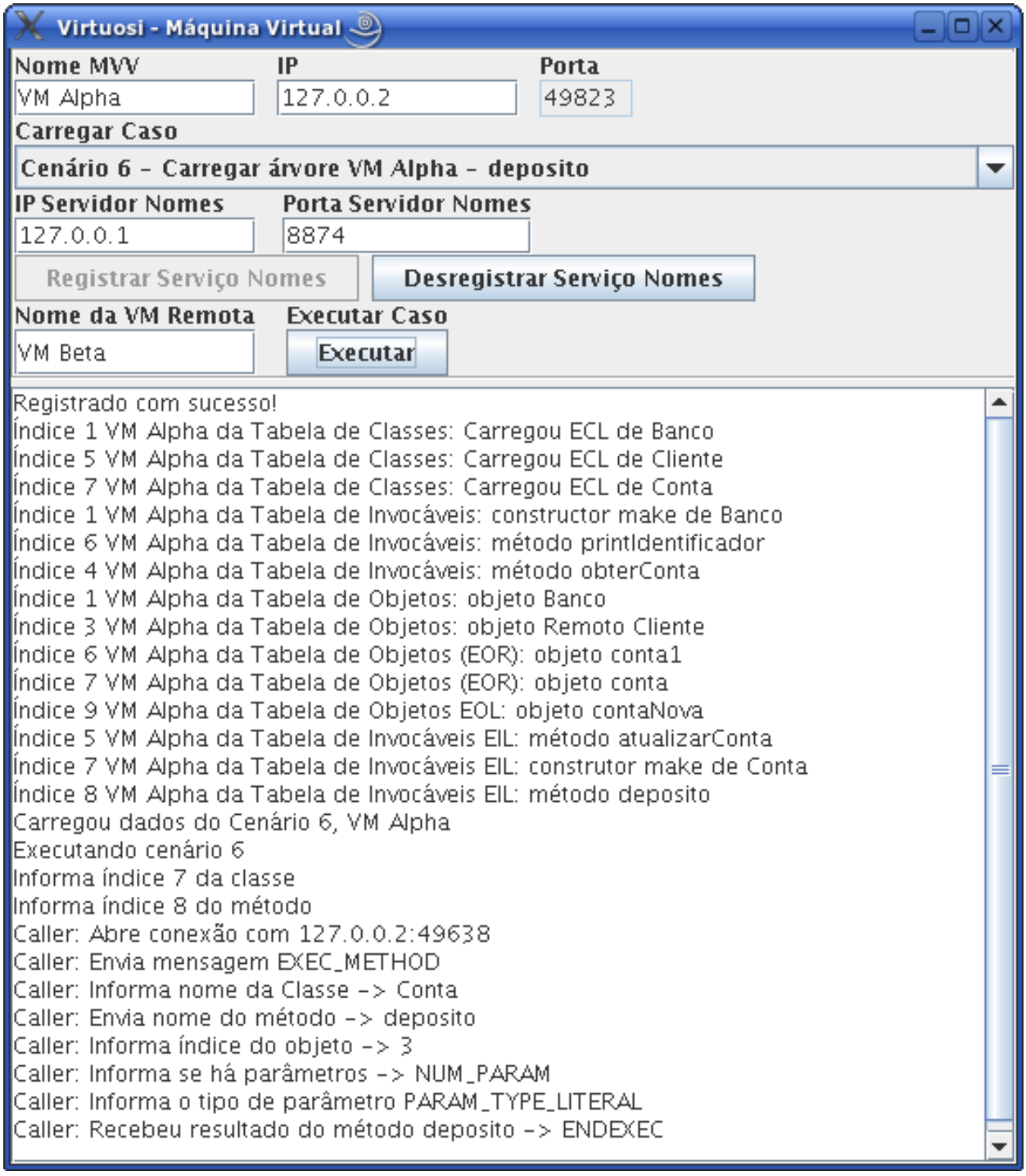


Figura 5.14: Interface da VM origem referente a invocação remota de método com parâmetro passado por valor.

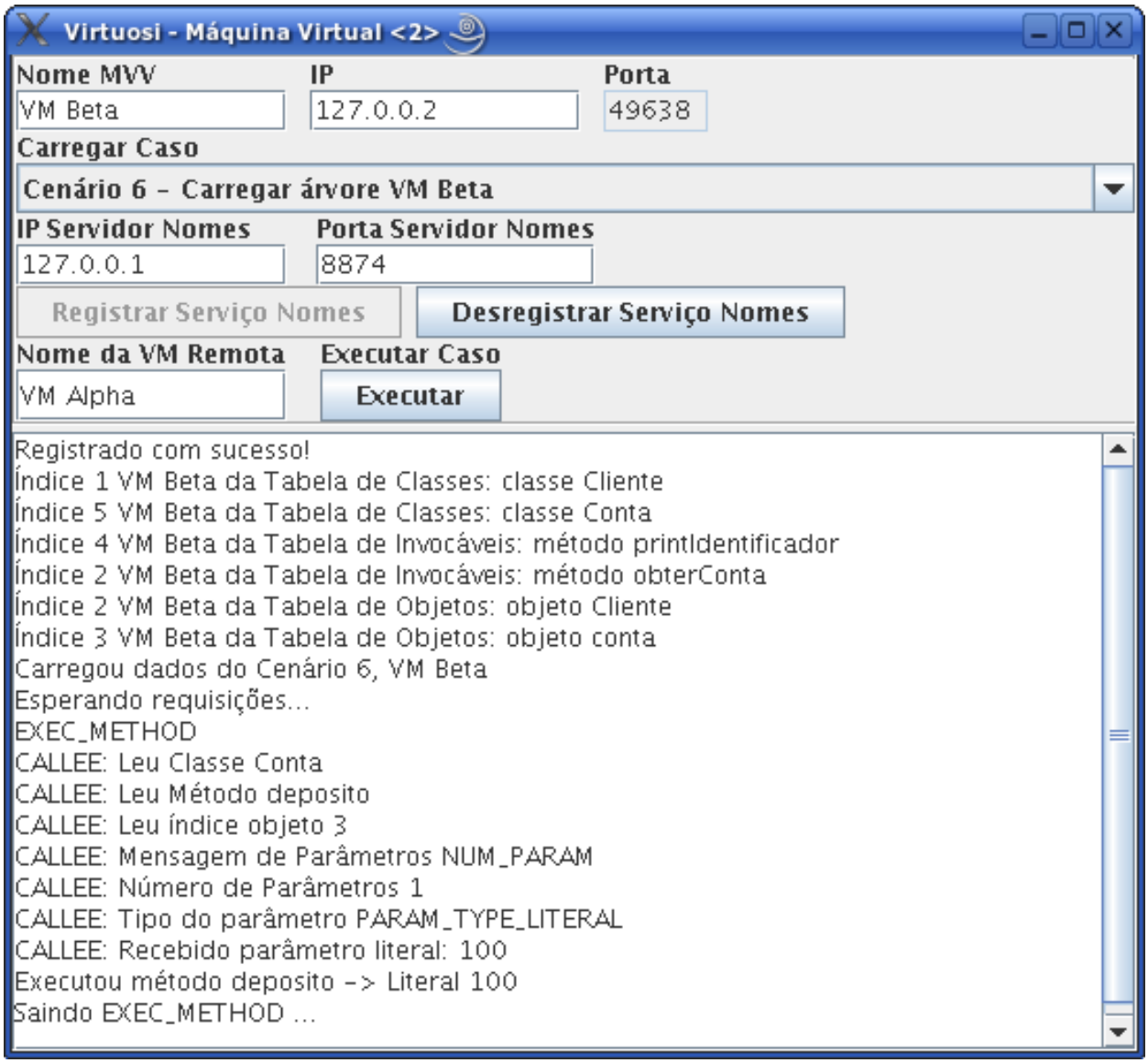


Figura 5.15: Interface da VM destino referente a invocação remota de método com parâmetro passado por valor.

Cenário 7	
VM Alpha	**VM Beta**
Objeto banco Objeto contaNova	Objeto cliente Objeto conta
Execução: Objeto Banco invoca a ação remota *saldoNegativo* do objeto conta	
Observação: A ação remota *saldoNegativo* retorna 0 (zero), pois no cenário anterior foi atribuído o valor 100 para o saldo.	

Tabela 5.7: Quadro resumo referente a invocação remota de ação.

A figura 5.16 mostra a interface da VM Alpha durante a execução do cenário 7.
A figura 5.17 mostra a VM Beta no momento da execução do cenário 7.

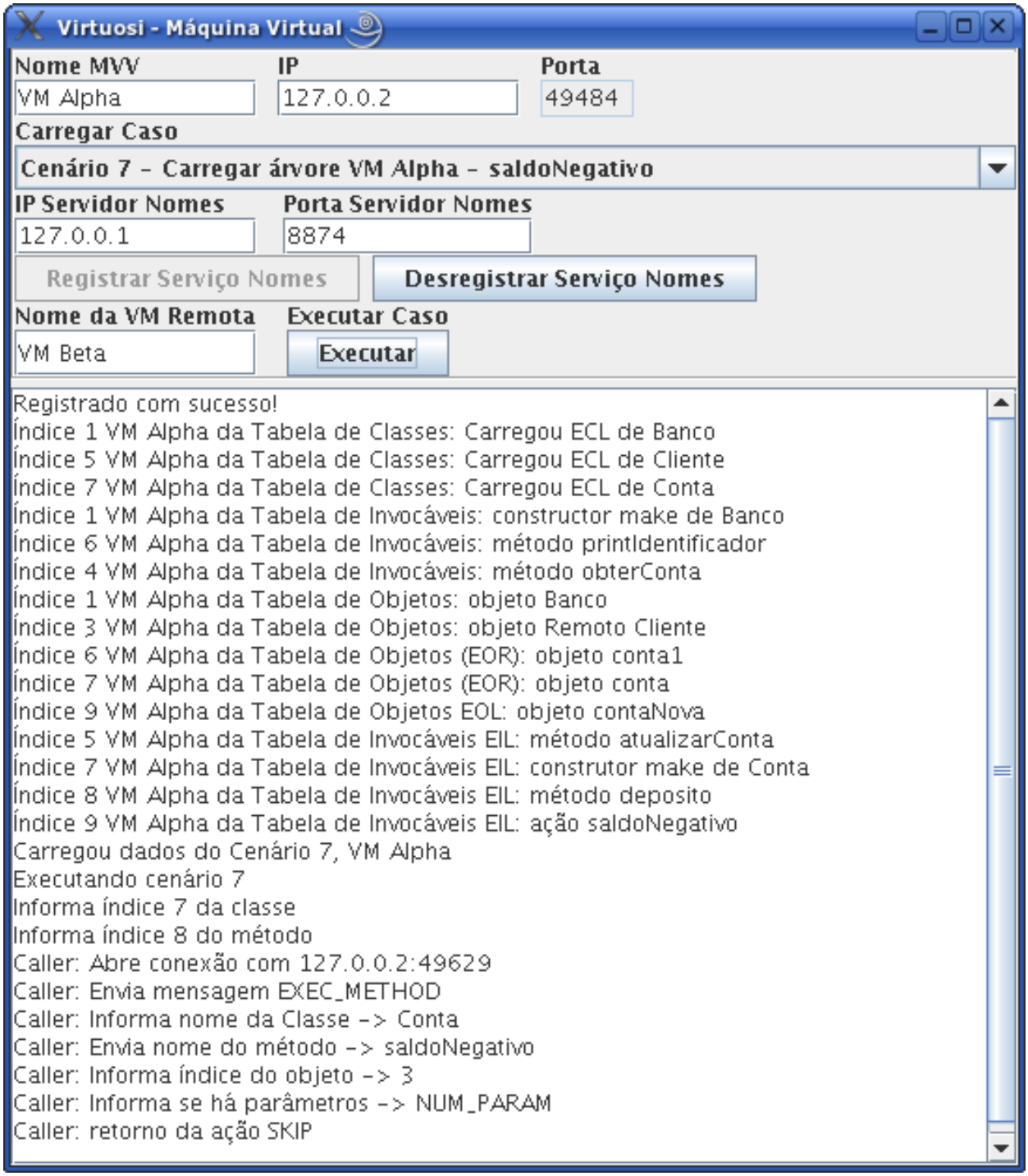


Figura 5.16: Interface da VM origem referente a invocação remota de ação.

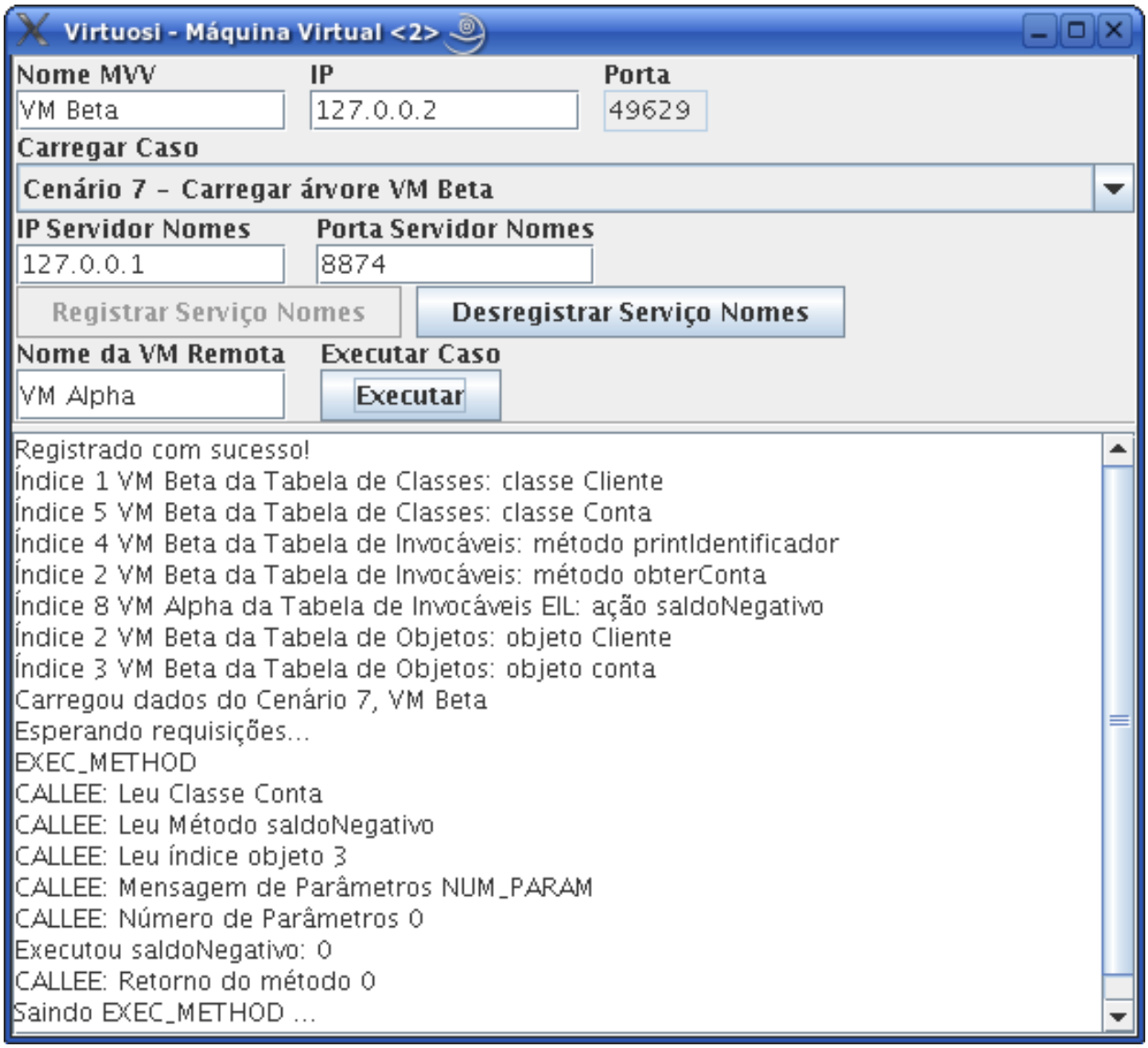


Figura 5.17: Interface da VM destino referente a invocação remota de ação.

5.6 Desempenho

A partir dos cenários apresentados acima realizou-se uma coleta de dados para verificar o desempenho apresentado durante a invocação dos métodos remotos.

O ambiente utilizado para avaliar o desempenho é composto por um computador com processador *Intel Celeron* 2.20GHz e 512MB de memória RAM, atuando como servidor (responsável pela recepção e processamento da invocação do método remoto). E outro computador com processador *AMD Atlon XP 2000* 1.26GHz e 256 MB de memória RAM, atuando como cliente (responsável pela invocação do método remoto). Ambos conectados por uma rede de computadores cuja velocidade de transmissão de dados é de 100Mbps.

A invocação de cada cenário apresentado acima foi executada 5 vezes, para cada execução foi coletado os tempos para o término da invocação remota do método.

Cenário 1	
Número da execução	**Tempo em milisegundos**
1	297
2	313
3	390
4	391
5	297
Tempo Médio: 337,6	

Tabela 5.8: Tempos para execução do cenário de invocação remota de método simples.

Cenário 2	
Número da execução	**Tempo em milisegundos**
1	2110
2	2312
3	2516
4	2422
5	2109
Tempo Médio: 2293,8	

Tabela 5.9: Tempos para execução do cenário de invocação remota de método simples com árvore de programa remota.

Cenário 3	
Número da execução	**Tempo em milisegundos**
1	297
2	406
3	296
4	391
5	312
Tempo Médio: 340,4	

Tabela 5.10: Tempos para execução do cenário da invocação remota de método com retorno, porém objeto de retorno se encontra na VM origem.

Cenário 4	
Número da execução	**Tempo em milisegundos**
1	297
2	390
3	406
4	313
5	296
Tempo Médio: 340,4	

Tabela 5.11: Tempos para execução do cenário da invocação remota de método com retorno.

Cenário 5	
Número da execução	**Tempo em milisegundos**
1	406
2	297
3	312
4	297
5	391
Tempo Médio: 340,6	

Tabela 5.12: Tempos para execução do cenário da invocação remota de método com parâmetro (referência para objeto).

Cenário 6	
Número da execução	**Tempo em milisegundos**
1	500
2	312
3	297
4	296
5	297
Tempo Médio: 340,4	

Tabela 5.13: Tempos para execução do cenário da invocação remota de método com parâmetro passado por valor.

Cenário 7	
Número da execução	**Tempo em milisegundos**
1	297
2	391
3	313
4	296
5	406
Tempo Médio: 340,6	

Tabela 5.14: Tempos para execução do cenário da invocação remota de ação.

Conforme observado nas tabelas de tempos dos cenários de execução de invocação de métodos remotos, os tempos médios para invocação de métodos remotos ficam em torno de 340 milisegundos. No entanto, como ilustrado na tabela 5.9, os tempos para execução do cenário 2 ficaram bem acima da média, o que denota a necessidade de se evitar situações das quais não é deixado uma cópia definição da árvore de programa, na máquina virtual origem, do objeto remoto invocado.

Capítulo 6

Conclusão e Trabalhos Futuros

O mecanismo de invocação remota de métodos para a VIRTUOSI, proposto neste trabalho, atende um dos principais requisitos defendidos por [Bir84] através do RPC, que é a transparência na invocação remota de métodos. O mecanismo de invocação remota de métodos para a VIRTUOSI, no que se refere aos objetivos da VIRTUOSI – pedagógico e experimental – atendeu de forma satisfatória nos aspectos mais relevantes como a transparência de acesso e localização, diferentemente de CORBA e Java RMI que atendem somente ao quesito de transparência de localização.

No entanto, tanto CORBA como o Java RMI contribuiram de forma positiva para a implementação do mecanismo de invocação remota de métodos para a VIRTUOSI. Por exemplo, a estrutura de recepção e controle de troca de mensagens do mecanismo de invocação de métodos remotos da VIRTUOSI foi inspirado no ORB do CORBA. E o serviço de nomes do Java RMI (*rmiregistry*) serviu como orientação para o desenvolvimento do serviço de nomes para a VIRTUOSI.

Um fator negativo, porém não restritivo, é que o mecanismo de invocação remota de métodos não permite a criação de objetos remotos. Necessariamente, os objetos devem ser criados localmente, em relação à máquina virtual, e devem passar a ser remotos por procedimento de migração, invocação de método remoto com retorno ou invocação de método remoto com passagem de parâmetro.

6.1 Contribuição Científica

Esta dissertação apresenta como contribuição científica os seguintes aspectos:

- Formaliza o mecanismo de invocação remota de métodos para a VIRTUOSI;
- Serve como base para o desenvolvimento de outros mecanismos de invocação remota de métodos;
- Valida o mecanismo de tabelas de referências para a distribuição de objetos;
- Valida o mecanismo de servidor de nomes para o ambiente da VIRTUOSI.

6.2 Trabalhos Futuros

Na elaboração desta dissertação pode-se identificar alguns trabalhos futuros:

- Conceitos essenciais que devem ser implementados:
  - Herança;
  - Polimorfismo;
  - Entrada e Saída;
  - Manipulação de índices de bloco de dados;
  - Lista de Exportação.
- Melhorias na implementação da Máquina Virtual Virtuosi (MVV):
  - Melhorar a estrutura interna com o objetivo de simplificar a implementação;
  - Construir uma implementação em uma linguagem com melhor desempenho do que Java;
  - Melhorar a ligação entre o compilador existente e a MVV;
  - Construir uma interface homem-máquina amigável (editor e depurador);
- Deve-se estender o metamodelo VIRTUOSI para contemplar:
  - Mais de uma linha de execução através de *multithreads*;
  - Controle de concorrência;
  - Tratamento de exceções;
  - Interface;
  - Classes abstratas;
- Segurança na comunicação;
- Aumento do desempenho de comunicação;
- *Garbage Collector*;
- Controle de referências inválidas no servidor de nomes;
- Construção de bibliotecas para os usuários finais (programadores);
- Passagem de objetos *por valor* como parâmetros de métodos.

Pode-se dizer enfim que este trabalho permitiu a definição e a implementação de um mecanismo de invocação remota de métodos que funciona de forma transparente para máquinas virtuais orientada à distribuição de objetos, e o mesmo pode servir como base para diversos trabalhos mais avançados.

Referências Bibliográficas

[Ald97] Aldrich, Jonathan,Dooley, James,Mandelsohn, Scott,Rifkin, Adam. Providing easier access to remote objects in client-server systems. Technical Report 1997.cs-tr-97-20, California Institute of Technology, 1997.

[Aro04] Aron Borges. Aram: uma linguagem de programação para o ambiente Virtuosi. Projeto Final II, Pontifícia Universidade Católica do Paraná, 5 2004.

[AV01] Marco Avvenuti and Alessio Vecchio. Embedding remote object mobility in java rmi. In *Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '01)*, Bologna, Italy, October 2001.

[BDV+98] Fabian Breg, Shridhar Diwan, Juan Villacis, Jayashree Balasubramanian, Esra Akman, and Dennis Gannon. Java RMI performance and object model interoperability: experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10(11–13):941–955, 1998.

[Ben87] J. K. Bennett. The design and implementation of distributed smalltalk. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 22, pages 318–330, New York, NY, 1987. ACM Press.

[Bir84] Birrell, Andrew,Nelson Bruce J. Implementing remote procedure calls. *ACM Transactions on Computer Systems - TOCS*, 2(1), 2 1984.

[Bir97] Kenneth P. Birman. Building secure and reliable network applications. In *WWCA*, pages 15–28, 1997.

[Bri88] Briot, J. P., Guerraqui, R., Lohr, K. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3), 9 1988.

[Bus98] Jackson,Leroy Buss, Arnold. Distributed simulation modeling: a comparison of hla, corba, and rmi. In *Proceedings of the 30th conference on Winter simulation*, pages 819–826. IEEE Computer Society Press, 1998.

[Cal00] Calsavara, Alcides. Virtuosi: Máquinas virtuais para objetos distribuídos. *Trabalho apresentado em concurso para professor titular da PUC-PR*, 2000.

[CHY+97] Emerald Chung, Yennun Huang, Shalini Yajnik, Deron Liang, Joanne C. Shih, Chung-Yih Wang, and Yi-Min Wang. DCOM and CORBA side by side, step by step and layer by layer, 1997.

[CNdC+04] Alcides Calsavara, Agnaldo Kiyoshi Noda, Juarez da Costa, Carlos Kolb, and Leonardo Nunes. A virtual-machine-based middleware. *Posters of the 2004 DOA (Distributed Objects and Applications) International Conference*, 3292/2004, October 2004.

[Cox03] Cox, Alan,Hu, Charlie Y.,Yu, Weimin,Wallach, Dan,Zwaenepoel Willy. Run-time support for distribuited sharing in safe languages. acm transactions on computer systems. . *ACM Transactions on Computer Systems*, 21(1), 2 2003.

[dCCF04] Juarez da Costa Cesar Filho. Mecanismo de mobilidade de objetos para a virtuosi. Master's thesis, Pontifícia Universidade Católica do Paraná– PUCPR, July 2004.

[dCM03] Carla de Carvalho Marchioro. Persistência de objetos baseada em reflexão computacional. Master's thesis, Pontifícia Universidade Católica do Paraná– PUCPR, January 2003.

[Die92] Dietmar Fauth - Siemens Nixdorf Informationssytema AG ,Chikong Sue - Open Software Foundation Inc. Remote procedure call: Technology, standardization and osf's distributed computing enviroment, 1992.

[DKS97] K.V. Dyshlevoi, V.E. Kamensky, and L.B. Solovskaya. Marshalling in distributed systems: Two approaches, 1997.

[DMN70] O. J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula 67 Common Base Language*. Norwegian Computing Center, 1970.

[Eri95] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Gro97] The Open Group. *DCE 1.1: Remote Procedure Call*. The Open Group, 10 1997.

[HJK+99] Michael W. Hicks, Suresh Jagannathan, Richard Kelsey, Jonathan T. Moore, and Cristian Ungureanu. Transparent communication for distributed objects in java. In *Java Grande*, pages 160–170, 1999.

[HV99] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison Wesley, 1 edition, February 1999.

[HYC+03] Y. Charlie Hu, Weimin Yu, Alan Cox, Dan Wallach, and Willy Zwaenepoel. Run-time support for distributed sharing in safe languages. *ACM Transactions on Computer Systems (TOCS)*, 21(1):1–35, 2003.

[Inc99] Sun Microsytems Inc. Java remote method invocation specification. `http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmiTOC.html`, 12 1999.

[Int04] Internet Assigned Numbers Authority. IANA Protocol Numbers and Assignment Services directory — Port numbers, 8 2004.

[Joh00] John Siegel. *CORBA 3 Fundamentals and Programming*. Wiley Computer Publishing, 2 edition, 2000.

[Jun02] Gildo Medeiros Junior. Uma abordagem contínua para sistemas de software baseados em estados e eventos. Master's thesis, Pontifícia Universidade Católica do Paraná– PUCPR, December 2002.

[KF99] Thomas Kistler and Michael Franz. A tree-based alternative to java bytecodes. *International Journal of Parallel Programming*, 27(1):21–33, 1999.

[Kol04] Carlos José Johann Kolb. Um sistema de execução para software orientado a objeto baseado em Árvores de programa. Master's thesis, Pontifícia Universidade Católica do Paraná– PUCPR, July 2004.

[LI86] Klaus-Peter LShr and Rainer Isle. RPC Stubs for Distributed Modules. In *ACM SIGOPS European Workshop*. ACM, September 1986.

[LPZ+05] Shiding Lin, Aimin Pan, Zheng Zhang, Rui Guo, and Zhenyu Guo. Wids: an integrated toolkit for distributed system development. *Tenth Workshop on Hot Topics in Operating Systems (HotOS X)*, 2005.

[Mae87] Maes, Patie. Concepts and experiments in computational reflection. *OOPSLA87 Proceedings*, 1987.

[Mey97] Meyer, Bertrand. *Object-Oriented Software Construction*. Prentice Hall PTR, 1997.

[Mic04] Sun Microsystems. Getting started with java idl, 2004.

[MMP88] Ole Lehrmann Madsen and Birger Møller-Pedersen. What object-oriented programming may be – and what it does not have to be. In S. Gjessing and K. Nygaard, editors, *Proceedings ECOOP '88*, LNCS 322, pages 1–20, Oslo, August 1988. Springer Verlag.

[MNV+01] Jason Maassen, Rob Van Nieuwpoort, Ronald Veldema, Henri E. Bal, Thilo Kielmann, Ceriel J. H. Jacobs, and Rutger F. H. Hofman. Efficient java RMI for parallel programming. *Programming Languages and Systems*, 23(6):747–775, 2001.

[Oli98] Oliva, A.,Garcia I. C.,Buzato L.E. The Reflexive Architecture of Guaraná. *Instituto de Computação Universidade Estadual de Campinas*, 1998.

[OMG02a] OMG - Object Management Group. Naming service specification, September 2002.

[OMG02b] Object Management Group OMG. Common object request broker architecture: Core specification, 12 2002.

[Par04] Terence Parr. Antlr another tool for language recognition. `http://www.antlr.org`, 2004.

[PZ97] Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[Rui00] Bao Ruixian. Distributed computing via rmi and corba, 2000.

[SM95] Inc. Sun Microsystems. Onc+ developer's guide, November 1995.

[Sri95] R. Srinivasan. RPC: Remote procedure call specification version 2. RFC 1831, Network Working Group, Aug 1995.

[Tan95] Tanenbaum, Andrew S. *Distributed Operating Systems*. Prentice Hall, 1995.

[TS03] E. Tilevich and Y. Smaragdakis. Nrmi: Natural and efficient middleware, 2003.

[TTK98] G. K. Thiruvathukal, L. S. Thomas, and A. T. Korczynski. Reflective remote method invocation. *Concurrency: Practice and Experience*, 10(11–13):911–925, 1998.

[WRW96] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems (COOTS)*, pages 219–232. USENIX Association, 1996.

[WW04] Ann Wollrath and Jim Waldo. Trail: Rmi, 2004.

Apêndice A

Metamodelo da VIRTUOSI

Esse Capítulo especifica os conceitos de orientação a objeto suportados pela Virtuosi através da formalização do metamodelo da Virtuosi. Para auxiliar o entendimento dos conceitos explicados ao longo desse Capítulo, serão utilizados trechos de código fonte de uma aplicação de software orientado a objeto escritos na linguagem Aram.

A.1 Especificação

Os conceitos de orientação a objeto suportados pela Virtuosi são formalizados através de um diagrama de classes da UML chamado de metamodelo da Virtuosi. O metamodelo da virtuosi possui classes e associações que representam os elementos encontrados em uma linguagem de programação orientada a objeto que suporte aos conceitos suportados pela Virtuosi. Por isso, as classes do metamodelo podem ser chamadas de meta-classes. Por exemplo, uma classe possui atributos; o metamodelo da Virtuosi, portanto, possui uma meta-classe para representar uma classe de aplicação, uma meta-classe para representar um atributo e através de uma associação, explicita se uma classe possui zero ou muitos atributos. O metamodelo da Virtuosi pode ser entendido como um diagrama de classes que descreve conceitos de orientação a objeto e a forma como tais conceitos se relacionam entre si.

A.1.1 Literais

A.1.1.1 Valor Literal

Um valor literal é uma seqüência de caracteres sem semântica definida. Um valor literal pode existir como:

1. parâmetro real em comandos de invocação;
2. origem da atribuição em comandos de atribuição de variáveis enumeradas;
3. segundo elemento em comandos de comparação de valor em variáveis enumeradas.

A Figura A.1 mostra as situações possíveis para o uso de valores literais.

```
class Principal
{
   constructor iniciar( ) exports all {
   ...
   Integer massa = Integer.make( 70 );// 70 é um valor literal
   }
   ...
}
...
class Boolean {
   enum { true, false } value = false;
   ...
   method void flip( ) exports all
   {
       if ( value == true )
           value = false;
       else
           value = true;
   }
   ...
}
```

Figura A.1: Código fonte em Aram mostrando os possíveis uso de um valor literal

A.1.1.2 Referência a Literal

Uma meta-classe que representa uma **referência a literal** se relaciona através de associações com outros componentes do metamodelo da VIRTUOSI nas seguintes situações:

1. como parâmetro formal;
2. como parâmetro real;
3. como origem de atribuição em um comando de atribuição de variável enumerada;
4. com uma das possibilidades para o segundo elemento de uma comparação de valor de variável enumerada.

Deve-se notar que uma referência a literal no papel de parâmetro real não pode sofrer atribuição – uma vez que não existe uma meta-classe que represente o comando de atribuição para referência a literal. Também não é possível declarar uma variável local do tipo referência a literal, visto que não existe uma meta-classe que represente o comando para declarar variáveis do tipo referência a literal.

A Figura A.2 mostra o relacionamento das meta-classes que representam valores literais e referências a literal com outros componentes do metamodelo da VIRTUOSI.

O uso de um valor literal e de uma referência a literal é detalhado na Seção A.1.7, especificamente na discussão sobre comandos de declaração de variáveis.

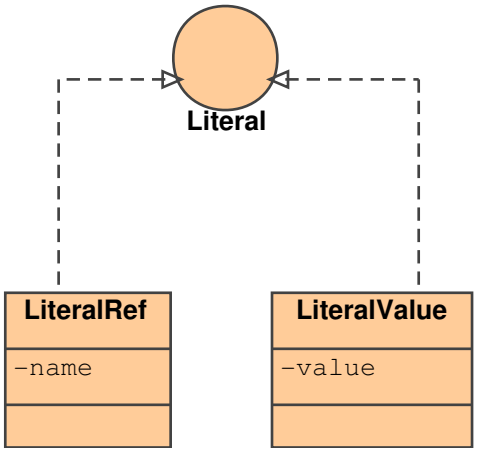


Figura A.2: Relacionamento das meta-classes que representam valores literais e referências a literal com outros componentes do metamodelo da VIRTUOSI

A.1.2 Bloco de Dados e Índice

A.1.2.1 Referência a Bloco de dados

Uma **referência a bloco de dados** consiste em uma referência para uma seqüência contígua de dados binários em memória. Esse tipo de referência é geralmente utilizado para a construção de classes pré-definidas (*Integer*, *Real*, *Boolean*, *Character* e *String*) utilizadas para compor outras classes. A referência a bloco de dados é discutida em detalhe na Seção A.1.4.

A meta-classe que representa uma referência a bloco de dados se relaciona através de associações com outros componentes do metamodelo da VIRTUOSI nas seguintes situações:

1. como parâmetro formal;
2. como parâmetro real;
3. como atributo de uma classe;
4. como atributo em um acesso a atributo bloco de dados (Este componente é explicado na Seção A.1.7, especificamente na discussão sobre o comando para atribuição de referência a objeto);
5. na comparação entre duas referência a bloco de dados;
6. na comparação entre uma referência a bloco de dados e uma referência nula;
7. como alvo da atribuição em um comando de atribuição de referência a bloco de dados;
8. como variável local;

9. como alvo da atribuição em um comando de atribuição de referência nula a bloco de dados;

10. como alvo de uma referência a índice;

11. como alvo dos muitos comandos de sistema para manipulação de blocos de dados de classes pré-definidas. Os comandos de sistema são detalhados na Seção A.1.8.1;

12. como alvo dos muitos testáveis especiais para manipulação de blocos de dados de classes pré-definidas.

A Figura A.3 mostra o relacionamento da meta-classe que representa a referência a bloco de dado com outros componentes do metamodelo da Virtuosi.

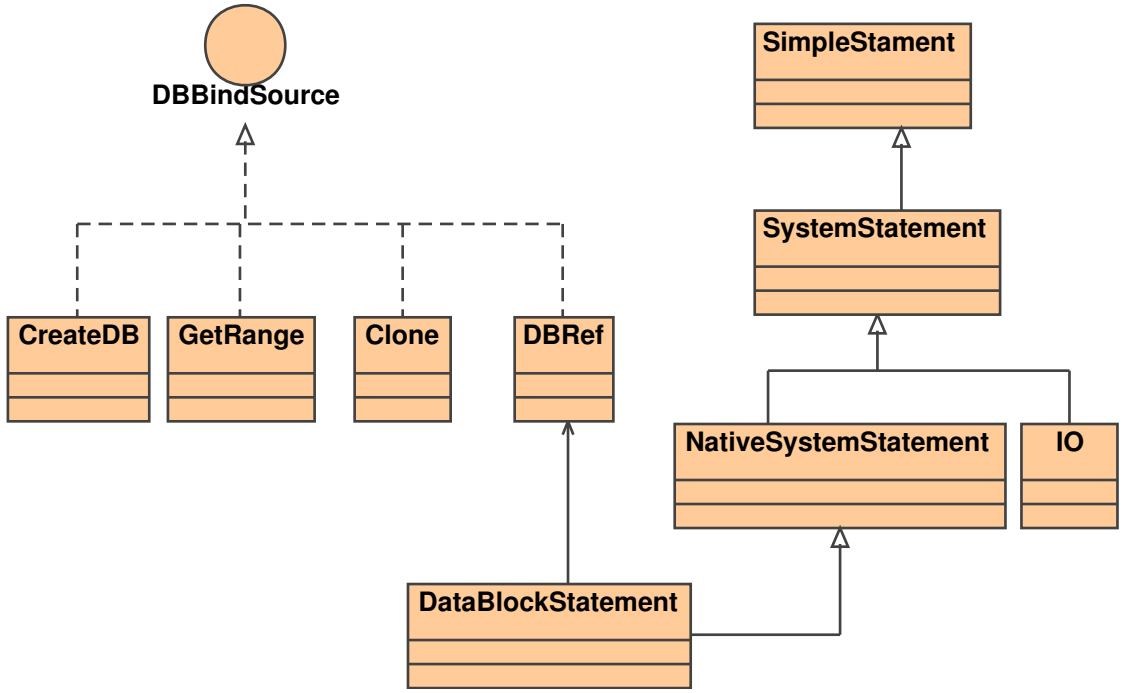


Figura A.3: Relacionamento da meta-classe que representa a referência a bloco de dado com outros componentes do metamodelo da Virtuosi.

Devido ao grande número de situações em que uma referência a bloco de dados existe, é preferível analisa-la separadamente em cada caso durante esta Seção.

A.1.2.2 Referência a Índice

Para a manipulação de um bloco de dados existe uma referência especial chamada **referência a índice**, ou simplesmente **índice**. Um índice pode ser obtido a partir de uma referência a bloco de dados. Quando isso acontece o índice passa a estar associado ao bloco de dados, ou seja, o índice passa a apontar para uma posição da seqüência contígua de

dados binários e, a partir desse momento, seu valor pode ser entendido como um número inteiro limitado entre zero e o tamanho do bloco de dados menos um. Um índice possui comandos tais como ir para frente, ir para trás, ir para determinada posição, etc. Também possui métodos de adição, subtração, multiplicação, etc. Estes métodos permitem o índice navegar na seqüência de dados binários.

O metamodelo da VIRTUOSI formaliza a relação entre um índice e uma referência a bloco de dados, conforme mostra a Figura A.4.

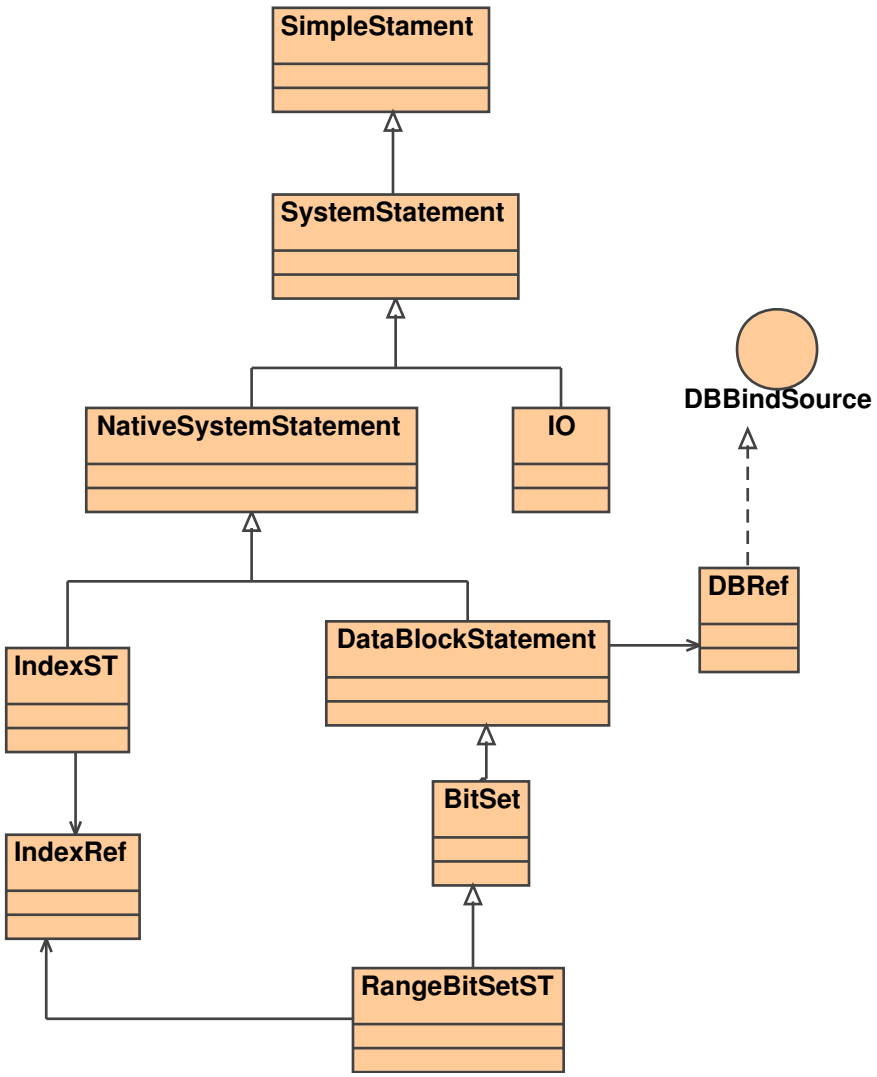


Figura A.4: Relação entre um índice e uma referência a bloco de dados

A meta-classe que representa uma referência a índice se relaciona através de associações com outros componentes do metamodelo da VIRTUOSI nas seguintes situações:

1. como parâmetro formal;
2. como parâmetro real;
3. como índice de uma referência a bloco de dados;

4. na comparação de referência a índice;
5. na comparação de referência a índice nula;
6. variável local;
7. como alvo da atribuição em um comando de atribuição de referência a índice;
8. com uma das possibilidades para a origem da atribuição em um comando de atribuição de referência a índice;
9. como parâmetro em alguns comandos de sistema para manipulação de blocos de dados de classes pré-definidas;
10. como parâmetro em alguns testáveis especiais para manipulação de blocos de dados de classes pré-definidas.

A Figura A.5 mostra o relacionamento da meta-classe que representa a referência a índice com outros componentes do metamodelo da VIRTUOSI, excetuando-se os comandos e testáveis especiais.

Devido ao grande número de situações em que uma referência a índice existe, é preferível analisa-la separadamente em cada caso durante esta Seção.

A.1.3 Classes

A meta-classe que representa uma **classe** se relaciona através de associações com outros componentes do metamodelo da VIRTUOSI nas seguintes situações:

1. como possuidora de atributos referência a objeto em um relacionamento associação;
2. como possuidora de atributos referência a objeto em um relacionamento composição exclusiva;
3. como possuidora de atributos referência a bloco de dados em um relacionamento composição exclusiva;
4. como possuidora de atributos de variáveis enumeradas em um relacionamento composição exclusiva;
5. como tipo de uma referência a objeto;
6. como possuidora de invocáveis;
7. como cliente da lista de exportação de um invocável;

A Figura A.6 mostra o relacionamento da meta-classe que representa uma classe de aplicação com outros componentes do metamodelo da VIRTUOSI, excetuando-se os relacionamentos com meta-classes descendentes.

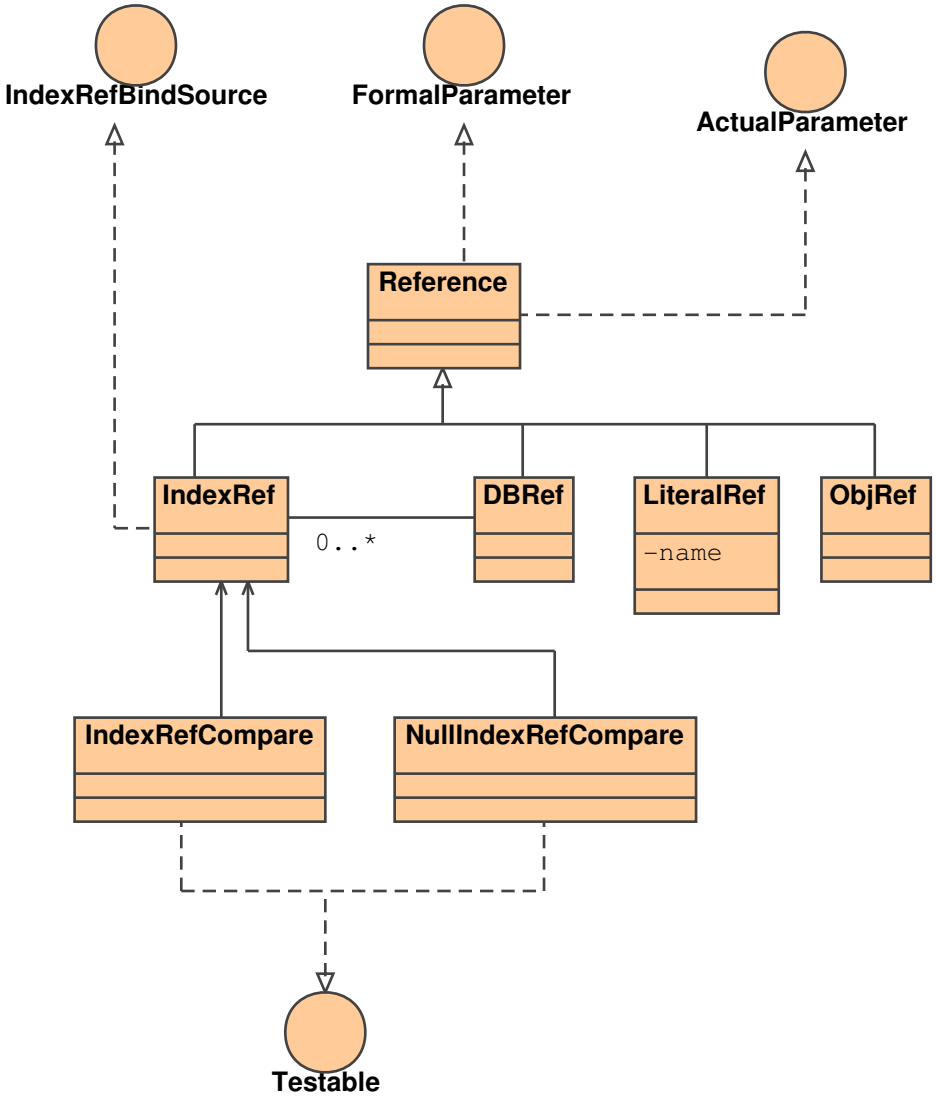


Figura A.5: Relacionamento da meta-classe que representa a referência a índice com outros componentes do metamodelo da VIRTUOSI

A.1.3.1 Herança

Duas classes podem estabelecer um relacionamento de herança entre si, tal que os invocáveis da classe herdeira podem acessar tanto o estado quanto o comportamento (métodos e ações) definidos pela classe ancestral, sem qualquer restrição. Em outras palavras, todas as definições de estado e comportamento existentes na classe ancestral são válidas para a classe herdeira. Segundo o metamodelo da VIRTUOSI, uma classe pode ter apenas uma classe ancestral direta[1], mas pode ter muitas classes herdeiras recursivamente. Entretanto, uma classe não pode ser direta ou indiretamente ancestral de si própria. Assim, um conjunto de classes pode ser organizado como um grafo acíclico dirigido no

[1]Essa propriedade é normalmente denominada herança simples, em contra-partida à herança múltipla, quando uma classe pode ter muitas ancestrais diretas.

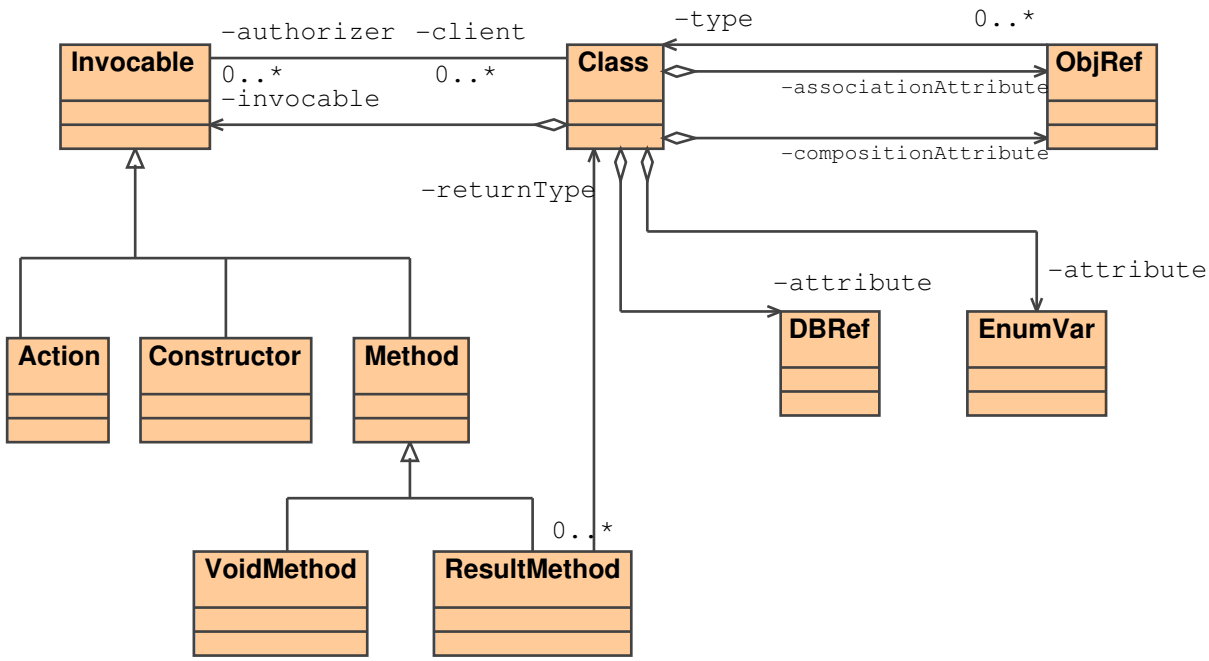


Figura A.6: Relacionamento da meta-classe que representa uma classe de aplicação com outros componentes do metamodelo da VIRTUOSI

qual a propriedade de herança é transitiva, isto é, uma classe herdeira assimila as definições de estado e de métodos de ancestrais diretas ou indiretas.

Uma conseqüência da transitividade da propriedade de herança é que uma referência de uma certa classe pode ter como alvo instâncias de distintas classes, desde que estas sejam herdeiras (diretas ou indiretas) da classe que define o tipo da referência, caracterizando assim a propriedade de polimorfismo.

O metamodelo da VIRTUOSI formaliza a relação de herança entre as classes, conforme mostrado na Figura A.7.

Existem dois tipos de classe, a **classe de aplicação** e a **classe raiz**. Toda classe da aplicação possui uma classe ancestral, sendo que esta pode ser uma outra classe de aplicação ou a classe raiz.

A.1.3.2 Classe Raiz

A classe raiz representa a classe ancestral – direta ou indireta – de todas as classes de aplicação, por este motivo não possui ancestral. Ela é única em toda a hierarquia de classes.

A.1.4 Atributos

O ambiente VIRTUOSI disponibiliza uma biblioteca de classes pré-definidas (*Integer*, *Real*, *Boolean*, *Character* e *String*) utilizadas para compor novas classes, as classes de aplicação.

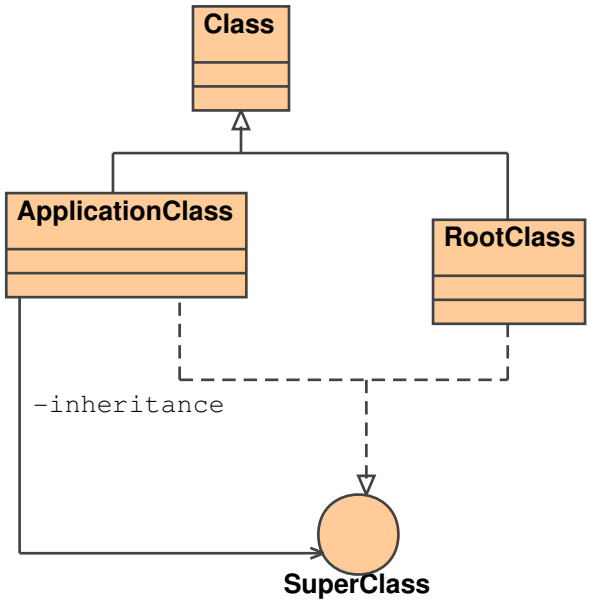


Figura A.7: Relação de herança entre classes segundo o metamodelo da VIRTUOSI

Os atributos de uma classe segundo o metamodelo da VIRTUOSI podem ser de três tipos, a saber:

- referência a objeto;
- referência a bloco de dados;
- variável enumerada.

O metadomelo da VIRTUOSI formaliza os três tipos de atributo que uma classe conforme mostra a Figura A.8.

A.1.4.1 Referência a Objeto

A meta-classe que representa uma **referência a objeto** se relaciona através de associações com outros componentes do metamodelo da VIRTUOSI nas seguintes situações:

1. como parâmetro formal;
2. como parâmetro real;
3. como atributo de uma classe por associação;
4. como atributo de uma classe por composição;
5. como sendo um tipo de classe;
6. como objeto em um acesso a atributo variável enumerada;

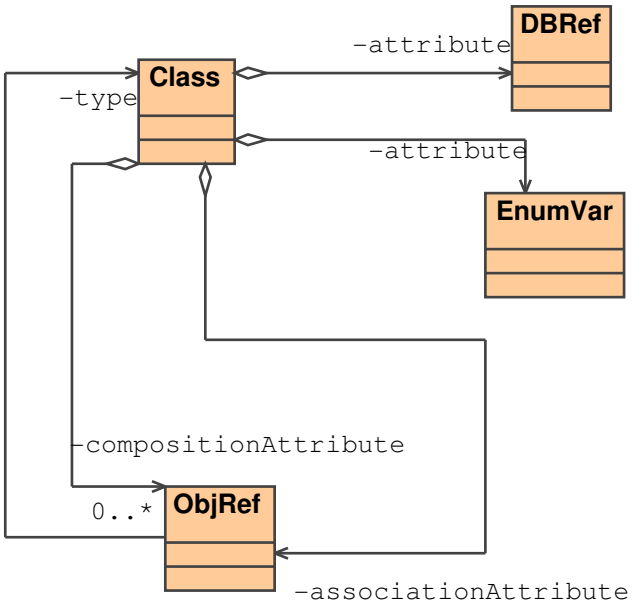


Figura A.8: Os três tipos de atributos possíveis em uma classe segundo o metamodelo da VIRTUOSI

7. como objeto em um acesso a atributo objeto (Este componente é explicado na Seção A.1.7, especificamente na discussão sobre o comando para atribuição de referência a objeto);

8. como atributo em um acesso a atributo objeto (Este componente é explicado na Seção A.1.7, especificamente na discussão sobre o comando para atribuição de referência a objeto);

9. como objeto em um acesso a atributo bloco de dados (Este componente é explicado na Seção A.1.7, especificamente na discussão sobre o comando para atribuição de referência a bloco de dados);

10. na comparação entre duas referência a objeto;

11. na comparação entre uma referência a objeto e uma referência nula;

12. como alvo da atribuição em um comando de atribuição de referência a objeto;

13. como alvo da atribuição em um comando de atribuição de referência nula a objeto;

14. com uma das possibilidades para a origem da atribuição em um comando de atribuição de referência a objeto;

15. com uma das possibilidades para o testável associado a um comando de desvio condicional (Tanto o componente testável quanto o comando de desvio condicional são explicados na Seção A.1.7, especificamente na discussão sobre o comando desvio condicional);

16. como referência retornada por um invocável;
17. como variável local;
18. como alvo de invocação de método;
19. como alvo de invocação de uma ação.

Devido ao grande número de situações em que uma referência a objeto existe, é preferível analisa-la separadamente em cada caso durante esta Seção.

A Figura A.9 mostra o relacionamento da meta-classe que representa a referência a objeto com outros componentes do metamodelo da VIRTUOSI.

A Figura A.10 mostra uma classe implementada em Aram – segundo a definição do metamodelo da VIRTUOSI – com três atributos do tipo referência a objeto. O primeiro e o segundo atributo são instâncias de uma classe pré-definida (*String*). O terceiro atributo é instância de uma classe de aplicação, no caso `Pessoa`.

Um atributo do tipo referência a objeto pode estar associado a classe por **composição** ou **associação**.

Composição: Observando a Figura A.10 nota-se que o primeiro atributo – uma *String* de nome *name* – possui como parte de sua declaração a palavra *composition*. A existência da palavra *composition* indica um relacionamento de **composição exclusiva** entre uma classe e um atributo. A Figura A.11 mostra um código fonte com uma relação de composição exclusiva entre classe e atributo e ilustra a semântica correspondente utilizando os objetos envolvidos. Em uma relação de composição exclusiva o objeto representado pelo atributo está contido no objeto representado pela classe. Como conseqüência do relacionamento de composição exclusiva um objeto contido somente pode ser referenciado por ele próprio, pelo seu contentor direto ou por outro objeto que seja contido no mesmo objeto contentor.

Associação: Observando-se novamente a Figura A.10 nota-se que o segundo atributo – uma *String* de nome *address* – possui como parte de sua declaração a palavra *association*. A palavra *association* indica um relacionamento de **associação** entre uma classe e um atributo. A Figura A.12 mostra um código fonte com uma relação de associação entre classe e atributo e ilustra a semântica correspondente utilizando os objetos envolvidos. Em uma relação de associação o objeto representado pelo atributo não faz parte do objeto representado pela classe, simplesmente está associado a ele. Como conseqüência do relacionamento de associação um objeto associado pode ser referenciado por qualquer outro objeto.

A Figura A.13 mostra como o metamodelo da VIRTUOSI formaliza as relações de composição e associação entre uma classe e seus atributos.

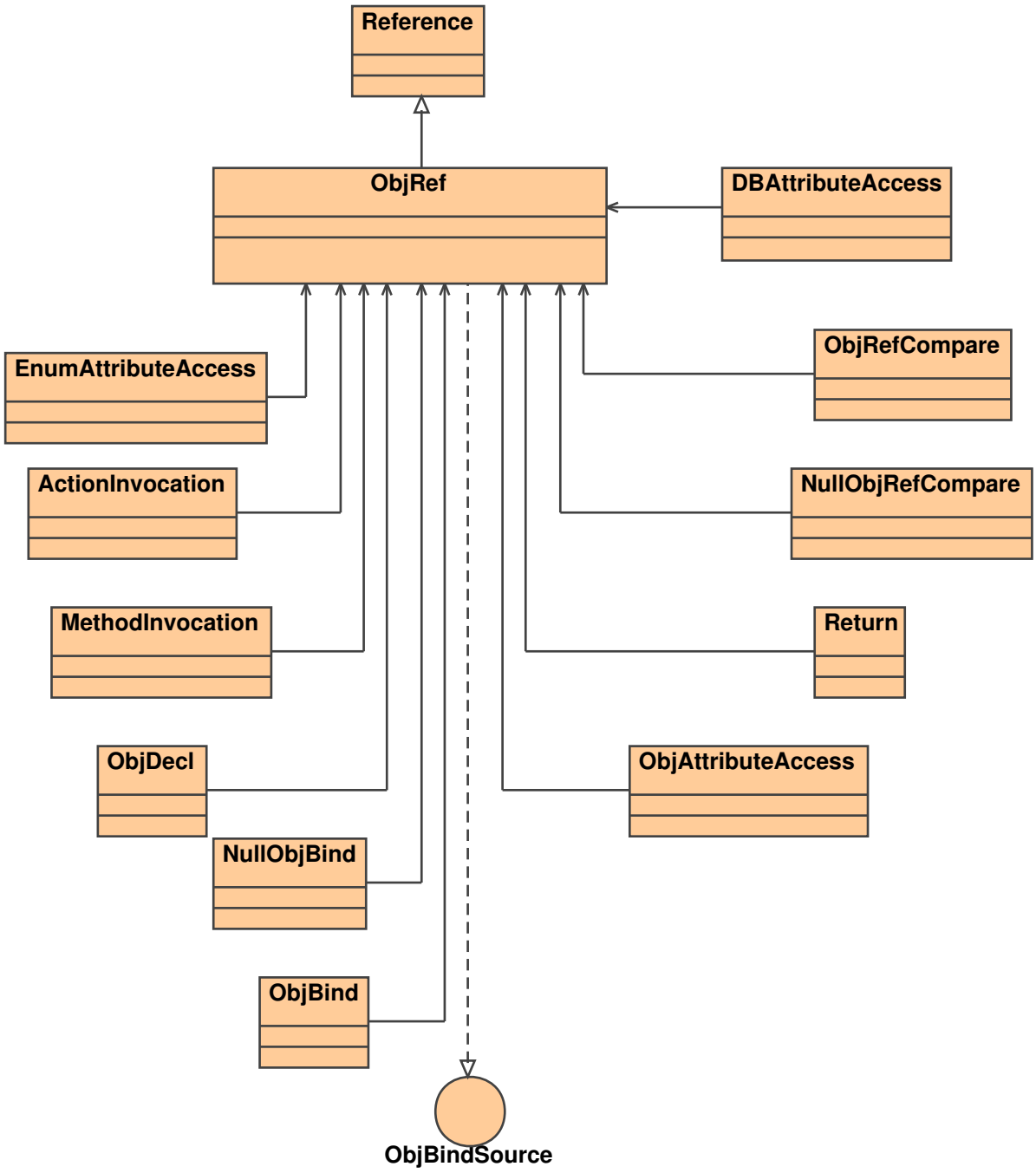


Figura A.9: Relacionamento da meta-classe que representa a referência a objeto com outros componentes do metamodelo da VIRTUOSI

A.1.4.2 Referência a Bloco de Dados

Segundo o metamodelo da VIRTUOSI, um programador tem a possibilidade de criar novas classes que não dependam de nenhuma outra classe pré-existente. Para tanto, um atributo pode referenciar um bloco de dados. Esse tipo de referência é chamada referência a bloco de dados. Uma referência a bloco de dados consiste em uma referência para uma

```
class Pessoa {
   composition String nome;     //classe pré-definida
   association String endereco;  //classe pré-definida
   association Person esposa;     //classe de aplicação
}
```

Figura A.10: Atributos em uma classe segundo o metamodelo da VIRTUOSI

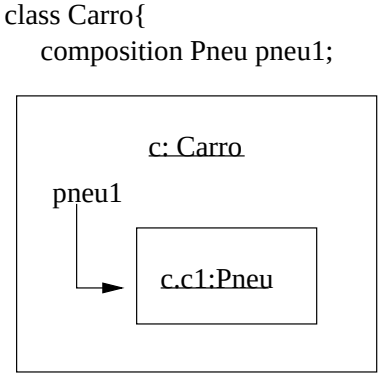


Figura A.11: Código fonte em Aram e diagrama de objeto de uma relação de composição entre uma classe e um atributo

seqüência contígua de dados binários em memória. Um atributo do tipo referência a bloco de dados obrigatoriamente tem uma relação de composição exclusiva com a classe que o possui. Esse tipo de referência é geralmente utilizado para a construção de classes pré-definidas. Cada classe pré-definida é responsável por dar o significado de sua seqüência de dados binários através de suas operações. Por exemplo, um objeto do tipo básico *Integer* pode armazenar um valor inteiro utilizando um bloco de dados de qualquer tamanho, nesse caso uma operação para adicionar um outro valor inteiro (armazenado em outro objeto do tipo *Integer* também utilizando um bloco de dados) ao valor inteiro deste objeto, deve conhecer a convenção utilizada na representação binária de ambos as seqüências. A Figura A.14 mostra o código fonte de uma classe que possui um atributo do tipo bloco de dado e uma ilustração de uma instância desta classe contendo o bloco de dados.

A.1.4.3 Variável Enumerada

Uma classe implementada segundo o metamodelo da VIRTUOSI, pode ainda, ter um atributo do tipo variável enumerada. Um atributo do tipo variável enumerada possui um conjunto de valores possíveis definidos na construção da classe. Os valores possíveis de uma variável enumerada não são objetos de nenhuma outra classe, são simples valores literais. Esse conjunto de valores possíveis de uma variável enumerada chama-se **enumerado**. Um atributo do tipo variável enumerada recebe um valor inicial durante sua declaração. A Figura A.15 mostra o código fonte de uma classe que possui um atributo

```
class Pessoa{
    composition String nome;
    enum {masculino, feminino} sexo = masculino;

    constructor make(String pNome, literal pSexo) exports{all}
    {
        nome = pNome;
        sexo = pSexo;
    }
    ...
} class Aviao{
    association Pessoa passageiro;

    constructor make() exports{all}
    {
    }

    method reservar(Pessoa pPessoa) exports{all}
    {
        passageiro = pPessoa;
    }
    ...
}
```

Figura A.12: Código fonte em Aram de uma relação de associação entre uma classe e um atributo

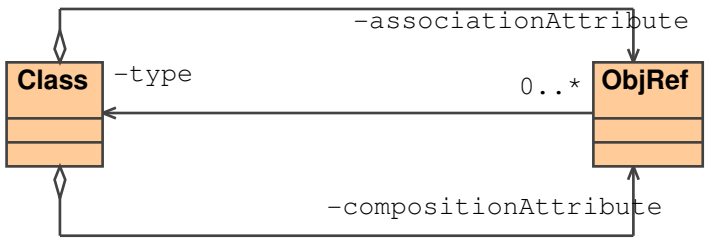


Figura A.13: As duas maneiras em que uma referência a objeto representa o papel de atributo segundo o metamodelo da VIRTUOSI

variável enumerada.

A.1.5 Referências

Existem quatro tipos de referência suportadas pelo metamodelo da VIRTUOSI, a saber:

- referência a objeto;

```
class Inteiro{
    datablock valor;

    constructor make(literal pValor) exports{all}
    {
        valor = datablock.make(32);
        valor.storeInteger(pValor);
    }
    constructor make(Inteiro pValor) exports{all}
    {
        this.set(pValor);
    }
    method void set(Inteiro i) exports{all}
    {
        datablock k = i.valor;
        valor = k.clone();
    }
    ...
}
```

Figura A.14: Um exemplo de atributo do tipo bloco de dados e uma representação de um objeto correspondente – código fonte em Aram

```
class Carro {
    enum { true, false } conversivel = false;
    constructor make() exports{all}
    {
    }
    ...
}
```

Figura A.15: Um exemplo de atributo do tipo enumerado – código fonte em Aram

- referência bloco de dados;
- referência a índice;
- referência a literal.

A Figura A.16 mostra o relacionamento de herança entre as referências na VIRTUOSI e mostra também que qualquer referência pode ser passada como parâmetro real e fazer parte dos parâmetros formais de um invocável. Deve-se notar que a meta-classe referência é abstrata, e é concretizada pelos quatro tipos de referência.

Observando-se a Figura A.16 nota-se que existe uma meta-classe chamada `This` herdeira da meta-classe que representa uma referência a objeto. Essa meta-classe representa uma referência para o objeto corrente durante a interpretação de um método. Um

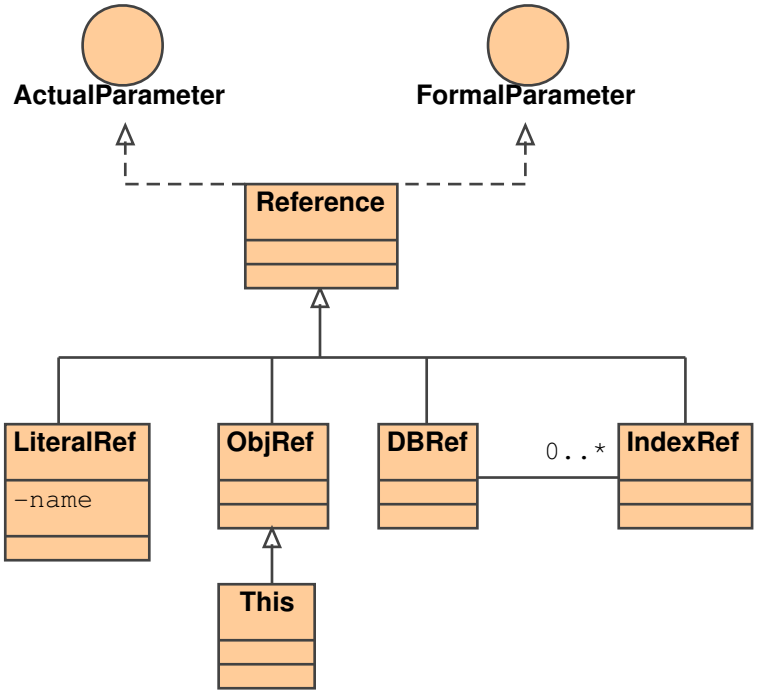


Figura A.16: Relacionamento entre as meta-classe que definem os tipos de referência na VIRTUOSI

método sempre é invocado através de uma referência a objeto sobre o objeto o qual ela aponta. Uma referência do tipo `This` é utilizada quando dentro de um método deseja-se invocar um método da própria classe e sobre o próprio objeto corrente.

A.1.6 Invocáveis

Segundo o metamodelo da VIRTUOSI, uma operação de uma classe pode ser implementada de duas maneiras, a saber:

- como um **método**;
- como uma **ação**;

Essas duas implementações descrevem o conjunto dos serviços que uma classe disponibiliza. Além das operações, uma classe precisa implementar um método especial utilizado para criar novas instâncias da classe, esse tipo de método especial é chamado de **construtor**.

O metamodelo da VIRTUOSI formaliza a relação entre uma classe e as possíveis implementações de suas operações e construtores, conforme mostra a Figura A.17.

Um método, um construtor ou uma ação podem sofrer invocação e, por isso, o metamodelo da VIRTUOSI os formaliza como **invocáveis**[2].

[2]Do inglês ***invocable*** (o termo **invocável** ainda não está registrado nos dicionários da língua portuguesa).

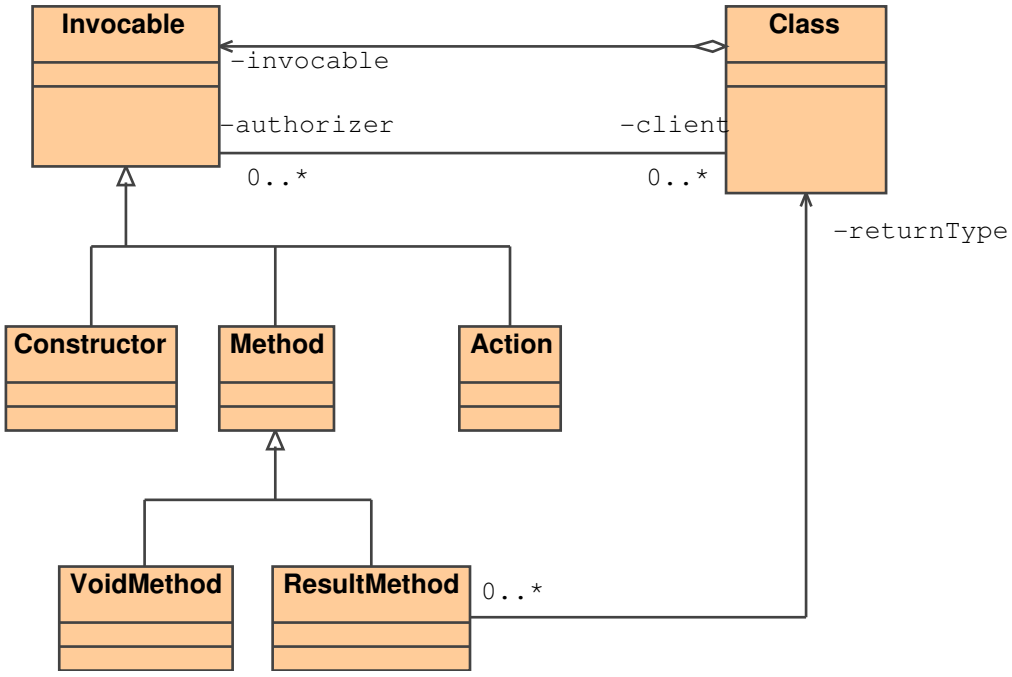


Figura A.17: Relação entre uma classe e as possíveis implementações de suas operações

A Figura A.18 mostra o relacionamento da meta-classe Invocável com outros componentes do metamodelo da VIRTUOSI.

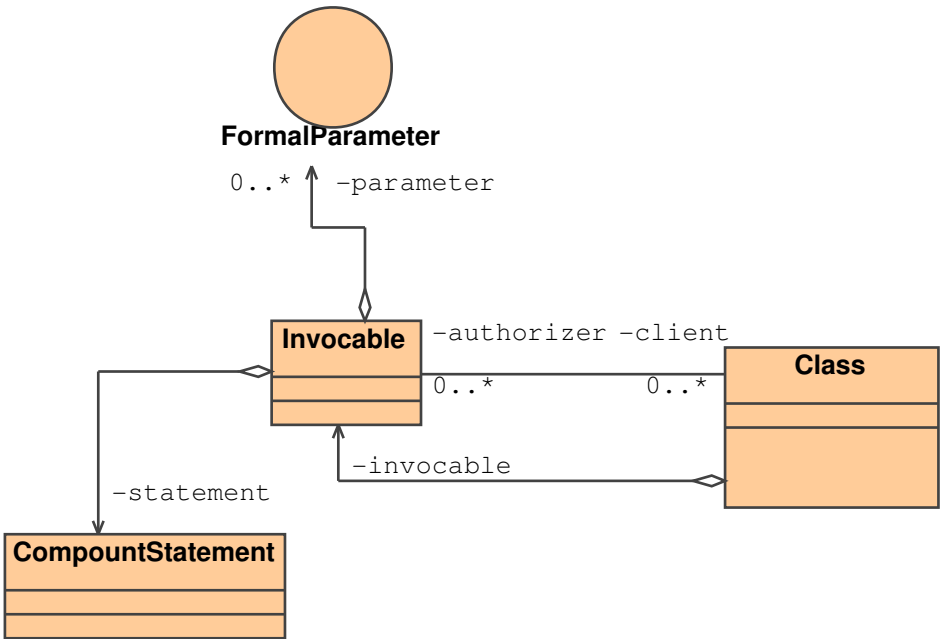


Figura A.18: Relacionamento da meta-classe Invocável com outros componentes do metamodelo da VIRTUOSI

A meta-classe que representa um invocável – independente do seu tipo (construtor, método ou ação) – se relaciona através de associações com outros componentes do metamodelo da VIRTUOSI nas seguintes situações:

1. como possuidora de parâmetros formais;
2. como possuidora de comandos;
3. como pertencente a uma classe;
4. como fornecedora para meta-classes que tem autorização para invocá-la;

Parâmetros:

Um invocável pode receber parâmetros. Por isso um invocável define uma seqüência de parâmetros que deve ser provida durante sua invocação (ou chamada).

Um parâmetro pode ser visto sob duas perspectivas diferentes. A primeira ocorre durante a construção de um invocável onde o parâmetro tem o papel de **parâmetro formal**[3], ou seja, ele define o nome, o tipo e a posição que determinado parâmetro possuirá na seqüência dos parâmetros formais daquela invocação. A segunda ocorre no momento da chamada do invocável, onde um parâmetro é uma referência a um objeto já existente, tendo assim, o papel de **parâmetro real**[4].

O conjunto de parâmetros formais de um invocável pode ser vazio ou composto de referências. Qualquer tipo de referência pode ser um parâmetro formal, inclusive uma referência a literal[5] utilizada para receber os parâmetros reais do tipo valor literal.

A Figura A.19 mostra um exemplo de uma classe de aplicação `Taxi` com dois métodos, um com uma lista de parâmetros vazia (`sairPassageiro`) e outro com uma lista contendo um parâmetro formal do tipo referência a objeto (`entrarPassageiro`). A Figura mostra também a classe de aplicação `Principal`, que invoca os dois métodos da classe de aplicação `Taxi`.

O metamodelo da VIRTUOSI formaliza a relação entre um invocável e seus parâmetros, conforme mostra a Figura A.20.

Lista de Exportação: Um invocável define explicitamente quais as outras classes cujos invocáveis podem realizar invocações sobre o invocável em questão. Para tanto, um invocável possui uma **lista de exportação**, ou seja, uma lista das classes cujos invocáveis podem invocá-lo.

Alguns exemplos do uso da lista de exportação podem ser observados na Figura A.21. A Figura mostra três casos distintos: o método *exportedToAllMethod* – exportado

[3]Do inglês ***formal parameter***.

[4]Do inglês ***actual parameter***.

[5]Embora um parâmetro seja utilizado pelo invocável da mesma forma que uma variável local, no caso de parâmetros do tipo referência a literal, o parâmetro não pode sofrer atribuição, visto que, não existe um comando de atribuição para referência a literal. Portanto, uma referência a literal sempre tem seu valor literal atribuído por um comando de invocação de invocável

```
class Principal
{
   constructor iniciar() exports { all }
   {
       Taxi corsa = Taxi.instanciar();
       ...
       Boolean entrou = corsa.entrarPassageiro(andrea);
       ...
       corsa.sairPassageiro();
       ...
   }
   ...
}
class Taxi {
   ...

   // método sem parâmetros
   method void sairPassageiro( ) exports { Principal }
   {
       ...
   }
   // método com parâmetros
   method Boolean entrarPassageiro( Pessoa p ) exports { Principal }
   {
       ...
   }
}
```

Figura A.19: Código fonte em Aram contendo um método sem parâmetros e um método com parâmetros

para toda e qualquer classe –, o método *nonExportedMethod* – não exportado para nenhuma classe – e o método *exportedToBandC* exportado para as classes B e C. Deve-se observar que nos dois primeiros casos foram utilizadas palavras reservadas da linguagem ao invés de uma lista de nomes de outras classes. Existem duas palavras reservadas que podem ser utilizadas no lugar de uma lista de exportação: *all* e *none*. A palavra *all* implica que o invocável em questão pode ser invocado a partir de invocáveis de toda e qualquer classe, enquanto que a palavra *none* implica que o invocável em questão somente pode ser invocado pelos invocáveis pertencentes a mesma classe que o possui. O uso da palavra *none* permite que invocáveis sejam acessados sem restrição por qualquer invocável da própria classe.

O metamodelo da VIRTUOSI formaliza a relação entre um invocável e sua lista de exportação, conforme mostra a Figura A.22.

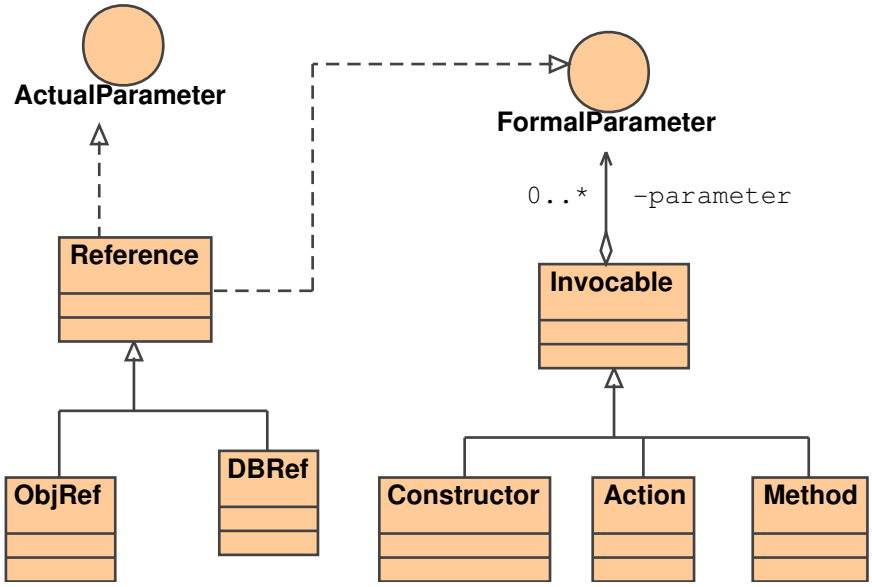


Figura A.20: Relação de um Invocável e seus parâmetros

```
class A {
   ...

   // método exportado para toda e qualquer classe
   method void exportedToAllMethod() exports all
   {
       ...
   }

   // método não exportado para nenhuma classe
   method void nonExportedMethod() exports none
   {
       ...
   }
   method void exportedToBandC() exports { B, C }
}
```

Figura A.21: Métodos com diferentes listas de exportação – Código fonte em Aram

Construtor: Um construtor não faz parte das operações definidas por um TAD pois não interfere no comportamento dos objetos representados pelo TAD. Porém, têm fundamental importância na implementação de uma classe, visto que, um objeto sempre é criado através de sua interpretação. O retorno da interpretação de um construtor é sempre um novo objeto, uma nova instância de uma classe.

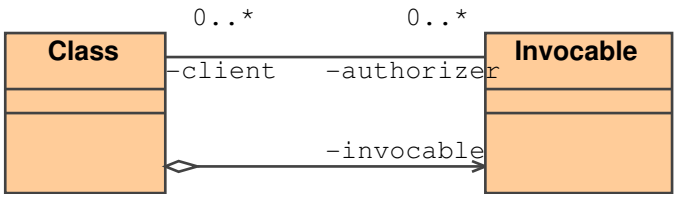


Figura A.22: Relação de um Invocável sua lista de exportação

Método: Um método é a maneira mais comum de implementar uma operação definida por um TAD. Existem dois tipos de métodos, a saber: **método sem retorno** – muitas vezes chamado de procedimento – e **método com retorno** – muitas vezes chamado função – conforme formalizado pelo metamodelo da VIRTUOSI e mostrado na Figura A.23.

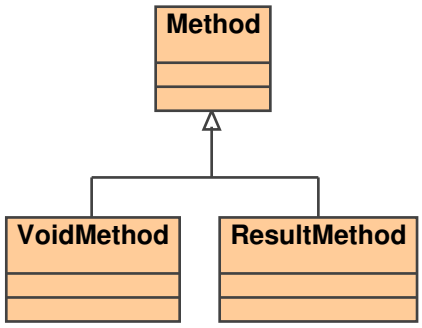


Figura A.23: Métodos com retorno e métodos sem retorno

A diferenciação entre métodos com e sem retorno se dá em parte pelo uso da palavra que fica entre a palavra *method* e o nome do método. Caso essa palavra seja *void* trata-se de um método sem retorno. Caso a palavra seja o nome de uma classe trata-se de um método com retorno. Outra diferença consiste no fato de um método com retorno sempre possuir ao menos um comando retorno. A Figura A.24 mostra um exemplo de código fonte em Aram contendo um método sem retorno e um método com retorno. O comando retorno é um comando simples e é discutido na Seção A.1.7.

Ação: A segunda maneira de implementar uma operação definida por um TAD é através de uma ação. Uma ação pode ser vista como uma operação cujo retorno é utilizado para a tomada de decisão referente à um comando de desvio. Em outras palavras, o retorno de uma ação permite o comando de desvio decidir qual dentre duas seqüências de comandos deve ser interpretada. Uma ação não retorna uma referência a objeto. Diferente de um método com retorno ou um construtor – onde uma referência é retornada – o retorno de uma ação é um comando simples chamado: **comando resultado de teste**. Tanto o comando de desvio quanto o comando resultado de teste são detalhados na Seção A.1.7.7.

```
class Taxi {
   ...

   // método sem retorno
   method void sairPassageiro( ) exports { Principal }
   {
       ...
   }
   // método com retorno
   method Boolean entrarPassageiro( Pessoa p ) exports { Principal }
   {
       ...
       return resultado;
   }
}
```

Figura A.24: Diferenciação entre métodos com retorno e métodos sem retorno

A Figura A.25 mostra um exemplo de código fonte com uma declaração de uma ação, segundo o metamodelo da VIRTUOSI.

```
class Pessoa {
   ...

   // ação
   action casado() exports all
   {
       ...
   }
}
```

Figura A.25: Código fonte em Aram contendo uma declaração de uma ação

A.1.7 Comandos

No ambiente VIRTUOSI, toda computação é realizada através da interpretação dos comandos que compõem um invocável. Esses comandos podem ser invocações de outros invocáveis, comandos responsáveis por controlar o fluxo da interpretação, comandos para manipular referências a objetos ou ainda comandos de sistema para a manipulação de referência a bloco de dados e manipulação de referência a índice. Esses comandos são interpretados a partir do momento que um invocável é invocado.

A.1.7.1 Composição de Comandos

Um invocável define uma seqüência de comandos. Comandos podem ser simples ou compostos. Um **comando simples**[6] pode ser uma atribuição, uma invocação de operação, um desvio condicional, conforme detalhado no restante dessa Seção. Um **comando composto**[7] é uma seqüência de comandos simples ou compostos, recursivamente.

Uma seqüência de comandos, simples ou compostos, pode ser agrupada em um comando composto. Esse relacionamento é mostrado na Figura A.26.

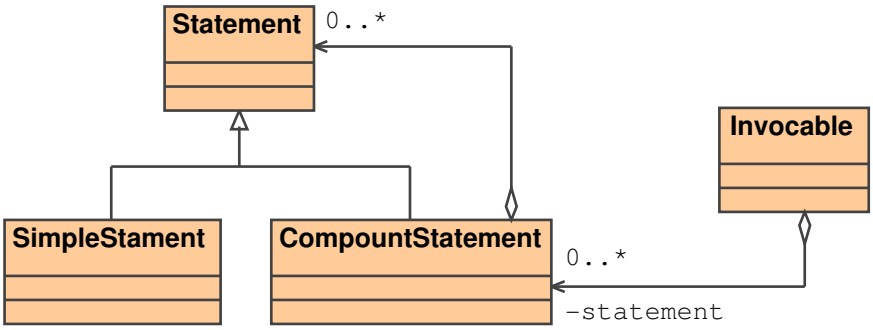


Figura A.26: Relacionamento de herança entre as meta-classes comando, comando simples e comando composto

A.1.7.2 Declaração de Variáveis

Para se invocar um invocável é preciso possuir uma referência para um objeto da classe que o define(com exceção de construtores que são invocados a partir do nome da classe). Segundo o metamodelo da VIRTUOSI, uma referência existe sob três formas, a saber:

1. atributo;
2. parâmetro;
3. **variável local**.

Ao contrário dos atributos e parâmetros que são definidos durante a construção da classe e seus respectivos invocáveis, uma variável local não existe até que seja declarada. Portanto, existem instruções para a declaração de alguns tipos de referência utilizadas como variáveis locais. A Figura A.27 mostra a hierarquia de classes que determina os comandos simples disponíveis para a declaração de variáveis locais, segundo o metamodelo da VIRTUOSI.

Conforme mostra a Figura A.27, é possível declarar variáveis locais do tipo:

[6] Do inglês ***simple statement***.
[7] Do inglês ***compound statement***.

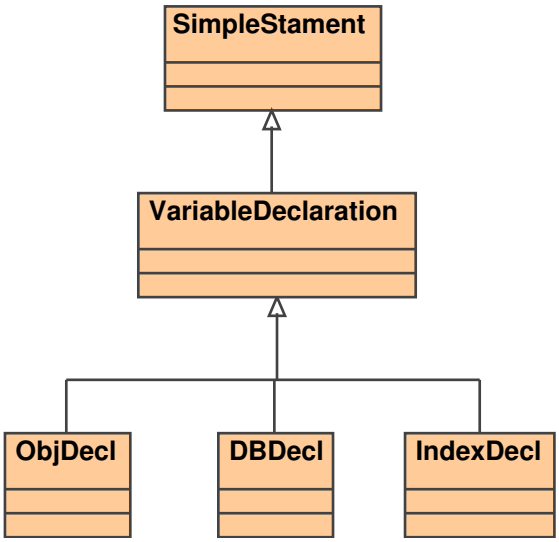


Figura A.27: Tipos de declarações para variáveis locais

- referência a objeto;
- referência a bloco de dados;
- referência a índice.

Observando novamente a Figura A.27 nota-se que não existe declaração de variável local do tipo referência a literal. Isto ocorre porque uma referência a literal somente é permitida como parâmetro de um invocável, ou seja, no momento da invocação o que é passado como parâmetro é um valor literal, e não uma referência a literal. A Figura A.28 mostra uma invocação de um método passando um valor literal e a declaração desse mesmo método contendo um parâmetro formal do tipo referência a literal.

Declaração de Variáveis do Tipo Referência a Objeto: A Figura A.29 mostra a declaração de uma variável local do tipo referência a objeto. Após a interpretação desse comando a referência não possui um alvo, ou seja, não está apontando para algum objeto em memória. Uma referência sem alvo é chamada de referência nula.

Declaração de Variáveis do Tipo Referência a Bloco de Dados: A Figura A.30 mostra a declaração de uma variável local do tipo referência a bloco de dados. Após a interpretação desse comando a referência não possui um alvo, ou seja, não está apontando para alguma seqüência de dados binários em memória. Da mesma forma que uma referência a objeto, uma referência a bloco de dados sem alvo também é uma referência nula.

Declaração de Variáveis do Tipo Referência a Índice: Além da declaração de variável local do tipo referência a objeto e do tipo referência a bloco de dados existe a

```
class Person {
   ...
   constructor create()
   {
       Integer age;
       age = Integer.make(26); // '26' é um valor literal
   }
}
...
class Integer {
   ...
   constructor make ( Literal charSequence){
       ...
   }
}
```

Figura A.28: Invocação de método passando como parâmetro um valor literal e a declaração do método invocado – código fonte em Aram

```
class Person {
   constructor make() exports all
   {
       // declaração de variável local inicialmente nula.
       String name;
       ...
   }
   ...
}
```

Figura A.29: Declaração de uma variável local do tipo referência a objeto – código fonte em Aram

declaração de variável local do tipo referência a índice. A Figura A.31 mostra a declaração de uma variável local do tipo referência a índice. Após a interpretação desse comando a referência não possui um alvo, ou seja, não está apontando alguma seqüência de dados binários em memória, ou seja, é uma referência nula.

A.1.7.3 Atribuição

Uma referência nula não permite uma invocação a partir dela. Isso é verdade tanto para variáveis locais quanto para atributos e parâmetros. Portanto, é preciso fazer com que uma referência aponte para um alvo, ou seja, uma referência a objeto precisa apontar para um objeto em memória, uma referência a bloco de dados precisa apontar para uma seqüência de dados binários em memória e uma referência a índice precisa apontar

```
class Integer {
   method void add( Integer i ) exports all
   {
       // declaração de variável local inicialmente nula.
       datablock d;
       ...
   }
   ...
}
```

Figura A.30: Declaração de uma variável local do tipo referência a bloco de dados – código fonte em Aram

```
class Integer {
   method void add( Integer i ) exports all
   {
       // declaração de variável local inicialmente nula.
       index i;
       ...
   }
   ...
}
```

Figura A.31: Declaração de uma variável local do tipo referência a índice – código fonte em Aram

para uma posição na seqüência de dados binários em memória. Isso ocorre através da interpretação de um comando de atribuição. Esse comando é identificado no código fonte pelo uso do caracter '=' chamado de caracter de atribuição. O que estiver a esquerda do caracter de atribuição é chamado **alvo da atribuição**, e o que estiver a direita do caracter de atribuição é chamado **origem da atribuição**.

Segundo o metamodelo da VIRTUOSI, os comandos de atribuição existentes são os seguintes:

- atribuição de referência a objeto;
- atribuição de referência nula a objeto;
- atribuição de referência a bloco de dados;
- atribuição de referência nula a bloco de dados;
- atribuição de referência a índice;
- atribuição de variável enumerada.

Atribuição de Referência a Objeto: No caso da atribuição de referência a objeto o alvo da atribuição sempre é uma referência a objeto, enquanto que a origem da atribuição pode ser:

- uma referência a objeto;
- um **acesso a atributo objeto**, ou seja, um acesso – somente de leitura – a um atributo do tipo referência a objeto, de outro objeto instância da mesma classe que implementa o invocável onde a atribuição ocorre;
- um comando de invocação de construtor (Este componente é explicado nesta Seção, especificamente na discussão sobre o comando de invocação de construtor);
- um comando de invocação de método com retorno (Este componente é explicado nesta Seção, especificamente na discussão sobre o comando de invocação de método com retorno).

Atribuição de Referência Nula a Objeto: Um comando de atribuição de objeto nulo, faz uma referência a objeto voltar a ser uma referência nula.

A Figura A.32 mostra o relacionamento das meta-classes que representam os dois comandos de atribuição de referência a objeto com outros componentes do metamodelo da VIRTUOSI.

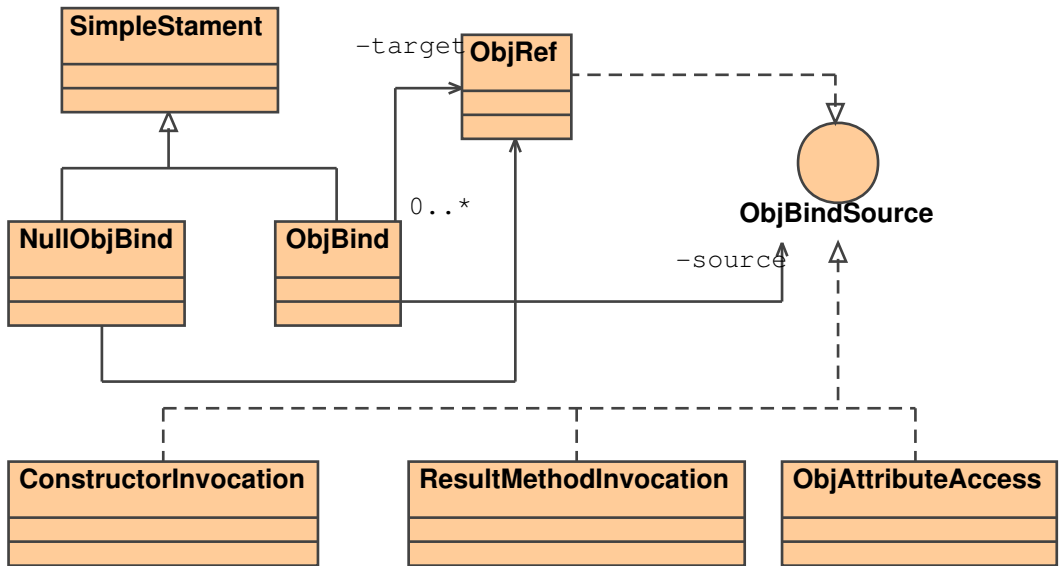


Figura A.32: Relacionamento das meta-classes que representam os comandos de atribuição de referência a objeto com outros componentes do metamodelo da VIRTUOSI

Atribuição de Referência a Bloco de Dados: No caso da atribuição de referência a bloco de dados o alvo da atribuição sempre é uma referência a bloco de dados, enquanto que a origem da atribuição pode ser:

- uma referência a bloco de dados;
- um acesso a atributo bloco de dados, ou seja, um acesso – somente de leitura – a um atributo do tipo referência a bloco de dados, de outro objeto instância da mesma classe que implementa o invocável onde a atribuição ocorre.

Atribuição de Referência Nula a Bloco de Dados: Um comando de atribuição de bloco de dados nulo, faz uma referência a bloco de dados voltar a ser uma referência nula.

A Figura A.33 mostra o relacionamento das meta-classes que representam os dois comandos de atribuição de referência a bloco de dados com outros componentes do metamodelo da VIRTUOSI.

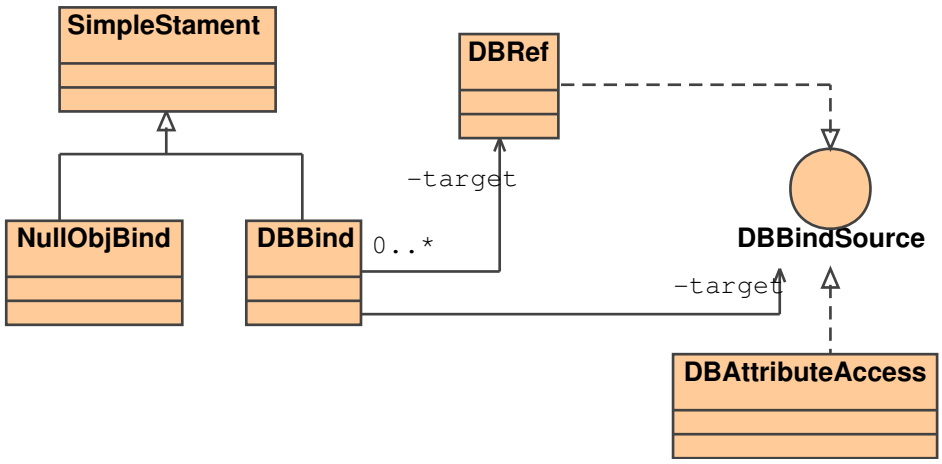


Figura A.33: Relacionamento das meta-classes que representam os dois comandos de atribuição de referência a bloco de dados com outros componentes do metamodelo da VIRTUOSI

Atribuição de Referência a Índice: No caso da atribuição de referência a índice o alvo da atribuição sempre é uma referência a índice, enquanto que a origem da atribuição pode ser:

- uma referência a índice;
- um comando especial para a manipulação de referência a bloco de dados.
- um comando de atribuição de índice nulo, faz uma referência a índice voltar a ser uma referência nula.

A Figura A.34 mostra o relacionamento da meta-classe comando de atribuição de referência a índice com outros componentes do metamodelo da VIRTUOSI.

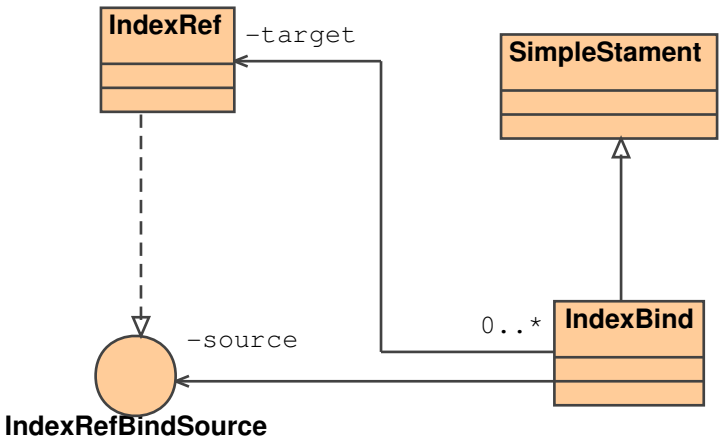


Figura A.34: Relacionamento da meta-classe comando de atribuição de referência a índice com outros componentes do metamodelo da VIRTUOSI

Atribuição de Variável Enumerada: No caso da atribuição de variável enumerada o alvo da atribuição sempre é uma variável enumerada, enquanto que a origem da atribuição pode ser:

- um valor literal;
- uma referência a literal;
- um acesso a atributo variável enumerada, ou seja, ou seja, um acesso – somente de leitura – a um atributo do tipo variável enumerada, de outro objeto instância da mesma classe que implementa o invocável onde a atribuição ocorre;

Um atributo do tipo variável enumerada recebe um valor inicial durante sua declaração e somente é permitido atribuir-lhe um valor literal pertencente ao enumerado definido.

A Figura A.35 mostra o relacionamento da meta-classe comando de atribuição variável enumerada com outros componentes do metamodelo da VIRTUOSI.

A.1.7.4 Retorno de Método

O comando de retorno utilizado por um método é chamado simplesmente de **retorno**, sendo que sempre retorna uma referência a objeto do mesmo tipo definido pelo tipo de retorno do método.

O metamodelo da VIRTUOSI formaliza o relacionamento entre um comando de retorno e uma referência a objeto, conforme mostra a Figura A.36.

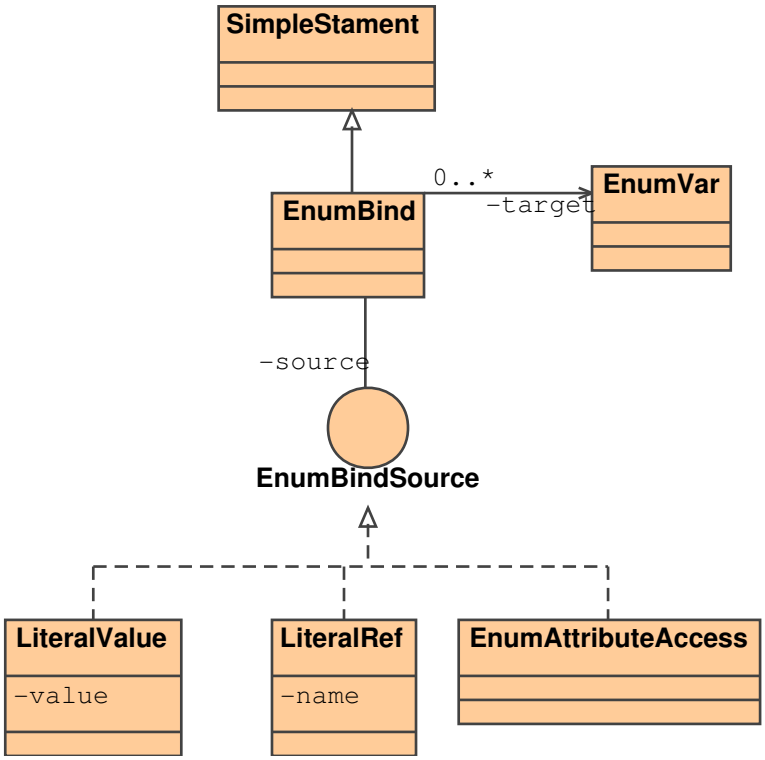


Figura A.35: Relacionamento da meta-classe comando de atribuição variável enumerada com outros componentes do metamodelo da VIRTUOSI

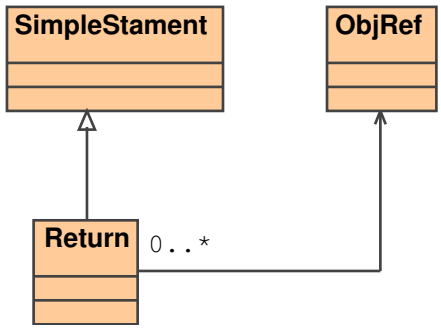


Figura A.36: Relacionamento entre um comando de retorno e uma referência a objeto

A.1.7.5 Retorno de Ação

Conforme explicado na Seção A.1.6, uma ação não retorna uma referência, ao invés disso, tem como retorno um comando *desvie* ou um comando execute. Um comando resultado de teste é o comando retornado por todos os componentes do metamodelo que são testáveis. Tanto o comando resultado de teste quanto os comandos testáveis são detalhados na discussão do comando de desvio condicional nessa Seção.

A.1.7.6 Invocação de Invocáveis

O metamodelo da VIRTUOSI define os comandos para realizar a invocação de construtores, métodos com retorno e métodos sem retorno. A invocação de um construtor é realizada a partir do nome da classe que o possui. Já a invocação de um método, com ou sem retorno, é realizada a partir de uma referência a objeto. Tanto o comando de invocação de método com retorno quanto o comando de invocação de método sem retorno podem ser generalizados como comandos de invocações de método. O comando de invocação de método e o comando de invocação de construtor podem ser generalizados como comandos de invocação.

A Figura A.37 mostra o relacionamento da meta-classe que representa um comando de invocação com sua hierarquia e outros componentes do metamodelo da VIRTUOSI.

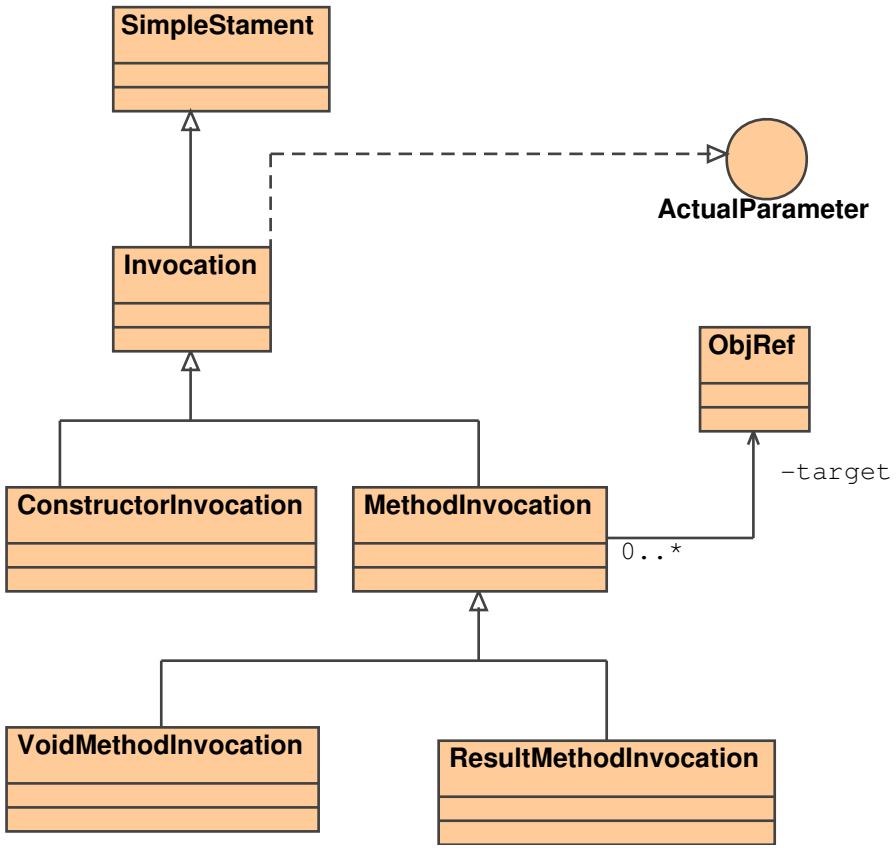


Figura A.37: Relacionamento da meta-classe que representa um comando de invocação com sua hierarquia e outros componentes do metamodelo da VIRTUOSI

O metamodelo da VIRTUOSI formaliza a relação entre os comandos de invocação e os respectivos invocáveis, conforme mostra a Figura A.38.

Conforme a Figura A.38 mostra, um comando de invocação de construtor causa a interpretação da seqüência de comandos definidos em um construtor; um comando de invocação de método sem retorno causa a interpretação da seqüência de comandos definidos em um método sem retorno; um comando de invocação de método com retorno

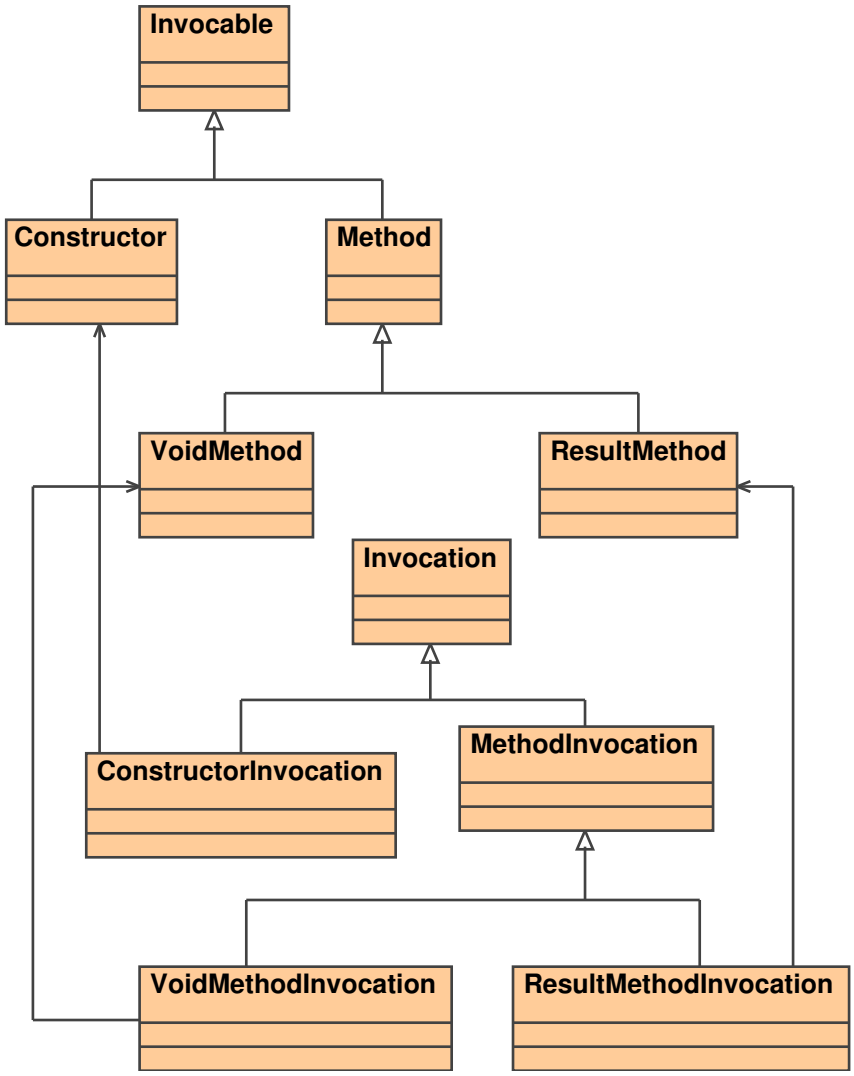


Figura A.38: Relação entre os comandos de invocação e os invocáveis

causa a interpretação da seqüência de comandos definidos em um método com retorno.

Deve-se notar que uma ação, embora seja um componente invocável, não possui um comando para invocação correspondente. A invocação de uma ação é abordada em detalhe na discussão sobre componentes testáveis.

A.1.7.7 Desvios

Dois comandos simples são responsáveis por controlar o fluxo de interpretação dentro de uma seqüência de comandos, a saber:

- **desvio incondicional**;
- **desvio condicional**.

Desvio Incondicional: Um comando de desvio incondicional quando interpretado faz com que uma seqüência de comandos específica seja interpretada. Nesse contexto essa seqüência de comandos é chamada de **caminho destino**. Um desvio incondicional não aparece explicitamente no código fonte, ele sempre é utilizado em conjunto de um comando de desvio condicional.

Desvio Condicional: Um desvio condicional tem duas seqüências de comandos que podem ser interpretados (exclusivamente). A primeira seqüência é chamada de **caminho destino** e a segunda é chamada **caminho alternativo**.

Testável: Na VIRTUOSI, para se interpretar um comando de desvio, não é necessário avaliar o valor de um objeto da classe *Boolean*, embora isso possa ser feito.

Um comando de desvio condicional está associado a algo que pode ser testado, um **testável**[8]. Um testável, quando interpretado, retorna um dentre dois comandos possíveis, a saber:

- comando **execute** ou ;
- comando **desvie**.

Caso o comando retornado seja o comando execute, isto faz com que o caminho destino seja interpretado. Caso o comando retornado seja o comando desvie, o caminho alternativo é interpretado.

Visto que ambos os comandos – execute e desvie – são os dois resultados possíveis de um testável, diz-se que ambos são comandos resultado de teste.

O metamodelo da VIRTUOSI formaliza os comandos de desvio e como estes se relacionam com as seqüências de instruções, conforme mostra a Figura A.39.

Entre os componentes definidos como testáveis pelo metamodelo da VIRTUOSI, está a invocação de uma ação. Uma ação, embora seja um componente invocável, não possui um comando de invocação correspondente. Uma ação também é invocada a partir de uma referência a objeto, contudo, sua utilização *sempre* está associada a um comando de desvio condicional, ou seja, a invocação de uma ação é realizada a partir de uma referência a objeto somente quando esta invocação for o elemento testável de um comando de desvio condicional.

Deve-se notar a diferença entre uma ação e uma invocação de ação. Uma invocação de ação causa a interpretação da seqüência de comandos definidos por uma ação. Um comando de desvio condicional interpreta um testável, para que este retorne um resultado de teste. O testável em questão, *pode* ser uma invocação de ação. A invocação de ação interpreta cada um dos comandos definidos pela ação até encontrar um comando resultado de teste. Esse resultado de teste é utilizado pelo comando de desvio condicional.

O metamodelo da VIRTUOSI formaliza a relação de um desvio condicional, um testável, uma invocação de ação e uma ação, conforme mostra a Figura A.40.

A Figura A.41 mostra um exemplo de utilização de comando de desvio condicional cujo testável é uma invocação de ação a partir de um objeto da classe pré-definida *Integer*.

[8] Do inglês ***testable*** (o termo testável ainda não está registrado nos dicionários da língua portuguesa).

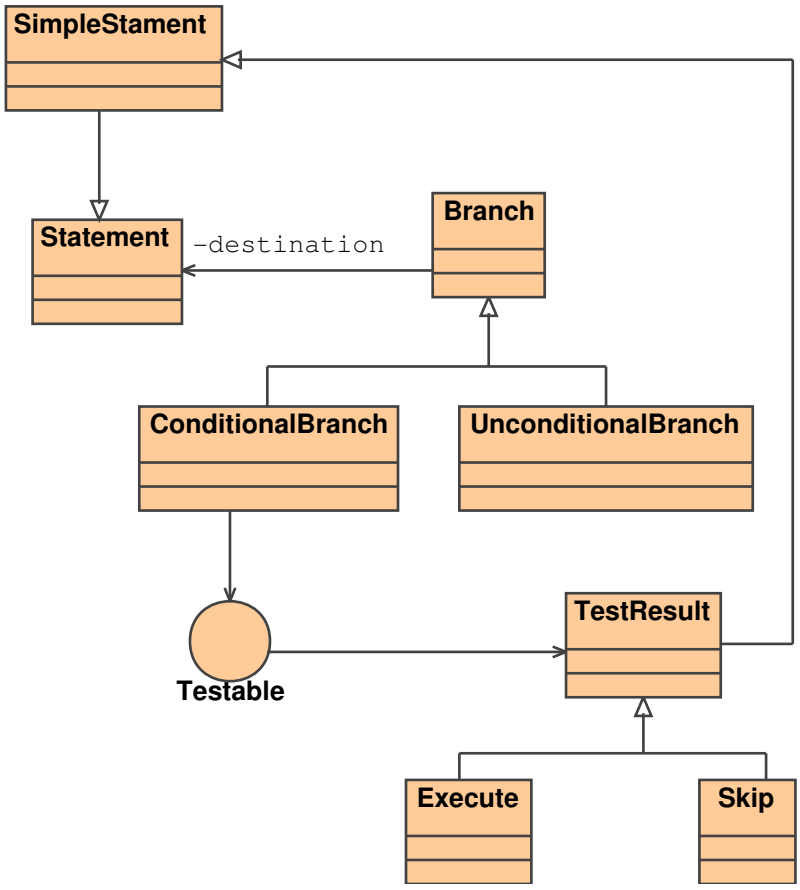


Figura A.39: Instruções de desvio como se relacionam com as seqüências de instruções

A Figura A.42 mostra um exemplo de utilização de comando de desvio condicional cujo testável é uma comparação de valor entre uma variável enumerada e um valor literal.

Essa abordagem é bem diferente da abordagem normalmente utilizada por linguagens de programação, onde um desvio condicional sempre depende da avaliação de uma expressão que retorna um valor verdadeiro ou falso. Isto permite, por exemplo, que qualquer classe defina uma ou mais ações que podem ser utilizadas para a tomada de decisão em um desvio condicional. Além disso, toda classe pode definir uma ação chamada `default` ou ação padrão. Esta ação padrão não precisa ser explicitamente chamada, ou seja, se um comando de invocação tiver como teste simplesmente uma referência a objeto, isto significa que a ação invocada deve ser a padrão. A Figura A.43 mostra o exemplo da definição de uma ação padrão e seu respectivo uso por um comando de desvio. Portanto, embora o funcionamento seja diferente, é possivel utilizar o valor de um objeto da classe *Boolean* como testável de um comando de desvio.

Além de uma invocação de ação, um testável pode ser:

- uma comparação entre duas referências a objeto, chamada **comparação de referência a objeto** – utilizada para verificar se ambas as referências apontam para o mesmo objeto em memória;

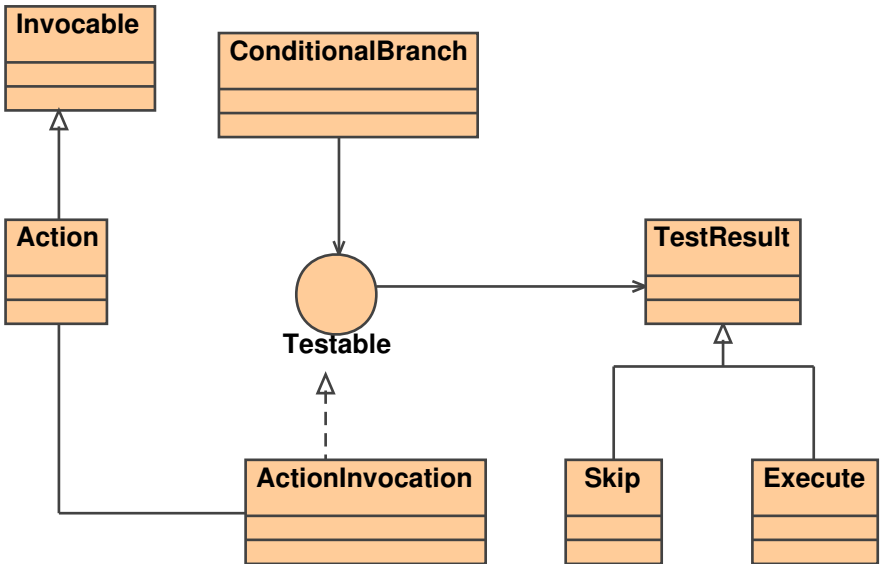


Figura A.40: Relação de um desvio condicional, um testável, uma invocação de ação e uma ação

```
class Pessoa
{
   ...
   action obesa() exports all
   {
       ...
       if (massa_.gt(h))// gt significa greater than
           execute;
       else
           skip;
   }
 ...
}
```

Figura A.41: Desvio condicional com um testável que é uma invocação de uma ação – código fonte em Aram

- uma comparação entre duas referências a bloco de dados, chamada **comparação de referência a bloco de dados** – utilizada para verificar se ambas as referências apontam para a mesma seqüência de dados binários em memória;
- uma comparação entre duas referências a índice, chamada **comparação de referência a índice** – utilizada para verificar se ambas as referências apontam para a mesma posição em uma mesma seqüência de dados binários em memória;
- uma comparação entre uma referência a objeto e uma referência nula, chamada

```
class Boolean
{
   enum { true, false } value = false;

   ...
   method void flip( ) exports all
   {
       if ( value == true )
           value = false;
       else
           value = true;
   }
}
```

Figura A.42: Desvio condicional com um testável que é uma comparação de valor entre uma variável enumerada e um valor literal – código fonte em Aram

comparação de referência nula a objeto- utilizada para verificar se uma referência é nula;

- uma comparação entre uma referência a bloco de dados e uma referência nula a bloco de dados, chamada **comparação de referência nula a bloco de dados** – utilizada para verificar se uma referência é nula;
- uma comparação entre uma referência a índice e uma referência nula, chamada **comparação de referência nula a índice** – utilizada para verificar se uma referência é nula;
- uma comparação de **valor** entre uma variável enumerada e um segundo elemento do tipo variável enumerada, valor literal, referência a literal ou ainda um acesso a atributo variável enumerada, chamada **comparação de valor de variável enumerada** – utilizada para comparar valores;

O metamodelo da VIRTUOSI formaliza todos elementos testáveis que podem ser utilizados por um desvio condicional, conforme mostra a Figura A.44.

A.1.7.8 Repetição

Utilizando um comando de desvio condicional associado a um comando de desvio incondicional (não visível no código fonte) é possível realizar o comportamento de uma estrutura de repetição no estilo *enquanto–faça* ou *faça–enquanto* conforme a Figura A.45 mostra.

```
class Boolean
{
   enum { true, false } value = false;
   ...
   action default() exports all /ação padrão da classe Boolean
   {
       if (value==true) {
           execute;
       }
       else {
           skip;
       }
   }
   ...
}
class Principal
{
   constructor iniciar() exports all
   {
       ...
       Boolean entrou = corsa.entrarPassageiro(andrea);

       if (entrou) {// uso da ação padrão da classe Boolean
           Integer distancia = Integer.make(10);
          ...
       }
       ...
   }
   ...
}
```

Figura A.43: Definição de uma ação padrão e seu respectivo uso por um comando de desvio

A.1.7.9 Vazio

O metamodelo da VIRTUOSI define um comando que não causa nenhum efeito, esse comando é chamado de comando vazio.

A.1.8 Chamadas de Sistema

Conforme citado na Seção A.1.4, o ambiente VIRTUOSI disponibiliza uma biblioteca de classes pré-definidas (*Integer*, *Real*, *Boolean*, *Character* e *String*) para a construção de novas classes chamadas de classes de aplicação.

Contudo, o ambiente VIRTUOSI também disponibiliza comandos simples e testáveis de sistema para a construção de classes que utilizem referências a bloco de dados. As próprias classes pré-definidas disponibilizadas pelo ambiente VIRTUOSI são construídas

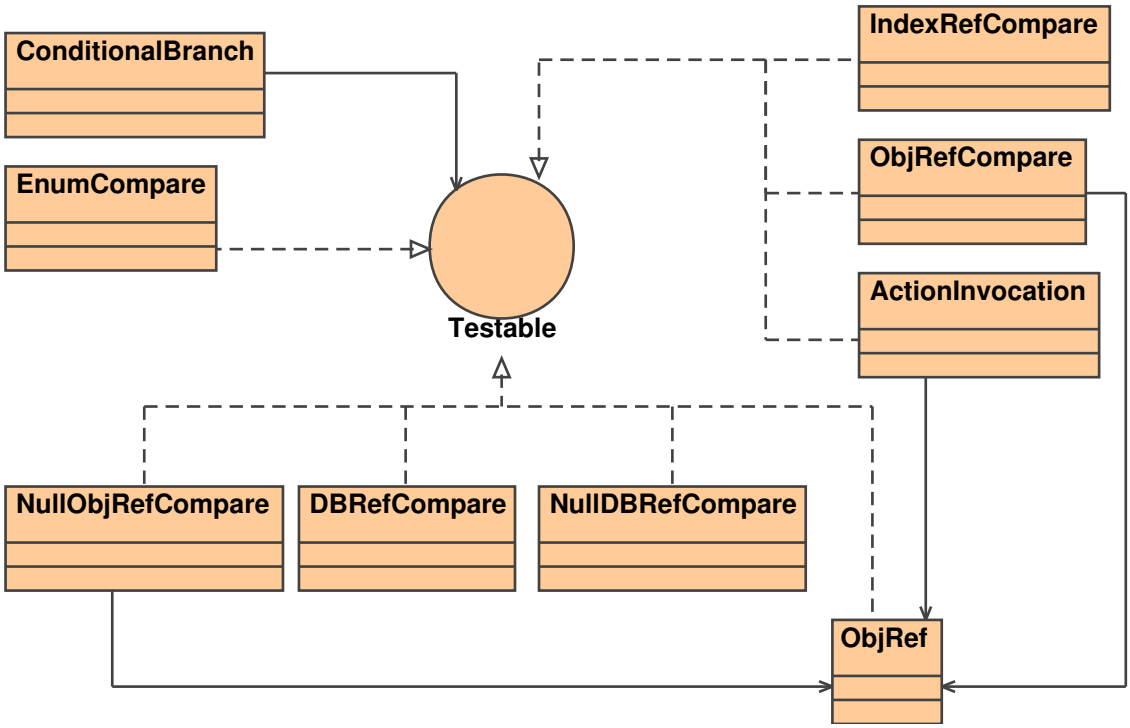


Figura A.44: Relação de um desvio condicional com todas os *testables* possíveis

```
class Principal
{
   constructor iniciar() exports  all
   {
        ...
        Integer t = Integer.make(2);
        while (andrea.obesa()) {
            andrea.emagreca(t);
        }
   }
}
```

Figura A.45: Estrutura de repetição realizada pela combinação de um desvio condicional e um desvio incondicional – código fonte em Aram

com estes comandos e testáveis de sistema.

Para cada uma das classes pré-definidas existe um conjunto definido de comandos de sistema e um conjunto definido de testáveis de sistema. A lista completa dos comandos e testáveis de sistema é apresentada no Apêndice **??**.

A.1.8.1 Comandos de Sistema

Os comandos simples disponibilizados pelo ambiente VIRTUOSI são chamados de comandos de sistema. Com o intuito de organizar a hierarquia das meta-classes que representam estes comandos, o metamodelo da VIRTUOSI define uma hierarquia de comandos de sistema, conforme mostra a Figura A.46.

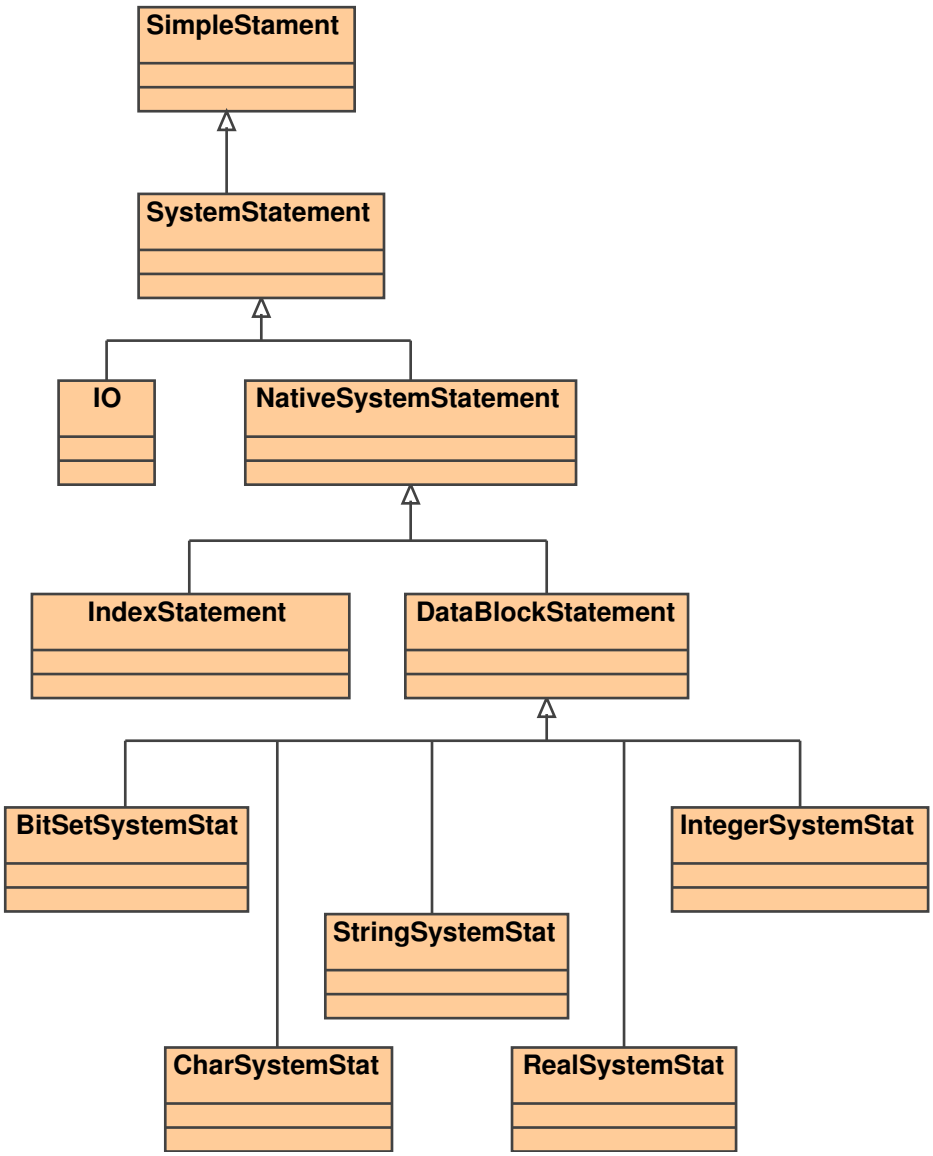


Figura A.46: Comandos de sistema disponibilizados pelo sistema

Observa-se que abaixo do comando de sistema, existem os comandos de entrada e saída e os comandos nativos.

Abaixo dos comandos nativos, existem dois tipos de comando:

- comandos para a manipulação de bloco de dados;
- comandos para manipulação de índices.

Abaixo dos comandos para manipulação de bloco de dados, existem cinco tipos de comando:

- comandos de sistema para seqüência de dados binários, que manipulam dados binários de forma neutra;
- comandos de sistema para valores inteiros;
- comandos de sistema para valores reais;
- comandos de sistema para valores caracter;
- comandos de sistema para valores conjunto de caracter;

A Figura A.47 mostra um exemplo de código fonte de uma classe pré-definida fornecida pelo ambiente VIRTUOSI, no caso uma classe para manipulação de valores inteiros, que utiliza dois comandos de sistema: *createDB* – representado no código fonte pela invocação do construtor make e *storeInteger* – utilizado para armazenar uma seqüência de dados binários no formato apropriado para a representação de números inteiros.

```
class Integer
{
   datablock value;

   constructor make( literal k ) exports all
   {
       value = datablock.make( 32 );
       // 32 é um valor literal utilizado para especificar
       // o tamanho do bloco de dados.
       value.storeInteger( k );
   }
   ...
}
```

Figura A.47: Uso de dois comandos de sistema para facilitar a construção de uma classe pré-definida *Integer* – código fonte em Aram

Observa-se que um comando sistema – disponibilizado pelo sistema – pode retornar uma referência para bloco de dados ou referência para índice. Nota-se também, que através da utilização desses comandos de sistema, o programador pode, por exemplo, definir novas classe que armazenem valores inteiros com bloco de dados de tamanho diferente. Isso é possível porque os comandos de sistema que manipulam bloco de dados com a representação de valores inteiros podem receber como parâmetros um valor literal indicando o tamanho do bloco de dados que deve ser criado.

A.1.8.2 Testáveis de Sistema

Os testáveis disponibilizados pelo ambiente VIRTUOSI são chamados de testáveis de sistema. Com o intuito de organizar a hierarquia das meta-classes que representam estes testáveis, o metamodelo da VIRTUOSI define uma hierarquia de testáveis de sistema, conforme mostra a Figura A.48.

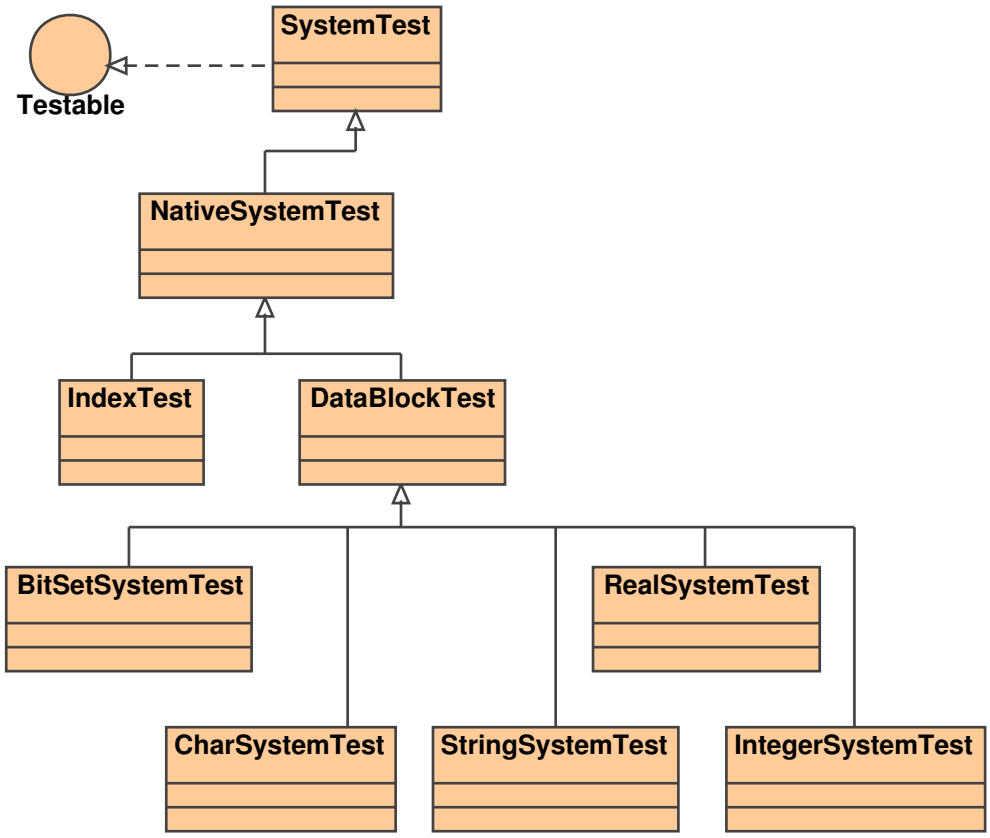


Figura A.48: Testáveis de sistema disponibilizados pelo sistema

Observa-se que abaixo da meta-classe representado os testáveis de sistema, existe somente os testáveis nativos.

Abaixo dos testáveis nativos, existem dois tipos de testáveis:

- testáveis para a manipulação de bloco de dados;
- testáveis para manipulação de índices.

Abaixo dos testáveis para manipulação de bloco de dados, existem cinco tipos de testáveis:

- testáveis de sistema para seqüência de dados binários, que manipulam dados binários de forma neutra;
- testáveis de sistema para valores inteiros;

- testáveis de sistema para valores reais;
- testáveis de sistema para valores caracter;
- testáveis de sistema para valores conjunto de caracter;

A Figura A.49 mostra um exemplo de código fonte de uma classe pré-definida fornecida pelo ambiente VIRTUOSI, no caso uma classe para manipulação de valores inteiros. O exemplo mostra a implementação de duas ações da classe *Integer*. A ação chamada *equals* faz uso de um testável de sistema para seqüência de dados binários chamado *sameBits*. A ação chamada *greaterOrEqual* faz uso de um testável de sistema para valores inteiros chamado *geq*, que retorna um comando execute caso o valor inteiro armazenado no bloco de dados seja maior ou igual ao passado como parâmetro.

```
class Integer
{
   datablock value;
   ...
   action equals( Integer i ) exports { all }
   {
       databalock k = i.value;
       if ( value.sameBits( k ) )
           execute;
       else
           skip;
   }
   action greaterOrEqual( Integer i ) exports { all } // greater or equal
   {
       databalock k = i.value;
       if ( value.geqInteger( k ) )
           execute;
       else
           skip;
   }
}
```

Figura A.49: Uso de testáveis especiais nos comandos de desvio utilizados na construção de uma classe pré-definida *Integer* – código fonte em Aram