Pontifícia Universidade Católica do Paraná – PUCPR
Programa de Poś-Graduação em Informaática Aplicada

Mecanismo de Mobilidade de Objetos para a Virtuosi

Juarez da Costa Cesar Filho

Dissertação apresentada ao Programa de
Pós-Gradução em Informática Aplicada da
Pontifícia Universidade Católica do Paraná
como requisito parcial para obtenção do título de
Mestre em Informaática Aplicada

Julho de 2004

Agradecimentos

Ao amigo e orientador Alcides Calsavara, que transcendeu suas funções de professor – e seus horários – para possibilitar a conclusão deste trabalho. Ao colega, amigo e colaborador Kolb, que percorreu comigo do primeiro ao último dia – último dia literalmente – os desafios do Mestrado. Aos meus Pais, e em especial minha Mãe, que da cartilha até esta dissertação corrige minha escrita.

Resumo

O objetivo deste trabalho é criar um mecanismo para mobilidade de objetos para a Virtuosi. A Virtuosi é um sistema computacional distribuído orientado a objetos que atua sobre máquinas virtuais cooperantes. A sua arquitetura é baseado no conceito de árvores de programa para representação de classes e de referências indiretas através de *handle tables* entre suas entidades. Como outros sistemas distribuídos tem em sua concepção a mobilidade de objetos. O mecanismo projetado deve ter como requisitos garantir a integridade do sistema, possuir um desempenho adequado e tratar possíveis faltas no ambiente. Em complemento a isto, o mecanismo deve disponibilizar funcionalidades – por interface de programação – para o controle da mobilidade dos objetos e, ser compatível com a definição atual do kernel e módulos já implementados. Este trabalho de pesquisa define um mecanismo para mobilidade de objetos baseado em diversos outros documentados na literatura e descreve uma implementação para o mesmo.

Palavras-chaves: mobilidade de objetos; migração de objetos; máquina virtual; metamodelagem.

Abstract

The main goal of this work is to define an Object Mobility Mechanism for Virtuosi. Virtuosi is a distributed computing system based on collaborative virtual machines. Virtuosi architecture uses program trees to represents object classes and indirect references using *handle tables* among its entities. As some other distribuited environments, Virtuosi aims at supporting object mobility. Such mechanism must preserve object integrity, with adequate performance and fault tolerance. Additionaly, it should provide funcionality – through some programming interface – to controle object mobility, and be compatible with Virtuosi's current core and already implemented modules. This research work defines an object mobility mechanism based on several similar mechanisms documented in the literature and describes an implementation for it.

Key-words: object mobility; object migration; virtual machines; metamodeling.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Diversos sistemas computacionais que fazem uso de processamento distribuído têm descrito ou implementado mecanismos de *mobilidade*[1] de objetos. Sistemas como o DCE[Tanenbaum, 1994], CORBA[Crowcroft, 1996], Emerald[Jul et al., 1988], Chorus/COOL[Amaral et al., 1992] e SOS[M. Shapiro, 1989] se beneficiam das várias vantagens que a mobilidade de objetos oferece [Nutttal., 1994]. Existem várias situações onde podemos verificar estas vantagens. A movimentação de um objeto para o mesmo local onde se encontram os recursos como arquivos, periféricos ou mesmo outros objetos com os quais haja bastante interação traz ganhos pela redução do tráfego da rede. Nodos que estão sobrecarregados podem distribuir a carga para outros nodos mais ociosos. Objetos podem necessitar de algum suporte específico, como, por exemplo, um ambiente seguro, ou ainda em redes heterogêneas atividades de um objeto podem necessitar de algum suporte específico fornecido somente por uma plataforma disponível em algum nodo [Jul et al., 1988]. Aplicações onde a mobilidade de objeto é intrínseca como os dispositivos computacionais portáteis, os quais estão sendo cada vez mais utilizadas.

A Virtuosi é um sistema computacional distribuído orientado a objetos que atua sobre máquinas virtuais cooperantes. Como outros sistemas distribuídos, tem em sua concepção a mobilidade de objetos. A arquitetura da Virtuosi tem como características a utilização de árvores de programa para representação de classes e o modo de referências indiretas através de *handle tables* [HU et al., 2003]. Visando estas características é necessário uma especificação formal de um mecanismo de mobilidade de objetos que atenda os requisitos

propostos pela Virtuosi.

1.2 Objetivos

O objetivo deste trabalho é definir formal e precisamente o processo de mobilidade de objetos na Virtuosi. Esta definição deve se adaptar as suas características de forma que utilize suas peculiaridades da melhor forma possível em benefício do desempenho, segurança e integridade do sistema. Para uma melhor compreensão, por ser o tema abrangente, o separamos em quatro objetos de estudo:

- Requisitos de mobilidade: Análise dos requisitos que um mecanismo de mobilidade deve atender.
- Arquitetura: Estudo da arquitetura de sistemas distribuídos orientados a objetos.
- Primitivas: Estudo das primitivas de mobilidade necessárias ao contexto distribuído.
- Protocolo da mobilidade: Definição do protocolo a ser seguido durante a movimentação de um objeto. Tratamento das atividades em andamento, atualização de referências, seqüências possíveis e modelo transacional.

O trabalho utilizará a linguagem gráfica representacional definida para Virtuosi, criando uma extensão desta para os casos ainda não concebidos pelo modelo [Calsavara, 2000]. O estudo contribuirá para a construção da Virtuosi e servirá de subsídio para trabalhos futuros correlatos à área de distribuição e mobilidade de objetos para Sistemas Computacionais Distribuídos (SCD).

1.3 Problemática

Ao desenvolver este trabalho científico pesquisou-se diversos SCDOO (Sistemas Computacionais Distribuídos Orientados a Objetos), de todos estes a arquitetura da Virtuosi se

[1]Mobilidade: Termo para a habilidade de movimentação de um objeto ou processo por entre diferentes nodos de uma rede. Migração foi comumente empregado com o mesmo sentido em artigos menos recentes que compõem este trabalho. Assim adotaremos ambos como sinônimos.

mostrou bastante diferente em importantes aspectos. O conceito de árvores de programa e de *handle tables* [HU et al., 2003] para todas as 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. O enfoque pedagógico e experimental da Virtuosi [Cal, 2004] também influenciou no peso dos requisitos a serem atendidos para o mecanismo.

1.4 Delimitação de Tema

Este trabalho se propõe ao estudo e desenvolvimento de um mecanismo de mobilidade que atenda a arquitetura da Virtuosi, garantindo que o processo de mobilidade atenda os requisitos estabelecido pelo sistema. Aqui não será contemplado o mecanismo de RPC[2], e nem a definição de políticas para a migração, como balanceamento de carga, disponibilidade do sistema, ou qualquer outro tópico que possa se utilizar do mecanismo porém não faça parte dele.

1.5 Organização de Trabalho

O capítulo 2 apresenta o contexto atual sobre mobilidade, apresentando as vantagens da movimentação de objetos, a conotação dos termos utilizados e os requisitos que devem ser atendidos pelos mecanismos. Citamos alguns Sistemas Computacionais Distribuídos e descrevemos com maior profundidade o Emerald, Distributed Oz e CORBA, os quais são parâmetros para este trabalho. O capítulo 3 descreve o funcionamento da arquitetura da Virtuosi em todos os aspectos importantes para a mobilidade: metamodelo, árvores de programa, modelo referencial, objetos, atividades, classes e Máquina Virtual. No capítulo 4 detalhamos o mecanismo de mobilidade proposto, começando pelas primitivas disponíveis, as condições necessárias para que ocorra a migração e a descrição do funcionamento do mecanismo. O capítulo 5 descreve alguns cenários para a mobilidade. No capítulo seguinte

[2]Remote Procedure Call: Protocolo que um programa pode utilizar para requisitar um serviço a outro programa localizado remotamente, sem que seja necessário o conhecimento de detalhes da rede.

é descrito a implementação do mecanismo. O capítulo 7 conclui este trabalho e apresenta proposta de trabalhos futuros dentro do tema de Sistemas Distribuídos Orientados a Objetos.

Capítulo 2

Mecanismos de Mobilidade

2.1 Introdução

O mecanismo de mobilidade de objetos é algo poderoso, porém ainda pouco explorado nos SCDOO (Sistemas Computacionais Distribuídos Orientados a Objeto). O movimento que um objeto faz de um nodo para outro pode ser motivado por alguma requisição externa ou por ele mesmo. Existem muitas implicações nesta ação, pois a total integridade e funcionalidade do objeto devem ser mantidas no novo local, incluindo o estado de suas atividades no momento da movimentação, as referências a outros objetos, processos e demais recursos que ele esteja usando. O inverso também é verdadeiro, pois todos os demais objetos devem atualizar suas referências para o novo local onde se encontra o objeto deslocado de forma automática e em nenhum momento do processo comprometer o sistema. Existem diversas vantagens conseguidas através da mobilidade dos objetos como o controle de balanceamento de carga, centralização de objetos com atividades com grande interação entre si, aumento da disponibilidade e outras que serão explicadas no tópico seguinte. Dentre todos os SCDOO definimos três para um estudo mais aprofundado, são eles o Distributed Oz, Emerald e CORBA. O critério usado para a seleção do Distributed Oz e Emerald foi que estes sistemas, de forma semelhante a Virtuosi, não implementam a distribuição adicionando uma camada sobre uma linguagem centralizada existente, o que ocorre em outros sistemas como DCE [Tanenbaum, 1994],Java [Arnold and Gosling, 1996] e Erlang [Wikstrõm, 1994] [Haridi et al., 1997]. Embora CORBA implemente a distribuição pela adição de uma camada sobre uma linguagem centralizada, o estudo da especificação sobre agentes móveis trouxe contribuições para elaboração dos passos necessários a serem seguidos por um mecanimos de mobilidade.

2.2 Motivação para a Movimentação

A motivação para a mobilidade de objetos vem das vantagens que ela traz aos SCDOO. A mobilidade é um mecanismo indispensável para a solução de várias situações encontradas em ambientes distribuídos como as citadas abaixo:

Balanceamento de carga: Nodos sobrecarregados podem ser desafogados pela mobilidade de objetos com atividades para outros nodos mais ociosos.

Desempenho de comunicação: Objetos que possuam grande interatividade de comunicação entre si podem ser dispostos em um mesmo nodo, reduzindo assim o tempo gasto com a utilização da rede. O mesmo acontece com interatividade entre objetos com arquivos, periféricos ou interfaces com outros aplicativos.

Disponibilidade: Objetos podem ser deslocados para nodos mais estáveis aumentando a eficiência de cobertura a falhas do sistema.

Utilização de capacidades especiais: Determinado objeto pode precisar de algum suporte especial existente somente em algum nodo. Por exemplo, efetuar suporte a operações que exijam um ambiente seguro, ou privilégios de execução para processamento em tempo real, ou em redes heterogêneas algum serviço específico prestado por uma plataforma.

Movimentação de dados: Encapsulando arquivos ou informações dentro de objetos, temos uma maneira fácil de movimentar estes dados sem a necessidade de tratar separadamente como uma transferência de arquivo ou como um envio de mensagem.

Coletor de lixo: Mobilidade de objetos simplificam o trabalho do Coletor de Lixo por mover os objetos para locais onde estes sejam referenciados [Hewitt, 1980] [Vestal, 1987].

Os casos citados acima justificam a motivação para que haja mecanismos de mobilidade de objetos para Sistemas Computacionais Distribuídos.

2.3 Requisitos de um Mecanismo de Mobilidade

De toda a bibliografia estudada não foram encontrados requisitos específicos para um mecanismo de mobilidade. Logo, teve-se que aumentar a abrangência partindo para os Sistemas

Distribuídos. Primeiramente fez-se um estudo sobre os requisitos dos sistemas distribuídos e adotou-se a abordagem do Distributed Oz que prevê quatro requisitos [Haridi et al., 1997]:

Transparência de Rede: significa que os processos computacionais no sistema devem se comportar da mesma forma, independente da estrutura distribuída em que se encontram [Cardelli, 1995]. A semântica da linguagem é inalterada independente de como a computação está distribuída entre os nodos. A distinção entre referências locais ou remotas deve parecer indiferente para o programador. A concorrência entre atividades deve ser suportada de forma que possa ser executada de forma distribuída ou centralizada em um único nodo.

Controle sobre a Comunicação de Rede: significa que os padrões de comunicação pela rede devem ser programáveis e previsíveis. Estes padrões devem estar disponíveis para a programação de modo a tornar possível o controle sobre o comportamento da rede. Isto se traduz no poder da programação em definir momento e local de onde deva ou não ocorrer a movimentação.

Tolerância à Latência: significa que a eficiência da computação deve ser minimamente afetada por atrasos acarretados pela rede. A utilização da rede deve ser feita de forma eficiente e racional.

Linguagem Segura: significa que a linguagem garante a integridade das atividades e dados, desde que haja *comunicação segura*[1]. *Language security* tem como objetivo garantir ao programador meios para restringir o acesso a dados através do escopo léxico da linguagem, ou seja, acessar dados somente onde exista uma referência explícita determinada por uma estrutura da linguagem.

Existem outros requisitos que poderiam ser avaliados, tais como: locatlização de recursos, suporte para múltiplas camadas, implementaçao segura e tolerância a falhas. Porém fogem do escopo que o mecanismo da Virtuosi vai atender em primeira instância. Além disso mesmo sendo a segurança um requisito básico não será contemplado neste trabalho.

Seguinte ao estudo, identificamos os pontos onde o mecanismo de mobilidade pode contribuir para os requisitos dos SCDOO. Então definimos os seguintes requisitos de um mecanismo de mobilidade.

[1]Entende-se por comunicação segura a proteção da integridade computacional contra possíveis intrusos que tenham acesso à implementação do sistema [Roy et al., 1997]

Integridade: é a garantia da integridade dos dados, das atividades e referências de tudo que faz parte ou é afetado na mobilidade. Todas as atualizações executadas sobre as referências devem ser precisas. A movimentação de dados deve ser segura e as atividades preservarem a semântica e ordem processual.

Desempenho: o mecanismo deve ser otimizado e ter desempenho compatível com as expectativas. A eliminação de redundâncias e o uso criterioso da rede são essencias para o seu atingimento.

Confiabilidade: o protocolo de comunicação deve tratar possíveis falhas do ambiente. Deve ser preemptivo quanto à ocorrências de indisponibilidade da rede.

Funcionalidade: o mecanismo deve dispor de controles que auxiliem na programação do comportamento da rede. Estes controles usualmente são disponibilizados através de primitvas.

A *Transparência da Rede* só é possível se houver um ambiente que resolva todas as questões de distribuição. Objetos locais ou remotos devem ser indiferentes à vista do programador. Este ambiente dinâmico e complexo só é possível se a integridade durante a movimentação dos objetos for garantida.

Para termos a *Controle sobre a Comunicação* é necessário que haja confiabilidade, o que somente é possível pela garantia de atomicidade do movimento. Outra relação da *Network Awareness* é com a funcionalidade, pois esta implica diretamente na facilidade de controle sobre o comportamento da rede.

Tolerância à Latência está relacionado diretamente aos requisitos de Desempenho e Funcionalidade. Mecanismos de mobilidade otimizados auxiliam diretamente na redução de atrasos ocasionados pela rede contribuindo para o desempenho geral do sistema. Outra forma de incrementar o desempenho é permitir ao programador, através de funcionalidades de controle, interferir no comportamento da distribuição.

O requisito de *Linguagem Segura* é suportado pela garantia de Integridade. A integridade das atividades e dos dados só é possível se preservada a integridade dos mesmos durante sua movimentação.

Outras relacões podem ser descritas, porém entendemos serem estas as mais importantes ao nosso trabalho. Nosso objetivo com isso é que estes requisitos nos auxiliem a avaliar os

mecanismos existentes e nos sirva como guia em nosso trabalho com a Virtuosi.

2.4 Emerald

Emerald é uma linguagem baseada em objetos e um sistema projetado para a construção de sistemas distribuídos [Amaral et al., 1992]. O Emerald não tem por objetivo ser utilizado em grandes redes, ele é concebido para ser utilizado em redes mais modestas, com não mais de cem nodos e com um ambiente homogêneo. Suporta *fine-grainned mobility*, ou seja, tem como unidade de migração estruturas que podem ser muito pequenas e assim estarem muito mais dispersas. Possui uma operação eficiente com referências locais conseguidas através das múltiplas implementações de seus objetos. Embora seus objetos possuam diversas implementações, estas se tornam invisíveis ao programador. A definição semântica de objeto é única para qualquer implementação do objeto independente se é local, objetos ativos ou somente com atributos, móveis ou distribuídos. O compilador é quem decide qual mecanismo que o objeto deve adotar.

2.4.1 Estruturas da Linguagem

No Emerald, não existem outras estruturas da linguagem que não o objeto. Todo objeto no Emerald possui quatro componentes: um nome global único no contexto do sistema, atributos, métodos, e um processo opcional. Somente objetos que possuem processo são ativos. Os demais são considerados meras estruturas de dados e de código. Emerald não possui hierarquia de classes. Objetos não pertencem a uma classe conceitualmente, cada objeto carrega seu próprio código. Porém, internamente objetos em um mesmo nodo compartilham código. Este código é armazenado no que denomina-se de *concrete type object*. Estes tipos concretos são imutáveis, logo podem ser copiados livremente pelos nodos. Quando um objeto é movimentado para um outro nodo seus dados são enviados sem seu *concrete type object*. Se existe um processo ativo para este objeto, a parte da pilha de execução onde se encontra o processo do objeto é enviada juntamente. Ao receber o objeto, o nodo verifica se precisa ou não do *concrete type object*, se precisar utiliza um algoritmo de procura que vai localizar o tipo necessário nos demais nodos do sistema, fará uma cópia dele, e o carregará dinamicamente.

2.4.2 Primitivas

A mobilidade do objeto no Emerald é determinada pelo uso de quatro primitivas:

***locate* obj** primitiva que retorna o nodo onde se encontra o objeto obj.

***move* obj *to* N** move o objeto obj para o nodo N.

***fix* obj *at* N** Fixa o objeto obj no nodo N.

***unfix* obj** Torna móvel o objeto obj.

***refix* obj** Executa o *Unfix*, *Move* e o *Fix* para o objeto obj.

As primitivas *fix* e *unfix* determinam se o objeto pode ser movido ou não. Depois de fixado somente o *unfix* lhe devolve a mobilidade. A primitiva *locate* é encapsulada em um objeto nodo, que é uma abstração de uma máquina física. Esta primitiva é utilizada para localizar o nodo onde se encontra o objeto. A primitiva *move* não necessariamente é executada, sendo somente uma sugestão. Caso o move seja encontrado na programação, o kernel não obrigatoriamente a executa, e se executa, o objeto não necessariamente vai permanecer no nodo destino. Emerald também possui um mecanismo de agrupar objetos. Isto é muito útil para evitar a separação de um objeto que tem grande interação dentro de um grupo. Caso não seja evitada esta separação, o custo de mantê-lo longe dos demais seria muito alto pela utilização de um grande número de chamadas remotas.

2.4.3 Parâmetros

A passagem de parâmetros no Emerald é sempre por referência (*call-by-reference*) independentemente se local ou remota. Como os objetos são móveis, nas chamadas remotas é possível obter ganhos de desempenho movendo o objeto referenciado no parâmetro para o nodo de onde ocorre a invocação. A movimentação do objeto é primeiramente determinada em tempo de compilação através dos critérios de otimização do compilador. Porém existe a possibilidade de o programador determinar se objeto deve ser migrado baseado em seu conhecimento do sistema. Para isto o programador deve utilizar-se do modo de passagem de parâmetro denominado *call-by-move*.

Exemplo:

```
operation Deliver
    Var aMailbox: Mailbox
    if ToList.Length=1 then
        aMailbox <- Tolist.getelement[ToList.lowerbound]
        aMailbox.Deliver [move self]
    else
        var i: integer <- ToList.lowerbound
        loop
            exit when i > ToLista.upperbound
            aMailbox <- ToList.getelement[i]
            aMailBox.Deliver [self]
            i <- i + 1
        end loop
    end if
end Deliver
```

No exemplo acima a operação *Deliver* entrega as mensagens de todas as $aMailboxes$ da lista $Tolist$. No entando, o caso mais comum é onde existe somente um destinatário, então $call - by - move$ é utilizado para movimentar a $aMailbox$ para o local destino. Se houver mais que um destinatário, somente a referência é passada. Não necessariamente o objeto movimentado no parâmetro permanecerá no nodo destino.

2.4.4 Tipos de objetos

Embora para a visão do programador exista somente uma definição de objeto, internamente isto não se reflete. Baseado no conhecimento do comportamento do objeto que pode ser deduzido durante a compilação, Emerald pode gerar três tipos de implementações para os objetos:

Global: este objeto pode ser referenciado por qualquer outro objeto no sistema. Não necessariamente o objeto que vai referenciá-lo precisa conhecê-lo durante a compilação.

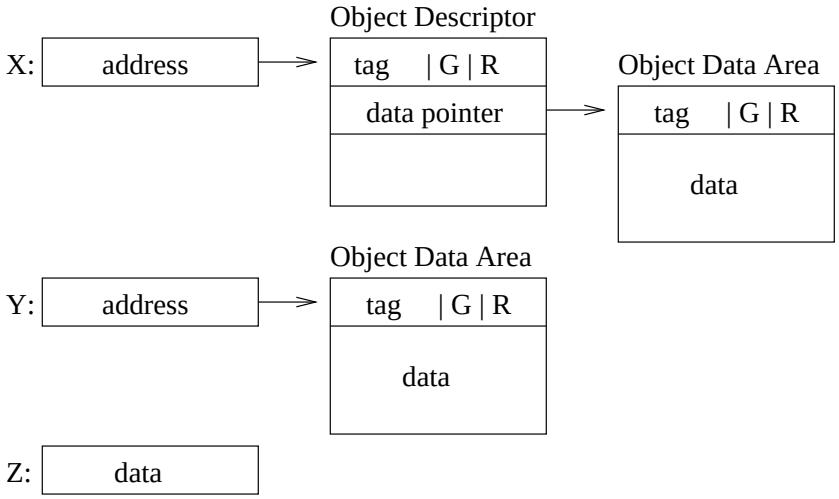


Figura 2.1 Tipos de endereçamento no Emerald

Local: sempre estará contido em outro objeto. Nunca poderá ser movimentado independente do objeto que o contenha.

Direct: é um objeto local o qual tem sua representação de dados diretamente contida junto à representação de dados do objeto que o contenha. Este estilo de objeto é usado principalmente para tipos primitivos como, por exemplo, o Integer.

Cada um destes objetos possui um tipo de endereçamento diferente como mostra a figura 2.1. A variável X é o nome de um objeto global, e o conteúdo de X é o endereço de um *local object descriptor*. Cada nodo contém um *object descriptor* para cada objeto global para o qual os objetos residentes neste nodo têm referências.

As informações sobre o estado do objeto e a localização são armazenadas no *object descriptor*. O campo preenchido com a letra G indica se o objeto é Global ou Local. Se o bit estiver com 1 é global senão local. O campo representado por R informa se o objeto é residente naquele nodo ou não. Se for o residente o *object descriptor* contém o endereço da área de dados do objeto, senão terá um endereço de outro *object descriptor* que possivelmente o tenha.

A variável Y representa o nome de um objeto local, no conteúdo desta variável está o endereço da área de dados do objeto. Neste caso o, campo G que esta junto a área de dados deve estar preenchido com 0 e o campo R com 1, pois objetos locais são sempre residentes. A variável Z é uma variável direct. Para este tipo de variável a área de dados esta representada

diretamente no seu conteúdo.

A localização de objetos no Emerald é baseada no conceito de *fowarding addresses* [Fowler, 1985]. Existe um OID (*Object Identifier*) para cada objeto global do sistema. Cada nodo possui uma tabela de acesso que mapeia o OID para o seu object descriptor correspondente. Em cada tabela de acesso existe uma entrada para cada objeto local o qual exista uma referência remota para ele, e para cada objeto remoto o qual exista uma referência local para ele. Sempre que o *object descriptor* não for residente, o endereço o qual ele apontará será um *forwarding address*. Este tipo de endereçamento é composto por um campo *timestamp* e o nodo, onde o nodo é a última localização conhecida do objeto e o *timestamp* determina a idade da localização. O *forwarding address* mais o OID são suficientes para encontrar o objeto.

2.4.5 Estrutura do Objeto

A estrutura do objeto no Emerald, como já dito anteriormente, é composta pela área de dados e o *concrete type object*. Toda a área de dados de um objeto contém um *control information*, onde estão as informações sobre o tipo de implementação do objeto e a localização, um ponteiro para o *concrete type object* e uma área a ser estruturada conforme um template que está contido no *concrete type object*. A Figura 2.2 mostra um exemplo de um objeto com um monitor, um atributo de um tipo primitivo integer de quatro bytes e duas referências para objetos, uma para si mesmo e outra para uma string.

Exemplo:

```
const simpleobject == object simpleobject
    monitor
        var myself: Any <- simpleobject
        var name: String <- "Emerald"
        var i: Integer <- 17
        operation GetMyName -> [n: String]
            n <- name
        end GetMyname
    end monitor
```

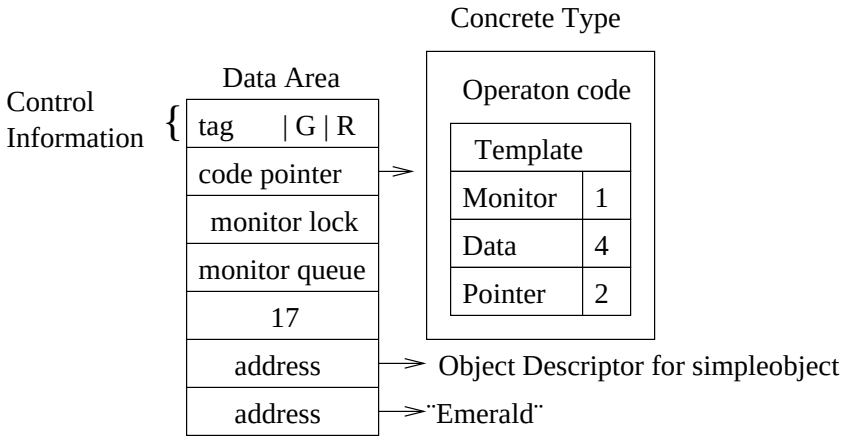


Figura 2.2 Representação de um objeto do Emerald em memória.

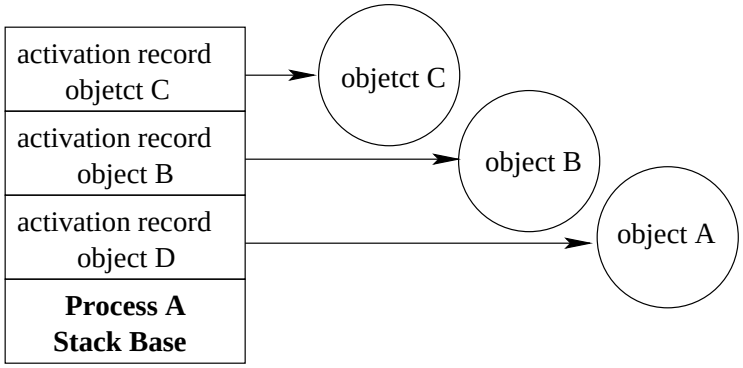


Figura 2.3 Representação de uma pilha de execução.

```
end simpleobject
```

2.4.6 Processos

Cada processo em Emerald é implementado como uma thread. Somente objetos com um processo podem iniciar um nova pilha de execução. Um processo pode conter várias operações de diversos objetos. Cada pilha de execução pode ser representada com um conjunto de *activation records* como demonstrado na Figura 2.3. O objeto A incia o processo e empilha as operações de outros dois objetos B e C.

Em Emerald quando uma invocação remota ocorre, o novo *activation record* é movido para o local da execução, e torna-se a base de uma nova pilha de execução. Assim a pilha

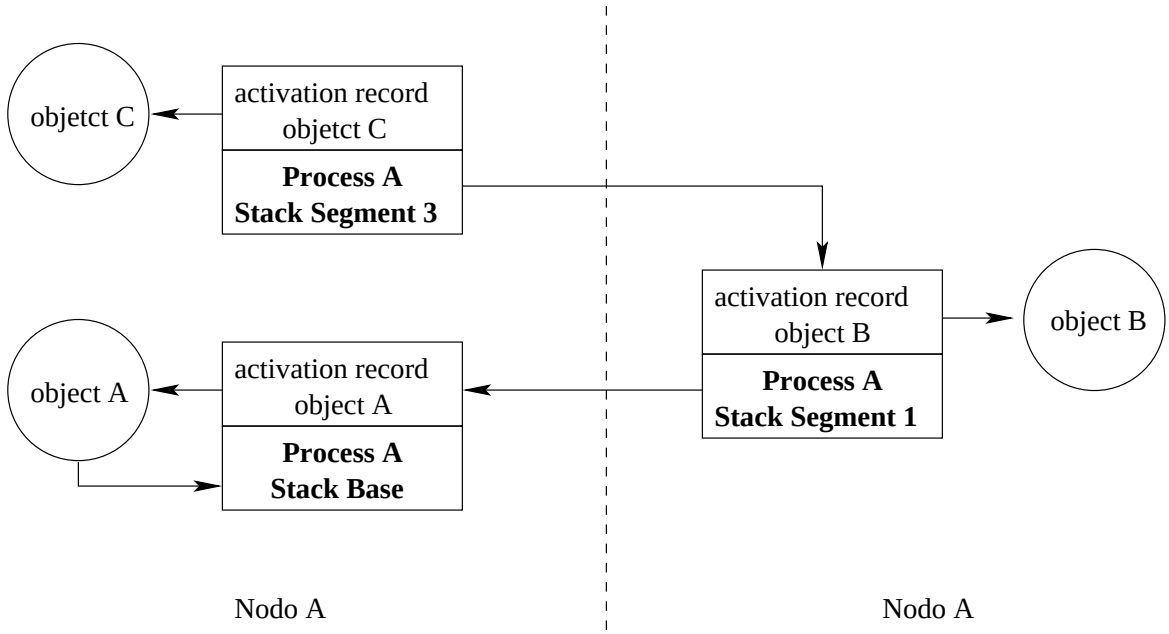


Figura 2.4 Representação distribuída de um processo através de suas pilhas de execução.

de execução pode ser distribuída entre os nodos.

Na figura 2.3 existe o caso onde o objeto B, que possui um *activation record* no meio da pilha de execução, é movido para outro nodo. Neste caso a pilha é particionada em três segmentos. A base e o *activation record* de A permanecem imóveis. Já o objeto B leva consigo seu *activation record* e forma a base de uma nova pilha do processo A em seu nodo destino. O *activation record* de C forma localmente um nova pilha de execução onde ele próprio é a base (ver Figura 2.4).

2.4.7 Movendo Objetos

Para mover um objeto, Emerald envia uma mensagem ao nodo destino contendo toda a área de dados do objeto e mais algumas informações que auxiliam o nodo destino a reorganizar as referências do objeto. Para ponteiros de objetos globais são enviados o OID, o *forwarding address* e o endereço do *object descriptor*. Para objetos locais, envia-se, junto à área de dados, o seu endereço. Assim que o nodo destino recebe toda a informação, o *kernel* aloca espaço para o objeto, copia a área de dados e cria uma tabela de tradução que mapeia os endereços originais para endereços do novo espaço alocado. OIDs são utilizados para localizar

os *object descriptors* para objetos globais já referenciados pelo nodo, ou para criar novos *object descriptors* onde necessários. Finalmente o *kernel* identifica o template dentro do *concrete type object* correspondente ao objeto e identifica a estrutura da área de dados. Com esta informação pode-se atualizar os ponteiros do objeto com os ponteiros correspondentes da tabela de tradução.

2.4.8 Movendo Atividades

Quando um objeto é movimentado, carrega consigo seus *activation records*, quebrando a pilha de execução em dois ou mais segmentos como demonstrado no subtópico sobre processos. Para isso é necessário criar uma lista de todas as atividades do objeto e atualizá-la em toda a invocação que iniciar ou finalizar, ou varrer as pilhas de execução atrás das atividades do objeto em questão. O Emerald utiliza em parte as duas abordagens, porém de forma mais otimizada. Existe uma lista das *activation records* em execução em cada objeto. Porém na invocação o *activation record* não está ligado a esta estrutura. Ao invés disto, um espaço é deixado para a ligação e marcado como *not linked*, esta é uma operação sem custo relevante. Quando algum processo de movimentação no Emerald é iniciado, a pilha é varrida a procura somente dos *not linked activation records*. Estes são então ligados com os descritores dos seus respectivos objetos. Se uma operação que possui uma ligação for concluída, ela deve se desvincular da lista de *activation records* em execução do objeto referido. Os *acivation records* são movimentados de maneira similar as áreas de dados do objeto.

Um problema adicional ao se mover *activations recods* é o gerenciamento dos registradores do processador. O compilador do Emerald otimiza o endereçamento de objetos armazenando variáveis locais em registros ao invés de *activation records*. Isto causa dependência da arquitetura do computador. Para solucionar este problema o *kernel* envia uma cópia dos registradores usados em uma invocação junto com o *activation record* movimentado. Utilizando informações do seu tipo concreto é possível determinar o mapeamento dos registradores e enviá-los com a correta semântica [Jul et al., 1988].

2.5 Distributed Oz

Distributed Oz é um sistema de programação distribuído que preserva o uso de uma semântica de linguagem centralizada. O conceito de rede é abstraído do programador. Chamadas a operações de rede podem ser executadas implicitamente pelo sistema com o uso de algum construtor da linguagem de forma invisível ao programador. Porém o programador tem o poder de controlar o fluxo da distribuição, pelo fato de que operações de rede usualmente têm um custo elevado [Haridi et al., 1997]. Diferentemente do Emerald, Oz não deixa uma trilha por onde os objetos passam (*forwarding address*). O comportamento da rede é melhor previsível, pois a comunicação segue diretamente para o endereço correto, sem a necessidade de passar por vários nodos até encontrar o objetivo [Roy et al., 1997].

2.5.1 Definição de Termos

Para um melhor entendimento conceitual e para sermos fiel aos nomes definidos dentro do universo de Distributed Oz, segue a definição de alguns termos utilizados:

***Language Entity*:** Ou somente entidade: É um item de dados básico da linguagem, como um objeto, uma procedimento, uma thread ou um registro [Roy et al., 1997].

***Statefull Entity*:** É uma entidade que pode ser alterada durante seu ciclo de vida. Em um determinado momento uma entidade $statefull$ está localizada em um particular nodo, chamado *home site* [Roy et al., 1997].

***Stateless Entity*:** É uma entidade que não pode ser alterada durante seu ciclo de vida.

***Mobily Control*:** É a habilidade das entidades $statefull$ de migrarem entre os nodos ou de permanecerem estacionárias de acordo com a intenção do programador [HU et al., 2003].

2.5.2 Linguagem

A linguagem Distributed Oz é dinamicamente escrita, ou seja, seus tipos de estruturas são verificados em tempo de execução [Roy et al., 1997]. A linguagem compreendida pelo *kernel*,

a qual o Distributed Oz é traduzido, é denominada OPM (*Oz Programming Model*). OPM é um modelo de programação concorrente o qual permite sincronização, orientação a objetos, e faz uma clara distinção entre referências para entidades *statefull* e *stateless*.

2.5.3 Estruturas da Linguagem

Distributed Oz define uma semântica distribuída para todas as entidades, significando que cada operação de cada entidade tem uma clara definição de seu comportamento na rede. As entities básicas compreendidas pelo OPM são valores, variáveis lógicas, procedimentos, *cells* e *threads*. Um valor nunca é alterado e é o mais primitivo dos dados. Todas as variáveis são variáveis lógicas (*stateless*), cujos conteúdos não podem ser alterados. Quando são necessárias variáveis que mudam o seu conteúdo ($statefull$) utilizam-se *cells*. $Threads$ e *cells* são as únicas entidade $statefull$. Para uma clara compreensão destas entidades é necessário contextualizá-las dentro da linguagem, como faremos a seguir ao descrevermos as sentenças do OPM. Um programa escrito em OPM consiste em uma composição de sentenças contendo descrição de valores, declaração de variáveis, definição de procedimentos e chamadas, declaração de estado e alteração, condicionais, declaração de *threads*, e tratamento de exceção.

Descrição de valores: Os valores são empregados através de registros, números, literais (nomes e átomos), e *closures*. Exceto por nomes e *closures*, os demais valores são definidos de forma usual. *Closure* é uma parte do procedimento e somente pode ser referenciada através do nome do procedimento. Os outros valores podem ser diretamente escritos ou referenciados por variáveis [Roy et al., 1997]:

Exemplo:

```
local V W X Y Z H T in
     V = queue(head:H tail:T) %Registro
     W = H|T                  %Registro (representando uma lista)
     X = 4324                 %Número
     Y = foo                  %Literal (atom)
     {NewName Z}              %Literal (nome)
 end
```

A chamada { NewName Z} cria um novo nome que é único para todo o sistema. Nomes são utilizados para representar cells, procedimentos e threads.

Declaração de variáveis: Todas as variáveis são variáveis lógicas. Elas precisam ser declaradas dentro de um escopo explícito situado entre um *local* e um *end*. Toda variável inicia sem o conhecimento de seu valor. Somente após uma operação de atribuição de valor, a variável passa a conhecê-lo. Por exemplo, a operação de atribuição X = Y, onde é atribuído o valor de Y para X. A sincronização entre threads pode ser executada através de variáveis, desde de que qualquer tentativa de utilizar uma variável a qual ainda não conheça seu valor irá paralisar as threads até que este valor seja conhecido.

Exemplo:

```
local X in                    %Declaração de X
    thread {Consumer X} end % Cria a thread que utiliza X
        {Producer X}        % Calcula o valor de X
end
```

No código acima, a chamada do procedimento *Consumer* tendo como parâmetro X, utiliza-se da variável X como base de cálculo para outras operações internas, porém não executa a operação de atribuição sobre ela. Já a *Producer* efetua atribuição sobre X. Neste caso a thread de *Consumer* irá iniciar, porém no primeiro instante em que fizer uso de X ficará bloqueada até que *Producer* faça a atribuição de um valor a X. Neste exemplo já demonstramos como é a sintaxe de criação de threads.

Definição de procedimento: A definição e chamada de procedimentos se dão conforme o exemplo abaixo:

Exemplo:

```
local
    MakeAdder Add3 X Y
in
    proc {MakeAdder N AddN}          %Definição da procedimento
```

```
        proc {AddN X Y} Y=X+N end
    end
    {MakeAdder 3 Add3}              %Chamada da procedimento
    { Add3 10 X}                    %X adquire o valor 13
    { Add3 1 Y}                     %Y adquire o valor 3
end
```

Executando a chamada {MakeAdder 3 Add3} define Add3 como um procedimento que adiciona 3 ao seu primeiro parâmetro. A execução da definição do procedimento cria um nome e um *closure*. Um *closure* é um valor que contém o código do procedimento e suas referências externas.

Declaração de estado e atualização: Variáveis sempre se referem a valores que não mudam. Quando é necessária a criação de um dado $statefull$, é necessário fazê-lo de forma distinta através da criação de uma *cell*. Uma *cell* é criada a partir da chamada {NewCell X C} onde C é o nome da *cell* e X seu valor inicial. Existem outras duas operações executadas sobre as cells: {Exchange C X Y} atualiza C com o conteúdo de Y e atribui a X o conteúdo antigo de C, {Accesss C X} atualiza C com o conteúdo de X.

Exemplo:

```
local C X1 X2 X3
in
    {NewCell bing C}
    % Cria C com conteúdo bing
    {Exchange C X1 bang}
    %bing é atribuído a X1 e bang atribuído a C
    {Exchange C X2 bong (me:C was:X2) }
    %bang é atribuído a X2 e bong(me:C was:X2) é atribuído a C
    {Access C X3}
    %bong(me:C was:X2) á atribuído a X3
end
```

Condicionais: São descritas em OPM como testes sobre valores de estruturas de dados, conforme o exemplo:

Exemplo:

```
local X in
    thread
        if X=yes then Z=no else Z=yes end
    end
    X = no
end
```

Declaração de Threads: cada thread executa seu código seqüencialmente. A thread será bloqueada se algum valor que ela precise não estiver disponível, e continuará seu processamento assim que obtê-lo. A concorrência é introduzida explicitamente pela criação de uma nova thread:

Exemplo:

```
local Loop in
     proc {Loop N}       % Define aprocedimento
         {Loop N+1}
     end
     thread              %Define a thread
         {Loop 0}
     end
end
```

Cada thread é identificada por um nome único que é obtido pela chamada $\{GetThreadIDT\}$ dentro da própria thread. Cells e threads são as únicas entidade $statefull$ em OPM. As entidades portas e objetos, que descreveremos na seqüência, são definidas em termos de cells.

Tratamento de Exceção: É um tratamento especial do fluxo seqüencial de controle das threads. Este tratamento permite um desvio no processamento caso ocorra uma operação ilegal no código.

Exemplo:

```
proc {AlwaysCalcX CalcX A X}
    try
        local Z in
            {CalcX A Z}
            Z = X       %Atribui X se não ocorrer exceção
            CalcX
        end
    catch E then
        {FailFix A X}
    end
end
```

Distributed Oz fornece duas entidades adicionais elaboradas a partir das entities do OPM que facilitam ao programador implementar o conceito de orientação a objeto, através de objetos e assincronismo através de portas. Objetos são definidos em termos de OPM como procedimentos e cells. O procedimento referencia uma *cell* a qual armazena o estado interno do objeto. Operações de leitura e alteração sobre o estado do objeto são executadas através do Access e Exchange. Métodos são representados por procedimentos.

Exemplo:

```
class Counter           %Define a classe
    attr val:0          %Atributo com valor inicial 0
```

```
    meth inc ( )        %Declaração de método
        val := @val + 1
    end
    meth get(X)         %Method com um argumento
        X = @val
    end
    meth reset
        val := 0
    end
end
```

A classe *Counter* definida acima tem um único atributo *val* que é iniciado com zero. Possui três métodos *inc*(), *get*(*X*), *reset*. Os parênteses são opcionais quando não se tem parâmetro. Atribuição de valor ocorre através da notação := e o acesso pelo símbolo @. A instanciação do objeto e a chamada de métodos ocorre conforme mostra o exemplo abaixo:

```
C = {New Lcounter} %Cria a instância
{C inc}            %Incrementa @val
{C get(X)}         %Pega o valor de @val para X
```

Porta é um canal assíncrono que suporta ordenação de comunicação de N para um e de N para N [Roy et al., 1997]. Uma porta consiste de uma procedimento *send* e uma lista. Ao final desta lista sempre está uma variável lógica. *send* é acionado assincronamente por diversas threads. Não há garantia de ordem nas mensagens recebidas entre as threads, porém respeita-se a ordem interna do envio de mensagens individual de cada thread. Se determinada thread T1 enviou duas mensagens T1M1 e T1M2 respectivamente, e outra thread T1 paralelamente a esta enviou mais duas mensagens T2M1 e T2M2 também nesta ordem, é garantido somente que T1M1 preceda T1M2 e T2M1 preceda T2M2. Sendo assim poderíamos ter seqüências como T2M1, T1M1, T2M2, T1M2 ou T2M1, T2M2, T1M1, T1M2 por exemplo. Sempre que uma nova mensagem é recebida na porta, atribui-se ao final da lista uma nova entrada e uma variável lógica. Caso o valor da variável lógica ainda não seja conhecido, várias threads podem estar aguardando o preenchimento desta informação. Assim que a atribuição seja executada o valor estará disponível para todas elas. Desde que

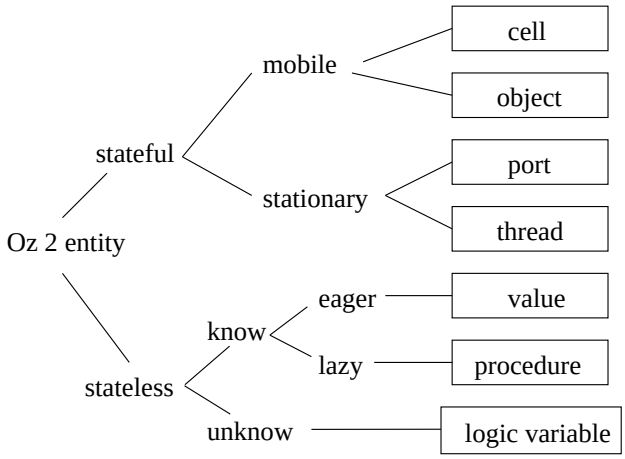


Figura 2.5 Classificação das entidades quanto a distribuição.

a lista seja *stateless* poderá ser suportado qualquer número de acessos de consulta a ela de forma rápida.

2.5.4 Comportamento Distribuído

Cada entidade tem um comportamento distribuído bem definido pela Distributed Oz. A distribuição semântica de cada entidade está resumida na Figura 2.5. Todas as entidades *stateless* são replicadas, pois nunca mudam durante seu ciclo de vida. Isto é feito objetivando incrementar o desempenho por reduzir o tráfego da rede evitando Chamadas Remotas de Procedimentos (CRP). A forma como ocorrerá a replicação destas entidades pode ser determinada conforme o desejo do programador. Ele pode escolher entre replicação *lazy* ou *eager*. Caso não seja determinado pelo programa, o sistema replica records, números e literais de forma *eager*, e procedimentos de forma *lazy*. Procedimentos e *closures* possuem uma identidade global no sistema, garantindo que não haverá mais que uma cópia de cada localmente. Esta identidade global é utilizada para que blocos de códigos sejam transmitidos somente onde eles ainda não existam. O mesmo tratamento não acontece com os números, literais e records.

As Variáveis Lógicas referenciam valores que ainda não são conhecidos. Uma vez que este valor não possa ser alterado indefinidamente a variável lógica é considerada *stateless*. Há duas operações sobre variáveis lógicas: a atribuição de valor ou no aguardo por um

valor[Roy et al., 1997]. Atribuindo-se um valor à variável de forma *eager*, este será repassado a todos os nodos onde exista uma referência. Este protocolo de atribuição é chamado protocolo de eliminação de variável. Quando é atribuído um registro para a variável lógica, este – o registro – é replicado de forma *eager*, quando atribui-se um procedimento, o nome desta também é replicado de forma *eager* . O *closure* utiliza o protocolo *lazy*.

Todas as entidades $statefull$ (*cell*, objeto, *port* e thread) são caracterizadas por sempre possuírem seu *home site*. O controle de mobilidade define o que irá acontecer no *home site* a cada operação sobre a entidade $statefull$. Uma entidade $statefull$ pode ser referenciada como móvel ou estacionária dependendo de seu $state - updating$ [Roy et al., 1997]. Ficou definido como padrão que os $states - updating$ da *cell* e do objeto são móveis e da porta e thread são estacionários.

Cells: A *cell* é móvel. Todos os nodos que conhecem o nome da *cell* podem acessá-la. A invocação de um Exchance causa um movimento síncrono do conteúdo da variável para o seu nodo. E a invocação de um $cell - access$ somente replica o valor da *cell*, sem que o *home site* seja alterado. A *cell* somente pode ser atualizada em seu *home site*.

Objects: O objeto é móvel, e sua distribuição obedece o comportamento distribuído do OPM que o compõe. Quando um método é acionado remotamente, os procedimentos correspondentes ao objeto e ao método são replicados. O método é então executado no *site* acionador. Quando o estado do objeto é alterado, todo ele é enviado para o *site* requisitante. De forma que o *home site* do objeto é alterado para que se possa efetuar localmente a alteração.

Portas: A porta é estacionária. O envio de mensagens vindas de qualquer *site* causam a aparição de uma nova entrada na lista da porta. A operação de *send* (envio de mensagem) é definida como assíncrona e ordenada.

Threads Threads são estacionárias. A computação das atividades de uma thread é executada localmente em seu *home site*. A thread nunca muda de local desde sua criação. Comandos vindos de outros *sites* que não o *home site* são enviados sincronamente e ordenados como um RPC.

Com as definições do comportamento distribuído de cada entidade pode-se prever o comportamento do sistema distribuído como um todo.

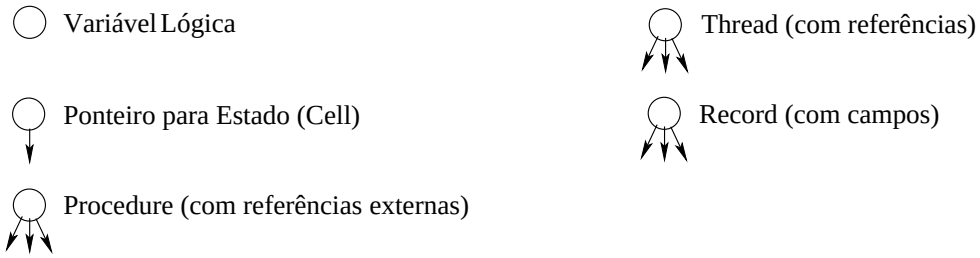


Figura 2.6 Representação gráfica das entities.

2.5.5 Gráfico de Linguagem e Distribuição

Para facilitar o entendimento, demonstraremos as entidades do OPM através do gráfico de linguagem. Cada entidade do OPM é representada por um nodo, exceto as entidades compostas (portas e objetos) como mostrado na Figura 2.6. Cada nodo pode ser considerado como uma entidade ativa com um estado interno, que tem o poder de enviar mensagens assincronamente para outros nodos. Objetos são entidades compostas e como tais são representadas com um conjunto de nomes.

Para representarmos o ambiente distribuído é necessário introduzir o conceito de *site*. Os nodos são dispostos em um conjunto finito de *sites* com referências entre eles. Um nodo referenciado por outro nodo em um *site* diferente do seu, necessariamente possui uma estrutura de mapeamento para esta referência. No exemplo da Figura 2.7, demosntra-se este mapeamento. O nodo N2 é referenciado por outro dois nodos remotamente, N1 e N3 (Gráfico de Linguagem). No Gráfico de Distribuição é possível visualizar as estruras de referências de acesso criadas. No caso do exemplo citado, teremos para N2 o *access struct* {P1,P2,P3,M}. Definimos com *access struct* as entidades necessárias para este mapeamento. Para tanto teremos um nodo proxy Pi para cada diferente *site* e um nodo *manager* M para toda a estrutura. O gráfico onde se demonstra o *access struct* é denominado Gráfico de Distribuição [Roy et al., 1997] .

2.5.6 Mecanismo de Migração

No gráfico distribuído, o objeto é mostrado como uma entidade composta, formada a partir de um registro de objeto, um registro de classe contendo procedimentos (métodos), um ponteiro

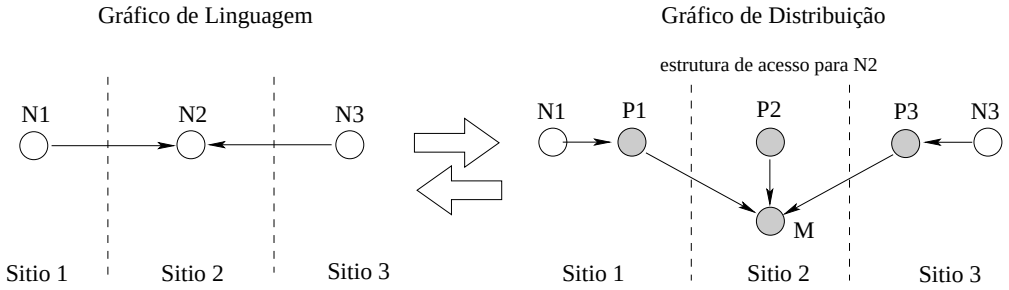


Figura 2.7 Gráfico de linguagem versus gráfico de distribuição.

```
class Account
    attr bal:0
    meth trans (Amt)
        bal<− @bal + Amt
    meth getBal(B)
        B = @bal
    end
end
A={New Account trans(100)}
```

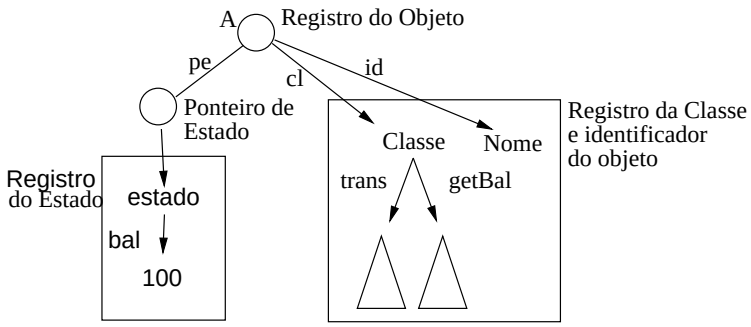


Figura 2.8 Visualização de um objeto com um atributo e dois métodos.

para estado e um registro para o estado do objeto. O comportamento do objeto é derivado do comportamento de suas partes. Na Figura 2.8 é mostrada uma classe A contendo um atributo *bal* e dois métodos *trans* e $getBal$. O campo *pe* contém o ponteiro para o ponteiro do estado do objeto. O ponteiro de estado define o *site* em que operações de atualização podem ser feitas sem operações de rede. O campo *cl* contém o registro da classe, onde estão as procedimentos *trans* e $getBal$. O campo *id* contém o identificador único do objeto $Nome$ [Roy et al., 1999].

Na Figura 2.9 um objeto A está no *site* 1. Não existe qualquer referência remota a ele. Em seguida na figura 2.10 já existe um *access struct* formado por $\{Pa2, Ma\}$. O objeto A agora é um objeto Global. Quando uma mensagem referenciando o objeto A deixa o *site* 1, um nodo de gerenciamento é criado, e quando a mensagem chega ao *site* 2 é criado um nodo proxy $Pa2$ para o objeto.

Digamos que exista uma thread T referenciando o ponteiro $Pa2$, e que T executará uma operação de atualização sobre o estado do objeto A. Como descrito no tópico anterior,

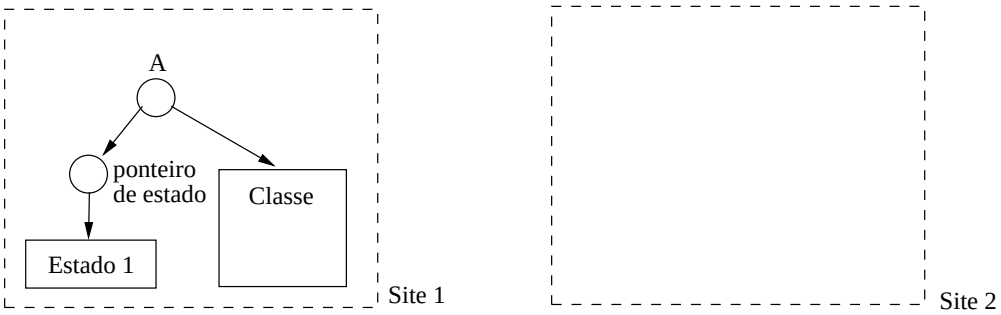


Figura 2.9 Objeto local - sem referências remotas.

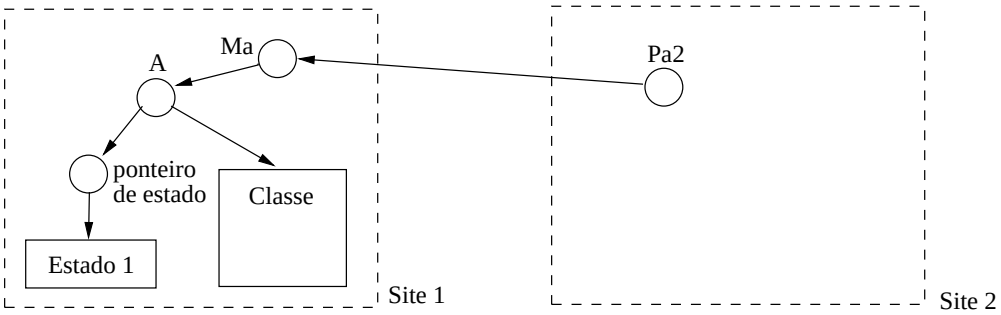


Figura 2.10 Objeto global com uma referência remota.

a thread é estacionária e o objeto móvel. Como todas as atualizações são sempre feitas localmente é necessária a transferência de A para o *site* 2. Para isto, o proxy $Pa2$ solicita ao nodo de gerenciamento Ma uma cópia do registro do objeto. A classe deve ser copiada de imediato. O registro da classe e um ponteiro proxy para o estado é enviado para o *site* 2 (veja Figura 2.11).

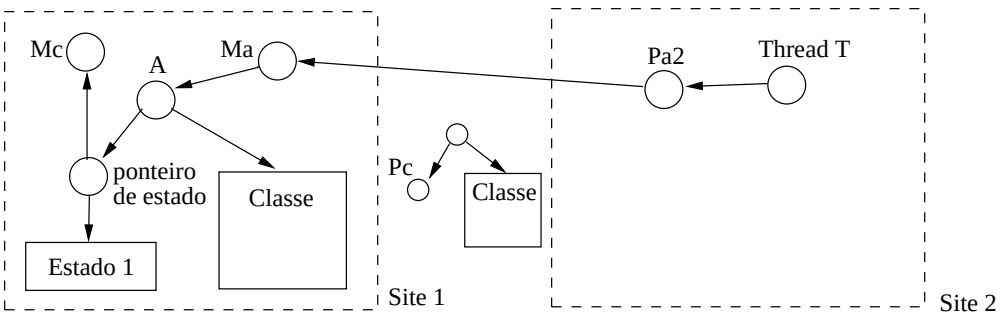


Figura 2.11 Invocação remota sobre o objeto (1).

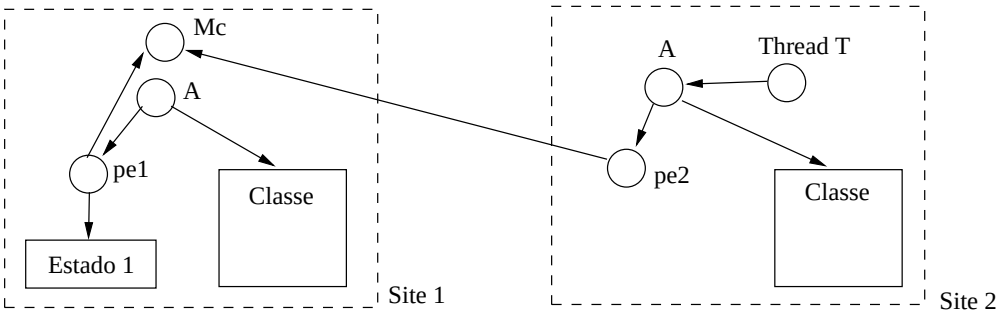


Figura 2.12 Invocação remota sobre o objeto (2).

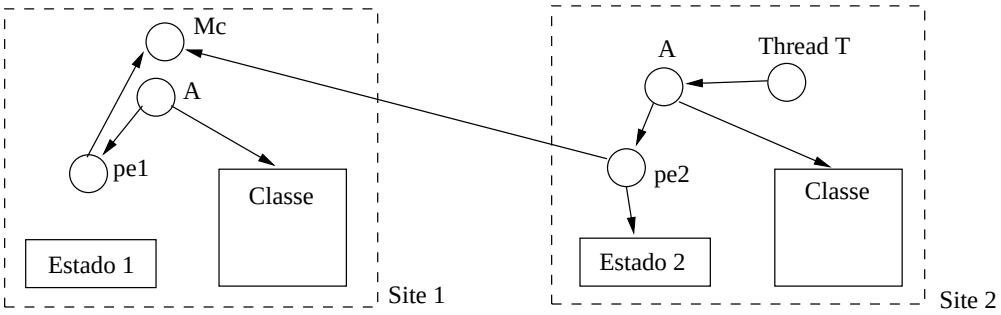


Figura 2.13 Invocação remota sobre o objeto (3).

Com a chegada da mensagem, um segundo proxy $pe2$ é criado. A classe é copiada para o *site* 2 e o proxy $Pa2$ torna-se o registro do objeto A. Então o protocolo de mobilidade de estado, que veremos na seqüência, transfere o ponteiro de estado para o *site* 2. O registro do objeto tem um nome global. Isto implica que todas as mensagem direcionadas para A no *site* 1, devem ser enviadas diretamente para o *site* 2.

A Figura 2.13 mostra a conclusão da operação. O novo estado permanece no *site* 2. O ponteiro de estado do *site* 1 é redirecionado para o Mc (nodo gerente).

Caso haja a invocação de um método no *site* 1 que atualiza o estado do objeto A, o ponteiro de estado é transferido novamente para o *site* 1. O novo estado 3 é criado no *site* 1. O estado antigo do objeto (estado 2) permanecerá no *site* 2, mas o ponteiro $Pc2$ já não apontará mais para ele (ver Figura 2.14) [Roy et al., 1999].

Temos algumas observações importantes a fazer sobre este mecanismo.

- O método do objeto é sempre executado localmente.

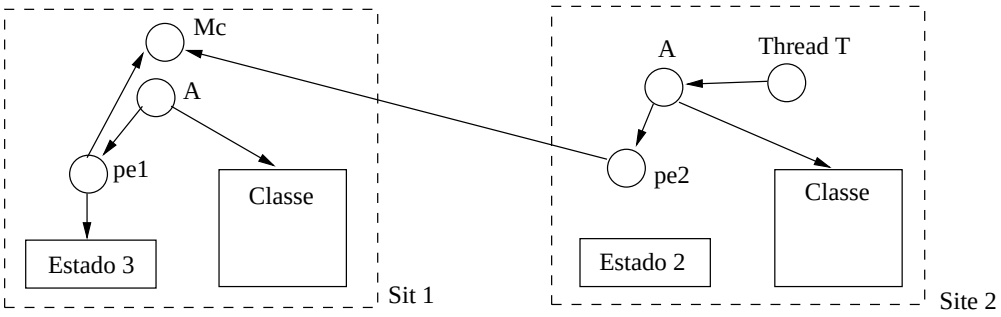


Figura 2.14 Visualização de um objeto com um atributo e dois métodos.

- A classe é enviada somente uma vez.
- Nas demais movimentações somente o ponteiro de estado é movimentado reduzindo o custo da mobilidade.
- Todas as requisições são direcionadas para o nodo gerenciador do ponteiro de estado. Isto simplifica o protocolo mas cria a dependência do *site* do nodo gerenciador.
- É possível tornar um objeto estacionário, através do encapsulamente do objeto por uma porta, porém tornar o objeto estacionário implica na criação de threads remotas, sincronização entre diferentes *sites*, e passagens de exceções entre estes. Conceitualmente é suficiente para sistemas distribuídos termos threads móveis ou objetos móveis [Roy et al., 1999].

O protocolo de mobilidade de estados deve garantir consistência entre estados consecutivos. Se estes estados estão em diferentes nodos, então é necessária uma transferência atômica do nodo gerenciador do ponteiro de estado entre os nodos. Quando um *site* solicita o ponteiro de estado, este deve fazê-lo para o gerenciador do ponteiro, e este envia uma mensagem para o *site* que tem, ou tinha, o ponteiro para o estado. Conseqüentemente, o gerenciador somente precisa armazenar o nome do *site* que eventualmente possui o ponteiro de estado [Roy et al., 1997].

Na Figura 2.15 o ponteiro de estado *pe*1 é referenciado a partir de dois *sites*. Inicialmente o estado está localizado no *site* 1. A thread T executa localmente um método que atualizará o estado do objeto. Neste momento T solicilita uma atualização no estado do objeto enviando uma mensagem para *pe*2. A thread referencia o novo estado através da variável Y. Quando a atualização finalizar, T também referenciará o antigo estado através da variável X. Como o proxy *pe*2 não possui o ponteiro de estado, ele deve solicita-lo ao seu nodo gerente.

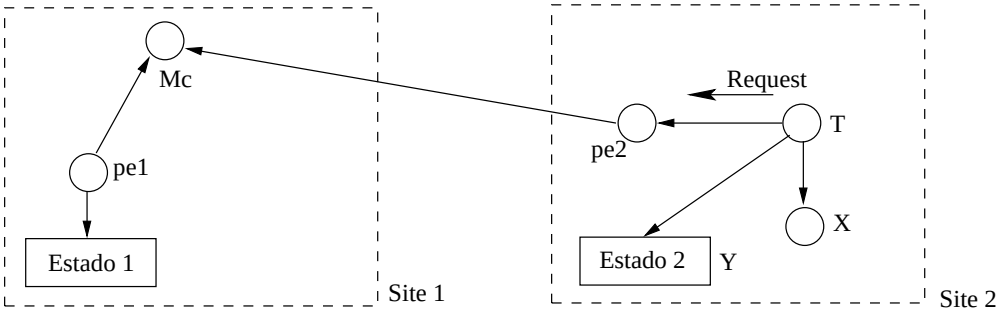


Figura 2.15 Thread requisita a atualização do estado apontado por pe2.

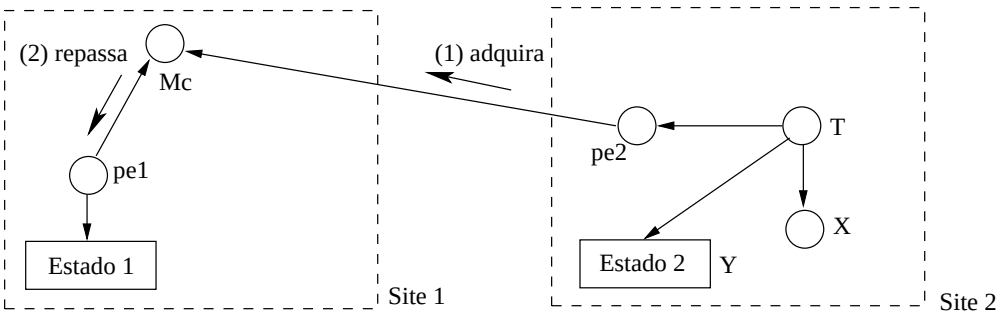


Figura 2.16 (1)Mensagem de aquisição do estado. (b)Redirecionamento da mensagem de aquisição.

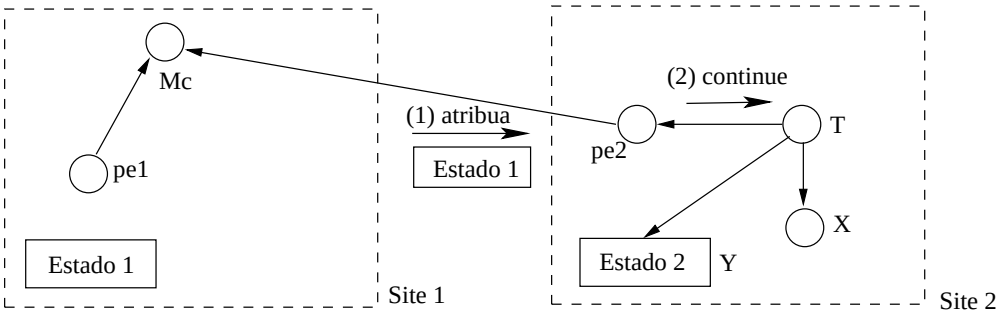


Figura 2.17 (1)Repasse do ponteiro de estado. (2) Mensagem para prosseguimento de transação.

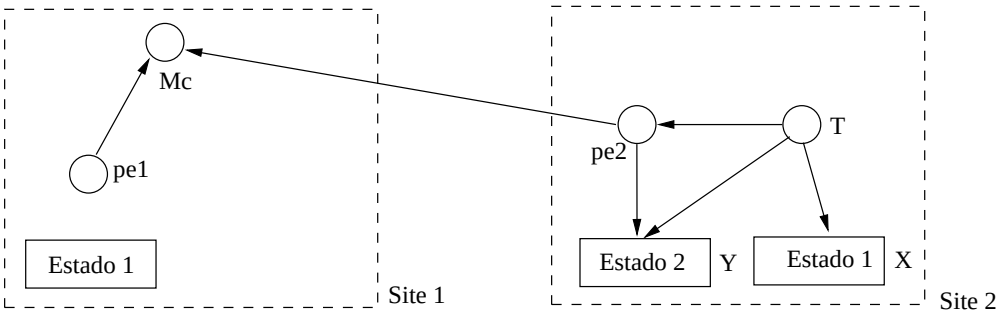


Figura 2.18 Fim da movimentação do estado do objeto para o *site* 2.

Na Figura 2.16(1) $pe2$ solicita o ponteiro de estado enviando um mensagem *get* para para o gerenciador Mc, e (2) o gerenciador envia uma mensagem de $forwarding$ para o proxy que eventualmente vai ter o ponteiro de estado ($pe1$). O gerenciador pode aceitar outra requisição imediatamente, não precisando esperar até o ponteiro de estado ser transferido completamente [Roy et al., 1999].

Na seqüência $pe1$ envia para $pe2$ uma mensagem *put* contendo o estado antigo, Estado 1 (Figura 2.17), e o $pe2$ envia para T uma mensagem *proceed* informando que a transferência esta completa. O estado antigo continua existindo no nodo para $pe1$. Mas $pe1$ não mais o referencia. A Figura 2.18 mostra a situação final da transação.

O protocolo descrito oferece um comportamento de rede previsível. São necessários no máximo três pontos de rede para que o ponteiro de estado mude de *site*. Somente dois se o gerenciador estiver no mesmo local que o estado e nenhum se o ponteiro de estado estiver no mesmo *site* que o requisitante. A alteração do estado do objeto é feita em uma ordem consistente global.

2.6 CORBA

2.6.1 Introdução

CORBA (*Common Object Resquest Broker Architecture*) é uma arquitetura aberta para computação distribuída orientada a objetos. É mantida e padronizada pela *OMG*[2]. CORBA automatiza muitas atividades comuns na programação em ambientes distribuídos como registros de objetos, localização, *parameter marshalling* e RPC.

Nas especificações CORBA sobre **Mobile Agent Facility**(MAF) [OMG, 2000] e **Life Cycle Service** [OMG, 2002] constam as definições sobre a mobilidade de objetos e agentes. Desde de que um agente é em princípio um objeto com atividades, focaremos no estudo da mobilidade de agentes em CORBA por este ser mais complexo e interessante ao objetivo deste trabalho. A especificação MAF engloba os aspectos essencias que precisam ser padronizados para que seja possível a operacionalização dos agentes:

Gerenciamento: Sistema administrador que, através de funções comuns, pode gerenciar agentes de diferentes tipos.

Transferência: O processo de mobilidade do agente.

Nomes: Atribuição de nomes aos agentes.

Tipos: A identificação do *tipo do agente*[3], permite a verificação se a plataforma do nodo destino suporta o tipo a ser migrado.

Localização: A sintaxe da localização precisa ser comum para que seja compreendida por todos os agentes.

Como o objetivo deste trabalho é a mobilidade de objetos, veremos somente os aspectos que são específicos da transferência de *agentes móveis*[4], não serão vistos os demais aspectos como segurança, autenticação, escolha de protocolos de rede, sistema de regiões entre outros.

[2] *Object Managment Group* é um consórcio aberto sem fins lucrativos que produz e mantém especificações para a interoperabilidade de aplicações comerciais.

[3] Tipo do agente: É a identificação do perfil do agente. Por exemplo um tipo *Aglet*, é implementado pelo sistema de agente da IBM, implementa Java como linguagem e utiliza *Java Object Serialization*

[4] Um agente pode ser móvel ou estacionário. Um agente móvel é caracterizado pelo capacidade de mover

2.6.2 Transferência

Quando um agente se transfere para outro *sistema de agente*[5], inicia-se uma requisição de migração. Junto à requisição, o agente envia o nome e o endereço do destino. O agente também especifica a qualidade do serviço de comunicação requerida para a transferência do agente. O MAF não especifica como deve ser a comunicação, deixando a critério do sistema implementador do agente.

Se o agente conseguir comunicar com o sistema de agente destino, este deve completar a transferência ou retornar um indicador de falha para o agente. Se o agente não conseguir comunicar com o sistema de agente destino, então o indicador de falha deve ser retornado ao sistema de agentes origem.

Se o sistema de agente destino concordar com a transferência, as credenciais de segurança, o estado do agente, e se necessário o seu código são transferidos. O sistema de agente destino reativa o agente, e continua sua execução. A Figura 2.19 mostra a relação de encapsulamento das estruturas da arquitetura CORBA e a conexão entre dois sistemas de agente.

Segue o detalhamento dos passos de inicialização da transferência, recepção do agente e transferência de classes, necessários à transferência de um agente.

A inicialização da transferência ocorre pela identificação do local do sistema de agente destino. Se o local não for especificado, o local $default$ do sistema de agente destino será selecionado. Após estabelecida a conexão, o agente solicita ao sistema de agente origem a sua transferência. Então o sistema de agente executa as seguintes ações:

(1) Suspende as atividades do agente.

(2) Identifica as partes do estado do agente que serão transferidas.

(3) Serializa a instância da classe e do estado do agente

(4) Codifica o agente serializado conforme o protocolo de transporte.

---

a si próprio para outros locais. Um agente estacionário é executado somente no sistema em que foi criado. Utiliza basicamente serviços de comunicação e transporte quando da necessidade de interação remota com outros agentes, serviços ou dados.

[5]sistema de agente: É uma plataforma que pode criar, interpretar, executar, transferir, e finalizar agentes. Um nodo pode conter um ou mais sistemas de agente

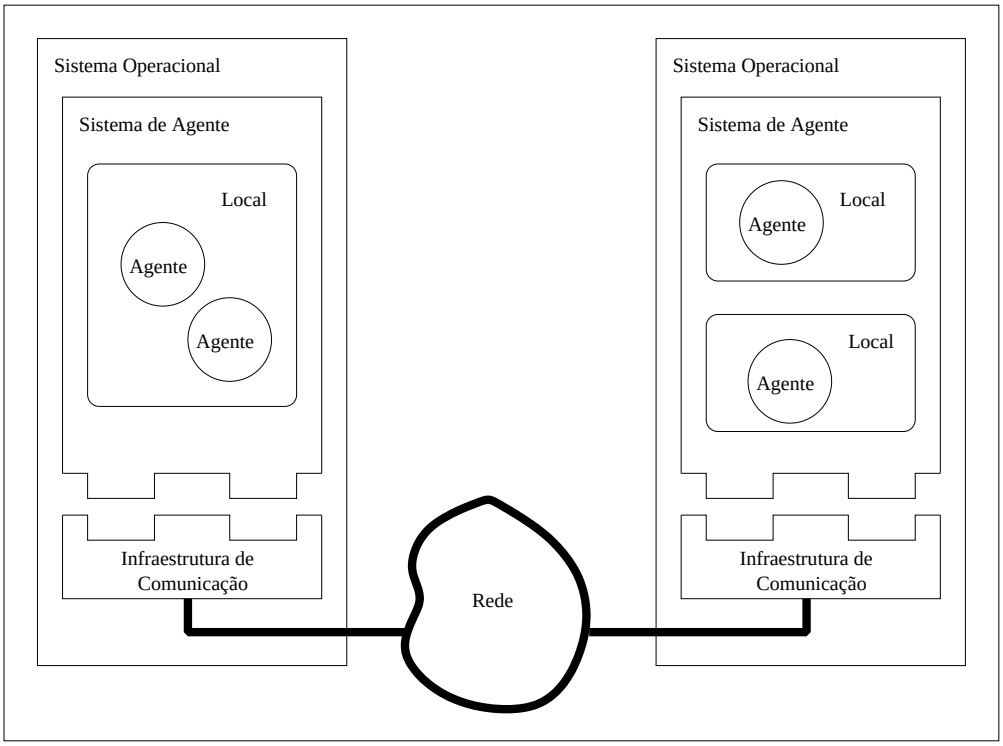


Figura 2.19 Interconexão entre dois sistemas de agente.

(5) Autentica o cliente.

(6) Transfere o agente.

Antes da recepção do agente pelo sistema de agente destino, este verifica se é possível interpretar o agente recebido. Se a verificação for positiva então procede-se com as seguintes ações:

(1) Autentica o cliente.

(2) Decodifica o agente.

(3) Desserializa o estado e a classe do agente.

(4) Instancia o agente.

(5) Restaura o estado do agente.

(6) Dá continuidade à execução do agente.

A transferência de classes é específica para agentes construídos sobre a orientação a objetos – não necessariamente os agentes em CORBA são objetos – quando a classe não

existe no sistema de agente destino. Existem quatro tipos possíveis de implementações para a transferência de classes:

Transferência automática de todas as classes: É o envio de todas as classes que o agente irá precisar para sua criação. Isto elimina a necessidade de solicitações posteriores.

Transferência automática somente da classe do agente, outras classes são enviadas conforme demanda: Somente a classe do agente é enviada, outras classes que o agente precise são requisitadas conforme demanda. Com isso a utilização da rede é otimizada, uma vez que as classes que o agente usará no sistema de agente destino podem existir localmente no destino eliminando o custo da transferência. Porém, se o sistema de agente destino posteriormente não tiver acesso as classes necessárias, ocorrerá uma falha.

Transferência da classe do agente e depois de todas das outras classes paralelamente a criação do agente: Quando a criação de um agente remota é ativada por um cliente que não está conectado permanentemente, por exemplo um notebook ou outro dispositivo portátil que não permaneça conectado.

Transferência de uma lista dos nomes das possíveis classes junto com requisição de transferência: O sistema de agente origem envia uma lista das classes necessárias para a criação e a execução de operações específicas do agente. Então o sistema de agente destino solicita somente as classes que não estão carregadas localmente. Esta técnica é eficiente, porém requer que o sistema de agente origem conheça as classes que o agente utiliza.

2.6.3 Considerações

CORBA têm por objetivo definir padrões para interoperabilidade de sistemas distribuídos abertos. Não é objetivo definir a implementação dos sistemas que adotam a sua arquitetura. Por esta razão não encontraremos nas especificações de CORBA detalhes de mais baixo nível sobre mecanismo de mobilidade de objetos. Estes detalhes podem ser encontrados na descrição de implementação dos sistemas que adotam a especificação.

2.7 Aglet

2.7.1 Introdução

Aglets é uma solução para a implementação de agentes móveis desenvolvida sobre a plataforma Java. Os Aglets têm o poder de encerrar sua execução, movimentar-se de um nodo para outro, e iniciar sua execução novamente. Em sua movimentação os Aglets levam consigo o seu código e estado.

2.7.2 Implementação

O Aglet Java estende o modelo de código de rede-móvel conhecido pelos applets Java [Venners, 1997]. Os arquivos de classe de um aglet podem migrar através da rede pelo mesmo mecanismo que um applet. Mas diferentemente de um applet, quando um aglet migra ele também pode levar o seu estado. Um applet é código estático que tem o poder de ser movido pela rede de um servidor para um cliente. Um aglet é um programa Java em execução que pode mover de um servidor para outro na rede mantendo seu estado interno antes da movimentação. Um aglet Java é similar a um applet na maneira que é executado, como uma thread (ou múltiplas threads) dentro do contexto de uma aplicação Java. Para executar applets, um navegador Web dispara uma aplicação Java para abrigar quaisquer applets que possam ser encontrados, como um usuário navega de página a página. Aquela aplicação instala um gerenciador de segurança para reforçar restrições nas atividades de quaisquer applets não confiáveis. Para baixar arquivos de classe applets, a aplicação cria classes de carregamento que "sabem"como pedir arquivos de classes de um servidor HTTP. Da mesma maneira, um aglet requer uma aplicação Java servidor (um servidor aglet) para ser executado no computador antes que ele possa visitar aquele computador. Quando aglets viajam através de uma rede, eles migram de um servidor aglet para outro. Cada servidor aglet instala um gerenciador de segurança para reforçar restrições nas atividades de aglets não confiáveis. Os servidores carregam aglets através de classes de carregamento que recuperam arquivos de classes e o estado de um aglet de um servidor aglet remoto.

2.7.3 Ciclo de Vida

O ciclo de vida de um aglet pode ter vários eventos em sua vida, os quais podem ser:

Criar: Um aglet é criado pela inicialização da execução de uma thread e a criação do seu estado interno.

Copiar: Uma cópia do aglets é criada.

Enviar: Um aglet viaja para um outro servidor.

Recuperar: Um aglet, previamente enviado, é trazido novamente para um servidor remoto.

Desativar: Um aglet tem suas atividades paralizadas. Seu estado é armazenado em algum dispositivo de armazenamento, usualmente um arquivo.

Ativar: Um aglet desativado é trazido de volta para atividade.

Descartar/Eliminar: Um aglet é finalizado.

Cada atividade além da criação e eliminação envolve tanto duplicação, transmissão através da rede, ou armazenagem persistente do estado do aglet. [Oshima et al., 1998] Cada uma dessas atividades utiliza o mesmo processo de serialização para obter o estado fora de um aglet. Na figura 2.20 são mostradas as fases do ciclo de vida de um aglet.

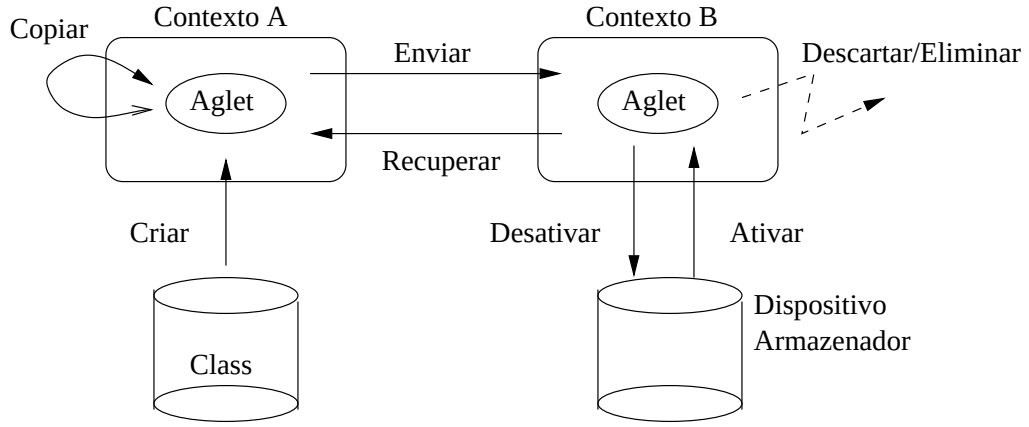


Figura 2.20 Ciclo de Vida de um aglet

2.7.4 Serialização

Os servidores aglets utilizam a serialização de objetos, disponível em Java, para exportar o estado de um objeto aglet para uma sequência de bytes. Em um processo reverso, o estado de um aglet pode ser reconstruído a partir desta sequência. Serialização permite que uma imagem do heap (o estado de um heap) seja exportada para uma sequência de bytes (tal como um arquivo) e então reconstruída daquela sequência de bytes. Porém, o estado da pilha de execução e contadores de programa das threads pertencentes ao aglet não são serializadas. Serialização de objetos diz respeito somente a dados no heap, e não na pilha ou no contador de programa. Assim, quando um aglet é enviado, clonado ou desativado, qualquer estado relevante em qualquer pilha de um aglet em execução, bem como o contador de programa corrente de qualquer thread, é perdido. Na teoria, um agente de software deveria ser capaz de migrar com todos os seus estados: heap, pilha de execução, e registradores. Alguns consideram a inabilidade de aglets para fazer isso um defeito na implementação de aglet. Esta característica de aglets originou-se da arquitetura do JVM, o qual não permite um programa acessar e manipular diretamente pilhas de execução. Isto é parte do modelo de segurança construído no JVM. A menos que haja uma alteração na JVM, aglets e qualquer outro agente baseado em Java móvel serão incapazes de levar o estado de suas pilhas de execução com eles quando os mesmos migram. Antes de ser serializado, um aglet deve colocar no heap todas as informações necessárias para que ele possa ser "ressucitado"corretamente como um aglet ativado, enviado, ou então um clone. Ele não pode deixar qualquer informação na pilha, porque esta nao será reproduzida na nova vida do aglet. Como resultado, o servidor aglet informa um aglet que ele vai ser serializado para que o aglet possa preparar-se. Quando o aglet é informado de uma possível serialização, ele deve colocar na pilha qualquer informação que ele precisará para continuar sua execução corretamente quando ele for ressucitado. Do ponto de vista prático, a inabilidade de um aglet migrar com sua pilha de execução é uma limitação razoável. Isso força o programador a pensar de uma certa maneira quando o mesmo escreve aglets. Ele deve entender o aglet como uma máquina de estado finito com o heap como um único repositório de um estado de máquina. Se em qualquer ponto na vida de um aglet pode-se saber seu estado apenas olhando no seu heap, então ele pode ser serializado em qualquer tempo. Caso contrário, deve-se ter uma maneira de gravar informação suficiente no heap antes da serialização, de maneira tal a continuar corretamente quando o aglet é "ressucitado".

2.7.5 Segurança

Sistemas de agentes móveis como aglets requerem níveis de segurança altos, pois eles representam ainda outra forma de transmitir um programa malicioso. Antes de aglets poderem ser utilizados na prática, os servidores aglets devem possuir uma infraestrutura de prevenção contra aglets não confiáveis a fim de evitar danos no servidor. Segurança é amplamene fornecida na arquitetura intrínsica do Java e nas características de segurança extra do JDK, mas alguns ataques são ainda possíveis, como ocorre também nos applets. Atualmente, os servidores aglets da IBM (Tahiti and Fiji) possuem restrições de segurança bastante severas nas atividades de qualquer aglet que não foi originado localmente.

Capítulo 3

Arquitetura da Virtuosi

Esse Capítulo especifica os conceitos de orientação a objeto suportados pela Virtuosi através da formalização do metamodelo da mesma. 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 objetos escritos na linguagem Aram[1].

3.1 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 dê 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 como tais conceitos se relacionam entre si. Na seqüência segue uma descrição resumida de algumas meta-classes da Virtuosi (o detalhamento das

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

principais meta-classes é apresentado no Apêndice A).

3.1.1 Literal

Valor Literal

Um valor literal é uma seqüência de caracteres sem semântica definida. A Figura 3.1 mostra as situações possíves para o uso de valores literais.

Referência a Literal

É a meta-classe que representa uma **referência a literal**. Uma referência a literal não pode sofrer atribuição. Também não é possível declarar uma variável local do tipo referência a literal.

3.1.2 Bloco de Dados

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.

3.1.3 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;

(2) como possuidora de atributos de variáveis enumeradas;

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

Figura 3.1 Código fonte em Aram mostrando os possíveis uso de um valor literal

(3) como tipo de uma referência a objeto;

(4) como possuidora de **invocáveis**[2]

A Figura 3.2 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 (ver seção .

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

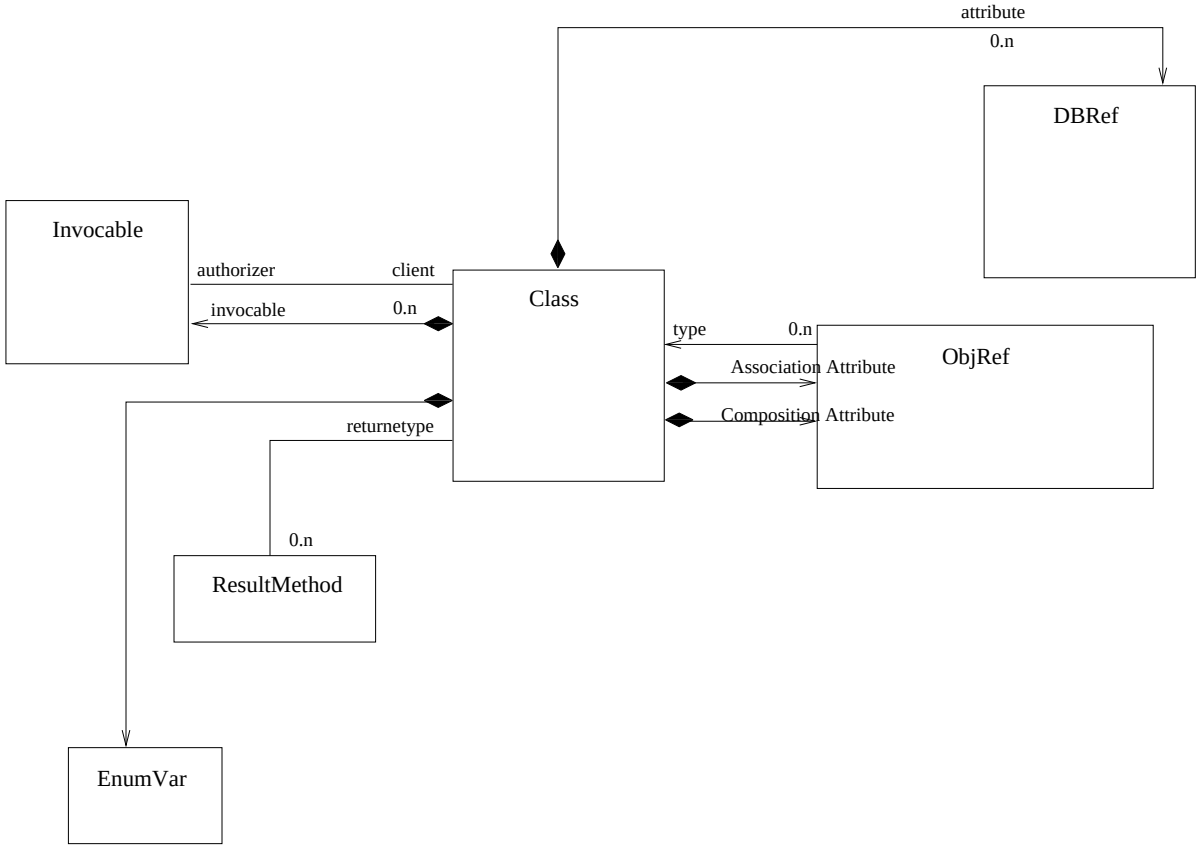


Figura 3.2 Relacionamento da meta-classe que representa uma classe de aplicação com outros componentes do metamodelo da Virtuosi

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[3], mas pode ter muitas classes herdeiras recursivamente. Entretanto, uma classe não pode ser direta ou indiretamente ancestral de si própria. Assim, um conjunto de clas-

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

ses 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.

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. 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.

3.1.4 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.

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 metamodelo da Virtuosi formaliza os três tipos de atributo de uma classe conforme mostra a Figura 3.3.

Referência a Objeto

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

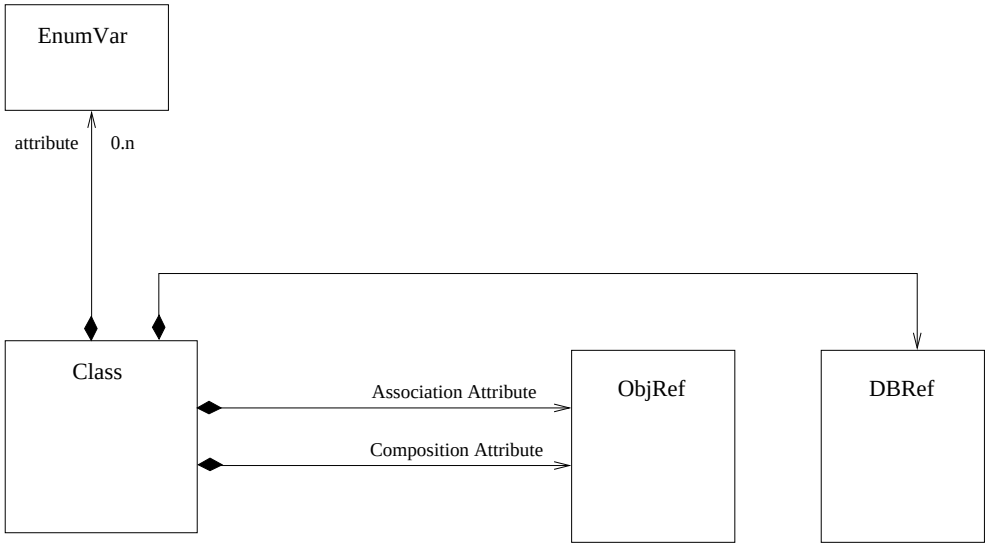


Figura 3.3 Os três tipos de atributos possíveis em uma classe segundo o metamodelo da Virtuosi

(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 3.4 mostra o relacionamento da meta-classe que representa a referência a objeto com outros componentes do metamodelo da Virtuosi.

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, e consiste em uma referência para uma seqüência contígua de dados binários em

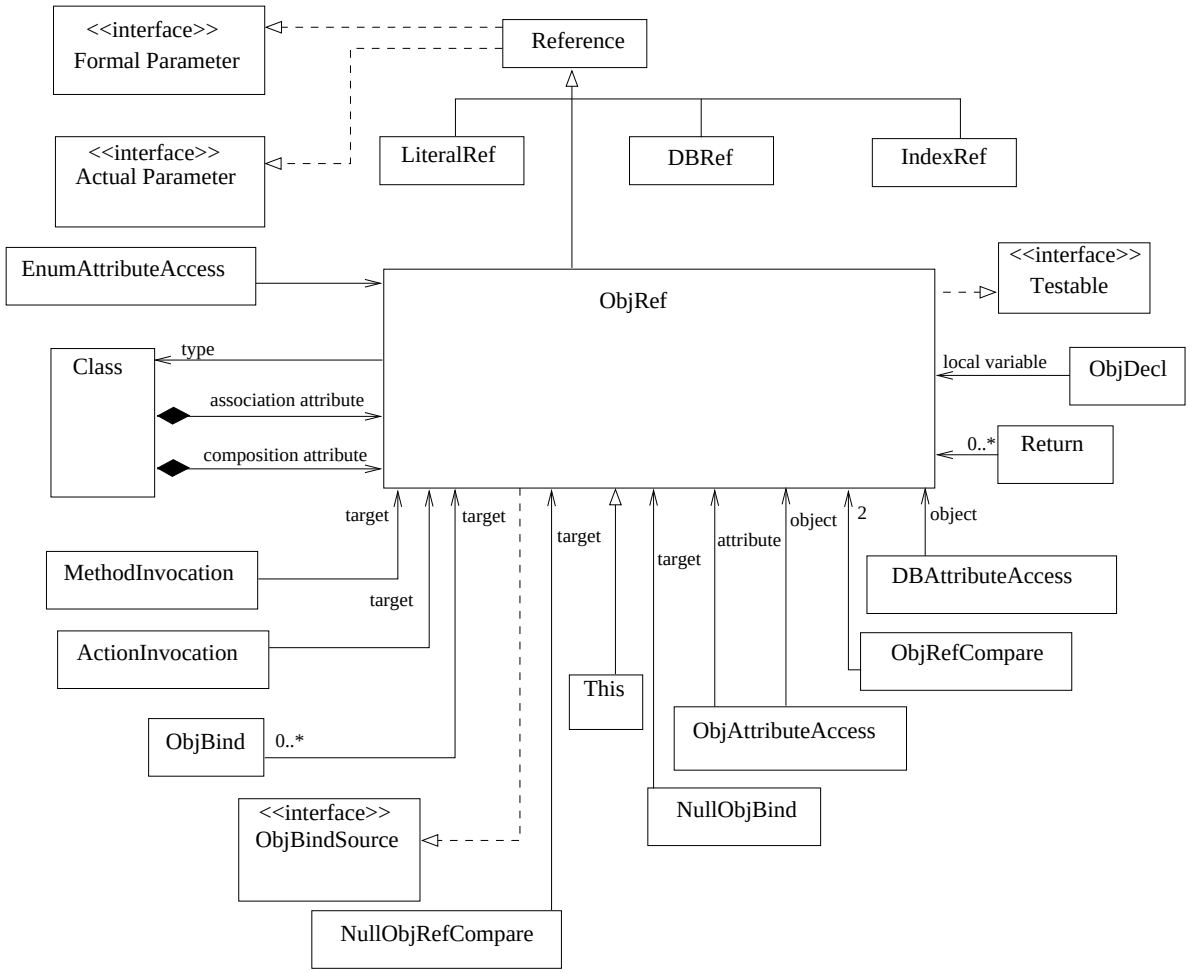


Figura 3.4 Relacionamento da meta-classe que representa a referência a objeto com outros componentes do metamodelo da Virtuosi

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 ambas as seqüências.

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.

3.1.5 Referências

Existem quatro tipos de referência suportadas pelo metamodelo da Virtuosi, a saber:

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

Um método sempre é invocado através de uma referência a objeto sobre o objeto ao 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.

3.1.6 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**.

Um método, um construtor ou uma ação podem sofrer invocação e, por isso, o metamodelo da Virtuosi os formaliza como **invocáveis**.

Construtor: Um construtor não faz parte das operações definidas por um TAD (Tipo de Dado Abstrato) 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.

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 de valor** – muitas vezes chamado de procedimento – e **método com retorno de valor** – muitas vezes chamado 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.

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**.

3.1.7 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.

Composição de Comandos

Um invocável define uma seqüência de comandos. Comandos podem ser simples ou compostos. Um **comando simples**[4] pode ser uma atribuição, uma invocação de operação, um desvio condicional, conforme detalhado no restante dessa Seção. Um **comando composto**[5] é uma seqüência de comandos simples ou compostos, recursivamente.

Uma seqüência de comandos, simples ou compostos, pode ser agrupada em um comando composto.

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.

[4]Do inglês ***simple statement***.
[5]Do inglês ***compound statement***.

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.

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.

3.1.8 Chamadas de Sistema

A Virtuosi disponibiliza diversos grupos de operações (comandos e testáveis) para a construção de classes de aplicação. Entre eles o grupo de operação de bloco de dados, o grupo de operações de entrada e saída, e o grupo de operações para a movimentação de objetos (seção 4.1).

3.2 Árvore de Programa

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 dados 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. 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 embutidos[6], em *applets* na Internet ou ainda em aplicações multi-plataforma. A portabilidade de Java é realizada através da compilação de um

[6]Do inglês, **embedded systems**

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 [Kistler and Franz, 1997]. 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 chamadas de árvores de programa.

No ambiente Virtuosi, utilizou-se a idéia de árvores sintáticas abstratas encontradas em [Kistler and Franz, 1997] 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.

3.2.1 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.

A Figura 3.5 ilustra o relacionamento entre as árvores de programa que compõem uma aplicação. Neste caso as classes correspondentes são chamadas *A*, *B*, *C* e *D*.

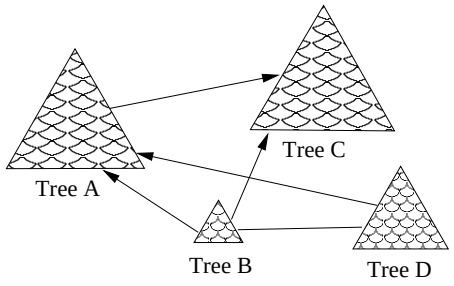


Figura 3.5 Conjunto de árvores de programa que compõem uma aplicação de software

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

Exemplo Básico

A Figura 3.6 apresenta o fragmento de código fonte da classe `Pessoa`.

A Figura 3.7 mostra a árvore de programa correspondente ao código fonte da Figura 3.6 através de um diagrama de colaboração UML.

Com base na Figura 3.7, 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 3.6 Fragmento de código fonte da classe `Pessoa`

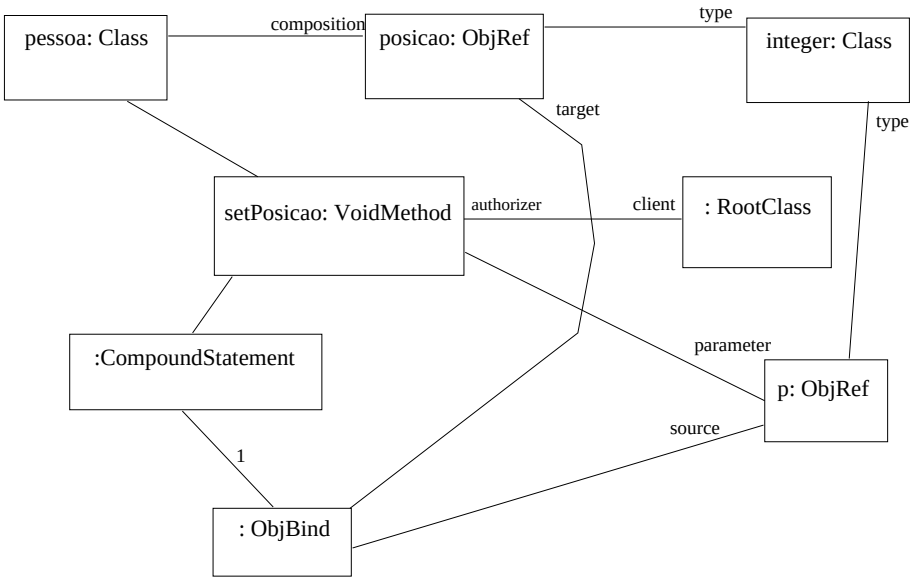


Figura 3.7 Árvore de programa parcial referente à classe Pessoa

(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` que no caso é do tipo *Integer* e tem o

papel de parâmetro do método *setPosicao*;

(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.

Exemplo Avançado

A Figura 3.8 apresenta o fragmento de código fonte da classe `Pessoa`.

A Figura 3.9 mostra a árvore de programa correspondente ao código fonte da Figura 3.8 através de um diagrama de colaboração UML.

Com base na Figura 3.9, 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`;

```
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 3.8 Fragmento de código fonte da classe `Pessoa`

(2) Associado 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

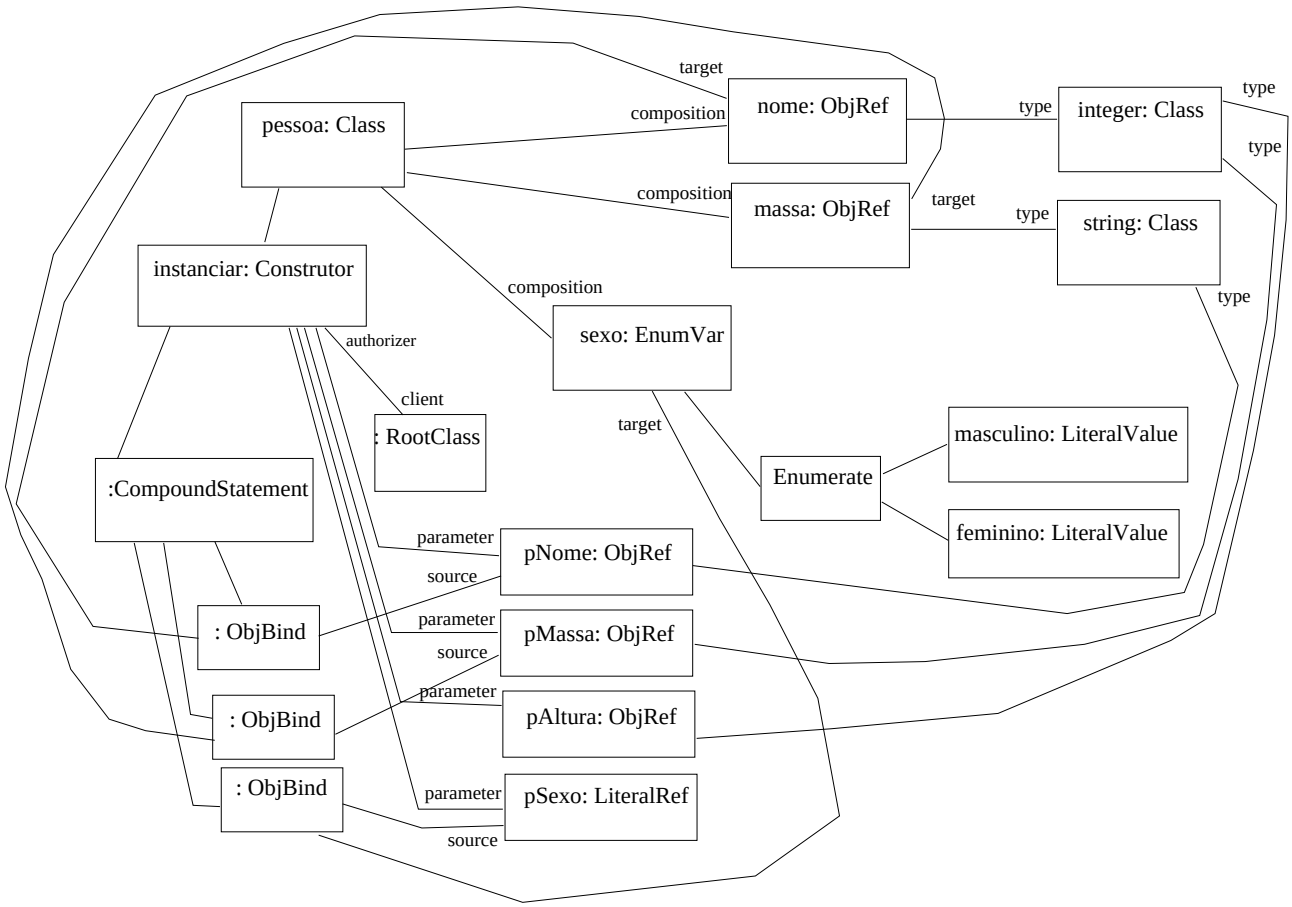


Figura 3.9 Árvore de programa parcial referente à classe Pessoa

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 repre-

senta 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 3.10 ilustra a ocorrência de referências simbólicas entre árvores de programa.

Para facilitar o trabalho do carregador de árvores (detalhado na Seção 3.3.2) na tarefa de resolver as referências simbólicas entre árvores, cada árvore de programa possui duas listas, a saber:

(1) lista de referências a classes;

(2) lista de referências a invocáveis.

3.2.2 Lista de Referências a Classe

A lista de referências a classes armazena um apontador para cada uma das classes as quais referencia.

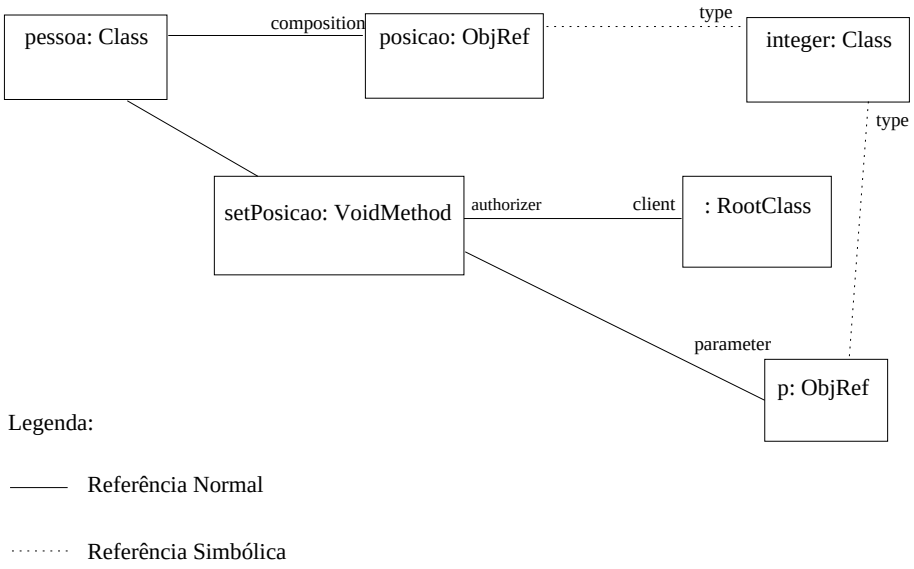


Figura 3.10 Referências simbólicas entre árvores de programa

3.2.3 Lista de Referências a Invocáveis

A lista de referências a invocáveis armazena um apontador para cada um dos invocáveis aos quais referencia.

3.3 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 **Máquina Virtual Virtuosi (MVV)**.

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 como estes interagem.

3.3.1 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 à **área de objetos**;

(3) A máquina virtual cria uma nova **atividade** (ver seção 3.3.4) – 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 atividade são empilhadas – recursivamente – na pilha de atividades em resposta à 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 da pilha de atividades e, caso não hajam outras pilhas com atividades, a aplicação é encerrada. A existência de outras pilhas de atividade é discutida na Seção 3.3.4.

3.3.2 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 árvores sempre são feitas de forma indireta, ou seja, os **pontos de ligação** entre duas árvores de programa nunca apontam diretamente para seus respectivos alvos. Essas duas ligações na perspectiva da aplicação significam, respectivamente, o tipo de um objeto e o método de uma invocação.

No caso de 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.

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 apontador para a área de memória onde a árvore está localizada. Uma árvore pode referenciar uma árvore localizada em outra máquina virtual. Portanto, existem dois tipos de entrada na tabela de classes, a saber:

- Entrada de Referência de Classe Local (ERCL);
- Entrada de Referência de Classe Remota (ERCR) – neste caso a entrada também armazena o nome da classe de aplicação, mas ao invés de possuir um apontador para uma área de memória, a entrada possui a identificação da MVV remota e a posição da entrada da tabela de árvores remota.

A Figura 3.11 ilustra a tabela de classes contendo entradas locais e entradas 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**.

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 apontador 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 Referência de Invocável Local (ERIL);

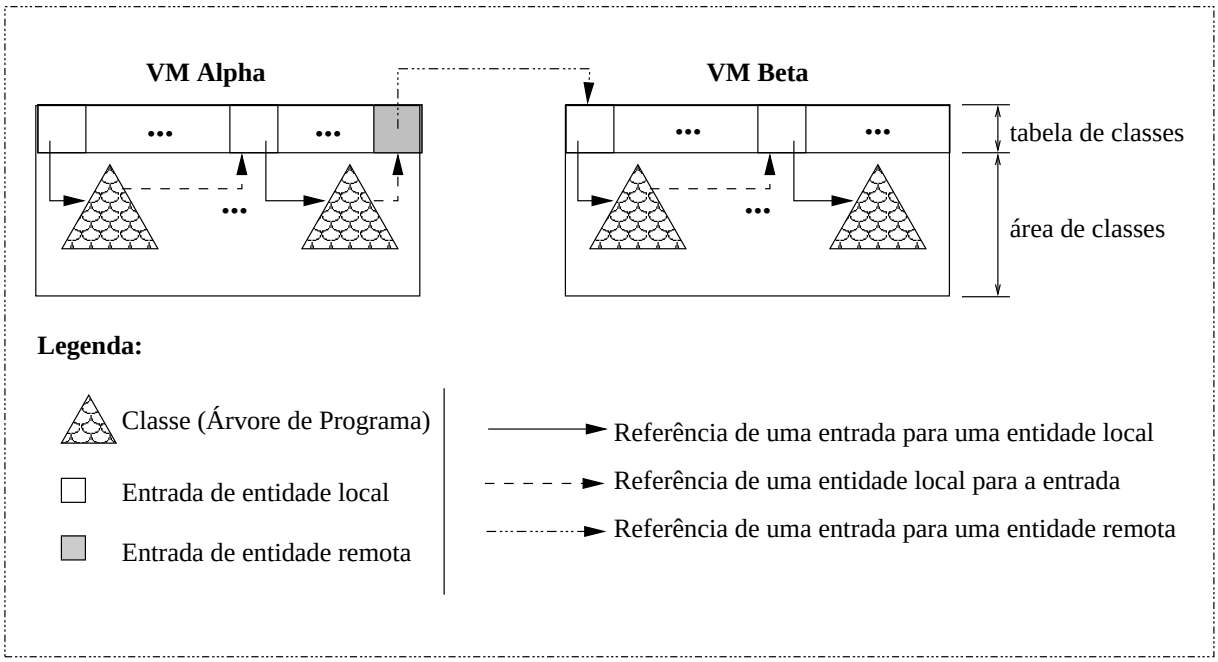


Figura 3.11 Tabelas de classes com referências locais e remotas

- Entrada de Referência de Invocável Remoto (ERIR) – neste caso, a entrada armazena a identificação da MVV remota e a posição da entrada da tabela de invocáveis remota.

A Figura 3.12 ilustra a tabela de invocáveis contendo entradas locais e entradas remotas.

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 deverá realizar 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;

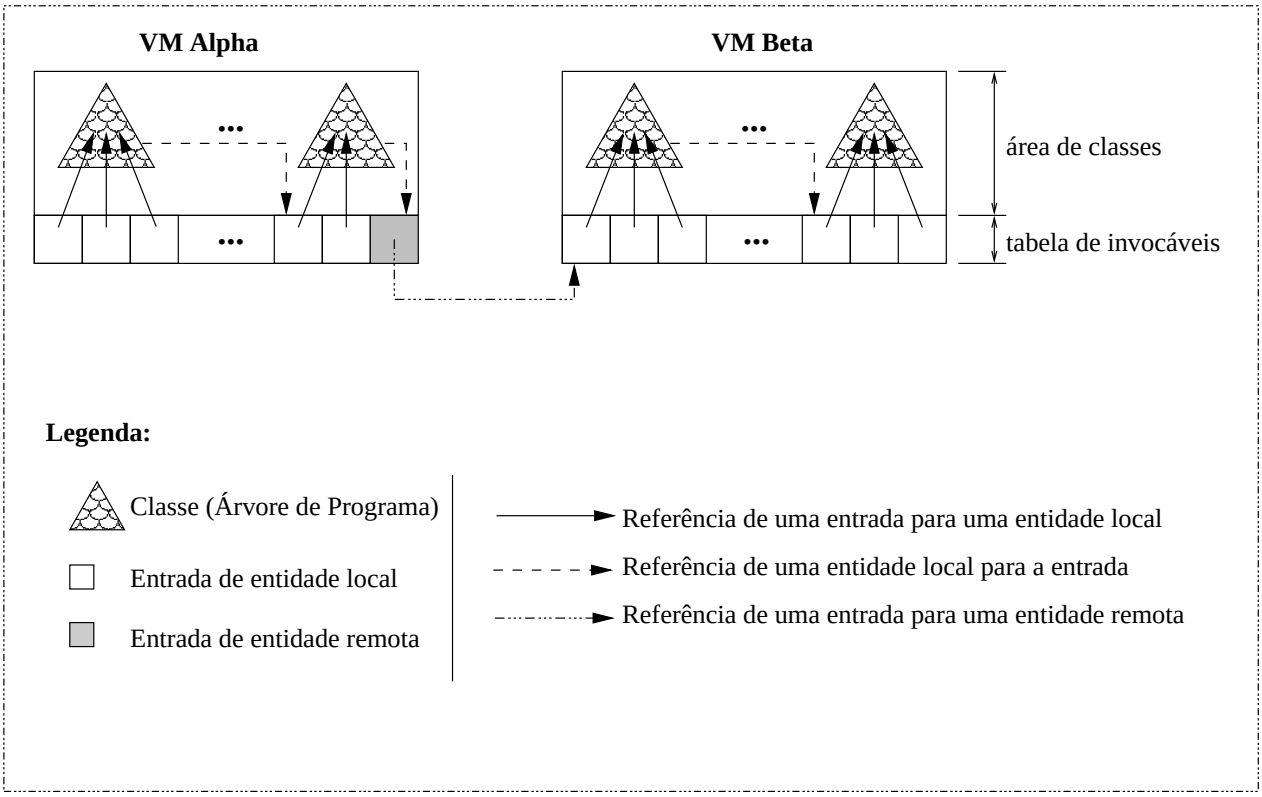


Figura 3.12 Tabelas de invocáveis com referências locais e remotas

(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 classes; 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 3.13 ilustra (de forma simplificada) a carga das árvores de programa de duas classes de aplicação situadas na mesma MVV. No caso, a classe `Pessoa` possui um atributo (uma referência a objeto) do tipo definido pela classe `Integer` e um método da classe `Pessoa`

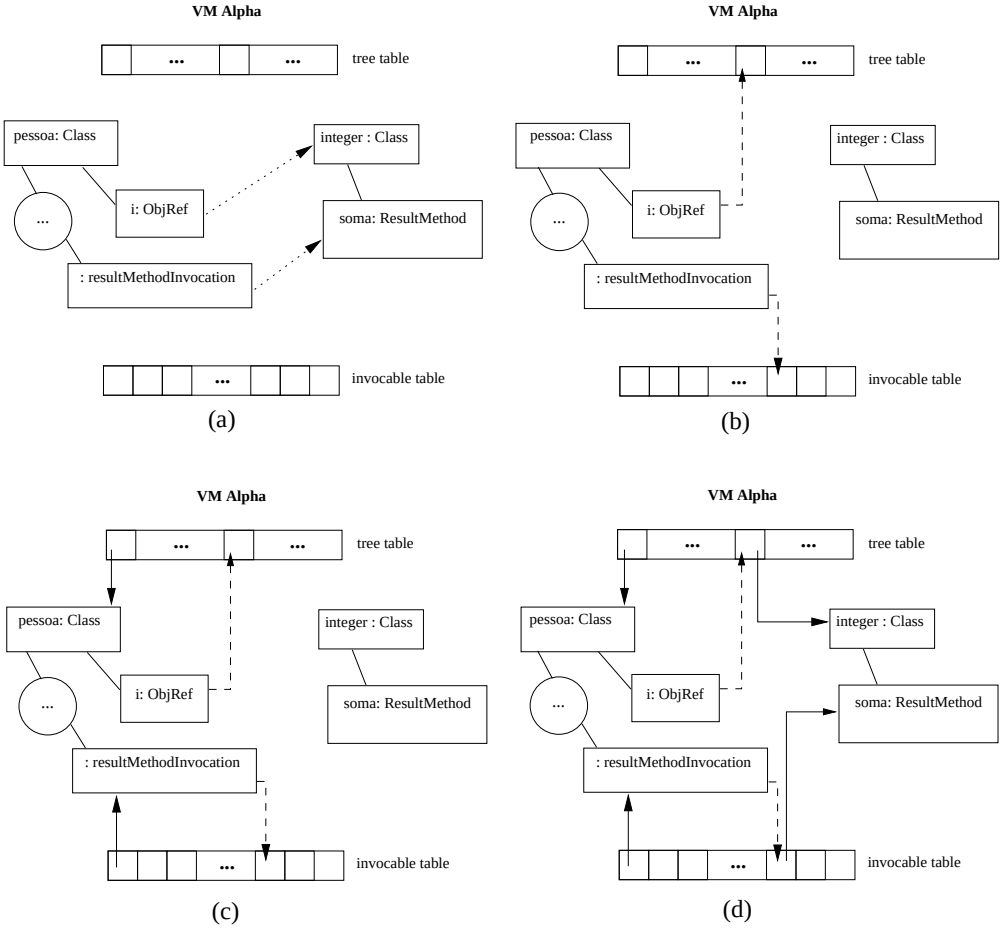


Figura 3.13 Ilustração da carga de duas árvores de programa ligadas entre si

faz uma invocação de um método sobre o atributo da classe `Integer` (soma, por exemplo). Observa-se na Figura 3.13.a a situação da classe `Pessoa` ainda não carregada e com os pontos de ligação ainda como referências simbólicas. A Figura 3.13.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 3.13.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 3.13.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 além disso, as entradas em ambas as tabelas já apontando para a árvore da classe `Integer`.

3.3.3 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. A ligação entre os objetos é feita sempre através da **tabela de objetos**. Essa estratégia é utilizada devido à natureza distribuída do ambiente Virtuosi, onde o objeto está localizado na própria instância da MVV ou remotamente em outra instância da MVV.

Tabela de Objetos

Para cada um dos objetos que a MVV instancia, há uma entrada correspondente na tabela de objetos. Cada entrada possui o nome do objeto para o qual aponta e um apontador 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 Referência de Objeto Local (EROL);
- Entrada de Referência de Objeto Remoto (EROR) – ao invés de possuir um apontador para uma área de memória, a entrada possui a identificação da MVV remota e a posição da entrada da tabela do objetos remota.

A Figura 3.14 ilustra a tabela de objetos contendo entradas locais e entradas remotas entre duas MVV.

A Entrada de Referência de Objeto Local, tem outra atribuição além de que somente referenciar o objeto. Existe um campo de um bit denomidado $freeze$ em sua estrutura. Quando o campo $freeze$ está igual a zero indica que o objeto está disponível para alteração, senão o objeto fica indisponível e as alterações devem aguardar até que o objeto esteja disponível novamente.

Para implementação das primitivas foi necessário estender a estrutura da entrada de referência para objeto local (ver Capítulo 4 - Estrutura do Objeto) incluindo um novo campo

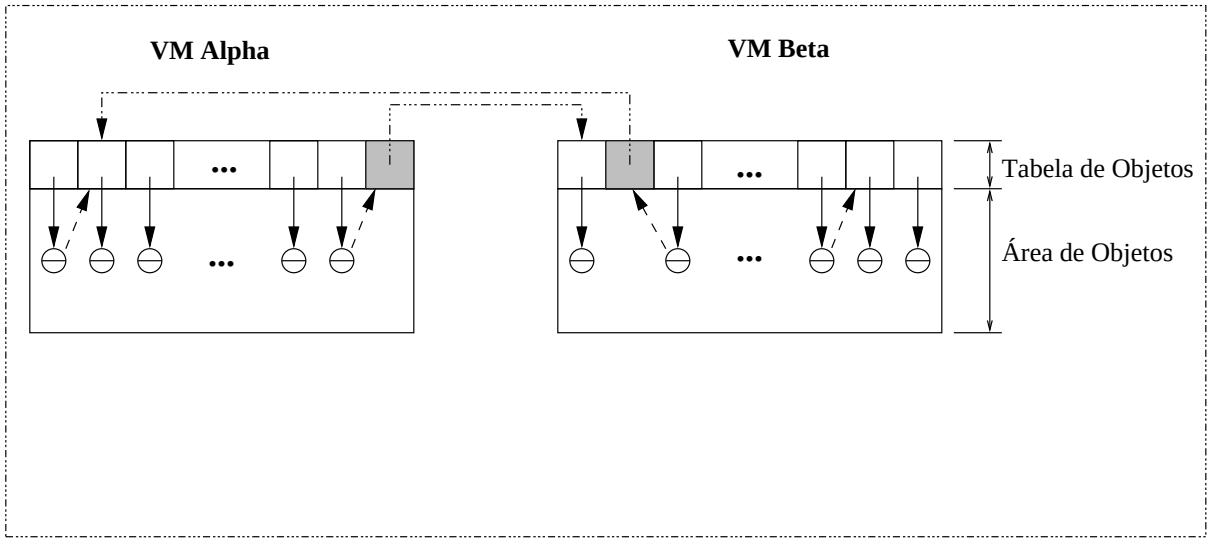


Figura 3.14 Tabelas de objetos com referências locais e remotas

de um bit denominado fix. Este campo quando diferente de zero indica que o objeto está fixo. A execução da primitiva fix atribui o valor 1 (um) para a campo, enquanto a $unfix$ atribui 0 (zero).

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)[7];

(2) um conjunto de atributos formado por apontadores para a Tabela de Objetos;

(3) um conjunto de blocos de dados;

(4) um conjunto de variáveis enumeradas;

(5) um apontador direto para a 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.

[7]O mecanismo gerenciador de nomes está em desenvolvimento no contexto da pesquisa referênte ao mecanismo de RPC

3.3.4 Área de Atividades

Conforme descrito em [Calsavara, 2000], a Virtuosi tem como forma mais básica de comunicação entre objetos 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.

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[8]). 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 um encadeamento de atividades, envolvendo um conjunto de objetos relacionados. 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 **ativividade invocada** o modo de invocação com relação ao sincronismo.

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

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

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.

No ambiente distribuído é possível que existam atividades síncronas localizadas em nodos diferentes. Para que seja possível a sincronia remota é necessário atribuir funções adicionais para as atividades:

Invocadora Remota: A atividade com esta função permanece bloqueada até que a mensagem de finalização da atividade remota síncrona a ela retorne.

Invocada Remota: A atividade com esta função retorna uma mensagem de finalização para a atividade remota subseqüente a ela.

Em determinados casos uma atividade pode ter função invocada e invocadora remota ao mesmo tempo.

Estrutura de uma Atividade Conforme supracitado, uma atividade é referente a 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:

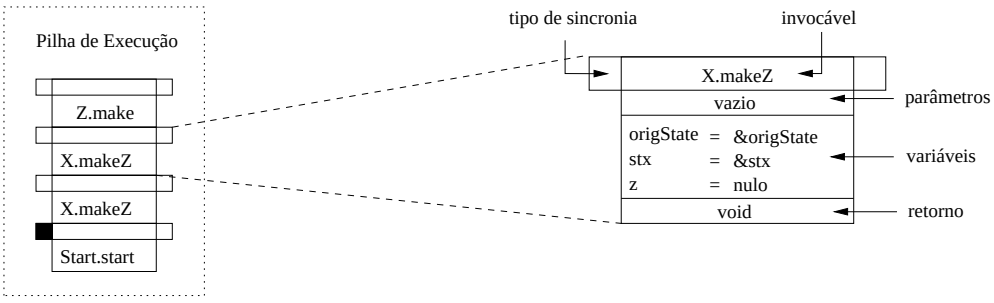


Figura 3.15 Exemplo de uma pilha de execução e o detalhe de uma atividade.

(1) um apontador para o objeto dono da atividade localizado na área de objetos;

(2) um apontador para o invocável sendo interpretado;

(3) um conjunto de parâmetros;

(4) um conjunto de variáveis locais.

Para atividades que assumem funções de invocadora ou invocada remota, é necessário adicionar à sua estrutura uma referência remota – identificador da MVV remota mais um ponteiro para a atividade. No caso da atividade que assume as duas funções, adiciona-se duas referências remotas.

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 apontador para uma entrada na tabela de objetos.

A Figura 3.15 mostra a notação gráfica para a representação de uma atividade de objeto.

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.

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.

A Figura 3.16 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 ao topo da pilha de atividades. No caso de um método com retorno uma referência a objeto é adicionada ao topo da pilha em resultado da interpretação de um comando de retorno. No caso de uma ação, o que é adicionado ao 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 resultados de teste.

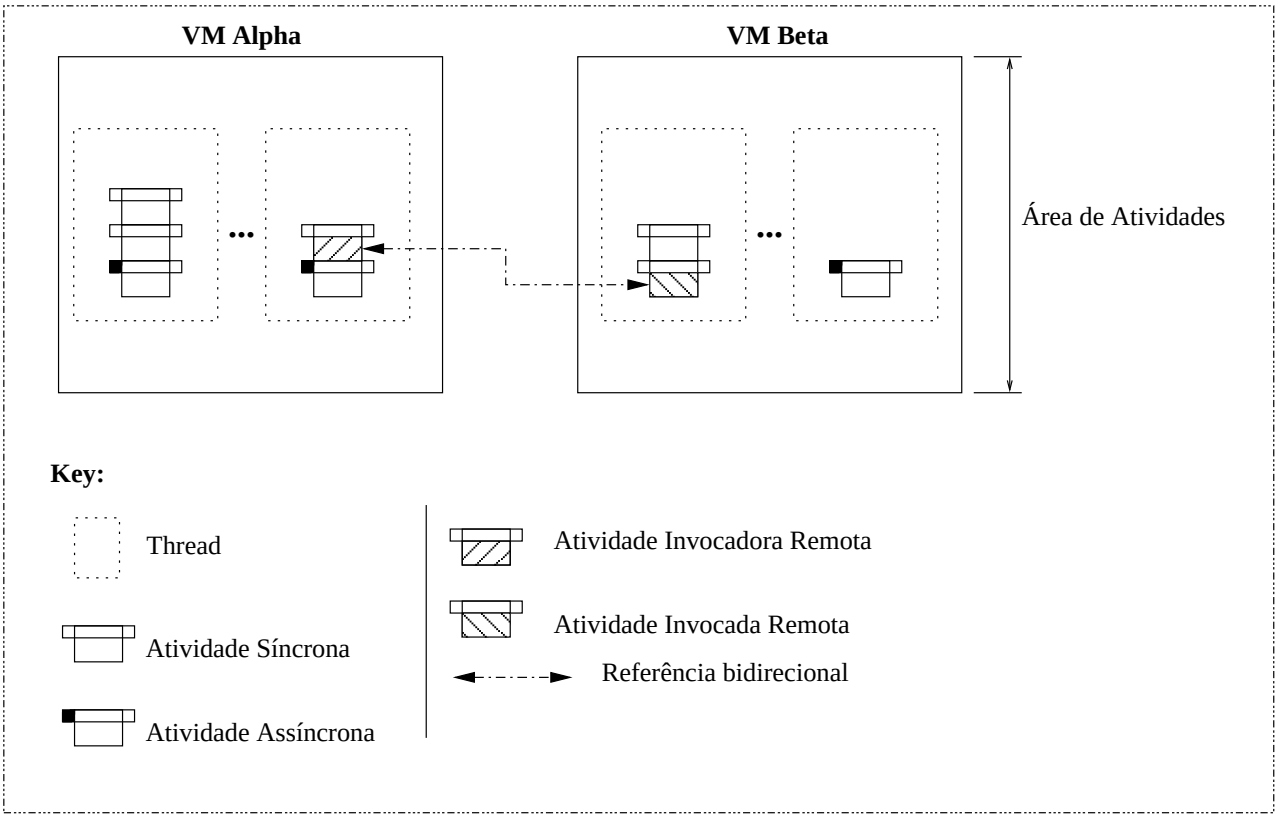


Figura 3.16 Uma atividade assíncrona remota

3.3.5 Resumo da Arquitetura da Máquina Virtual Virtuosi

Uma visão geral de todas as áreas que compõem a arquitetura da MVV é fornecida na Figura 3.17.

3.3.6 Funcionamento da Máquina Virtual Virtuosi

Além das estruturas e áreas de memória da MVV, esse trabalho define as funções principais realizadas pela MVV a fim de interpretar uma aplicação já carregada na área de classes. 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 objeto é instância. Em seguida a MVV interpreta os comandos definidos pelo invocável definido pela árvore de programa.

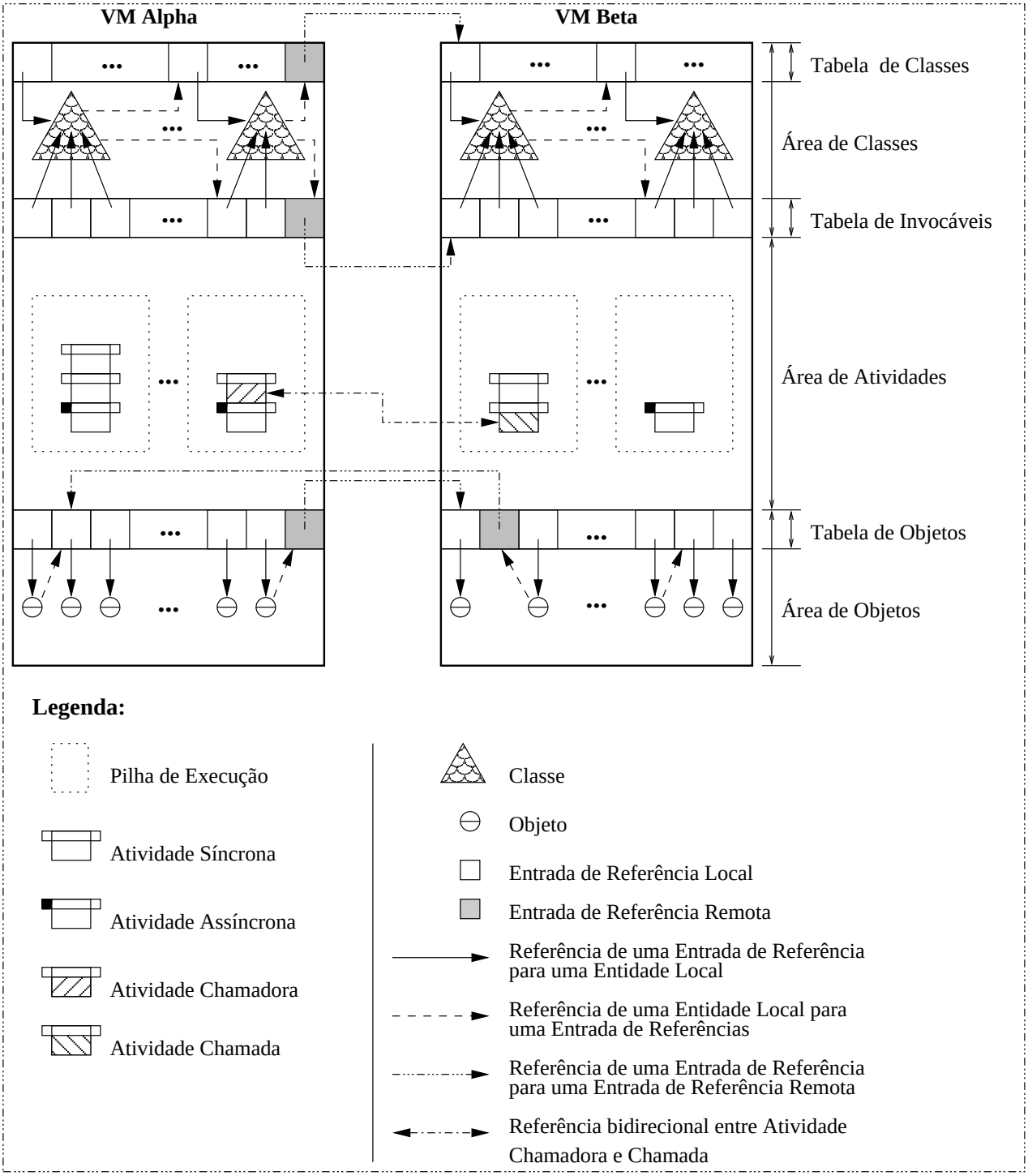


Figura 3.17 Arquitetura da Máquina Virtual Virtuosi

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[9]. De posse do nome do objeto e um apontador para a árvore de programa correspondente à classe de aplicação do objeto, deve-se realizar a seguinte seqüência de ações:

(1) atribuir o nome ao objeto;

(2) atribuir ao apontador 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 apontador nulo;

(4) para cada um dos atributos blocos de dados definidos na árvore de programa correspondente à classe de aplicação da qual o objeto é instância, declarar um apontador 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 objeto 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.

Também nota-se 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.

[9] O mecanismo gerenciador de nomes não faz parte do escopo deste trabalho.

Criação de uma Atividade

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

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.

Após a criação de uma atividade, a MVV a adiciona ao 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, consiste em 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.

Capítulo 4

Mobilidade de Objetos na Virtuosi

Na Virtuosi é premissa que todos os objetos tenham a habilidade de se mover livremente. Esta habilidade pode somente ser restringida por meio de primitivas. Este capítulo formaliza os aspectos que tangem a mobilidade de objetos na Virtuosi. Primeiro são descritas as primitivas de mobilidade. Segundo, a definição dos pré-requisitos necessários para a movimentação de um objeto. Terceiro, o detalhamento do protocolo de migração. Quarto, as fragmentações referenciais. Quinto, algumas considerações sobre o protocolo. Por fim, uma avaliação comparativa.

4.1 Primitivas

Mobilidade dos Objetos na Virtuosi é suportada por cinco primitivas básicas da linguagem que são associadas através de invocáveis, a saber:

fix (): Fixa o objeto na MVV local.

unfix (): Torna o objeto móvel.

refix (MVVNome): Move e fixa o objeto na MVV cujo identificador corresponde ao objeto MVVNome.

locate (): MVVNome: Retorna o objeto MVVNome que identifica a MVV onde o objeto se encontra.

move (MVVNome): Move o objeto para a MVV cuja identificação é representada pelo objeto MVVNome.

As primitivas são semelhantes às adotadas pelo Emerald [Jul et al., 1988], porém a forma com que são implementadas são diferentes, desde que o mecanismo de migração e o modelo de referências são outros. O fix é utilizado pelo programador para impedir que ocorra a migração do objeto. Esta primitiva deve ser utilizada de forma criteriosa, pois pode impedir que outros aplicativos que gerenciam as políticas administrativas do sistema percam o poder de geri-las. Por exemplo, um balanceador de carga limita sua funcionalidade na medida que não pode movimentar objetos. Em uma manutenção preventiva de equipamento, o objeto fixado não poderia ser realocado.

O $unfix$ permite que o objeto tenha mobilidade novamente. Para isto atualiza o mesmo campo da estrutura da entrada que o fix. O $refix$ é a composição das primitivas $unfix$, *move* e fix. O *locate* é útil quando é relevante ao programador a informação de onde se encontra o objeto. O *move* efetiva a migração do objeto da MVV local para a destino, desde que o objeto não esteja fixado. Exemplos da sintaxe das primitivas estão na seção .

A utilização do fix é importante por permitir ao programador a segurança que o objeto criado não será transposto para um local diferente ao qual ele foi projetado. Por exemplo, se o objeto tem a necessidade de rodar em uma plataforma específica dentro da rede, ou o ganho de desempenho é bastante significativo pela permanência do objeto em um determinado nodo. Para tanto, o objeto deve ser fixado. Uma medida de segurança a mais é restringir a licença do uso da primitiva $unfix$ e $refix$ para outros objetos. Assim não há como o objeto ser movimentado pela solicitação de um objeto externo.

Para implementação das primitivas foi necessário estender a estrutura da entrada de referência para objeto local (seção 3.3.3) incluindo um novo campo de um bit denominado fix. Este campo quando diferente de zero indica que o objeto está fixo. A execução da primitiva fix atribui o valor 1 (um) para a campo, enquanto a $unfix$ atribui 0 (zero).

A informação sobre a fixação somente existe na entrada de referência local para o objeto. Referências remotas não contém o campo fix. (ver Figura 4.1).

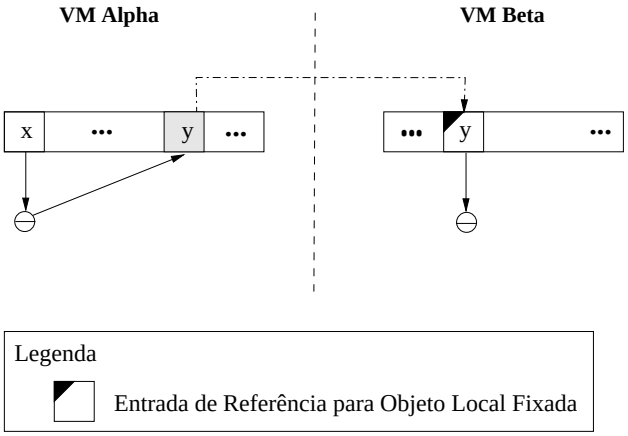


Figura 4.1 Objeto Y fixado por meio de primitiva e o objeto X disponível para movimentação.

4.2 Pré-requisitos para migração

Na Virtuosi o único pré-requisito existente para que um objeto possa ser migrado, é que ele não esteja fixado por força de alguma primitiva. Caso a primitiva *move* seja invocada para um objeto fixado, será disparada uma exceção.

4.3 Mecanismo

Nesta seção é descrito passo a passo o mecanismo de migração de objetos para a Virtuosi. A questão da mobilidade de atividades será descrita em seguida, fora da seqüência cronológica do protocolo, para facilitar o entendimento. Algumas considerações sobre o protocolo finalizam a seção.

4.3.1 Protocolo de Mobilidade

A movimentação do objeto ocorre sempre quando a primitiva *move* é invocada. O protocolo de mobilidade possui três passos. O cenário inicial descrito pelas Figuras 4.2 e 4.3 será utilizado para demonstrar cada um dos passos.

PASSO 1: Verificação da necessidade da classe. (a) O objeto a ser migrado envia

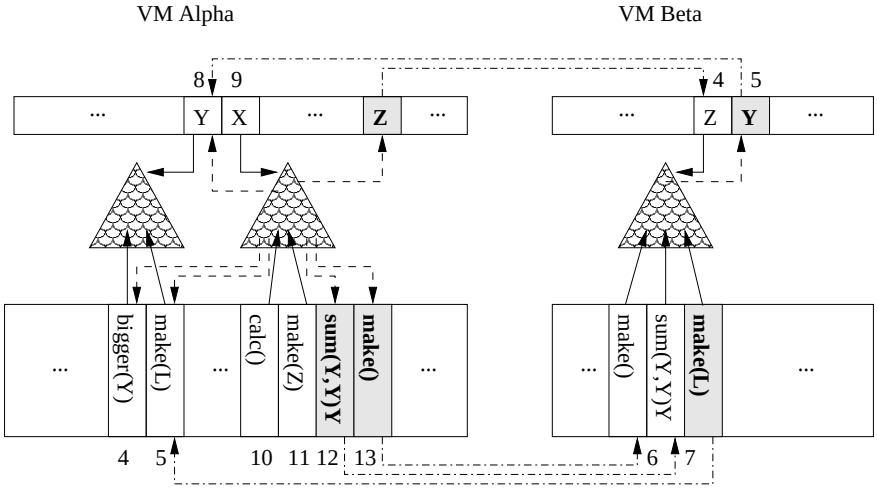


Figura 4.2 Cenário inicial antes da migração do objeto X para a MVV Beta.

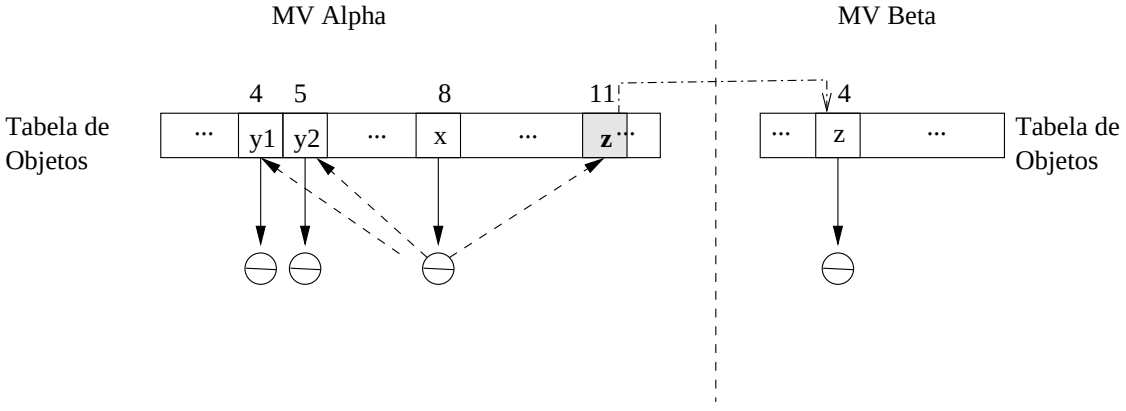


Figura 4.3 Cenário inicial antes da migração do objeto X para a MVV Beta.

uma mensagem informando o nome de sua classe para a MVV destino. (b) A partir do nome recebido, a MVV destino identifica a existência ou não da classe. A identificação é feita pela busca na Tabela de Classes de uma entrada cujo o atributo *nome* seja igual ao nome transmitido pela MVV origem. (c) Uma mensagem é retornada informando o resultado da busca. (d) Caso se confirme a existência da classe, o PASSO 2 do protocolo não será executado, seguindo diretamente ao PASSO 3.

PASSO 2: Replicação da classe. (a) São criadas duas tabelas auxiliares com as informações das referências externas da classe. Estas tabelas são criadas a partir das informações da Lista de Referências a Classes e da Lista de Referência a Invocáveis

Lista para Classes

Nome	Y	Z
Refer	&Y	&Z

Lista para Invocáveis

Nome	Y.bigger(Y)	Y.make(L)	Z.make()	Z.sum(Y,Y)Y
Refer	&(Y.bigger(Y))	&(Y.make(L))	&(Z.make())	&(Z.sum(Y,Y)Y)

Tabela Auxiliar para Classes

Nome	Y	Z
VM	Alpha	Beta
Id	8	4

Tabela Auxiliar para Invocáveis

Nome	Y.bigger(Y)	Y.make(L)	Z.make()	Z.sum(Y,Y)Y
VM	Alpha	Alpha	Beta	Beta
Id	4	5	6	7

Figura 4.4 Demonstração das tabelas de referência auxiliares para Classes e Invocáveis pertinentes à classe X, geradas a partir das informações das listas de referências, da Tabela de Classes e da Tabela de Invocáveis conforme Figura 4.2.

(seção 3.2). Cada item destas listas é substituído por uma referência composta pelo identificador da MVV e o Id da respectiva entrada para a classe ou invocável. No caso da entrada ser uma referência remota, a referência apontada pela entrada é atribuída a tabela, senão é atribuída a própria entrada (ver Figura 4.4). A árvore de programa da Classe e as tabelas auxiliares são serializadas e enviadas para a MVV destino. (b) A árvore de programa e as tabelas são restauradas na MVV destino. O carregamento da classe é iniciado. No carregamento de uma classe replicada, o subsistema de carga de árvores de programa (seção 3.3.2) realiza uma seqüência de ações diferenciada:

(1) Carregar a árvore de programa na área de classes.

(2) Para cada referência da classe, procurar pela entrada correspondente na Tabela de Classes. Caso não encontre, criar uma nova entrada remota e atribuir as informações da linha da tabela auxiliar para a classe correspondente. Atualizar a referência para entrada.

(3) Para cada referência a um invocável, procurar pela entrada correspondente na Tabela de Invocáveis. Caso não encontre, criar uma nova entrada remota e atribuir as informações da linha da tabela auxiliar para invocáveis correspondente. Atualizar a referência para entrada.

(4) Substituir a entrada remota, ou caso não exista, criar uma nova entrada local na Tabela de Classes para a própria classe cuja árvore foi carregada. Ligar a entrada a classe.

(5) Substituir as entradas remotas, ou caso não existam, criar novas entradas locais

Nome	**MV**	**Id**
y1	Alpha	4
y2	Alpha	5
z	Beta	11

Tabela 4.1 Tabela de referência auxiliar para Objetos, gerada a partir do cenários descrito na Figura 4.3

na Tabela de Invocáveis para os próprios invocáveis cuja árvore foi carregada. Ligar as entradas aos invocáveis.

Ao fim das ações, a classe estará carregada e todas as suas referências atualizadas (ver Figura 4.5). Nas Figuras 4.2 e 4.5 temos respectivamente o cenário inicial e final da migração da classe X.

PASSO 3: Migração do objeto. (a) Pela alteração da $flag\ freeze$ (seção 3.3.3) o objeto torna-se indisponível para alterações de estado. Somente acessos de leitura continuam sendo possíveis. (b) É criada uma tabela auxiliar com as informações das referências do objeto. Para cada referência do objeto, cria-se uma referência composta pelo identificador da MVV e o Id da respectiva entrada (ver Tabela 4.1). No caso da entrada ser uma referência remota, o objeto apontado pela entrada é atribuído à tabela, senão a própria entrada é atribuída. O objeto mais a tabela auxiliar são serializados e enviados para a MVV destino. (c) Com o recebimento das estruturas, a MVV destino inicia o processo de instanciação do objeto conforme as seguintes ações:

(1) Carregar o objeto na área de objetos.

(2) Para cada referência do objeto, procurar pela entrada correspondente na Tabela de Objetos. Caso não encontre, criar uma nova entrada remota e atribuir as informações da respectiva linha da tabela auxiliar.

(3) Criar uma nova estrutura de entrada local para o objeto sem adiciona-la na Tabela de Objetos.

(4) Procurar por uma entrada correspondente ao objeto na Tabela de Objetos.

(5) Se encontrar, armazenar o Id (identificado da posição). Se não, reservar uma entrada na Tabela de Objetos e armazenar o Id.

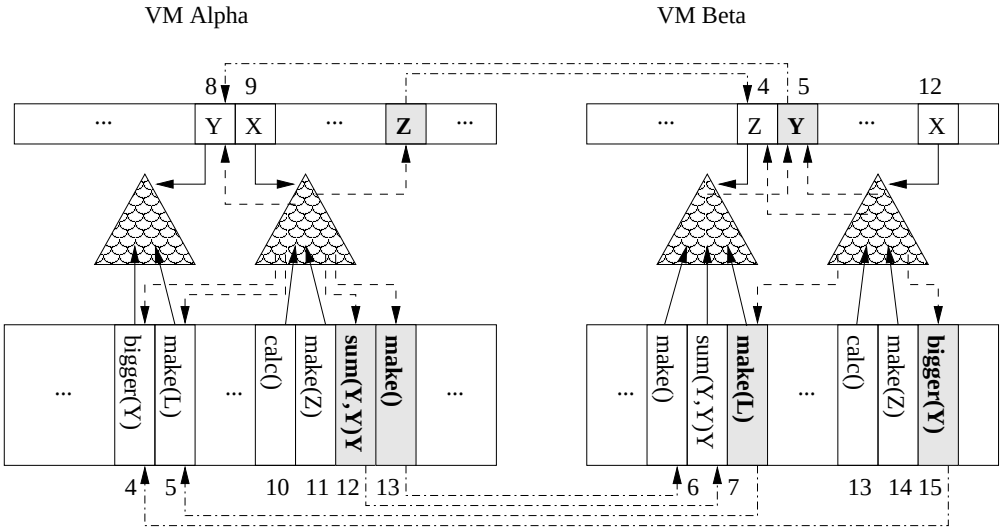


Figura 4.5 Cenário final após a migração da Classe X para a MVV Beta, baseado no Cenário inicial da Figura 4.2.

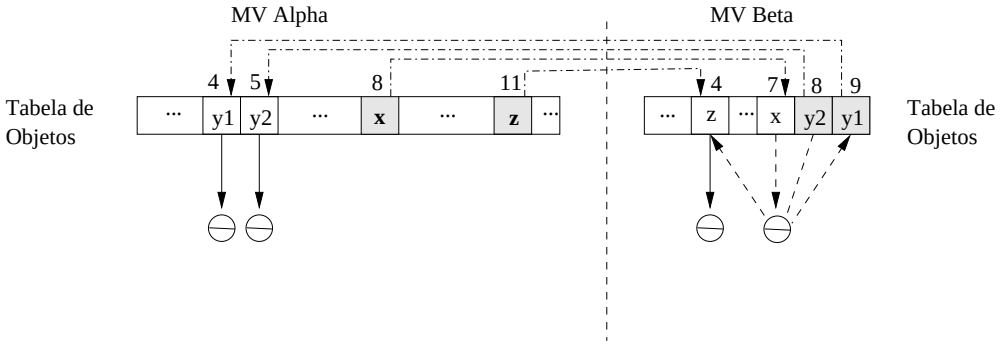


Figura 4.6 Cenário final após a migração do objeto X para a MVV Beta, baseado no Cenário inicial da Figura 4.3.

Os blocos de dados e os enumerados que possam fazer parte do objeto não são alterados. (d) Uma mensagem contendo o Id da posição da tabela armazenada é retornada a MVV origem. (e) Com o Id recebido mais a identificação da MVV destino, a entrada local do objeto migrado na MVV origem é substituída por um entrada remota (ver Figura 4.6). (f) A MVV origem notifica o término da transação para a MVV destino. (g) A MVV destino adiciona a entrada local para o objeto na posição armazenada da Tabela de Objetos. Nas Figuras 4.3 e 4.6 temos respectivamente o cenário inicial e final da migração do objeto x.

4.3.2 Mecanismo de Mobilidade de Atividades

Diferentemente do *DistributedOz*, a Virtuosi permite que objetos com atividades possam ser movimentados. A implementação deste mecanismo tem um custo caro para o sistema, já que é preciso relacionar as atividades com o objeto que as gerou. A abordagem escolhida para a Virtuosi é semelhante à adotada pelo Emerald.

O processo de movimentação das atividades pertence ao PASSO 3. Antes do congelamento do objeto – PASSO 3(a) – o processo de movimentação das atividade é iniciado. Todas as atividade vinculadas ao objeto estão registradas na Tabela de Referência para Atividades conforme descrito na Arquitetura da Virtuosi (Capítulo 4 - Estrutura do Objeto). (a) Para cada atividade do objeto, as seguintes ações são efetuadas:

(1) Verificar se a atividade já não foi congelada.

(2) Identificar se há [1]*atividades consecutivas* pertencentes ao mesmo objeto.

(3) Congelar a atividade corrente mais todas as atividades identificadas.

(4) Se existir uma atividade imediatamente abaixo das atividades congeladas:

(a) Selecionar a atividade.

(b) Transformar a atividade selecionada em um atividade *invocadora remota*.

(c) Transformar a atividade da base das atividades congeladas em um atividade *invocada remota* com referência à atividade *invocadora remota*.

(d) Atualizar a referência da atividade *invocadora remota* para a atividade *invocada remota*.

(5) Se existir uma atividade imediatamente acima das atividades congeladas:

(a) Selecionar a atividade.

(b) Transformar a atividade do topo das atividades congeladas em um atividade *invocadora remota*.

(c) Transformar a atividade selecionada em um atividade *invocada remota* com referência para a atividade *invocadora remota*.

(d) Atualizar a referência da atividade *invocadora remota* para a atividade *invocada remota*.

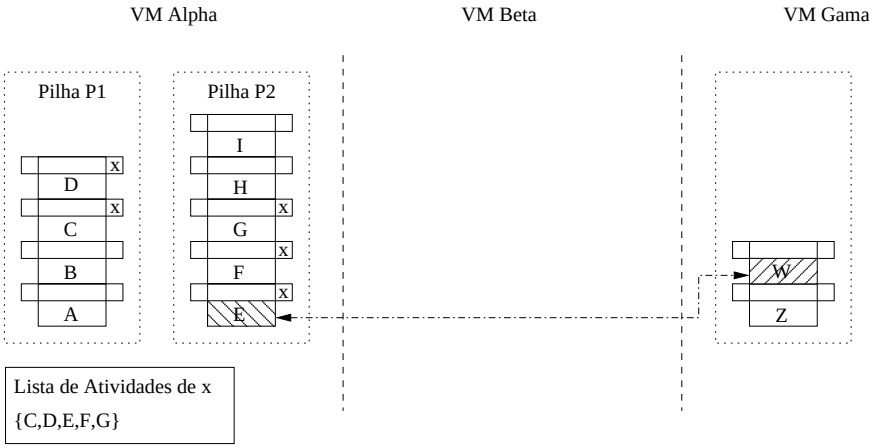


Figura 4.7 Cenário inicial antes da migração do objeto X para a MVV Beta.

(6) Copiar e agrupar as atividade congeladas preservando a ordem em que se encontram na pilha.

Caso a atividade a ser transformada já for uma atividade *invocada remota* ou *invocadora remota* esta assumirá os dois papéis. (b) O objeto é congelado e a tabela auxiliar é criada como no PASSO 3. Para cada variável local e parâmetro da atividade deve ser incluso um registro na tabela auxiliar de objetos da mesma forma como acontece para as referências do objeto. Juntamente com a tabela auxiliar e o objeto, as atividades são serializadas e enviadas. (c) Após a instanciação do objeto na MVV destino (PASSO 3(c)), procede-se com a seguintes ações:

(1) Para cada variável local ou parâmetro de cada atividade, procurar pela entrada correspondente na Tabela de Objetos. Caso não encontre, criar uma nova entrada remota e atribuir as informações da respectiva linha da tabela auxiliar. Ligar a entrada à variável ou parâmetro.

(2) Atualizar a referência da atividade para o objeto e para o invocável.

(3) Criar uma pilha de execução para cada grupo de atividades e bloquear.

(d) Junto com a mensagem do Id da entrada do objeto (PASSO 3(d)), retorna-se a referência de cada atividade *invocada* ou *invocadora remota* que tenha sido transferida.(e) A MVV

[1]Atividades Consecutivas: Conjunto de atividades em uma mesma pilha de execução que necessariamente estão dispostas uma sobre a outra sem sofrer segmentação por outras atividades que não pertencem ao grupo.

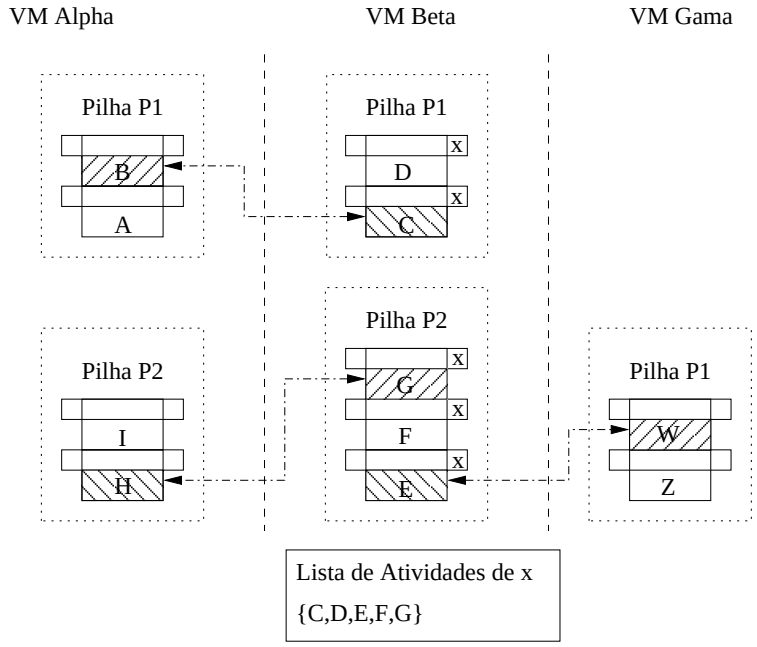


Figura 4.8 Cenário final depois da migração do objeto X para a MVV Beta, baseado no Cenário inicial da Figura 4.7.

origem atualiza as referências das atividades locais que foram transformadas em atividades *invocadas* e *invocadoras remotas*. Para as possíveis atividades remotas que referenciam as atividades transferidas, uma mensagem é envidada para as respectivas MVVs informando os novos endereços. Aguarda-se a confirmação de retorno de cada uma das atividades remotas. (f) Notifica-se a MVV destino do fim da transação. As pilhas de execução então são desbloqueadas e o protocolo de mobilidade é concluído. Nas Figuras 4.7 e 4.8 temos respectivamente o cenário inicial e final da migração das atividade do objeto x.

No caso de alguma atividade migrada *invocada* ou *invocadora remota* possuir somente referências para atividades da MVV destino, a atividade migrada não retornará sua referência para MVV origem (etapa (d)). A própria atividade, após a notificação de conclusão da transação, atualizará as referências somando suas pilhas às das atividades já existentes no destino (ver Figuras 4.9 e 4.10).

No exemplo descrito na Figura 4.7, existem três MVVs Alpha, Beta e Gama. O objeto x situado na MVV Alpha (origem), será migrado para a MVV Beta (destino). A atividade remota W residente na MVV Gama relaciona-se com a atividade E do objeto x. Segue a

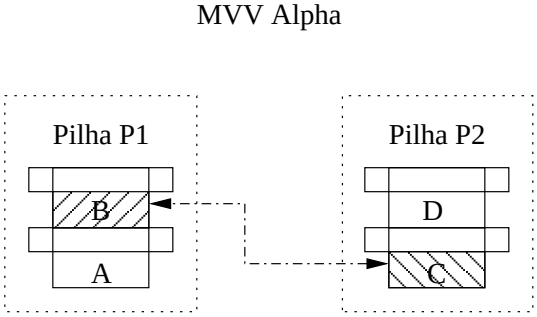


Figura 4.9 Cenário onde duas pilhas de execução possuem referências entre suas atividades localmente.

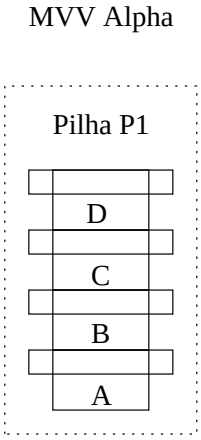


Figura 4.10 Cenário final após resolução das referências locais entre atividades.

descrição detalhada das principais etapas do protocolo para este exemplo: a) A primeira atividade selecionada que consta na lista é a atividade C.

(1) A atividade C não está congelada.

(2) Identifica a atividade D como sendo consecutiva a C.

(3) A atividade C e D são congeladas.

(4) Existe uma atividade imediatamente abaixo das atividades congeladas.

(a) A atividade B é selecionada.

(b) B é transformada em uma atividade *invocadora remota*

(c) A atividade C é transformada em uma atividade *invocada remota* com referência a atividade B.

(d) Atualizar a referência da atividade B para C.

(5) Não existem atividades imediatamente acima das atividades congeladas.

(6) As atividades congeladas C e D são agrupadas e copiadas.

A próxima atividade da lista é a atividade D. Verifica-se que está congelada. A próxima atividade é a E.

(1) A atividade E não está congelada.

(2) Identifica as atividades F e G como sendo consecutivas a E.

(3) As atividades E, F e G são congeladas.

(4) Não existe uma atividade imediatamente abaixo das atividades congeladas.

(5) Existem atividades imediatamente acima das atividades congeladas.

(a) Seleciona a atividade H.

(b) Transforma a ativivade G em um atividade *invocadora remota*.

(c) Transformar a atividade H em um atividade *invocada remota* com referência para a atividade G.

(d) Atualizar a referência da atividade G para H.

(6) As atividades congeladas E,F e G são agrupadas e copiadas.

As próximas atividades da lista são F e G as quais estão congeladas. Para cada variável local e parâmetro de cada atividade inclui-se um registro na tabela auxiliar de objetos. As atividades copiadas são serializadas e enviadas. (c) Na MVV destino cada parâmetro ou variável de cada atividade é tratado como uma referência de objeto conforme descrito no PASSO 3(c). Atualizam-se as referências das atividades migradas para o objeto, e para os respectivos invocáveis. Criam-se as pilhas $P1$ e $P2$. As pilhas criadas são bloqueadas. (d) As novas referências das atividades *invocadas remotas* E e C, e da atividade *invocadora remota* G são retornadas.(e) A MVV origem atualiza as referências da atividades *invocadora remota* B e *invocada remota* H para C e G respectivamente. Para a atividade remota W é enviada uma mensagem informando o novo endereço de E. A atividade W confirma a alteração. (f) Notifica-se a MVV destino do fim da transação. As pilhas de execução $P1$ e $P2$ são desbloqueadas e o protolocolo de mobilidade é concluído.

4.3.3 Fragmentação de Referências

A fragmentação de referências se dá pela movimentação dos objetos. Quanto mais dispersos os objetos se encontram pela rede maior tende a ser a fragmentação de suas referências. Uma referência remota entre entidades que precise percorrer mais de duas [2] *referências intermediárias* é considerada uma referência fragmentada. No caso das entidades estarem situadas localmente, a referência é considerada fragmentada quando existirem mais de uma referência intermediária. Na cenário descrito pela Figura 4.11, exitem referências otimizadas e fragmentadas entre os objetos. Na Figura 4.13 os quatro caminhos das referências da Figura 4.11 são separados. Observe que as referências b) e d) estão fragmentadas enquanto a a) e c) estão otimizadas.

As três entidades da Virtuosi têm comportamento diferenciado para com as referências durante a movimentação. No caso da classe, sempre que o carregador de árvores (seção 3.3.2) carrega uma classe, necessariamente todas as classes referênciadas são carregadas localmente. Seguindo o exemplo da Figura 4.12, a figura (a) monstra o estado inicial após o carregamento

[2]Referências Intermediárias: Estruturas que tem como objetivo redirecionar referências, não sendo o objetivo final do acesso. Na Virtuosi todas as entradas das tabelas de referências são consideradas referências intermediárias.

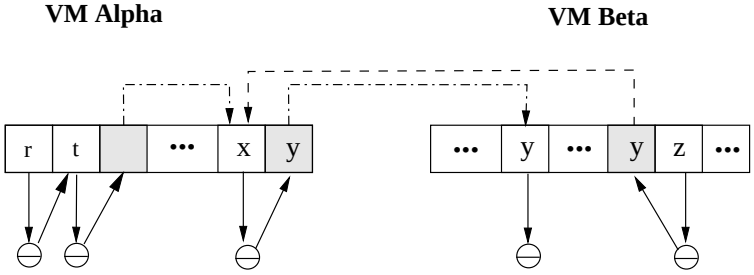
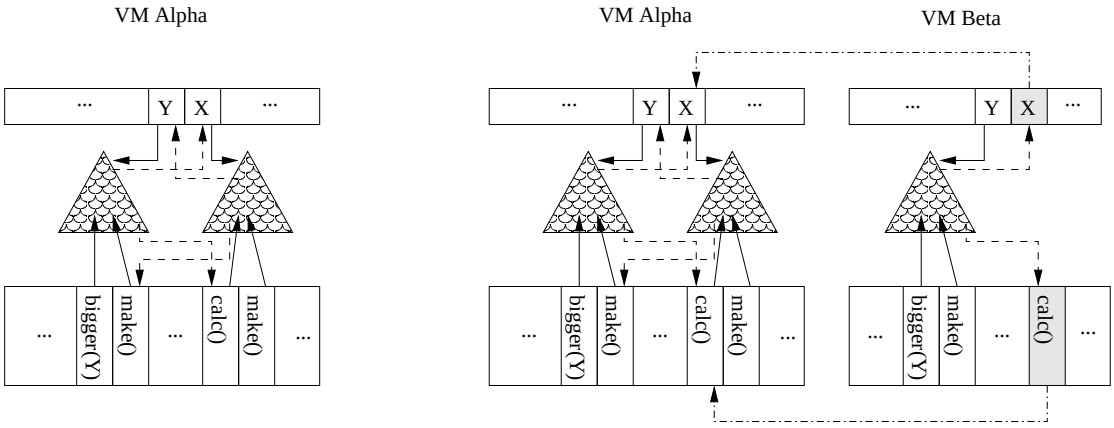


Figura 4.11 Demonstração da área de objeto onde existem referências fragmentadas e otimizadas.

das classes. Todas as classes referênciadas devem ser carregadas. Para o exemplo são as classes X e Y. Na figura (b), a classe Y é replicada para a MVV Beta passando a referênciar remotamente a classe X. Na seqüência (c), Y é replicada da MVV Beta para a MVV Gama. Observa-se que as referências continuam otimizadas. Independente da replicação das classes X e Y para qualquer MVV nunca haverá fragmentação de referências.

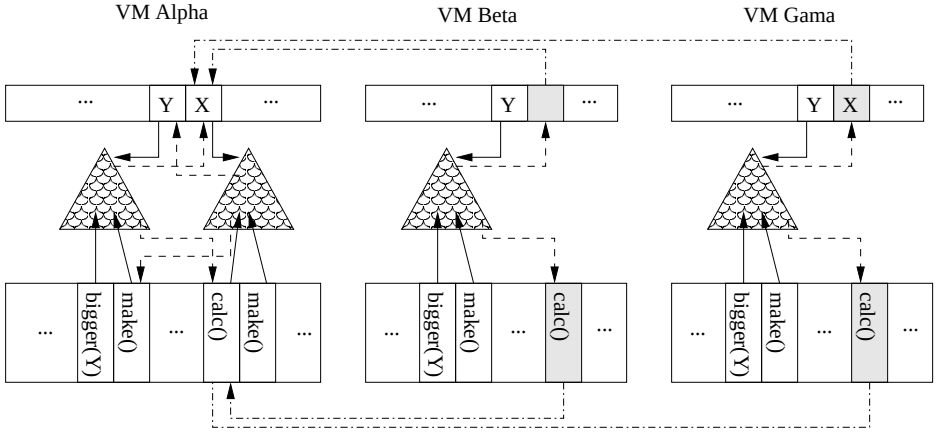
Atividades não se utilizam de referências intermediárias entre si, as referências sempre são diretas e bidirecionais. Logo não existe fragmentação referencial entre atividades.

O objeto tem por características possuir referências unidirecionais para com outros objetos e sempre de forma indireta. Diferentemente da classe, o objeto não poder ser replicado, sendo uma entidade única no sistema. A partir do objeto referenciado não é possível saber quais outros objeto o referenciam. Sem esta informação, não há como notificar estes objetos sobre a nova localização do objeto migrado, exceto pela utilização de um *broadcast*, o que inviabilizaria o sistema para redes maiores. No caso (b) e (d) da Figura 4.13, o protocolo consegue evitar a fragmentação identificando previamente na tabela de objetos qualquer referência ao nome do objeto recebido. Por esta razão é que se justifica idetificar os objetos com um nome único. Na Figura 4.14, o objeto z referência y remotamente de forma otimizada. Conforme mostra a Figura 4.15, o objeto y foi migrado para a MVV Gama e conseqüentemente a referência de z para y tornou-se fragmentada. Quanto mais y prosseguir migrando para diferentes MVVs, mais fragmentada a referência se torna. Neste caso o protocolo de mobilidade é inoperante.



(a) Estado inicial após o carregamento das classes.

(b) A classe Y é replicada para a MVV Beta.



(c) Da MVV Beta a classe Y é replicada para a MVV Gama.

Figura 4.12 Demonstração da seqüência de movimetações da classe Y. Cenário iniciado a partir da conclusão da carga das classes pelo carregador de árvores.

4.3.4 Considerações

Existem algumas considerações a se fazer sobre o protocolo. A criação das tabelas auxiliares para classes e invocáveis geram um custo que poderia ser evitado se subtituídas pelas Listas de Referências a Classes e a Invocáveis. Neste caso, durante a carga do objeto (PASSO 2 - (c)), a MVV destino solicitaria à MVV origem somente as informações sobre as entradas

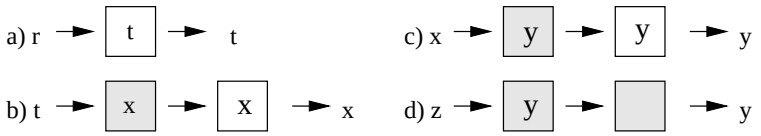


Figura 4.13 Demonstração de referências fragamentadas: as referências **b** e **c** estão fragmentadas enquanto a **a** e **d** estão otimizadas (informação baseada na Figura 4.11).

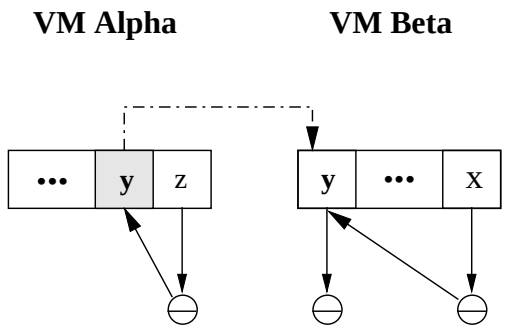


Figura 4.14 Cenário onde todas as referências estão otimizadas.

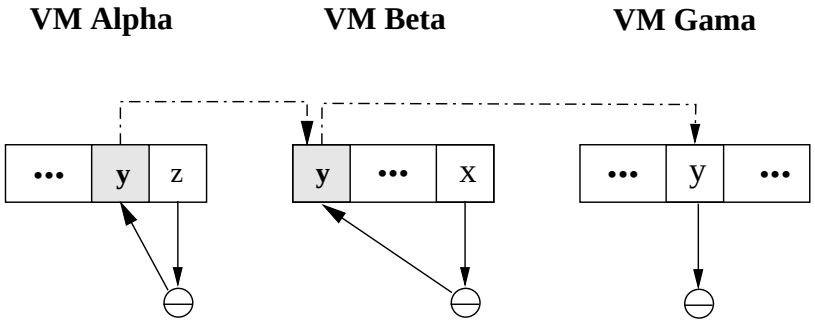


Figura 4.15 Cenário após a migração do objeto y para a MVV Gama tendo como base a Figura 4.14.

de referências as quais não fossem encontradas equivalentes localmente. Como conseqüência teríamos mais dois acessos à rede, a solicitação e o retorno. Para o protocolo da Virtuosi sempre prevalece a minimização da utilização da rede em prol do maior uso de processamento local.

Mesmo com o esforço despendido pelo mecanismo não é possível manter todas as referências atualizadas. Nestes casos a otimização das referências fragmentadas serão deixadas a cargo do mecanismo de RPC[3] ainda a ser desenvolvido.

4.3.5 Tolerância a Faltas

A segurança do funcionamento mecanismo é garantida pelo fato de que somente após a notificação de finalização da transação a entrada para o objeto é disponibilizada na Tabela de Objetos. No caso de uma falha de comunicação entre as MVVs a integridade fica garantida. Se ocorrer uma falha de comunicação após a conclusão do PASSO 2, nada que foi efetuado até o momento será desfeito pois nenhuma referência estaria inválida para as classes. No caso da falha ocorrer durante o PASSO 3, as estruturas montadas na MVV destino são desprezadas. Em ambos os casos a MVV origem lança uma exceção, restabelece as atividades congeladas caso existam, e disponibiliza o objeto novamente para acesso.

4.3.6 Avaliação Comparativa

Nesta seção descreve-se o Mecanismo de Migração da Virtuosi conforme os requisitos de Integridade, Performance, Confiabilidade e Funcionalidade definidos (seção 2.3) a priori. Utilizaremos a comparação com os sistemas Emerald e DistributedOz.

Integridade: O protocolo garante a ordem processual global e a integridade referêncial através da atomicidade do mecanismo. O protocolo somente disponibiliza alterações no estado do objeto após a finalização da transação. Todos as sistemas atendem a este requisito.

[3]Remote Procedure Call: Protocolo que um programa pode utilizar para requisitar um serviço a outro programa localizado remotamente, sem que seja necessário o conhecimento de detalhes da rede.

Desempenho: O DistributedOz é superior tanto ao Emerald quanto a Virtuosi por duas características:

(1) **Liguagem:** O modelo de programção OPM (seção 2.5.2) permite a total fragmentação do objeto. Com isso somente as partes do código da classe que serão usadas são replicadas, eliminando a necessidade de transferência de toda a classe.

(2) **Entidades Estacionárias:** A definição de que a atividade é uma entidade estacionária e de que o estado do objeto é uma entidade móvel, elimina todo o custo acarretado pela mobilidade de atividades.

O Emerald, por utilizar três tipos de endereçamento para o objeto (seção 2.4.4), consegue obter um ótimo desempenho nos processos que ocorrem localmente. A Virtuosi utiliza o conceito de *handle tables* como padrão de endereçamento entre as entidades. Embora reduza a complexidade, gera-se um custo maior que o Emerald para os processamentos locais, já que a referências são sempre indiretas. Porém o mecanismo de *handle table* facilita e agiliza o processo de coleta de lixo [HU et al., 2003].

O desempenho do mecanismo de mobilidade da Virtuosi, ignorando-se a movimentação de atividades, é representado através das variáveis abaixo:

Custo = Número total de entradas de referências incluídas e alteradas nas tabelas de classes, invocáveis e objetos.

ro = Número de referências para outros objetos.

rc = Número de referências para outras classes.

ri = Número de referências para invocáveis de outras classes.

ni = Número de invocáveis da classe.

n = Número de movimentações do objeto.

Para um cenário onde, a migração do objeto ocorra para nodos onde previamente não existam a classe do objeto e nem outros objetos que sejam referênciados por este, utilizamos a seguinte equação:

$$\mathbf{Custo = n(ro + 2) + n(rc + 1) + n(ri + ni)}$$

Confiabilidade: Na Virtuosi e no Emerald, a fragmentação de referências causada pela movimentação dos objetos gera uma dependência de um maior número de pontos da rede do que no caso que estas tivessem otimizadas. Isto fragiliza a confiabilidade do mecanismo em caso de falha de um nodo o qual faz parte do caminho da referência. No DistributedOz, a referência ao estado do objeto nunca passa de duas referências intermediárias, reduzindo o risco de uma falha romper o caminho de uma referência. Por outro lado, no caso de uma detecção prévia de uma futura falha ou uma manutenção preventiva em algum nodo do sistema, o DistributedOz não tem como mover as entidades estacionárias para outro lugar, sendo inevitável aguardar o fim de todas as atividades daquele nodo antes do desligamento.

Funcionalidade: Embora o Emerald disponibilize primitivas de mobilidade para a programação, não há garantia que ela seja efetivamente executada. Isto porque o fluxo de mobilidade dos objetos é determinado em tempo de compilação. No DistributedOz, o sistema pré-define quais objetos são móveis ou estacionários, também já determinando o fluxo de mobilidade. Na Virtuosi, a migração só ocorre por força de primitiva. Isto permite que seja agregado ao sistema mecanismos de distribuição de carga modulares e viabiliza o suporte a agentes por permitir autonomia de mobilidade.

Com esta avaliação comparativa demonstramos que o Mecanismo de Mobilidade de Objetos desenvolvido para a Virtuosi atinge os requisitos estabelecidos. Atendendo de forma satisfatória as necessidades do sistema.

Capítulo 5

Estudo de Cenários

Será demonstrado através de estudo de cenários o comportamento do protocolo de migração da Virtuosi. Os cenários serão expressos pela notação gráfica da Vituosi e pela linguagem ARAM. Todos os cenários sempre iniciam pelo método *start*() da classe *Start*. Esta classe somente será utilizada para auxiliar a formação do cenário e não será representada dentro da MVV.

5.1 Cenário 1

O cenário 1 é a situação mais simples de migração. A transferência do TAD não é necessária, pelo fato do TAD do objeto a ser migrado já existir na MVV destino. O objeto x referencia y unidirecionalmente e y não possui referências para outros objetos. Não será demonstrada a área de atividades.

Código em ARAM:

```
class y {
    constructor make ( ) exports (all) { }
}

class X {
    Y y;
    constructor make ( ) exports (all) {
        y = Y.make();
        y.migrate("VMBeta");
    }
}
```

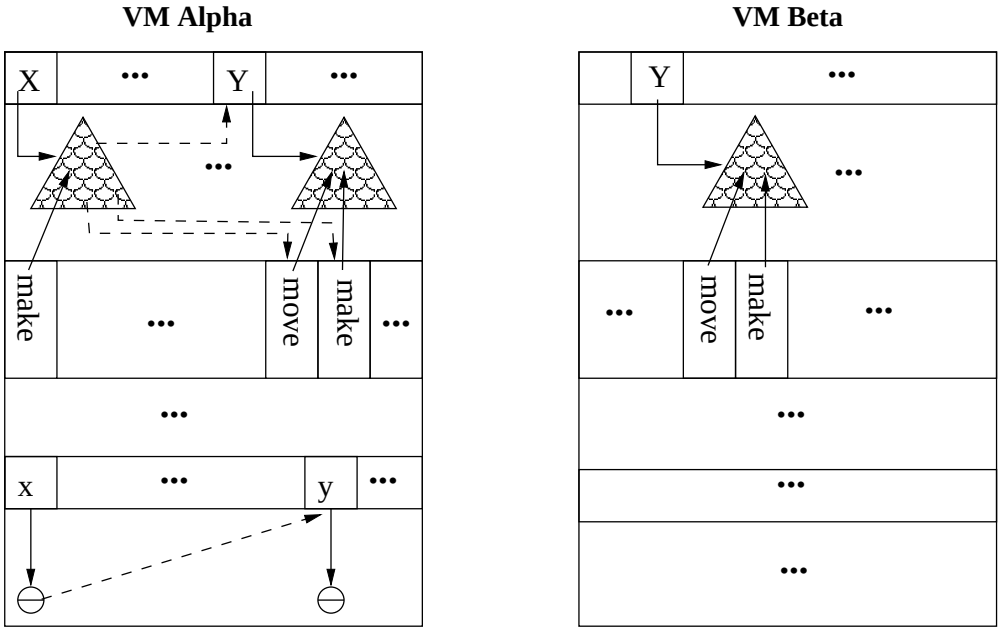


Figura 5.1 Cenário 1 (a): Disposição inicial das entidades e sua referências.

```
class Start {
    void method start ( ) exports (all) {
        X x;
        x = x.make();
    }
}
```

Resolução: O PASSO 1 do protocolo identifica que a classe já existe na MVV destino, conseqüêntemente o PASSO 2 não é executado. O PASSO 3 cria uma nova entrada local na MVV destino para o objeto. A entrada de referência local passa a ser remota na MVV origem.

5.2 Cenário 2

Neste contexto teremos dois objetos sendo que existe uma referência bidirecional entre eles. O objeto migrado precisará referenciar o outro que permaneceu na VM origem remotamente. A classe do objeto migrado não existe na MVV destino. Também não existem referências a

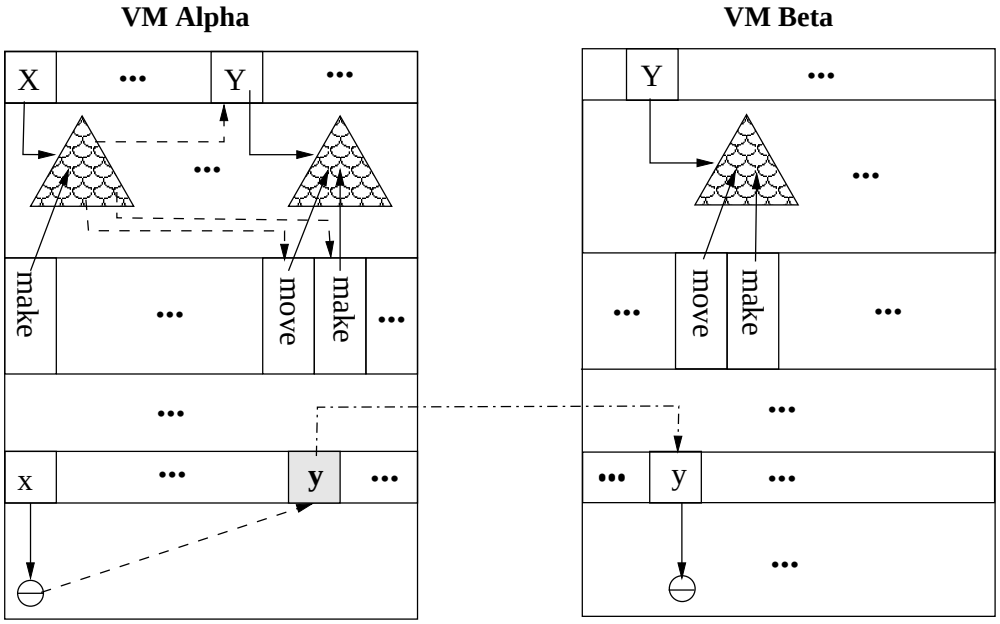


Figura 5.2 Cenário 1 (b): Objeto Y já migrado para a MVV Beta.

invocáveis de ambas as classes. Não será demonstrada a área de atividades.

Código em ARAM:

```
class Y {
    X x;
    constructor make () exports (all) { }
    void method setX (X px) exports (all) {
        x = px;
    }
}
class X {
    Y y;
    constructor make () exports (all) {
        y = Y.make();
    }
    Y method getY () exports (all) {
        return y;
    }
}
class Start {
    void method start ( ) exports (all) {
```

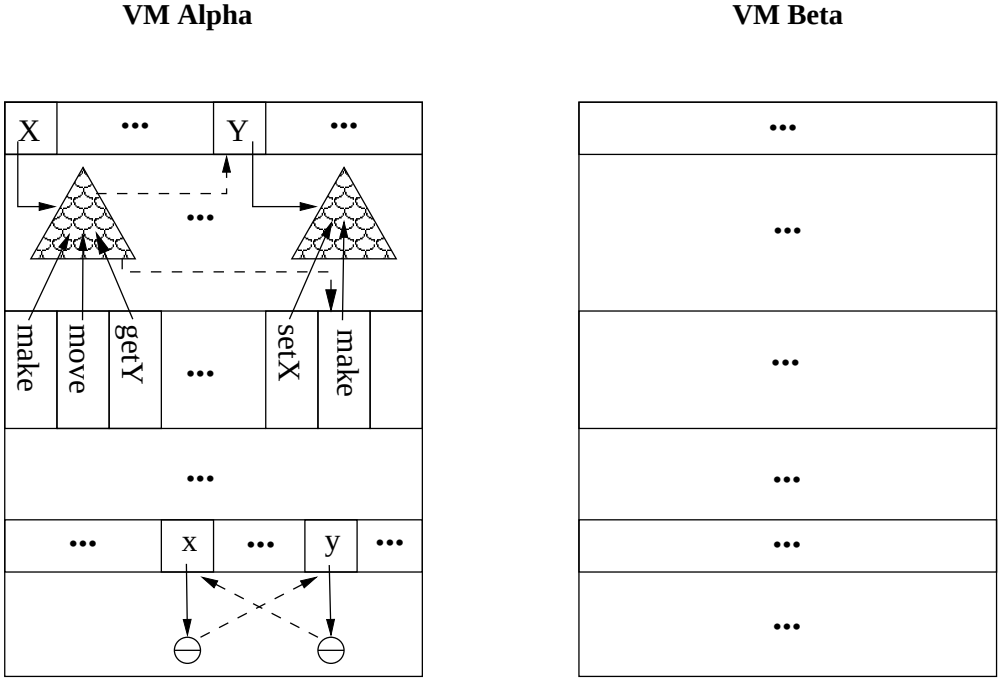


Figura 5.3 Cenário 2 (a): Disposição inicial das entidades e sua referências.

```
        X x = X.make();
        Y y = x.getY();
        y.setX(x);
        x.move(MVVBeta);
    }
}
```

Resolução: O PASSO 1 do protocolo identifica que a classe não existe na MVV destino. O PASSO 2 copia a classe e cria referências remotas para classe Y e para o invocável *make* de Y. O PASSO 3 cria uma nova entrada local para x e uma entrada remota para y na MVV destino. A entrada local de x na MVV origem é transformada em uma entrada remota.

Este cenário atende as restrições para a aplicação da fórmula de desempenho. Com isto é obtido o custo de atualização das referências (ver Seção 4.3.6):

$$\mathbf{Custo = n(ro + 2) + n(rc + 1) + n(ri + ni)}$$

$$\mathbf{Custo = 1(1 + 2) + 1(1 + 1) + 1(1 + 3)}$$

$$\mathbf{Custo = 9}$$

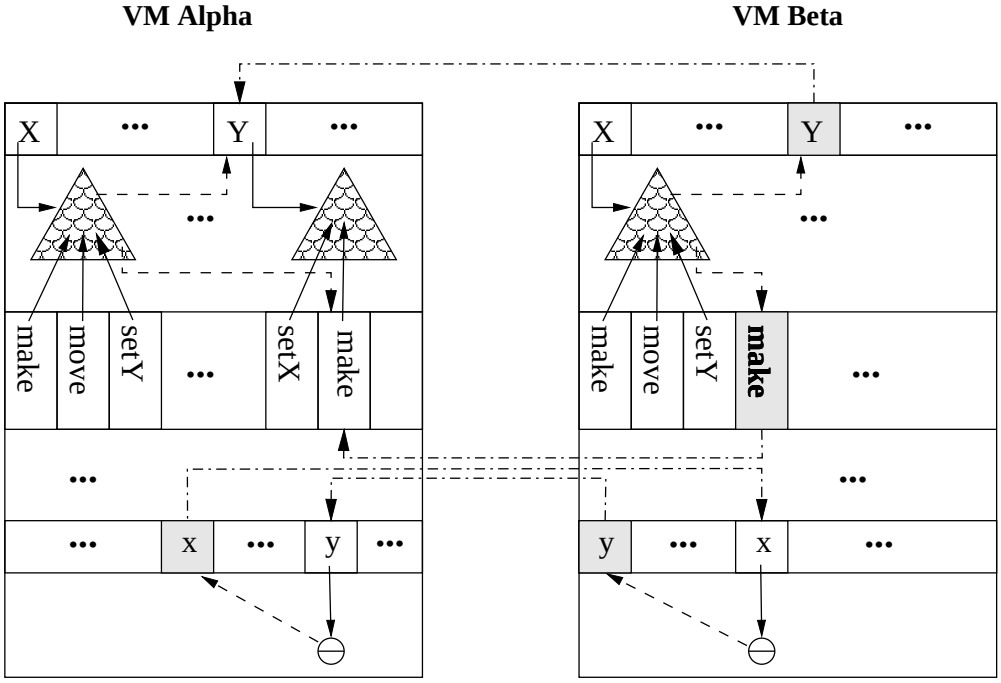


Figura 5.4 Cenário 2 (b): Objeto Y já migrado para a VM Beta.

5.3 Cenário 3

Partindo do final do Cenário anterior (Figura 5.4), moveremos o objeto x para uma terceira MVV Gama onde não existe a classe X. O código das classes X,Y e $Start$ é mantido conforme Cenário 2, exceto pela adição da primitiva $x.move(MVVGama)$ ao final do método $start()$. A configuração final é demonstrada na Figura 5.5.

Código em ARAM:

```
class Start {
    void method start ( ) exports (all) {
        X x = X.make();
        Y y = x.getY();
        y.setX(x);
        x.move(MVVBeta);
        x.move(MVVGama);   // Linha acionada
    }
}
```

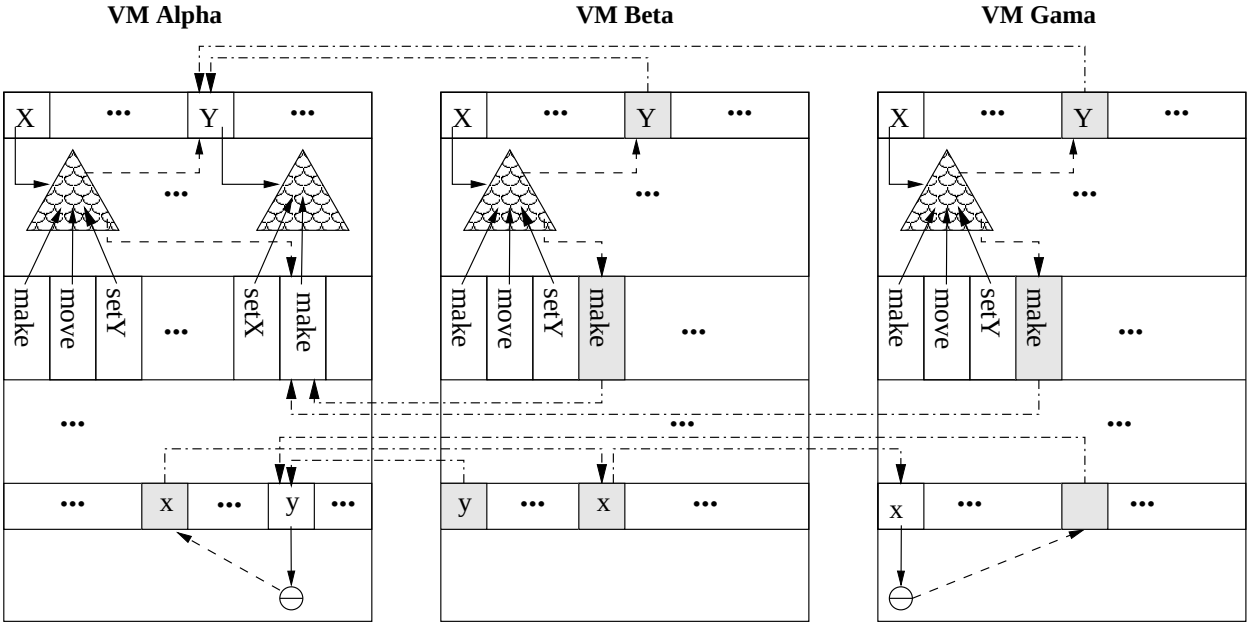


Figura 5.5 Cenário 3: Disposição das entidades ao final do código do método *start*().

Resolução: O PASSO 1 do protocolo identifica que a classe X não existe na MVV Gama. O PASSO 2 copia a classe e cria referências remotas para classe Y e para o invocável *make* de Y. Para as classes não há fragmentação de referências. O PASSO 3 cria uma nova entrada local para x e uma entrada remota para y na MVV Gama. A entrada local de x na MVV Beta é transformada em uma entrada remota. Observe que a referência do objeto y para o x torna-se fragmentada.

5.4 Cenário 4

Este Cenário é composto por três classes X,Y e $Calc$. O objetivo destas classes é que através de redução ou incremento do objeto y declarado no método *calc* da classes $Calc$, este torne-se igual ao número interno de x, que no caso é 23. Antes da migração ocorrer – conforme linha 11 da classe X – teremos a pilha de atividades descrita na Figura 5.6.

```
1 class X {
2     Y y;
3
4     constructor make () exports (all) {
```

```
5         y = Y.make(23); // Número secreto
6     }
7
8     method void test(Y p) exports all {
9
10        if (y.equals(p))    // Encontrou o número
11            move(MVVBeta);  // Move para Beta
12            return;
13        else {
14            if (y.greater(p))
15                p.more();
16            else
17                p.less();
17        }
18    }
19 }
```

```
1 class Y {
2     datablock value1;
3     X x;
4
5     contructor make(Literal k) {
6         value2 = datablock.createDataBlock(32);
7         value2.storeInteger(k);
8     }
9     void setX(X p) {
10        x = p;
11    }
12    void method less(){
13        value.addInteger(-1);
14        x.test(this);
15    }
16    void method more(){
17        value.addInteger(1);
18        x.test(this);
19    }
20    action equals(Y y){             // Se o parâmetro for igual
21        if (value.equals(y.value))  // que o value, executa
22            execute;
23        else
```

```
24              skip;
25      }
26      action grater(Y y){              // Se o parâmetro for maior
27          if (value.greater(y.value))  // que o value, executa
28              execute;
29          else
30              skip;
31      }
32  }
```

```
1   class Calc {
2
3       constructor make() {
4       ...
5       calc();
6       }
7
8       void method calc() {
9           X x = X.make();
10          Y x = make(22);
11          Y.setX(x);
12          x.test(y)
13      }
14  }
```

```
1   class Start {
2
3       void method start() {
4           C c = C.make();
5       }
6   }
```

Resolução: Na migração das atividades do objeto x foi necessário transformar quatro atividades da pilha $P1$ em uma atividade *invocadora remota* (c.calc), duas atividades *invocada invocadora remotas* (x.test) e uma atividade *invocada remota* (y.more). Não houve envio de mensagens à outras MVVs pois originalmente todas a atividades migradas eram atividade normais.

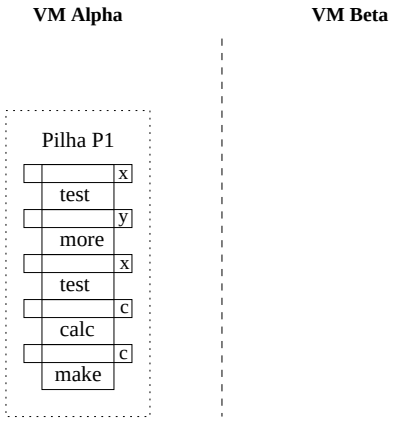


Figura 5.6 Cenário 4: Disposição incial das atividades.

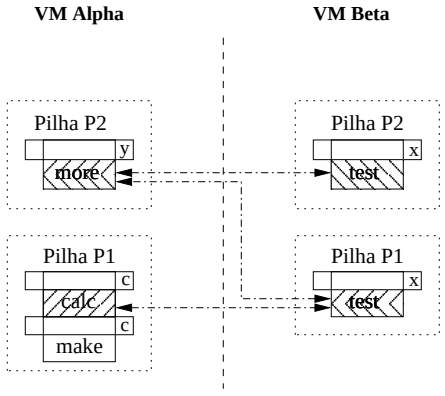


Figura 5.7 Cenário 4: Disposição das atividade após a migração do objeto x.

Capítulo 6

Implementação

O mecanismo foi implementado sobre o protótipo da MVV desenvolvido no trabalho que definiu o modelo de execução das árvores de programa da Virtuosi [Kolb, 2004]. Este protótipo foi implementado com o objetivo de validar o metamodelo da Virtuosi e o uso de árvores de programa como representação intermediária para máquinas virtuais.

6.1 Ambiente

Para a construção do protótipo foi utilizada a linguagem de programação Java pelo fato do protótipo original da MVV já ter adotado esta linguagem. O ambiente distribuído foi elaborado por um rede composta por dois computadores comunicando-se através do protocolo TCP/IP. Optou-se pela utlização de Sockets por ser um conceito fundamental sem depedência, diferente, por exemplo, de RMI. Durante os testes, cada instância da MV Java executou uma instância da MVV, sendo possível abrir mais que uma MV Java em cada computador. Sob a MV Java utilizamos o sistema operacional Windows 2000.

6.2 Limitações

A implementação do mecanismo de mobilidade tem algumas limitações ao modelo proposto no Capítulo 4:

(1) não consegue migrar objetos com atividades;

(2) não existe um serviço de nomes para os nodos. Previamente no programa já deveria constar o nome e porta de comunicação do nodo destino;

O protótipo MVV no qual o mecanismo foi implementado também possui algumas limitações:

(1) dá suporte somente à interpretação de uma pilha de atividades por instância da MVV, de forma que somente invocações síncronas podem ser interpretadas[1];

(2) não dá suporte à herança entre classes de aplicação, embora o metamodelo da Virtuosi dê suporte a esta propriedade;

6.3 Diagrama de Classes da MVV

A Figura 6.1 mostra um diagrama de classes com as principais classes que compõem a MVV. As classes VM, ObjectTable, ObjectTableEntry, InvocableTable e TreeTable foram estendidas para atender a mobilidade.

As funcionalidades necessárias à distribuição da MVV foram implementadas nas seguintes classes:

VMD: É a extensão da classe VM. A partir desta classe foram implementados os serviços de comunicação de rede.

ObjectTableD: Gerada como extensão da classe ObjectTable, foi implementada com a função de buscar entradas de referências tendo como índice o nome do objeto. A classe representa a tabela de objetos.

ObjectTableEntryD: Foi adicionado o atributo fix e os respectivos métodos de atualização. A classe representa a tabela de objetos.

InvocableTableD : Implementação semelhante ao ObjetTableD. A classe representa a tabela de invocáveis.

TreeTableD : Implementação semelhante ao ObjetTableD. A classe representa a tabela de classes.

[1]Concorrência é objeto de estudo de outra dissertação de mestrado.

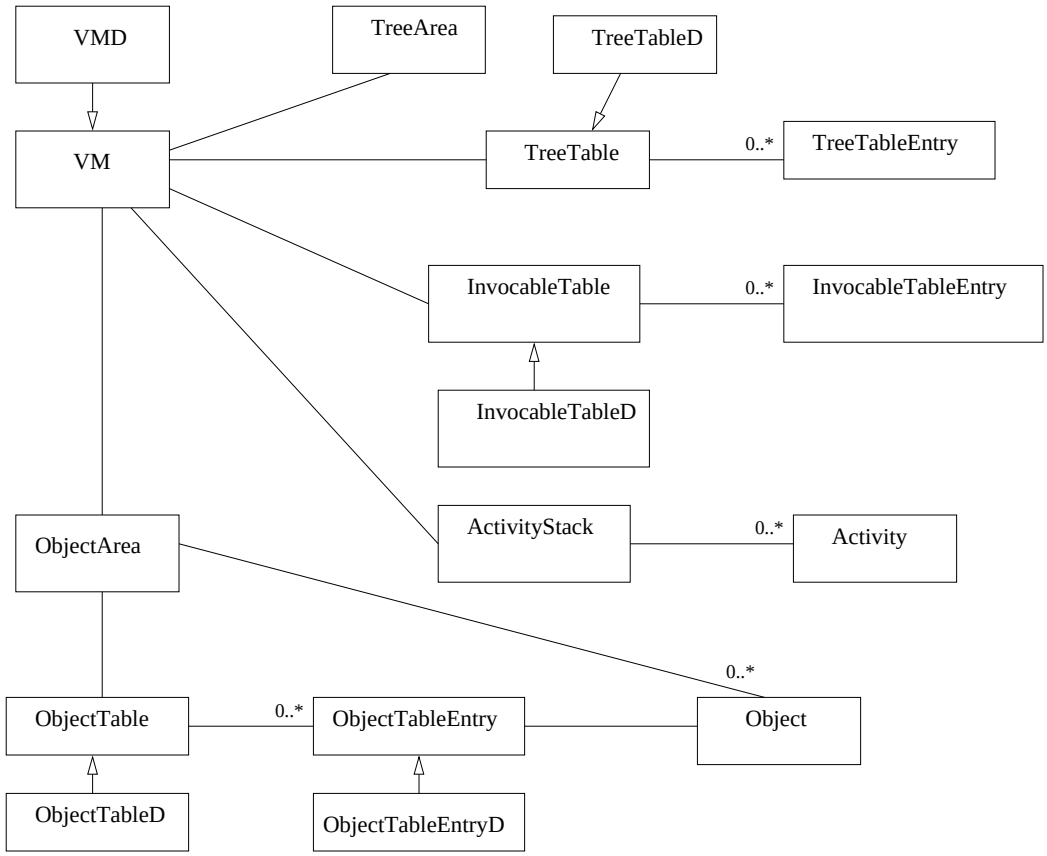


Figura 6.1 Diagrama das principais classes da Máquina Virtual Virtuosi

6.4 Ciclo de Vida da MVV em Termos das Classes que a Compõem

No início do ciclo de vida o método estático `start` da classe `VM` é invocado. Este – o método `start` – recebe dois parâmetros: o nome da classe de aplicação inicial e o nome do construtor que deve ser interpretado. Após isso, os seguintes passos são executados dentro do método `start`.

(1) A tabela de árvores é inicializada;

(2) A tabela de invocáveis é inicializada;

(3) A tabela de objetos é inicializada;

(4) A área de árvores é inicializada;

(5) A área de objetos é inicializada;

(6) O controlador de nomes de objeto é inicializado;

(7) A área de atividades é inicializada;

(8) A MVV invoca o carregador de árvores passando como parâmetro o nome da classe de aplicação inicial. O carregador, através de um processo recursivo já descrito na Seção 3.3.2, carrega para a área de árvores todas as árvores correspondentes às classes utilizadas pela aplicação. Deve-se notar que o carregador de árvores cria todas as entradas necessárias nas tabelas de árvore e tabela de invocáveis, além de resolver todas as referências simbólicas entre árvores;

(9) A MVV cria um objeto na classe de aplicação (um nome único para o objeto e o nome da classe de aplicação são fornecidos como parâmetros) inicial e recupera a respectiva entrada na tabela de objetos;

(10) Com a entrada na tabela de objetos do objeto recém criado, a entrada na tabela de invocáveis que aponta para o construtor inicial (para recuperar a entrada na tabela de invocáveis deve-se invocar um método da tabela de invocáveis passando como parâmetro o nome do construtor, o nome da classe que o possui e a concatenação dos tipos de seus parâmetros) e um conjunto de parâmetros (neste caso vazio), a MVV cria uma nova atividade – a atividade inicial;

(11) A MVV então adiciona a atividade inicial ao topo da pilha de atividades (neste momento vazia). A pilha de atividades então é encarregada de dar início à interpretação da atividade;

(12) Após o final da interpretação da atividade inicial, e por conseqüência o final da interpretação da aplicação, a MVV retira a atividade inicial do topo da pilha de atividades e a MVV termina sua computação;

6.5 Tabela de Referências

As tabelas de referências foram definidas como coleções de objetos em Java. Nestas coleções existem entradas para refências locais e remotas. As entradas locais são compostas por um nome, uma referência à entidade a que se refere, um campo indicando se o objeto está

fixado e outro campo indicando se está congelado. Para a entrada remota a única diferença seria que a referência para a entidade é composta pelo endereço de rede mais a porta de comunicação da MVV destino e a posição onde se encontra a entrada de referência para o objetivo alvo.

6.6 Programação das Primitivas

As primitivas de mobilidade foram programadas no nível do metamodelo. Para isto ele foi estendido com a criação de uma chamada de sistema para cada primitiva. Com isto atendemos o requisito de que o mecanismo é intrínseco ao sistema, já que a mobilidade é disponibilizada no mesmo nível das chamadas do sistema e não adicionando-se uma camada a uma linguagem centralizada existente.

6.7 Cenários de Testes

Foram elaborados os seguintes cenários para validar o mecanismo de mobilidade:

Cenário 1

MVV Alpha	**MVV Beta**
Uma classe X sem referência a outras	Sem classes
Um objeto x sem referência a outros	Sem objetos
Execução: O objeto x é movimentado para a MVV Beta.	

O validação dos resultados dos cenários foram obtidas através de logs geradas pela implementação da MVV. Nestas logs constavam todo o conteúdo das tabelas referencias e das estruturas de todos os objetos instanciados. A cada etapa de cada passo as informações eram colhidas e verificadas.

Cenário 2

MVV Alpha	**MVV Beta**
Duas classes X e Y. Y referenciando X. Dois objetos x e y. y referenciando x.	Sem classes e sem objetos
Execução: O objeto x é movimentado para a MVV Beta.	

Cenário 3

MVV Alpha	**MVV Beta**
Duas classes X e Y. Y referenciando X e Z. y referenciando x e z.	Uma classe Z e um objeto z
Execução: O objeto y é movimentado para a MVV Beta.	

Cenário 4

MVV Alpha	**MVV Beta**
Duas classes X e Y. Y referenciando X e Z. Dois objetos x e y. y referenciando x e z.	Uma classe Z e um objeto z
Execução: O objeto z é movimentado para a MVV Alpha.	

Cenário 5

MVV Alpha	**MVV Beta**
Três classes X, Y e Z. Y referenciando X, e Z referenciando X. Três objetos x, y e z. y referenciando x, e z referenciando x.	Sem classe sem objetos
Execução: O objeto z, x e y são movimentado para a MVV Beta respectivamente.	

Cenário 6

MVV Alpha	**MVV Beta**	**MVV Gama**
Três classes X, Y e Z. Todas se referenciam. Três objetos x,y e z. Todos se referenciam.	Sem classe sem objetos	Sem classe sem objetos
Execução: Os objeto x é movimentado para a MVV Beta, e depois para a MVV Gama. O mesmo ocorre com y e x respectivamente.		

Cenário 7

MVV Alpha	**MVV Beta**
Duas classes X e Y. Y referenciando X e Z, e Z referenciando Y. Dois objetos y e x. y referenciando x e z.	Duas classes Y e Z. Um objeto z referenciando y.
Execução: O objeto z é movimentado para a MVV Alpha e depois retorna para a MVV Beta.	

Capítulo 7

Conclusão e Trabalhos Futuros

O mecanismo de mobilidade de objetos para a Vituosi proposto neste trabalho atende os requisitos de Integridade, Desempenho, Segurança e Funcionalidade definidos (seção 2.3). Na avaliação comparativa com Emerald e Distributet Oz, o mecanismo de mobilidade da Virtuosi obteve uma posição satisfatória nos aspectos mais relevantes aos objetivos da mesma (pedagógico e experimental). Já está previsto no mecanismo, a mobilidade de objetos ativos, o que é um requisito para atender a especificação CORBA para agentes (MAF).

7.1 Contribuição Científica

O trabalho desenvolvido nesta dissertação de mestrado tráz uma contribuição científica nos seguintes aspectos:

- Serve como base para o desenvolvimento de outros mecanismos de mobilidade de objetos.
- Formaliza o mecanismo de mobilidade para a Virtuosi.
- Valida o mecanismo de referências indiretas através de *handle tables*
- Oferece suporte à mobilidade de objetos ativos, o que é um requisito importante para o suporte a agentes.

7.2 Trabalhos Futuros

Através do desenvolvimento deste trabalho é possível prever os seguintes trabalhos futuros:

- Conclusão e melhorias na implementação e modelo do mecanismo:
  - Implementar a movimentação de atividades.
  - Elaborar uma avaliação de desempenho do mecanismo.
  - Permitir a mobilidade de grupos de objetos.
  - Projetar modelo de segurança (segurança contra intrusos, autenticação e permissões de acesso).
  - Criação de um serviço de nomes para as MVVS.
- Trabalhos que utilizarão como base o mecanismo:
  - Mecanismo de RPC.
  - Mecanismo de balanceamento de carga.
  - Suporte a agentes.

Portanto, o trabalho atingiu um nível de robustez, tanto conceitual quanto operacional, a ponto de permitir a sua própria evolução de forma estruturada, além de permitir a interação com outros módulos a ele relacionados na Arquitetura da Virtuosi.

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[1].

A.1 Especificação

Os conceitos de orientação a objeto suportados pela Virtuosi são formalizados através de um diagrama de classes 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.

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

A.1.1 Literais

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 o uso de valores literais em cada uma das situações supracitadas.

Referência a Literais

É 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 à 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 à literal. Também não é possível declarar uma variável local do tipo referência à literal, visto que não existe uma meta-classe que represente o comando para declarar variáveis do tipo referência à literal.

A Figura A.2 mostra o relacionamento das meta-classes que representam valores literais e referências à literal com outros componentes do metamodelo da Virtuosi.

O uso de um valor literal e de uma referência à literal é detalhado na Seção A.1.7, especificamente na discussão sobre comandos de declaração de variáveis.

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

Figura A.1 Código fonte em Aram mostrando os possíveis usos de um valor literal

A.1.2 Bloco de Dados e Índice

Referência a Bloco de dados

Uma **referência a bloco de dados** consiste em uma referência à 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 detalhes na Seção A.1.4.

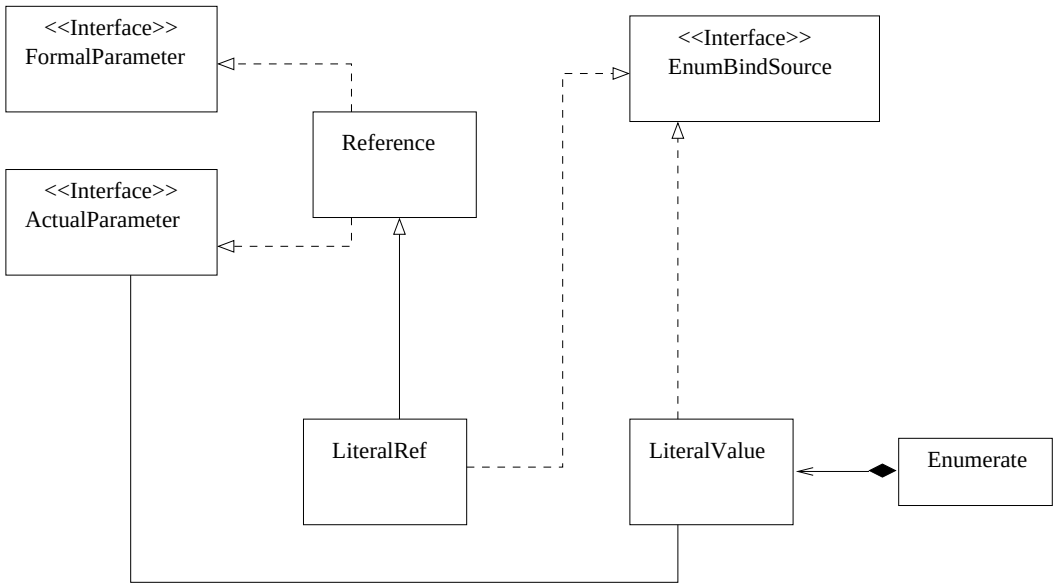


Figura A.2 Relacionamento das meta-classes que representam valores literais e referências à literal com outros componentes do metamodelo da Virtuosi

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 referências 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;

(12) como alvo dos muitos testáveis especiais para manipulação de blocos de dados de classes pré-definidas.

A Figura A.3 mostra o referência a literal-classe que representa a referência a bloco de dado com outros componentes do metamodelo da Virtuosi, excetuando-se os comandos e testáveis especiais.

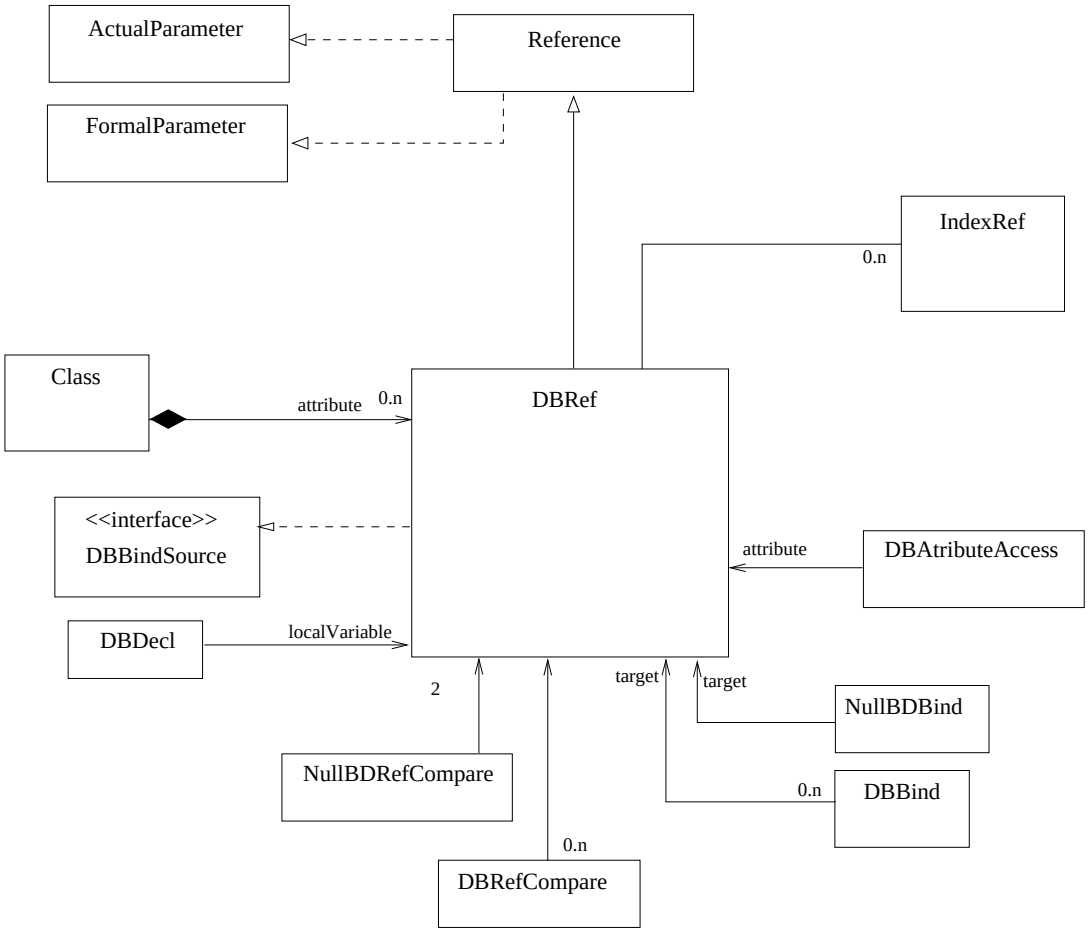


Figura A.3 Referência a literal-classe que representa a referência a bloco de dado com outros componentes do metamodelo da Virtuosi

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.

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 ao 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) como 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[2].

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

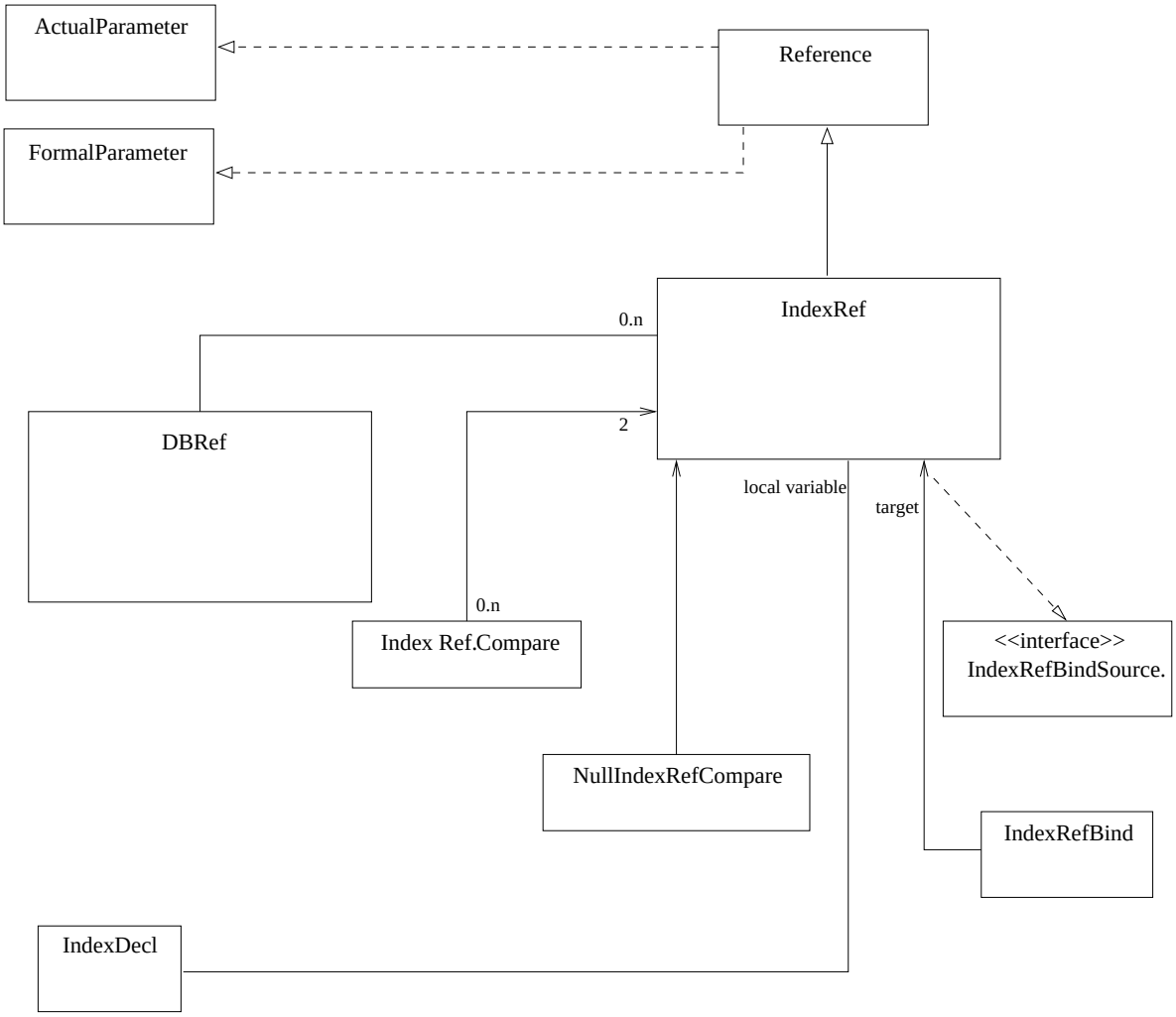


Figura A.5 Relacionamento da meta-classe que representa a referência a índice com outros componentes do metamodelo da Virtuosi

(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.

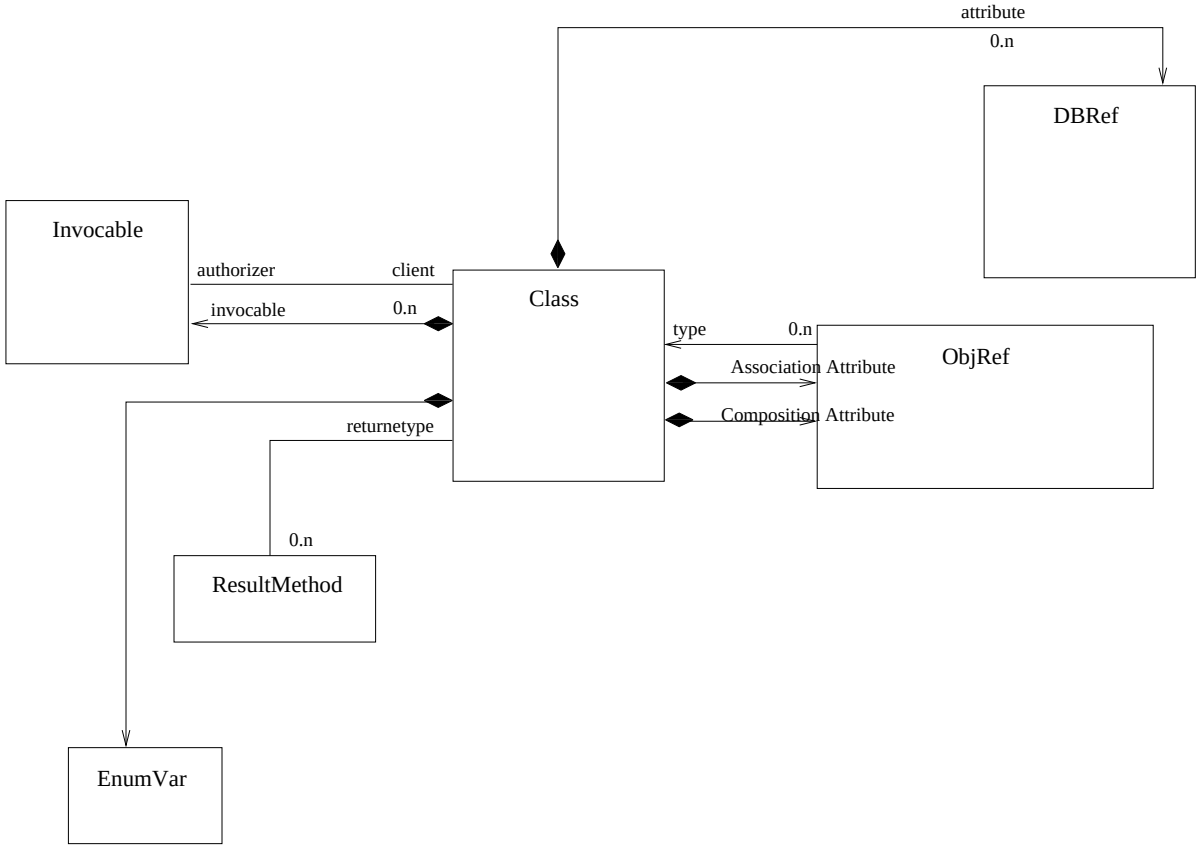


Figura A.6 Relacionamento da meta-classe que representa uma classe de aplicação com outros componentes do metamodelo da Virtuosi

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[3], mas pode ter muitas classes herdeiras recursivamente. Entretanto, uma classe não pode ser direta ou indiretamente ancestral de si própria. Assim, um conjunto de clas-

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

ses 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.

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.

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.

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 para uma classe conforme mostra a Figura A.8.

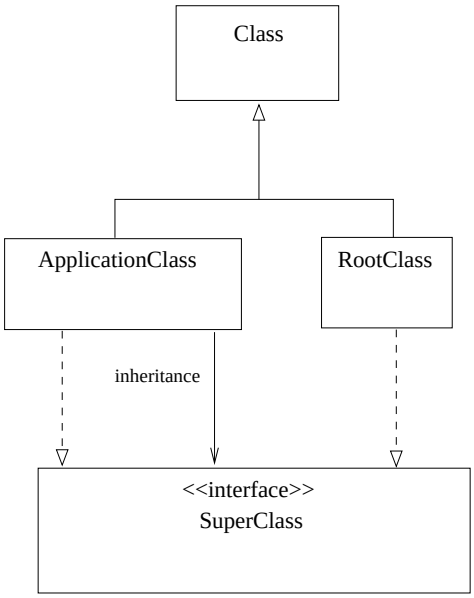


Figura A.7 Relação de herança entre classes segundo o metamodelo da Virtuosi

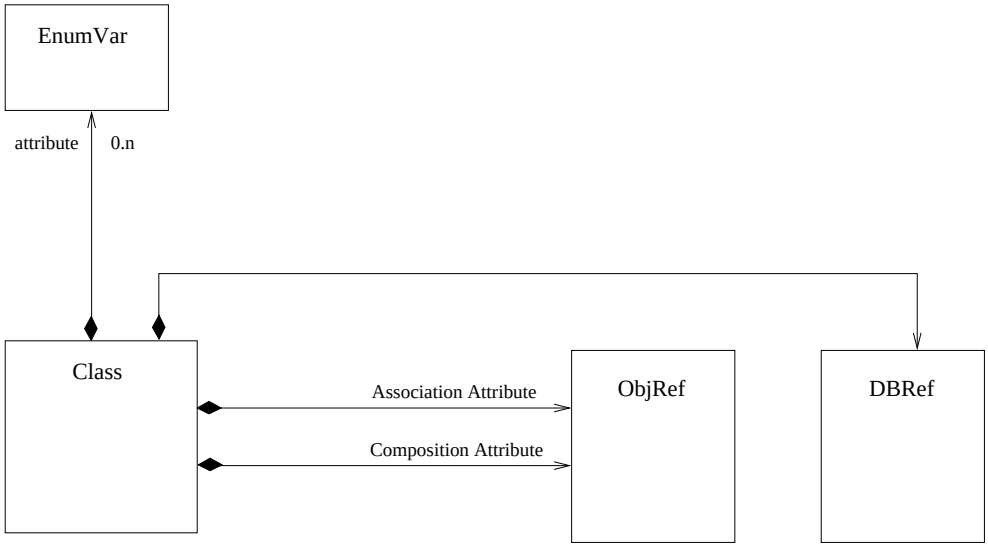


Figura A.8 Os três tipos de atributos possíveis em uma classe segundo o metamodelo da Virtuosi

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;

(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 referências 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) como uma das possibilidades para a origem da atribuição em um comando de atribuição de referência a objeto;

(15) como 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.

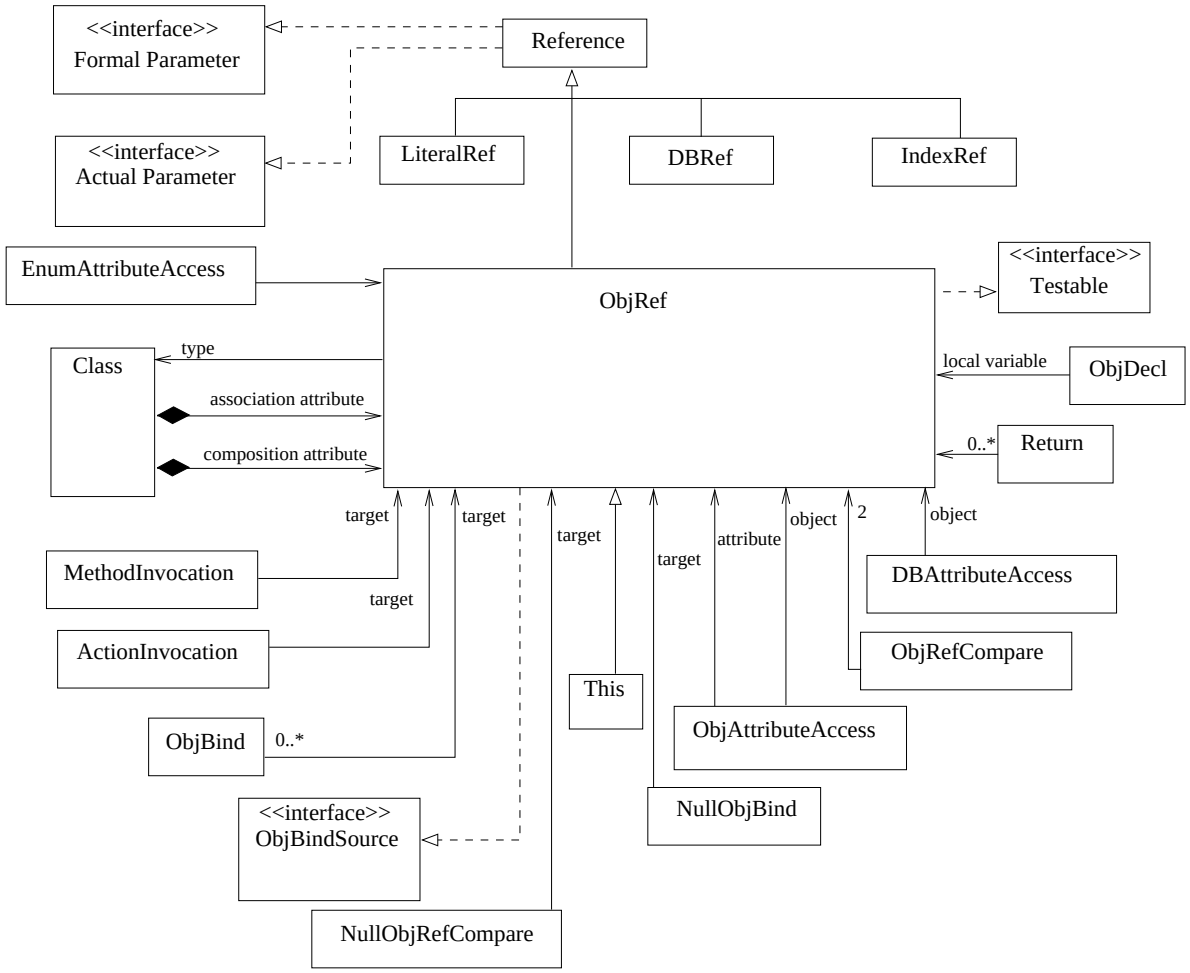


Figura A.9 Relacionamento da meta-classe que representa a referência a objeto com outros componentes do metamodelo da Virtuosi

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`.

```
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

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 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

```
class Pessoa   {
    composition String nome;    //classe pre−definida
```

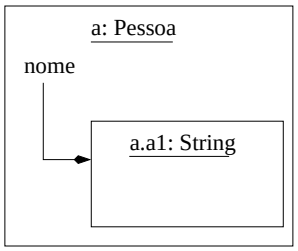


Figura A.11 Código fonte em Aram e diagrama de objeto de uma relação de composição entre uma classe e um atributo

associação um objeto associado pode ser referenciado por qualquer outro objeto.

```
class Pessoa   {
    association String endereco;    //classe pre−definida
```

Figura A.12 Código fonte em Aram e diagrama de objeto de uma relação de associação entre uma classe e um atributo

A Figura A.13 mostra como o metamodelo da Virtuosi formaliza as relações de composição e associação entre uma classe e seus atributos.

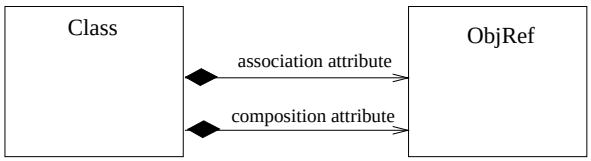


Figura A.13 As duas maneiras em que uma referência a objeto representa o papel de atributo segundo o metamodelo da Virtuosi

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 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 ambas 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.

```
class Integer   {
    datablock value;
```

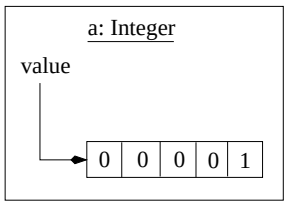


Figura A.14 Um exemplo de atributo do tipo bloco de dados e uma representação de um objeto correspondente – código fonte em Aram

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 de variável enumerada e uma ilustração de uma instância desta classe contendo um enumerado. O código fonte da Figura abstrai um dado cuja face virada para cima sempre assume um dos valores definidos, no caso um(1) a seis(6).

```
class Boolean   {
    enum { true, false}   value = false ;
```

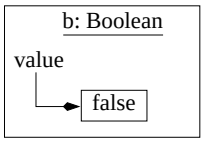


Figura A.15 Um exemplo de atributo do tipo enumerado e uma representação de um objeto correspondente – código fonte em Aram

A.1.5 Referências

Existem quatro tipos de referência no metamodelo da Virtuosi, a saber:

- referência a objeto;
- referência a 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.

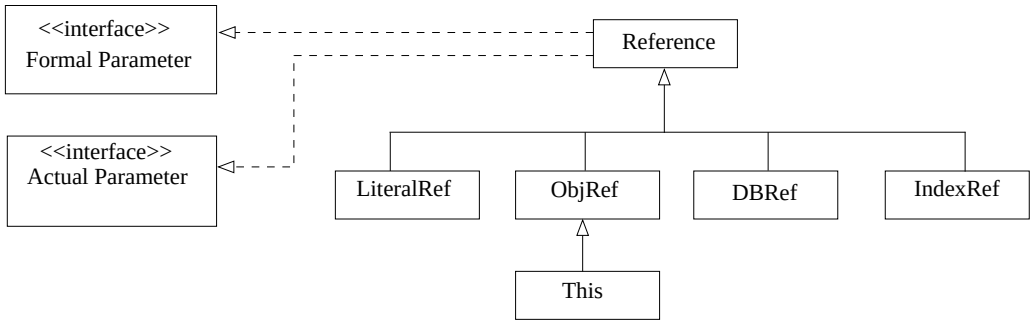


Figura A.16 Relacionamento entre as meta-classes que definem os tipos de referência na Virtuosi

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. Uma referência do tipo `This` é utilizada quando dentro de um método deseja-se invocar um método da própria classe e sobre a própria instância 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.

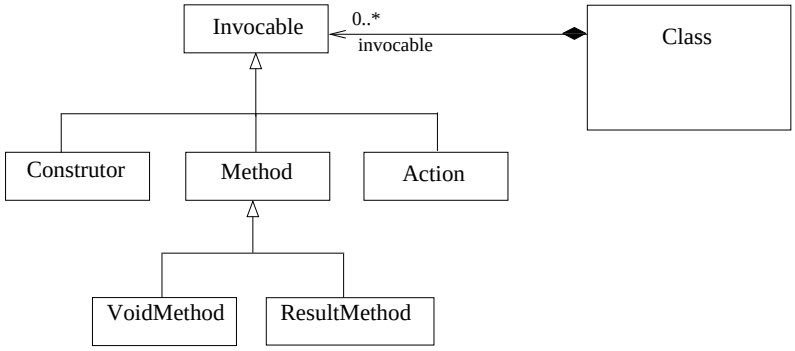


Figura A.17 Relação entre uma classe e as possíveis implementações de suas operações

Um método, um construtor ou uma ação podem sofrer invocação e, por isso, o metamodelo da Virtuosi os formaliza como **invocáveis**[4].

A Figura A.18 mostra o relacionamento da meta-classe Invocável com outros componentes do metamodelo da Virtuosi.

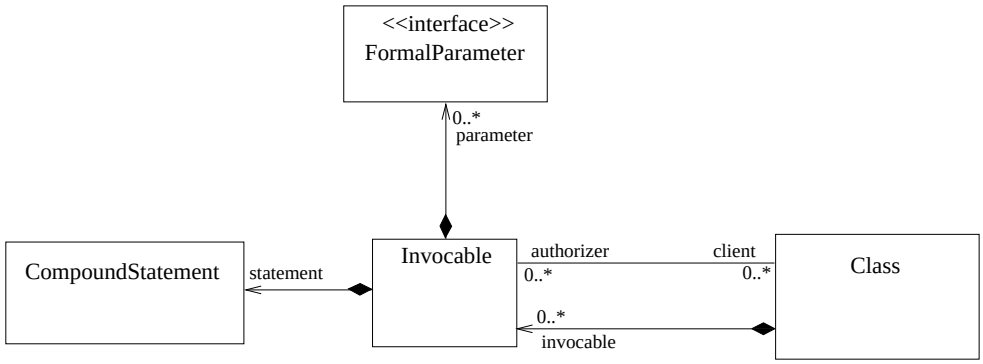


Figura A.18 Relacionamento da meta-classe Invocável com outros componentes do metamodelo da Virtuosi

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

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**[5], 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**[6].

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 à literal[7] 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`.

[5]Do inglês ***formal parameter***.

[6]Do inglês ***actual parameter***.

[7]Embora um parâmetro seja utilizado pelo invocável da mesma forma que uma variável local, no caso de parâmetros do tipo referência à literal, o parâmetro não pode sofrer atribuição, visto que, não existe um comando de atribuição para referência à literal. Portanto, uma referência à 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

O metamodelo da Virtuosi formaliza a relação entre um invocável e seus parâmetros, conforme mostra a Figura A.18.

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.20. A Figura mostra três casos distintos: o método *exportedToAllMethod* – exportado 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.21.

Construtor Um construtor não faz parte das operações definidas por um TAD (Tipo Abstrato de Dado) 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.

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 de valor** – muitas vezes chamado de procedimento – e **método com retornode valor** – muitas vezes chamado função – conforme formalizado pelo metamodelo da Virtuosi e mostrado na Figura A.22.

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

```
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.20 Métodos com diferentes listas de exportação – Código fonte em Aram

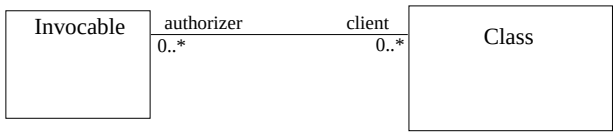


Figura A.21 Relação de um Invocável e sua lista de exportação

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.23 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

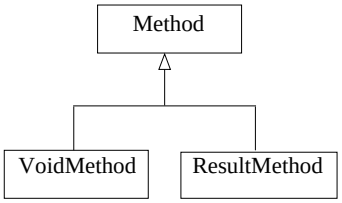


Figura A.22 Métodos com retorno e métodos sem retorno

```
class Taxi {
   ...

   // método sem retorno
   method void sairPassageiro( ) exports { Principal }
   {
       ...
   }
   // método com retorno
   method Boolean entrarPassageiro( Pessoa p ) exports { Principal }
   {
       ...
       return resultado;
   }
}
```

Figura A.23 Diferenciação entre métodos com retorno e métodos sem retorno

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**. Tanto o comando de desvio quanto o comando resultado de teste são detalhados na Seção A.1.7.

A Figura A.24 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.24 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.

Composição de Comandos

Um invocável define uma seqüência de comandos. Comandos podem ser simples ou compostos. Um **comando simples**[8] pode ser uma atribuição, uma invocação de operação, um desvio condicional, conforme detalhado no restante dessa Seção. Um **comando composto**[9] 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.25.

[8]Do inglês ***simple statement***.
[9]Do inglês ***compound statement***.

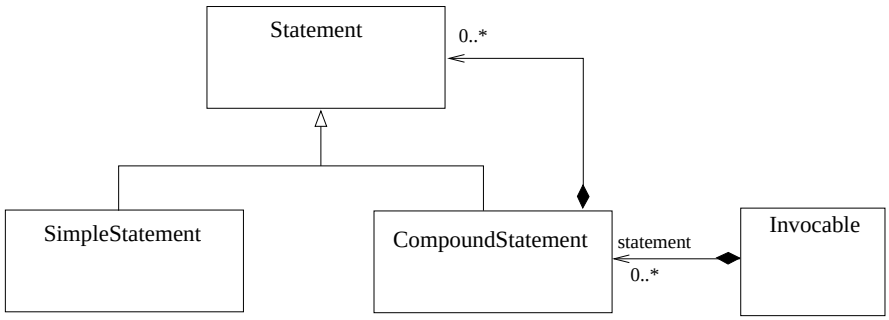


Figura A.25 Relacionamento de herança entre as meta-classes comando, comando simples e comando composto

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 comandos para a declaração de alguns tipos de referência utilizadas como variáveis locais. A Figura A.26 mostra a hierarquia de meta-classes que determina os comandos simples disponíveis para a declaração de variáveis locais, segundo o metamodelo da Virtuosi.

Conforme mostra a Figura A.26, é possível declarar variáveis locais do tipo:

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

Observando novamente a Figura A.26 nota-se que não existe declaração de variável local do tipo referência à literal. Isto ocorre porque uma referência à literal somente é permitida

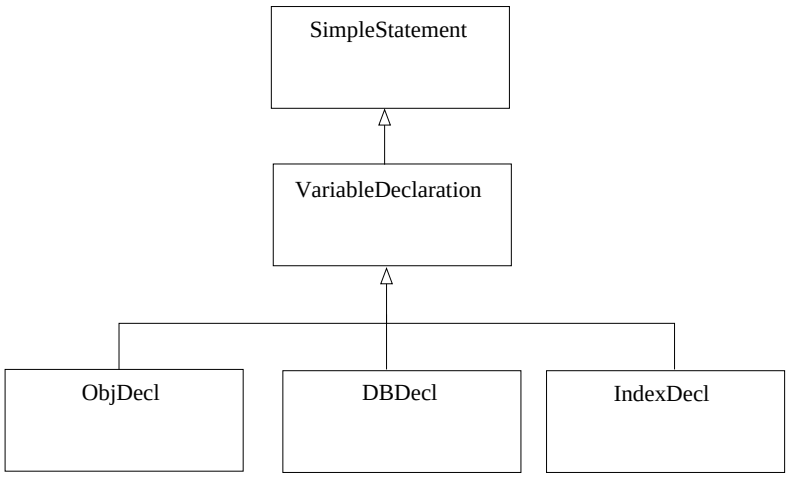


Figura A.26 Tipos de declarações para variáveis locais

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 à literal. A Figura A.27 mostra uma invocação de um método passando um valor literal e a declaração desse mesmo método contendo um parâmetro formal do tipo referência à literal.

Declaração de Variáveis do Tipo Referência a Objeto A Figura A.28 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.29 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 declaração de variável local do tipo referência a índice. A Figura A.30 mostra a declaração de uma variável local do tipo referência a índice. Após a interpretação desse comando a referência não possui

```
class Person {
   ...
   constructor create()
   {
       Integer age;
       age = Integer.make(26); // '26' é um valor literal
   }
}
...
class Integer {
   ...
   constructor make ( Literal charSequence){
       ...
   }
}
```

Figura A.27 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.28 Declaração de uma variável local do tipo referência a objeto – código fonte em Aram

um alvo, ou seja, não está apontando alguma seqüência de dados binários em memória, ou seja, é uma referência nula.

```
class Integer {
   method void add( Integer i ) exports all
   {
       // declaração de variável local inicialmente nula.
       datablock d;
       ...
   }
   ...
}
```

Figura A.29 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.30 Declaração de uma variável local do tipo referência a índice – código fonte em Aram

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 caractere '=' chamado de caractere de atribuição. O que estiver à esquerda do caractere de atribuição é chamado **alvo da atribuição**, e o que estiver à direita do caractere 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.31 mostra o relacionamento das meta-classes que representam os dois comandos de atribuição de referência a objeto com outros componentes do metamodelo da Virtuosi.

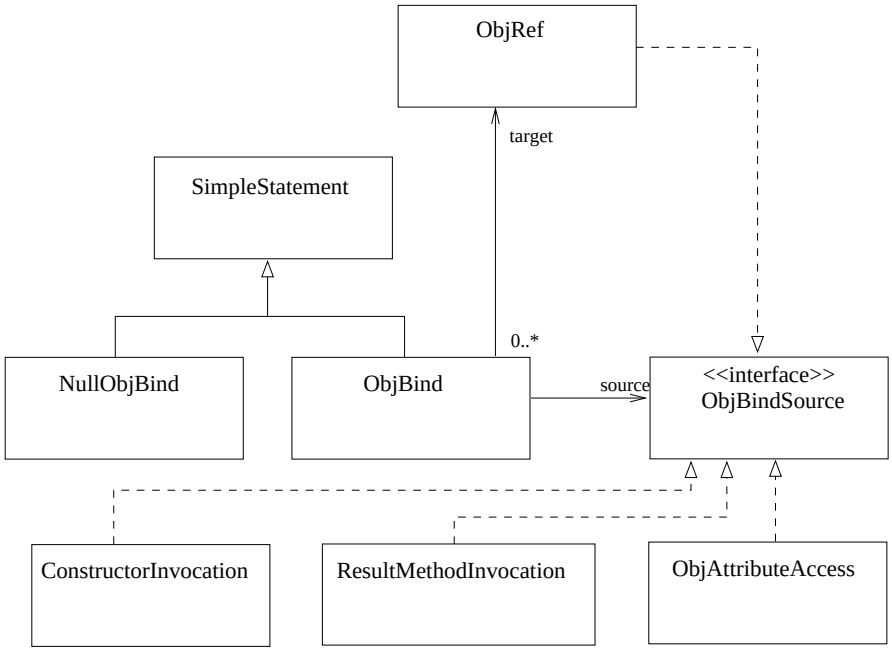


Figura A.31 Relacionamento das meta-classes que representam os comandos de atribuição de referência a objeto com outros componentes do metamodelo da Virtuosi

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.32 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.

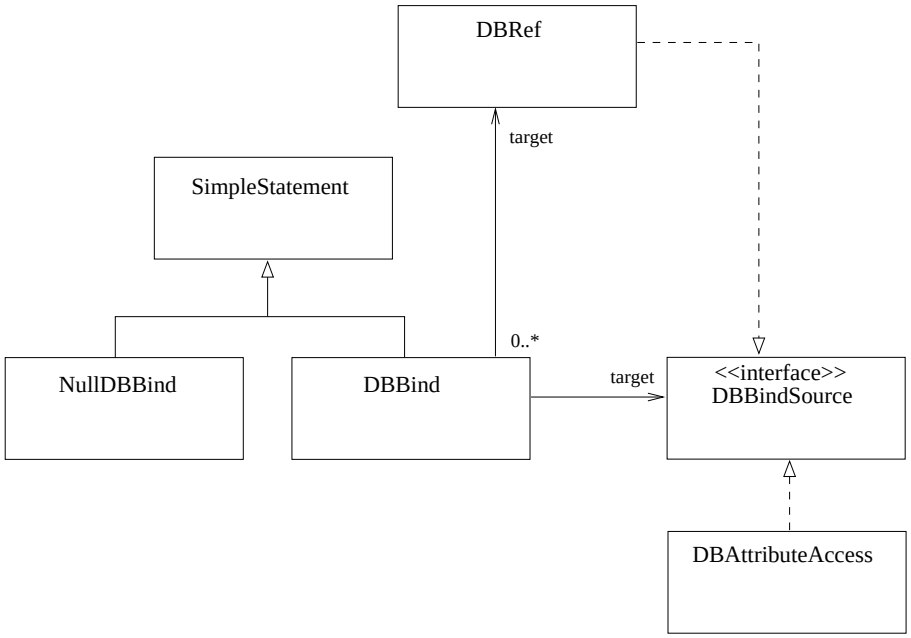


Figura A.32 Relacionamento das meta-classes que representam os dois comandos de atribuição de referência a bloco de dados com outros componentes do metamodelo da Virtuosi

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.33 mostra o relacionamento da meta-classe comando de atribuição de referência a índice com outros componentes do metamodelo da Virtuosi.

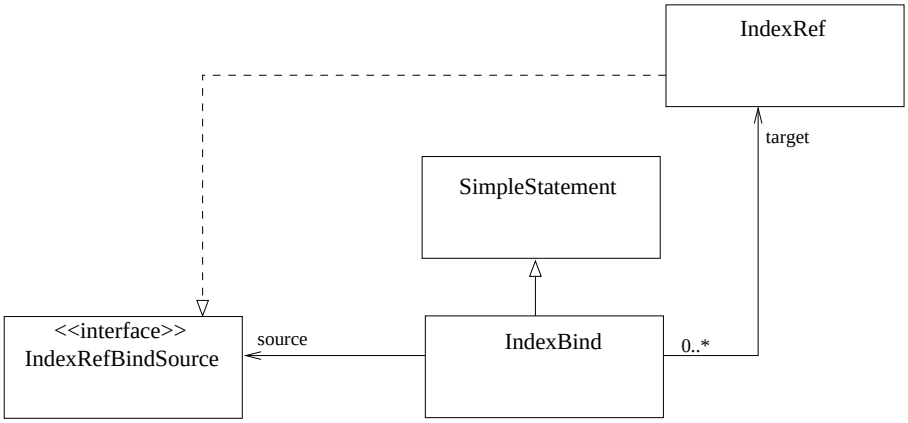


Figura A.33 Relacionamento da meta-classe comando de atribuição de referência a índice com outros componentes do metamodelo da Virtuosi

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 à 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.34 mostra o relacionamento da meta-classe comando de atribuição variável enumerada com outros componentes do metamodelo da Virtuosi.

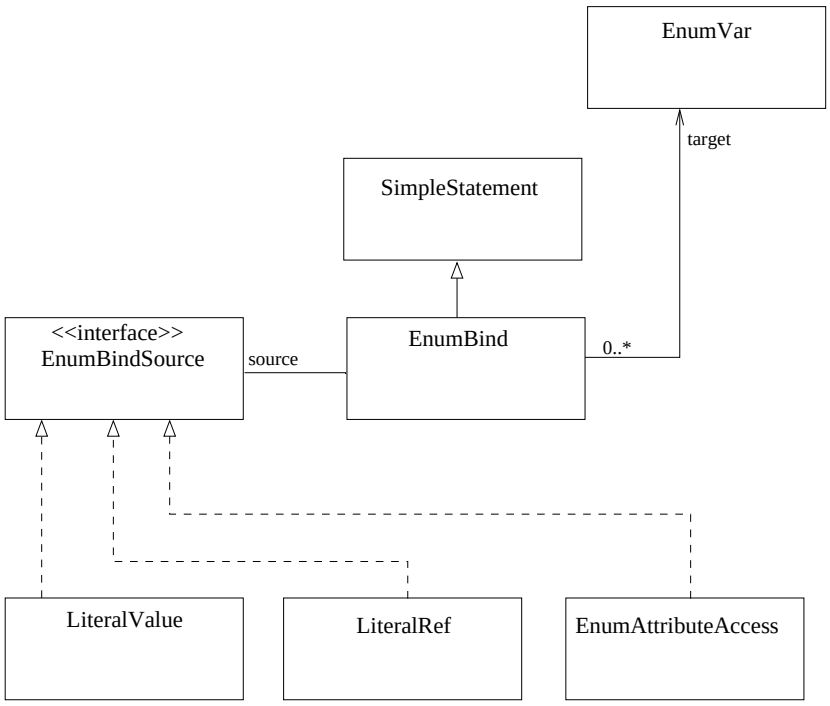


Figura A.34 Relacionamento da meta-classe comando de atribuição variável enumerada com outros componentes do metamodelo da Virtuosi

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.35.

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

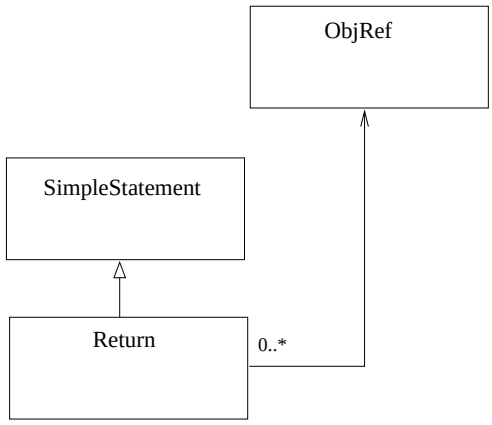


Figura A.35 Relacionamento entre um comando de retorno e uma referência a objeto

comando de desvio condicional nessa Seção.

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.36 mostra o 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.37.

Conforme a Figura A.37 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 causa a interpretação da seqüência de comandos definidos em um método com retorno.

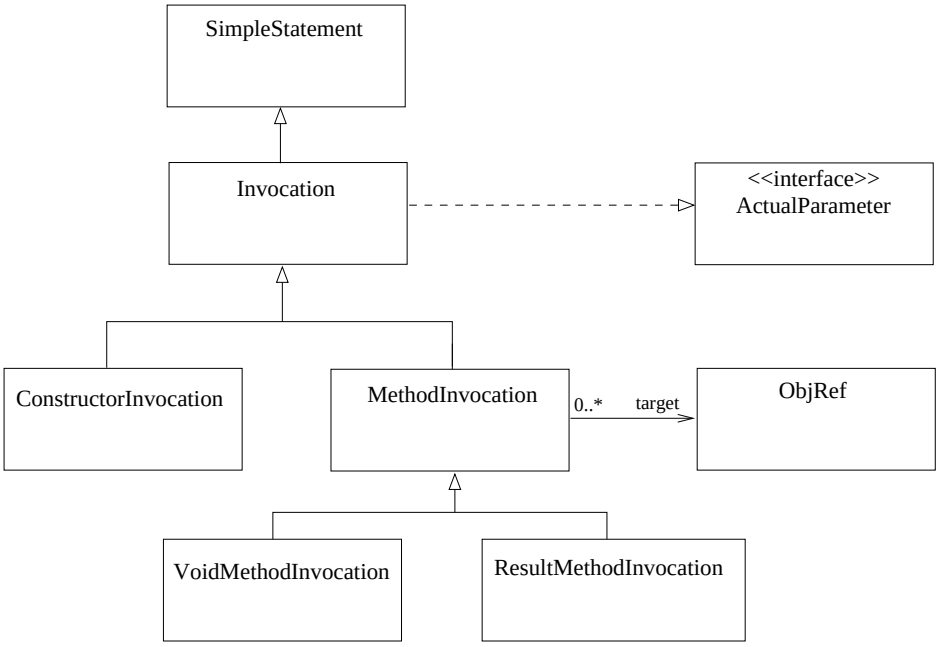


Figura A.36 Relacionamento da meta-classe que representa um comando de invocação com sua hierarquia e outros componentes do metamodelo da Virtuosi

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.

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 ex-

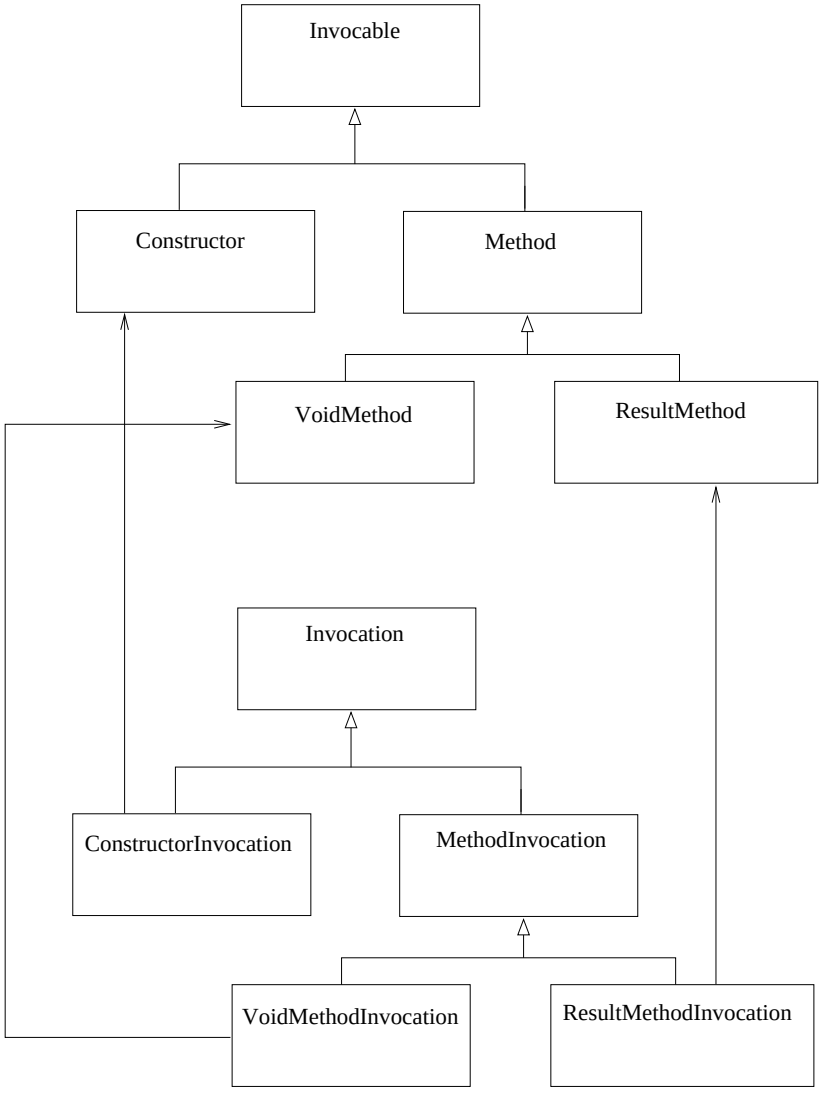


Figura A.37 Relação entre os comandos de invocação e os invocáveis

plicitamente no código fonte, ele sempre é utilizado em conjunto a 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**[10]. 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 comandos, conforme mostra a Figura A.38.

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.

[10]Do inglês ***testable*** (o termo testável ainda não está registrado nos dicionários da língua portuguesa).

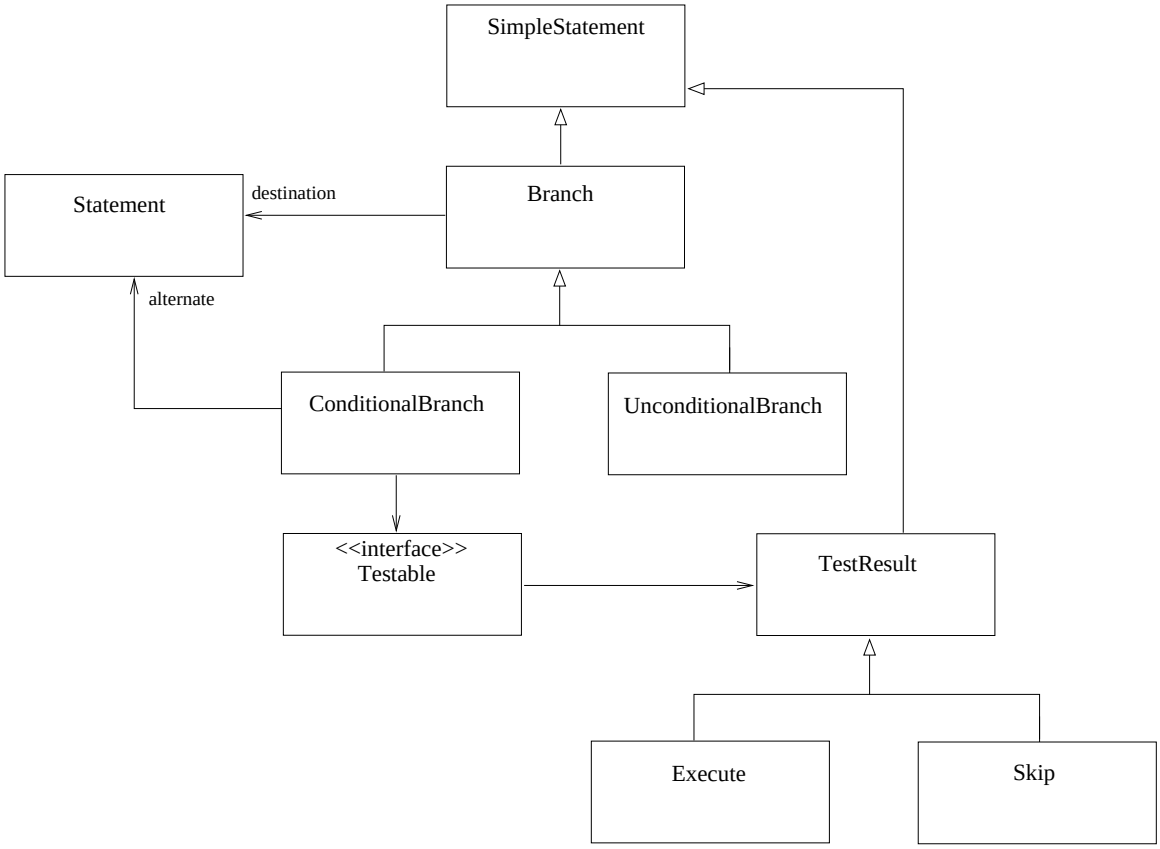


Figura A.38 Comandos de desvio e como se relacionam com as seqüências de comandos

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.39.

A Figura A.40 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*.

A Figura A.41 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

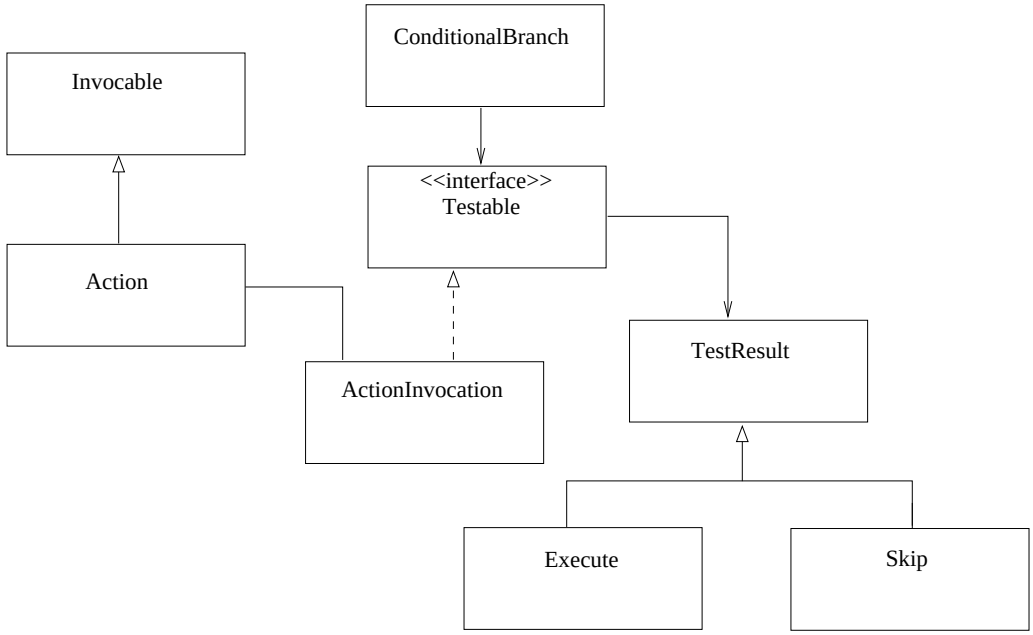


Figura A.39 Relação de um desvio condicional, um testável, uma invocação de ação e uma ação

```
class Pessoa
{
   ...
   action obesa() exports all
   {
       ...
       if (massa_.gt(h))// gt significa greater than
           execute;
       else
           skip;
   }
 ...
}
```

Figura A.40 Desvio condicional com um testável que é uma invocação de uma ação – código fonte em Aram

```
class Boolean
{
   enum { true, false } value = false;


   ...
   method void flip( ) exports all
   {
       if ( value == true )
           value = false;
       else
           value = true;
   }
}
```

Figura A.41 Desvio condicional com um testável que é uma comparação de valor entre uma variável enumerada e um valor literal – código fonte em Aram

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.42 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;
- 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

```
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.42 Definição de uma ação padrão e seu respectivo uso por um comando de desvio

posição em uma mesma seqüência de dados binários em memória;

- uma comparação entre uma referência a objeto e uma referência nula, chamada **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 à 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.43.

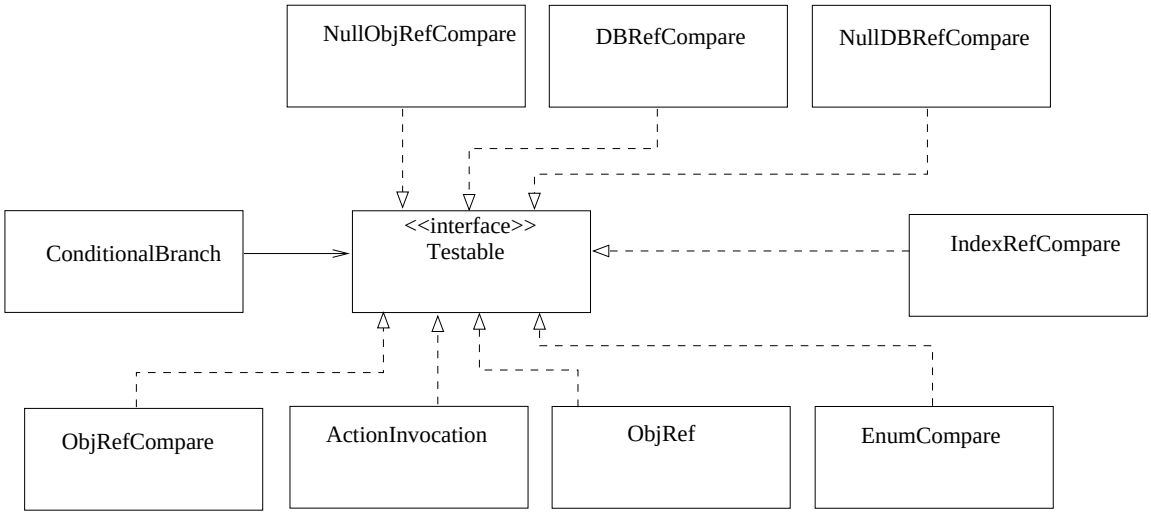


Figura A.43 Relação de um desvio condicional com todos os testáveis possíveis

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.44 mostra.

```
class Principal
{
   constructor iniciar() exports  all
   {
       ...
       Integer t = Integer.make(2);
       while (andrea.obesa()) {
           andrea.emagreca(t);
       }
   }
}
```

Figura A.44 Estrutura de repetição realizada pela combinação de um desvio condicional e um desvio incondicional – código fonte em Aram

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 cons- trução 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.

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.45.

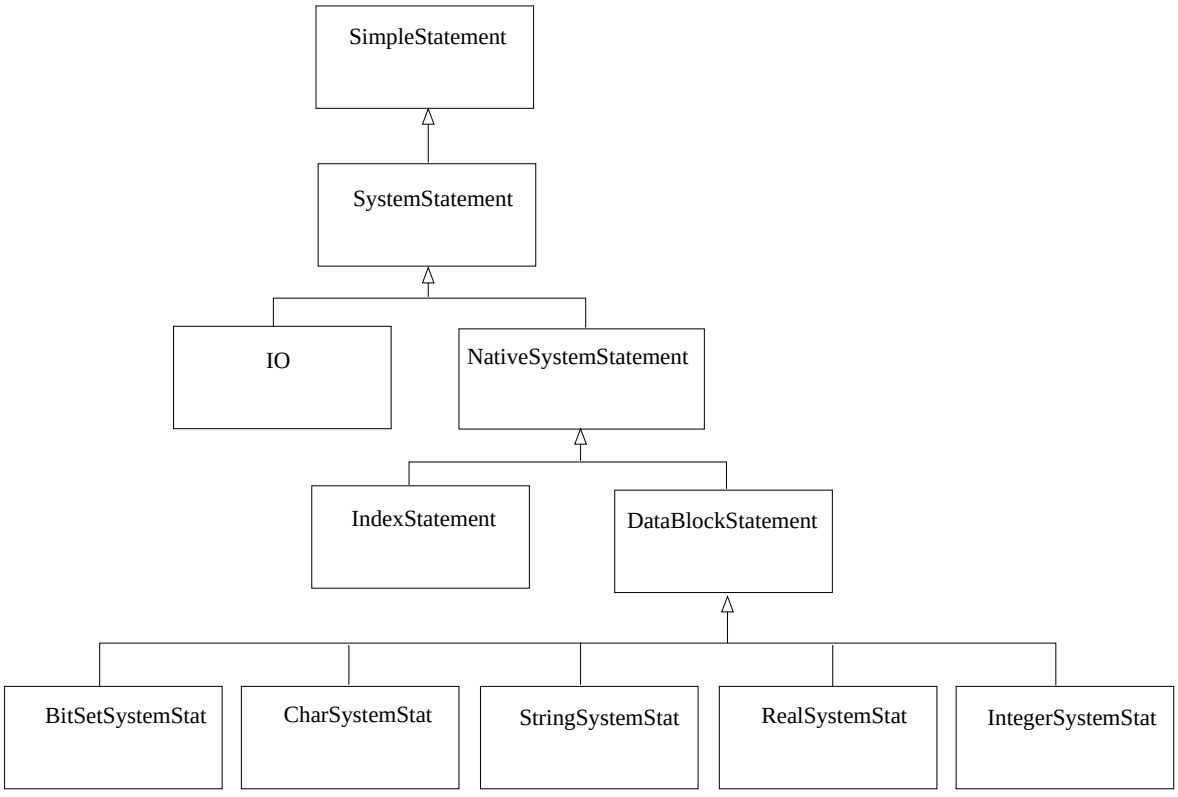


Figura A.45 Comandos de sistema disponibilizados pelo sistema

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.46 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.

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 classes que armazenem valores inteiros com bloco de dados de tamanho diferente. Isso é possível porque os comandos de sistema que manipulam blocos 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.

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

```
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.46 Uso de dois comandos de sistema para facilitar a construção de uma classe pré-definida *Integer* – código fonte em Aram

metamodelo da Virtuosi define uma hierarquia de testáveis de sistema, conforme mostra a Figura A.47.

Observa-se que abaixo da meta-classe representando 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;

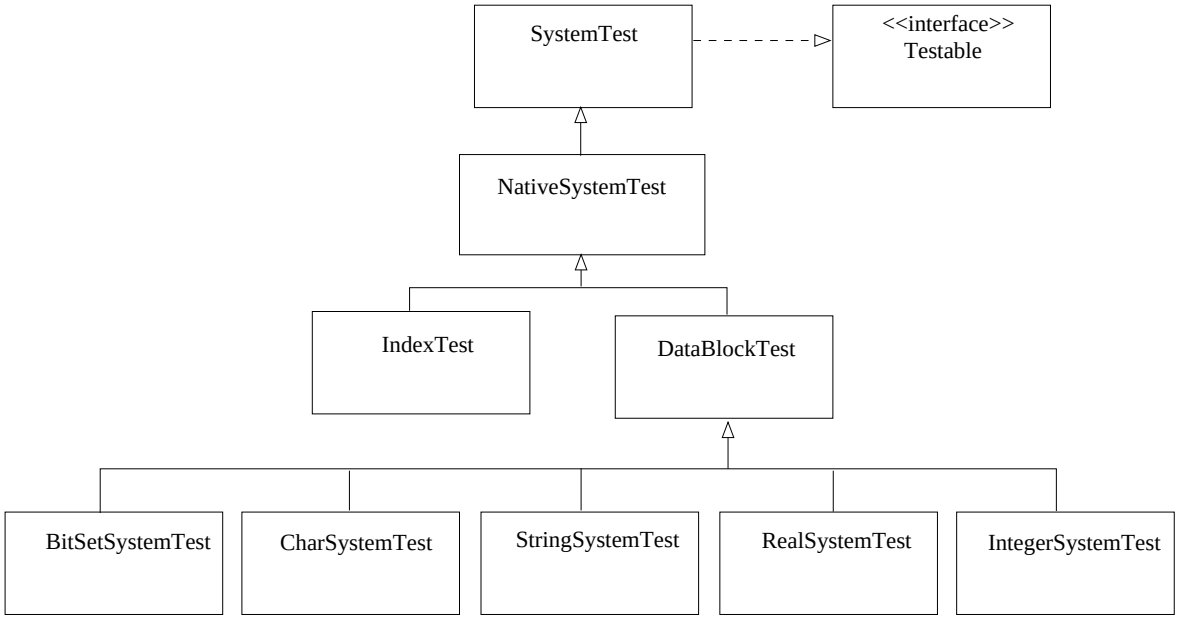


Figura A.47 Testáveis de sistema disponibilizados pelo sistema

- testáveis de sistema para valores de conjunto de caracter;

A Figura A.48 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.48 Uso de testáveis especiais nos comandos de desvio utilizados na construção de uma classe pré-definida *Integer* – código fonte em Aram

Referências Bibliográficas

[Cal, 2004] (2004). *Virtuosi Architecture Issues*, volume 3061 of *Lecture Notes in Computer Science*. Springer.

[Amaral et al., 1992] Amaral, P., Jacque, C., Jensen, P., Lea, R., and Mirowski, A. (1992). Transparent object migration in cool-2. In *ECOOP'92 European Conference on Object-Oriented Programming*.

[Arnold and Gosling, 1996] Arnold, K. and Gosling, J. (1996). *The Java Programming Language*. Addison Wesley.

[Calsavara, 2000] Calsavara, A. (2000). Virtuosi: Máquinas virtuais para objetos distribuídos. In *Trabalho apresentado em concurso para professor titular da PUC-PR*.

[Cardelli, 1995] Cardelli, L. (1995). A language with distributed scope. *ACM Transactions on Computer Systems*, pages 286–297. Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.

[Crowcroft, 1996] Crowcroft, J. (1996). *Open Distributed Systems*. Artech House, Inc.

[Fowler, 1985] Fowler, R. J. (1985). *Descentralized Object Finding Using Forwarding Addresses*. PhD thesis, University of Washington.

[Franz and Kistler, 1997] Franz, M. and Kistler, T. (1997). Does java have alternatives? In *Proceedings of the California Software Symposium CSS '97*, pages 5–10.

[Franz and Thomas, 1999] Franz, M. and Thomas, K. (1999). A Tree-Based Alternative to Java Byte-Codes. *In International Journal of Parallel Programming*.

[Haridi et al., 1997] Haridi, S., Roy, P. V., and Smolka, G. (1997). An overview of the design of distributed oz. *International Symposium on Parallel Symbolic Computation*, pages 176–187.

[Hewitt, 1980] Hewitt, C. (1980). Transparent object migration in cool-2. In *Lisp Conference*, pages 107–118, Palo Alto, California.

[HU et al., 2003] HU, Y. C., Cox, A., Wallach, D., and Zwaenepoel, W. (2003). Run-tim support for distributed sharing in safe languages. *ACM Transactions on Computer Systems*, 21(1).

[Jul et al., 1988] Jul, E., Levy, H., Hutchinson, N., and Black, A. (1988). Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133.

[Kistler and Franz, 1997] Kistler, T. and Franz, M. (1997). A tree-based alternative to java byte-codes. In *Proceedings of the International Workshop on Security and Efficiency Aspects of Java '97*. Also published as Technical Report No. 96-58, Department of Information and Computer Science, University of California, Irvine, December 1996.

[Kolb, 2004] Kolb, C. (2004). Um sistema de execução para software orientado a objeto baseado em Árvores de programa. In *Dissertação apresentada na defesa de Mestrado da PUC-PR*.

[M. Shapiro, 1989] M. Shapiro, Y. G. (1989). Sos:an object-oriented operationg system – assessment and perspectives. *Computing Systems*, 2:287–387.

[Nutttal., 1994] Nutttal., M. (1994). A brief survey of systems providing process or object migration. *ACM Operating Systems Review*, 28:64–80.

[OMG, 2000] OMG (2000). Mobile agent facility specification.

[OMG, 2002] OMG (2002). Life cicle service specification.

[Oshima et al., 1998] Oshima, M., Karjoth, G., and Ono, K. (1998). Aglets specification. *www.trl.ibm.com/aglets*.

[Otte et al., 1996] Otte, R., Patrik, P., and e Mark Roy (1996). *Understanding CORBA: The Common Object Request Broker Architecture*. Prentice Hall PTR.

[Roy et al., 1999] Roy, P. V., Haridi, S., Brand, P., and Collet, R. (1999). A lightweight reliable object migration protocol. *Lecture Notes in Computer Science*, 1686.

[Roy et al., 1997] Roy, P. V., Haridi, S., Brand, P., Smolka, G., Mehl, M., and Scheidhauer, R. (1997). Mobile objects in distributed oz. *ACM Transactions on Programming Languages and Systems*, 19:804–851.

[Stevens, 1990] Stevens, W. R. (1990). *Unix Networking Programming*. Prentice Hall software Series. Remote Procedure Calls.

[Tanenbaum, 1994] Tanenbaum, A. S. (1994). Distributed operation systems. *Prentice-Hall, Englewood Cliffs, N. J.*

[Venners, 1997] Venners, B. (1997). The architecture of aglets. *JavaWorld*.

[Vestal, 1987] Vestal, S. (1987). *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming*. PhD thesis, University of Washington.

[Wikstrõm, 1994] Wikstrõm, C. (1994). Distributed programming in erlang. In *Primeiro Simpósio Internacional sobre Computação Simbólica (PASCO '94)*, pages 412–421. PASCO '94.