Pontifícia Universidade Católica do Paraná – PUCPR
Centro de Ciências Exatas e de Tecnologia – CCET

Um Sistema de Execução para Software Orientado a Objeto Baseado em Árvores de Programa

Carlos José Johann Kolb

Dissertação apresentada ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná, como parte dos requisitos para obtenção do título de Mestre em Informática Aplicada

Julho de 2004

Abstract

Since the end of the 70's, many studies concerning portability of computer applications through the use of virtual machines can be observed. This tendency increased with the improve of network technology and the need for integration among computers from different platforms. Another tendency fully accepted is the object orientation paradigm. The combination of these tendencies is observed in technologies such as Java, Smalltalk and SELF. In addition, there are studies that aim at the evolution of the intermediate representation usually applied to achieve portability (the bytecode approach). This dissertation aims at defining and implementing a virtual machine architecture that interprets an intermediate representation in the form of a program tree. Moreover, the virtual machine architecture should facilitate distributed computation by accessing both object code and application object instances through handle-tables. A program tree should be viewed as an object graph whose objects represent the syntactic elements of a source code written in some object-oriented programming language. Each one of the objects that compose the program tree is an instance of a class defined by a class model – the Virtuosi metamodel – that establishes which concepts an object oriented programming language should have to be compliant with the Virtuosi execution system. With this execution environment it should be possible to assess the use of program trees as an intermediate representation for computational systems. This first step will allow the Virtuosi metamodel to be assessed and extended to accomplish with object distribution, object migration and reflection computation.

key-words: object orientation, programming languages, virtual machines, distributed systems, distributed objects, metamodel.

Resumo

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. Outra tendência amplamente aceita é o paradigma de orientação a objetos. A combinação destas tendências é observada em tecnologias como Java, SmallTalk, SELF, entre outras. Existem, ainda, estudos que visam a evolução da representação intermediária tradicionalmente utilizada para obtenção de portabilidade (abordagem no estilo do *bytecode*). Esta dissertação visa definir e implementar uma arquitetura de máquina virtual que interprete software orientado a objeto utilizando uma representação intermediária na forma de árvore de programa. Além disso, a arquitetura da máquina virtual deve facilitar a computação distribuída por acessar tanto código objeto quanto objetos instâncias de classes de aplicação de forma indireta através do uso de tabelas de manipulação. Uma árvore de programa deve ser vista como um grafo cujos nós são objetos que representam os elementos sintáticos presentes no código fonte de uma linguagem de programação orientada a objetos. Cada um dos objetos que compõem a árvore de programa é instância de uma classe definida por um modelo de classes, chamado de metamodelo Virtuosi, que define e restringe quais os conceitos que uma linguagem de programação orientada a objetos compatível com o ambiente de execução Virtuosi deve possuir. De posse do ambiente de execução, será possível avaliar a utilização de árvores de programa como representação intermediária de sistemas computacionais. Este primeiro passo permitirá que o metamodelo Virtuosi seja avaliado e estendido para contemplar outras características desejáveis, tais como distribuição de objetos, migração de objetos e reflexão computacional.

Palavras-chaves: orientação a objetos, linguagens de programação, máquinas virtuais, sistemas distribuídos, objetos distribuídos, metamodelos.

Agradecimentos

Agradeço muito ao meu orientador Alcides Calsavara por toda a paciência e grande ajuda (grande mesmo) prestada durante todo o período do mestrado! Ajuda tanto no trabalho realizado quanto nas situações difíceis.

A minha querida esposa Letícia pela paciência, pela dedicação e amor. E também por ter sido uma "viúva de marido vivo"desde que casamos...

A minha mãe e ao meu pai pela oportunidade de estudo, pela educação, por todo o suporte. E principalmente por ensinar as coisas que realmente importam na vida!

A minha família em geral, aos meus sobrinhos pela alegria de sempre, à minha irmã Luisa pela ajuda nas últimas horas (literais) e ao meus sogros pelo suporte de sempre!

Aos companheiros de projeto Agnaldo, Aron, Gildo, Juarez e Leonardo pelas nossas longas reuniões nas Segundas-Feiras.

Aos meus amigos, meus poucos e bons amigos, por ainda serem meus amigos.

Ao Professor Doutor Altair Olivo Santin e ao Professor Doutor Edmundo Roberto Mauro Madeira pela disposição de participar na banca de avaliação dessa dissertação.

Finalmente, agradeceço a Jeová por toda a criação do Universo e por todas as coisas boas da vida!

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A importância de computação distribuída tem crescido significativamente nos últimos anos devido ao incremento no uso da Internet como meio de implantação de sistemas de informação. Muitas aplicações novas têm surgido e outras são esperadas em futuro próximo, especialmente nos campos de software embarcado (*embedded systems*) e dispositivos móveis (*mobile devices*) – comumente denominado por *ubiquitous computing*. Esse cenário indica uma alta demanda por desenvolvimento de sistemas de software distribuído nos próximos anos.

Entretanto, computação distribuída introduz grande complexidade no desenvolvimento, na implantação e na manutenção de sistemas de software. Uma série de requisitos que normalmente não estão presentes em sistemas centralizados podem ser necessários se considerar em sistemas distribuídos, tal como a confiabilidade do protocolo utilizado para troca de mensagens entre processos. Mais ainda, requisitos que normalmente já estão presentes em sistemas centralizados podem ser mais difíceis de implementar em sistemas distribuídos, tais como segurança e tolerância a faltas. Conseqüentemente, desenvolver sistemas de software distribuído de qualidade é difícil e conta fundamentalmente com a proficiência dos programadores e com o apoio de boas ferramentas.

1.1 Justificativa para a Virtuosi

Um programador torna-se proficiente no desenvolvimento de sistemas de software distribuído primeiramente quando lhe são passados os conceitos de computação distribuída de forma adequada e, segundo, quando é suficientemente treinado no uso de artefatos tecnológicos

específicos, tal como uma linguagem de programação distribuída ou um *middleware* para execução distribuída, que lhe permitam experimentar os conceitos estudados. Quase invariavelmente, conceitos são assimilados através de experimentos com artefatos tecnológicos, permitindo que conhecimento em teoria e em prática sejam adquiridos conjuntamente. O processo de aprendizado é complexo e a curva de aprendizado depende de muitos fatores, mas, certamente, os artefatos tecnológicos empregados são determinantes.

Uma ferramenta para o desenvolvimento de sistemas de software distribuído pode prover uma série de facilidades para os programadores, desde modelagem de conceitos até instalação física do sistema. Naturalmente, a qualidade do sistema de software distribuído é fortemente influenciada pelas facilidades disponíveis na ferramenta e pelo modo como os programadores as empregam. Uma dessas possíveis facilidades que é determinante para o sucesso de um sistema é a fácil criação de protótipos, isto é, a criação de uma versão preliminar do sistema na qual os seus requisitos – estabelecidos em seu projeto ou introduzidos posteriormente, durante o desenvolvimento do próprio protótipo – possam ser rapidamente implementados, executados passo a passo, testados e simulados.

Atualmente, existem diversas ferramentas para o desenvolvimento de sistemas de software distribuído, mas estas raramente favorecem o aprendizado de computação distribuída, tampouco favorecem o desenvolvimento de protótipos. A razão disso é que essas ferramentas são tipicamente projetadas para satisfazer padrões industriais – *perspectiva industrial* – ou então para experimentar novos conceitos – *perspectiva de pesquisa*. Ferramentas industriais, tais como a plataforma Sun Microsystems J2EE, a plataforma Microsoft .NET e diversas implementações do padrão CORBA – por exemplo, Orbix –, têm seu foco em produtividade e em eficiência e robustez do software desenvolvido; dificilmente permitem ao programador desenvolver qualquer tarefa de maneira simples e focado em um único problema por vez, isto é, as ferramentas industriais invariavelmente exigem que o programador se preocupe com requisitos de versões operacionais de aplicações reais, independentemente do problema em estudo. Essa exigência certamente distrai o programador e pode comprometer tanto a curva de desenvolvimento do software como a curva de aprendizado. Por outro lado, ferramentas de pesquisa, tais como Arjuna [Parrington et al., 1995], ISIS [Birman, 1985] e Emerald [Jul et al., 1988], e, novamente, diversas implementações do padrão CORBA – por exemplo, MICO – e do padrão OSF DCE – por exemplo, DC++ [Schill, 1993] –, costumam ter interfaces de uso complexas e ainda requerem o conhecimento de conceitos particulares da

pesquisa correspondente. De uma forma geral, os programadores acham difícil usar ferramentas de pesquisa porque demandam um trabalho e um tempo considerável para construir qualquer aplicação, mesmo se esta for de pequeno porte.

Há muito se discute sobre paradigmas e linguagens para o ensino de programação de computadores. Diversas linguagens mostraram-se eficazes para esse fim, tais como a linguagem Pascal para o paradigma de programação procedural e as linguagens Eiffel e Smalltalk-80 para o paradigma de programação orientada a objetos, e, de fato, têm sido tradicionalmente usadas tanto em cursos introdutórios de programação, como em cursos avançados. O sucesso dessas justifica-se por satisfazerem – total ou parcialmente – os requisitos comumente estabelecidos na literatura – como por [Kolling, 1999] – para linguagens de programação com fins pedagógicos. Diz-se que essas linguagens possuem uma *perspectiva pedagógica*.

Entretanto, nota-se, atualmente, no âmbito de ensino de programação de computadores, uma tendência crescente no emprego de linguagens – e mesmo ferramentas de desenvolvimento – amplamente empregadas para fins industriais, isto é, as linguagens que possuem uma perspectiva industrial. Exemplos concretos dessa tendência são as linguagens C++ e Java, que muitas vezes são empregadas já em cursos introdutórios de programação. Um argumento comumente usado para justificar essa abordagem é que o aprendiz não precisará fazer transição de uma linguagem de ensino para uma linguagem industrial quando adentrar no mercado de trabalho, supondo que tal transição teria um alto custo. Outro argumento é que o aprendiz pode exercitar os conceitos através de problemas "reais", isto é, problemas que farão parte do seu cotidiano profissional, ao passo que linguagens de ensino seriam limitadas e permitiriam apenas a construção de exemplos teóricos ou imaginários. Há, por outro lado, argumentos contrários ao uso em ensino de linguagens com perspectiva industrial. Tipicamente, a perspectiva industrial prejudica a perspectiva pedagógica, pois as linguagens com perspectiva industrial têm como propósito essencial obter alto desempenho na execução de programas, em detrimento, muitas vezes, da perspectiva pedagógica.

Portanto, há uma necessidade de ferramentas para o desenvolvimento de software que permitam o programador aprender sobre computação distribuída – perspectiva pedagógica – e, ao mesmo tempo, construir sistemas de software distribuído de qualidade através de prototipação – perspectiva experimental. Uma ferramenta pedagógica deve implementar, de uma maneira clara, os principais conceitos de computação distribuída já estabelecidos, e

deve ser aberta para a introdução de novos conceitos, inclusive para fins de pesquisa. Uma ferramenta experimental deve ser compatível com as principais tecnologias bem estabelecidas, tal que seja possível converter um protótipo para a sua correspondente versão operacional.

1.2 Objetivos da Virtuosi

O Projeto Virtuosi visa o desenvolvimento de uma ferramenta com as perspectivas pedagógica e experimental para a construção (edição, compilação, e depuração) e a execução de sistemas de software distribuídos. O projeto engloba muitos aspectos de computação distribuída e de engenharia de software. Deve incluir artefatos para a construção de sistemas de software e uma plataforma de execução distribuída completa.

A perspectiva pedagógica permitirá o aprendizado de computação distribuída de uma maneira estruturada; conceitos e técnicas de programação distribuída poderão ser introduzidos e estudados individualmente, possibilitando acelerar a curva de aprendizado. A perspectiva pedagógica exige um ambiente no qual o programador possa escrever programas através de um conjunto de abstrações que sejam, ao mesmo tempo, simples e poderosas. Exige ainda que o emprego de todas essas abstrações possam ser verificados em detalhe em tempo de execução, isto é, as traduções das abstrações de programação para as estruturas de execução devem ser minimizadas, estabelecendo uma relação causal explícita conforme definido em [Riehle et al., 2001]. Outro requisito decorrente da perspectiva pedagógica é que a plataforma de execução deve ser completamente neutra com relação ao ambiente de execução hospedeiro a fim de evitar qualquer distração por parte dos programadores. Finalmente, a perspectiva pedagógica requer um ambiente no qual o programador possa facilmente selecionar quais aspectos de sistema devem ser transparentes ou translucentes em um certo momento.

A perspectiva experimental permitirá a criação de protótipos relativamente bem maduros e robustos; serão maduros porque poderão incorporar parte bastante significativa dos requisitos do sistema, e serão robustos porque será relativamente fácil realizar testes em unidades separadas, seguidos de testes de integração gradativa, sendo possível simular todas as configurações e circunstâncias pelas quais estará sujeita a versão operacional do sistema, independentemente de aspectos tecnológicos particulares. Devido a essa maturidade e robustez,

um protótipo será uma base bastante sólida para o desenvolvimento da correspondente versão operacional através do uso de tecnologias específicas. Como resultado final, a ferramenta auxiliará no desenvolvimento de sistemas de software distribuídos de alta qualidade e em períodos de tempo relativamente curtos, já que os programadores serão bem treinados e, além disso, poderão implementar e testar requisitos críticos do sistema em desenvolvimento de uma maneira bem controlada. A perspectiva experimental requer um ambiente no qual aplicações do mundo real possam ser rapidamente desenvolvidas e cuidadosamente testadas.

1.3 Justificativa da Abordagem para o Trabalho

Apesar da inadequação das ferramentas industriais e de pesquisa para fins pedagógicos e experimentais, o seu amplo uso tem ajudado a consolidar vários conceitos como fundamentais na construção de qualquer ferramenta similar. Dentre esses conceitos, destacam-se máquinas virtuais, programação orientada a objetos e reflexão computacional, pois podem trazer grandes benefícios às perspectivas pedagógica e experimental. Estes três conceitos são discutidos na seqüência.

1.3.1 Máquina Virtual

Uma máquina virtual é um sistema de software que executa sob a forma de um processo de usuário (acima do sistema operacional) e que emula um computador real, incluindo seus componentes de hardware e correspondente sistema operacional. Assim, deve ser capaz de hospedar qualquer sistema de software típico que armazena e processa dados, além de se comunicar com periféricos. A máquina virtual isola a aplicação do sistema operacional e do hardware, dando-lhe portabilidade. O agrupamento de máquinas virtuais em uma coleção na qual cada máquina é capaz de endereçar e enviar mensagens para qualquer outra permite que um sistema de software executando em uma máquina comunique com outro sistema de software executando em outra máquina, isto é, uma coleção de máquinas virtuais que se comunicam forma uma plataforma para a execução distribuída de sistemas de software. Como a comunicação ocorre numa camada acima do sistema operacional, a interoperabilidade entre aplicações fica bastante facilitada. De fato, uma plataforma assim pode ser

vista como um *middleware*, semelhantemente aos sistemas baseados no padrão CORBA, pois o sistema de software distribuído pode executar sobre um conjunto heterogêneo de computadores. Os benefícios decorrentes do emprego de máquinas virtuais com relação às perspectivas pedagógica e experimental podem ser sumarizados da seguinte forma:

- *arquitetura neutra*: Uma máquina virtual não está vinculada a qualquer arquitetura de computador em particular e deve implementar somente propriedades comuns às principais tecnologias de computador. Com relação à perspectiva experimental, isso garante que protótipos que executam sobre máquinas virtuais poderão ser facilmente traduzidos para suas versões operacionais que executam sobre máquinas reais, sem impedir possíveis otimizações de código a fim de explorar características específicas dessas máquinas. Por outro lado, com relação à perspectiva pedagógica, a simplicidade de uma máquina virtual a torna adequada para o treinamento de programadores, pois o número de conceitos a se trabalhar é relativamente pequeno; os programadores são motivados a combinar tais conceitos para construir aplicações complexas.
- *portabilidade e mobilidade*: Uma máquina virtual situa-se entre as aplicações e o sistema operacional; as aplicações interagem com a máquina virtual que, por sua vez, interage com o sistema operacional. Uma conseqüência é que é preciso haver uma implementação específica da máquina virtual para cada tipo de sistema operacional. Outra conseqüência é que uma aplicação que executa em uma implementação particular da máquina virtual executará sobre qualquer outra implementação. Em outras palavras, as aplicações têm portabilidade: executam em computadores heterogêneos, desde que haja implementação adequada da máquina virtual. Com respeito à perspectiva experimental, essa portabilidade ajuda na construção de protótipos quando um grupo de programadores que utilizam sistemas operacionais distintos trabalham de maneira cooperativa; eles podem compartilhar seu código sem se preocupar com detalhes específicos de cada sistema operacional e hardware, dessa forma aumentando a produtividade. Com respeito à perspectiva pedagógica, ajuda os programadores a escrever exercícios nos quais diversos computadores heterogêneos são empregados indistintamente, sem causar qualquer distração. Outra conseqüência é que as aplicações são móveis: podem se mover por computadores heterogêneos, em tempo de execução, desde que haja implementações adequadas da máquina virtual. Essa mobilidade pode ser bastante útil, já que é um requisito cada vez mais freqüente nas aplicações atuais.

- *execução controlada*: Como uma máquina virtual é um sistema de software que controla a execução de outro sistema de software, pode dar um forte apoio na depuração de aplicações; uma máquina virtual pode manter dados muito precisos sobre o contexto de execução, munindo, assim, o programador de informação bastante precisa quando alguma falha ocorrer na execução da aplicação. Com relação à perspectiva experimental, essa é uma facilidade essencial para aumentar a produtividade. E, com relação à perspectiva pedagógica, é bastante importante porque permite que programadores usem a facilidade de depuração para entender melhor o comportamento do software.
- *flexibilidade na configuração de rede*: Como uma máquina virtual é simplesmente um processo de usuário, pode haver qualquer quantidade de instâncias de uma máquina virtual executando em um único computador. Como conseqüência, uma coleção de n máquinas virtuais pode executar sobre uma rede composta por 1 a n computadores. Em uma situação extrema, então, nem há necessidade de uma rede de computadores, isto é, basta um computador para executar uma aplicação virtualmente distribuída. De acordo com [Silberchatz and Galvin, 1998], esse conceito foi experimentado pela primeira vez pelo sistema operacional VM da IBM, no qual um conjunto de máquinas virtuais executa sobre um único computador, dando a ilusão de que cada usuário possui seu próprio computador; a comunicação entre as máquinas virtuais ocorre através de uma rede virtual. Com respeito à perspectiva experimental, tal propriedade pode facilitar o desenvolvimento de protótipos, pois qualquer configuração de rede pode ser simulada. Com respeito à perspectiva pedagógica, pode auxiliar os programadores a experimentar a computação distribuída mesmo se houver disponibilidade de apenas um computador.

1.3.2 Orientação a Objetos

A programação orientada a objetos é, provavelmente, o paradigma mais amplamente aceito e utilizado para a computação distribuída, tanto no meio acadêmico como na indústria. Orientação a objetos foi inicialmente introduzida por Simula-67 como um meio de representar entidades do mundo real para fins de simulação, e ganhou popularidade após Smalltalk-80 e C++. Atualmente, há muitas linguagens de programação que incorporam os conceitos de programação orientada a objetos e são amplamente usadas no treinamento de programadores

por mais de uma década. Mais recentemente, com o incremento na demanda por aplicações baseadas na Internet, novas linguagens e ferramentas surgiram e, praticamente, todas são orientadas a objetos. Provavelmente, o exemplo mais significativo seja Java, que, mesmo sendo de caráter industrial, é muito utilizada em cursos introdutórios de programação e também motiva muito da pesquisa atual em computação distribuída. Outro exemplo importante é Eiffel, uma linguagem rigorosa na implementação dos conceitos de orientação a objetos.

De fato, o paradigma de programação orientada a objetos está presente em quase toda nova arquitetura desenvolvida pela comunidade de sistemas distribuídos. Por exemplo, tanto o Open Distributed Processing (ODP) quanto o Object Management Group (OMG), as principais iniciativas de padronização para a computação distribuída heterogênea, baseiam-se nos conceitos de objetos. Na indústria de software, dois exemplos muito importantes do uso dos conceitos de objetos são as plataformas Sun Microsystems' Java-based J2EE e Microsoft .NET.

A adoção do paradigma de orientação a objetos na definição da ferramenta de construção e execução de aplicações ajuda os programadores a desenvolver aplicações através do emprego de seus conceitos, com assistência de artefatos apropriados, tal como uma linguagem de programação rigorosamente orientada a objetos, e de ferramentas, tal como um compilador construído de acordo com a perspectiva pedagógica, isto é, um compilador que não apenas verifique o código fonte, mas que também auxilie no processo de treinamento do programador. A plataforma de execução (formada por uma coleção de máquinas virtuais cooperantes) deve preservar todas as abstrações orientadas a objetos a fim de minimizar as traduções que poderiam dificultar a depuração de aplicações; isso ajuda na construção rápida de protótipos – perspectiva experimental – além de ajudar os programadores a entender mais corretamente os conceitos de programação – perspectiva pedagógica.

1.3.3 Reflexão Computacional

Três abordagens distintas, embora complementares, para o emprego de orientação a objetos em sistemas distribuídos e concorrentes são discutidos por [Briot et al., 1998]:

- *abordagem de biblioteca*: Conceitos de orientação a objetos são disponibilizados através

de bibliotecas de classes. É apropriada para os programadores de sistema e objetiva a identificação de abstrações fundamentais para o tratamento de concorrência e distribuição – pode ser vista como uma abordagem bottom-up, na qual flexibilidade é prioritária. Sua principal limitação é que a programação é representada por conjuntos de conceitos isolados, o que requer grande conhecimento e habilidade dos programadores. Exemplos dessa abordagem incluem o sistemas ISIS e Arjuna.

- *abordagem integrativa*: Conceitos de orientação a objetos são unificados com conceitos de concorrência e de sistemas distribuídos, tais como os conceitos de objeto e de atividade. É adequado para programadores de aplicações e objetiva a definição de uma linguagem de programação de alto nível, com um pequeno conjunto de conceitos unificados, tornando os mecanismos mais transparentes. Sua desvantagem é a possibilidade de reduzir a flexibilidade e a eficiência dos mecanismos. Exemplos da abordagem integrativa são os sistemas operacionais Amoeba [Mullender et al., 1990] e Mach [Boykin et al., 1993].
- *abordagem reflexiva*: Integra bibliotecas de protocolos a uma linguagem de programação orientada a objetos; a aplicação fica isolada dos vários aspectos do contexto de implementação e de computação – separation of concerns – através da sua descrição em termos de metaprogramas, de acordo com o conceito de reflexão computacional, disseminado por [Maes, 1987]. É adequado tanto para programadores de aplicações como para programadores de sistema e, de fato, aproxima as duas abordagens anteriores, pois permite a integração de bibliotecas de protocolos em uma linguagem de programação ou sistema – combinação de flexibilidade com transparência.

Dessa forma, a abordagem reflexiva permite que os programadores modifiquem o comportamento do sistema em dois níveis: aplicação e plataforma de execução. Sob a perspectiva pedagógica, os programadores podem selecionar quais aspectos devem ser tratados de forma transparente, tal que torna-se possível estudar cada propriedade individualmente ou fazendo-se a combinação de algumas. E, sob a perspectiva experimental, os programadores são munidos de grande dinamismo para o desenvolvimento de sistemas de software como componentes que podem ser facilmente substituídos e testados. Dessa forma, a plataforma de execução pode ser vista como um middleware reflexivo, como as implementações baseadas no padrão CORBA DynamicTAO [Kon et al., 2000] e Open ORB [Blair et al., 2001]. O modelo de middleware reflexivo é uma maneira consistente e eficiente de se trabalhar com ambientes

altamente dinâmicos, além de favorecer o desenvolvimento de sistemas e aplicações flexíveis e adaptativas [Kon et al., 2002]. Naturalmente, tal flexibilidade pode ser difícil de se obter e exige um modelo consistente para a composição de recursos de sistema no nível meta, tal como o modelo proposto por [Venkatasubramanian, 2002].

A combinação de máquina virtual e orientação a objetos acarreta em uma propriedade interessante para a implementação de reflexão computacional: objetos e correspondente código podem ser explicitamente armazenados e manuseados em tempo de execução, permitindo, assim, reflexão sobre praticamente qualquer aspecto de computação, facilitando a modificação dinâmica do comportamento de sistemas. Em outras palavras, a máquina virtual pode ter acesso a toda informação semântica de uma aplicação, tornando possível aos programadores explorar suas propriedades reflexivamente, tanto para fins pedagógicos como para fins experimentais. Essa capacidade diferenciaria a plataforma de execução de outras que também contemplam reflexão computacional, tal como o Projeto Guaraná [Oliva, 1998], no qual uma Máquina Virtual Java foi modificada para incorporar reflexão computacional, mas preservando intacto o seu padrão de código executável – o então chamado *bytecode* – e também a compatibilidade da linguagem de programação. Difere também do Projeto PJama [Atkinson, 1998], no qual a Máquina Virtual Java foi modificada para incorporar um serviço de persistência de objetos ortogonal.

Uma maneira de se dispor da completa semântica da aplicação em tempo de execução é a representação e o armazenamento do código na forma de uma árvore de programa: um grafo de objetos que representa todos os elementos de um certo código fonte, incluindo os seus relacionamentos. Árvores de programa são empregadas com sucesso na Máquina Virtual Juice [Franz and Kistler, 1997] com o objetivo de transporte de código pela rede; quando um programa chega ao seu destino, é traduzido para o código de máquina correspondente para fins de execução eficiente. Como há um mapeamento direto entre uma árvore de programa e o correspondente código fonte, as regras para a construção de uma árvore de programa são as mesmas para a escrita de um programa orientado a objetos. Tais regras podem ser estabelecidas através de um modelo formal, também orientado a objetos, normalmente denominado de metamodelo. Assim, os objetos de uma árvore de programa seriam instâncias das classes do metamodelo.

1.4 Objetivos Específicos desta Dissertação

Esta dissertação visa três objetivos principais:

- definir um modelo de classes (metamodelo) que represente os elementos existentes em uma linguagem de programação orientada a objetos;
- projetar a arquitetura de uma máquina de interpretação (máquina virtual) para um sistema computacional baseado em objetos que facilite a computação distribuída;
- validar o metamodelo e arquitetura definidas (através da implementação da máquina virtual Virtuosi).

1.5 Estrutura da Dissertação

Esta dissertação está dividida em seis Capítulos, conforme descrito abaixo:

- Capítulo 1 – Introdução. Apresenta o Projeto Virtuosi e seus objetivos, justifica a abordagem adotada neste trabalho, explicita os objetivos específicos e fornece uma visão geral desta dissertação;
- Capítulo 2 – Trabalhos Relacionados. Fornece um resumo de algumas das principais máquinas virtuais e outros assuntos importantes para a consecução deste trabalho;
- Capítulo 3 – Conceitos de Orientação a Objetos. Discute os principais conceitos de orientação a objetos implementados pelas linguagens Java e Eiffel e que possuem um paralelo no ambiente Virtuosi;
- Capítulo 4 – Arquitetura da Virtuosi. Especifica os conceitos de orientação a objeto suportados pela Virtuosi através da formalização do metamodelo da Virtuosi. Em seguida, especifica um novo formato de representação intermediária na forma de árvore de programa interpretada pela máquina virtual Virtuosi. Por último, especifica a arquitetura da máquina virtual Virtuosi;
- Capítulo 5 – Implementação da Máquina Virtual. Mostra como a arquitetura definida na Seção 4.3.5 foi implementada, bem como as limitações do protótipo;
- Capítulo 6 – Conclusão. Apresenta os resultados obtidos com este trabalho e a contribuição científica do mesmo. Também discute os possíveis trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Esse Capítulo fornece um resumo de algumas das principais máquinas virtuais (focado em aspectos arquiteturais) e outros assuntos considerados importantes para a consecução deste trabalho.

2.1 A Arquitetura Java Virtual Machine

A máquina Virtual Java (JVM) pode ser entendida em três sentidos, dependendo do contexto: uma especificação abstrata, uma implementação concreta ou uma instância de tempo de execução. Em se tratando da especificação abstrata pode-se dizer que a arquitetura da máquina virtual Java define o comportamento externo de uma implementação concreta em termos de subsistemas, áreas de memória, tipos de dados e instruções [Venners, 1999].

A Figura 2.1 mostra os principais componentes da arquitetura da JVM. Cada máquina virtual possui seu carregador de classes[1] responsável por carregar em memória classes e *interfaces* a partir de seus nomes. Uma máquina virtual possui um motor de execução[2] responsável por executar as instruções contidas nos métodos das classes ou interfaces carregadas. Tais instruções são chamadas de ***bytecodes***. Além desses dois subsistemas – o carregador de classe e o motor de execução – uma máquina virtual precisa de áreas de memória para armazenar informações como os *bytecodes*, informações extraídas das classes carregadas, objetos instâncias das classes, parâmetros de métodos e resultados intermediários de computação.

[1] Do inglês, ***class loader***
[2] Do inglês, ***execution engine***

As classes carregadas pelo subsistema carregador de classe geralmente estão em um arquivo `.class`.

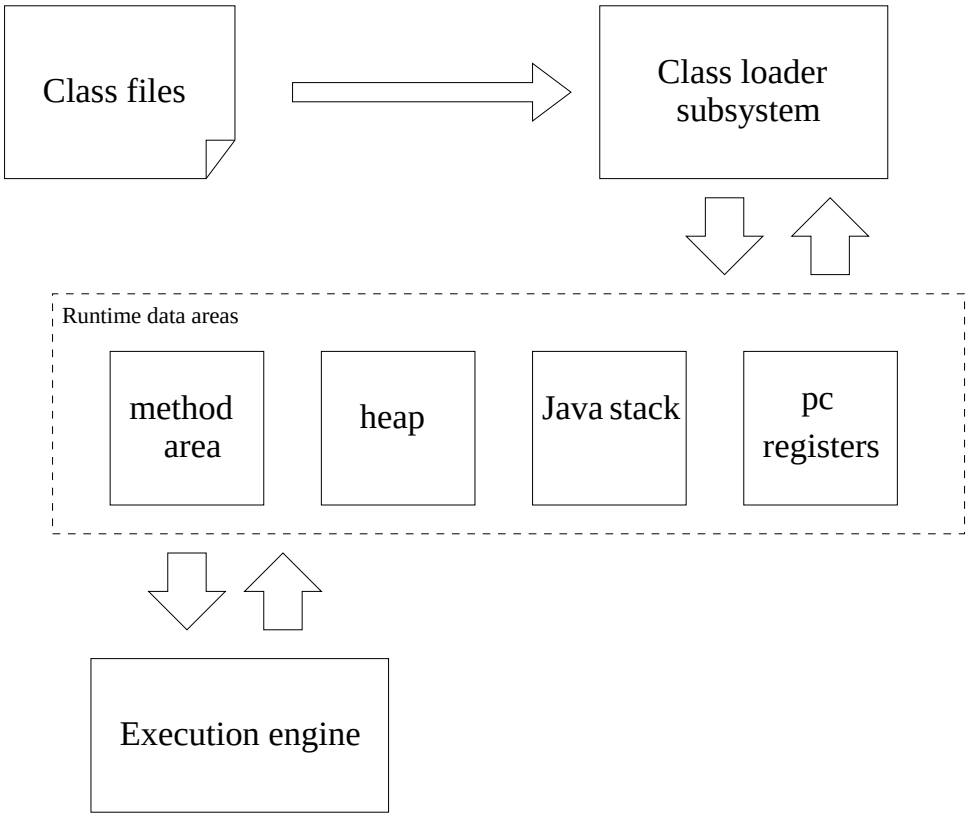

Figura 2.1 Principais componentes da Arquitetura da JVM

Para cada execução de aplicação, uma nova instância da JVM é criada. A JVM dá suporte a execução de mais de uma linha de execução[3].Coforme discutido nas Seções subseqüentes, existem componentes particulares a uma linha de execução e existem componentes compartilhados por todas as linhas de execução de uma mesma aplicação. Note que não é possível compartilhar nenhum componente entre instâncias diferentes da JVM.

A especificação da JVM define o comportamento destes componentes e a interação entre eles, mas não dita como esses devem ser implementados. O restante desta Seção discute os

[3]Do inglês, ***thread***

principais componentes da arquitetura da JVM em seus aspectos mais relevantes para este trabalho.

2.1.1 Tipos de Dados

A JVM realiza sua computação por executar operações específicas sob tipos de dados específicos. Os tipos de dados definidos pela JVM podem ser divididos em dois grupos: **tipos primitivos** e **tipos referência**. Atributos ou variáveis locais de tipos primitivos armazenam *valores primitivos* enquanto que atributos ou variáveis de tipos referência armazenam *referências*. Uma referência, como o próprio nome diz, referencia um objeto, ou seja, seu valor é um apontador para um objeto em memória. Um valor primitivo, não referencia outro objeto, ele é o próprio valor em uma posição de memória.

Todos os tipos primitivos da linguagem Java são tipos primitivos na JVM, com exceção do tipo *boolean* que é manipulado pela JVM como um valor inteiro do tipo *int* ou *byte*. Com a exceção do tipo boolean, os outros tipos primitivos formam os tipos numéricos da JVM. Os tipos numéricos da JVM são por sua vez divididos entre tipos integrais[4] e tipos ponto flutuante. Além dos tipos integrais e tipos flutuante a JVM também possui o tipo retorno de endereço[5] não disponibilizado pela linguaguem Java mas utilizado internamente pela máquina virtual.

Um tipo referência pode referenciar objetos de três tipos, a saber:

- objetos instâncias de classe;
- objetos vetor[6];
- objetos instâncias de classes que implementam uma *interface*.

Uma referência pode ainda não referenciar nenhum objeto, esse tipo de referência é chamada de referência nula e sua atribuição é indicada pela palavra null.

A Figura 2.2 resume os tipos de dados existentes na JVM.

[4]Do inglês, ***integral types***
[5]Do inglês, ***returnAddress***
[6]Do inglês, ***array***

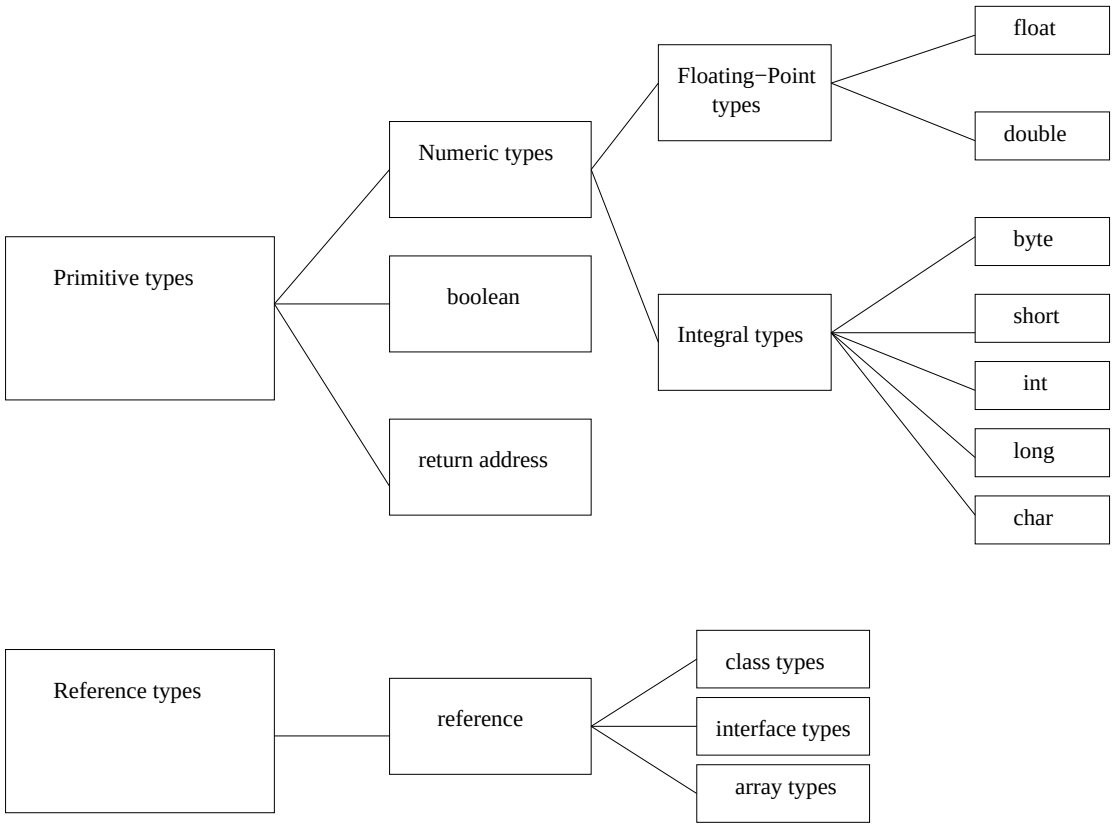

Figura 2.2 Tipos primitivos disponibilizados pela JVM

2.1.2 Carregador de Classes e Interfaces

A JVM possui dois tipos de carregadores de classe. Os carregadores disponibilizados pela própria máquina virtual (*bootstrap class loader*) e os carregadores definidos por usuário (*user-defined class loaders*).

Sempre que uma classe é carregada a JVM cria um objeto instância da classe `java.lang.Class` para representar o tipo. Este objeto da classe `java.lang.Class` é armazenado na *heap* como todas as outras instâncias de objeto, conforme é discutido na Seção 2.1.4, enquanto que as informações da classe são armazenadas na área de método, conforme é discutido na Seção 2.1.3.

O subsistema carregador de classes tem outras responsabilidades além de localizar e carregar classes. Ele também precisa validar as classes carregadas, alocar e inicializar memória

para variáveis de classe e dar suporte para a resolução de referências simbólicas. Estas atividades são executadas em uma ordem estrita, a saber:

(1) carga[7] – encontrar e importar os dados binários de um tipo;

(2) ligação[8] – executar a verificação, preparação e (opcionalmente) a resolução

- verificação – garantir a validade de um tipo importado;
- preparação – alocar memória para as variáveis de classe e inicializá-las com valores padrão;
- resolução – transformar as referências simbólicas de um tipo em referências diretas.

(3) inicialização – Invocar o código Java que inicializa as variáveis de classe com seus valores iniciais apropriados (quando estes existirem).

Classes carregadas por carregadores diferentes são armazenadas em espaços de nomes diferentes, isto torna possível a existência de mais de uma classe com o mesmo nome em uma mesma aplicação. Para resolver este aparente problema a JVM mantém uma referência para o carregador de classe de cada classe carregada. Dessa forma, quando a JVM precisa resolver uma referência simbólica, ela sempre procura a classe referenciada no mesmo carregador de classe que carregou a classe que tem a referência.

Deve-se notar que JVM utiliza uma estratégia de carga de classes sob demanda (carga dinâmica). Isto significa que o processo de carga de uma classe é iniciado a partir do momento que uma referência simbólica da mesma precise ser resolvida.

2.1.3 Área de Método

A área de método armazena informações sobre as classes carregadas em memória. Além das informações sobre um tipo, a área de método também aloca espaço para as variáveis da classe[9].

[7] Do inglês, ***loading***

[8] Do inglês, ***linking***

[9] Note que a JVM faz distinção entre variáveis de instância de uma classe e variáveis da própria classe. As variáveis de classe pertencem à classe e portanto são compartilhadas por todas as instâncias e podem

A área de método é compartilhada por todas as linhas de execução em uma instância da máquina virtual.

A JVM armazena na área de método as informações extraídas pelo carregador de classe. Para cada um dos tipos (classes e *interfaces*) as seguintes informações são armazenadas:

- o nome completo[10] do tipo;
- o nome completo do tipo ancestral direto (com exceção dos tipos interface ou da classe `java.langObject`, que não possuem ancestrais);
- indicação do tipo: se é uma classe ou *interface*;
- os modificadores de acesso;
- uma lista ordenada dos nomes completos de qualquer interface que a classe implemente.

Além dessas informações listadas acima, a máquina virtual também armazena:

- a tabela de símbolos[11] do tipo ;
- informações de atributos;
- informações de métodos;
- todas as variáveis de classe declaradas no tipo, exceto as constantes;
- uma referência para o carregador de classe;
- uma referência para o objeto da classe `java.lang.Class`.

Tabela de Símbolos Para cada tipo carregado, a JVM armazena uma tabela de símbolos. Uma tabela de símbolos é uma lista ordenada de constantes utilizadas pelo tipo, incluindo valores literais e referências simbólicas para tipos, atributos e métodos. As entradas na tabela de símbolo são referenciadas por um índice. Uma vez que armazena referências simbólicas para tipos, atributos, métodos utilizados pelo tipo, a tabela de símbolo desempenha um papel fundamental na fase de ligação executada pelo carregador de classe.

---

ser acessadas mesmo na ausência de uma instância. Já as variáveis de instância são particulares para cada instância.

[10] O nome completo de uma classe ou interface dentro de um arquivo `.class` é composto do nome do pacote mais uma barra '/' mais o nome da classe, recursivamente

[11] Do inglês, ***constant pool***

Informações de Atributos Para cada atributo do tipo são armazenadas as seguintes informações:

- o nome do atributo;
- o tipo do atributo;
- os modificadores do atributo.

Além disso, a ordem em que os atributos são declarados dentro da classe ou interface também é armazenada.

Informações de Métodos Para cada método do tipo são armazenadas as seguintes informações:

- o nome do método;
- o tipo do valor de retorno do método (ou a palavra *void* caso não haja retorno);
- o número e tipos dos parâmetros do método (ordenados);
- os modificadores do método.

A ordem em que os métodos são declarados dentro da classe ou interface também é armazenada.

Além disso, para cada um dos métodos não abstratos (métodos que possuem implementação) são armazenados:

- os *bytecodes* do método;
- os tamanhos da pilha de operandos e das seções de variáveis do item da pilha do método.
- uma tabela de exceção.

Variáveis de Classe Variáveis de classe são compartilhadas por todas as instâncias de uma classe e podem ser acessadas mesmo na ausência de uma instância. Essas variáveis são associadas a classe – não a instâncias – portanto fazem parte das informações contidas na área de método. Antes da JVM utilizar uma classes, ela precisa alocar memória na área de método para cada uma das variáveis de classe não constantes, conforme explicado na Seção 2.1.2. As variáveis constantes são armazenadas na tabela de símbolo de cada tipo.

Referência para o Carregador de Classe Um tipo pode ser carregado por um carregador disponibilizado pela própria máquina virtual (*bootstrap class loader*) ou por um carregador definido pelo usuário. No caso dos tipos carregados por um carregador de classe definido por usuário, a JVM mantém uma referência para o carregador, essa referência é utilizada durante a fase de ligação dinâmica[12].

Referência para o Objeto da Classe `java.lang.Class` Uma instância da classe `java.lang.Class` é criada pela JVM para cada tipo carregado. A máquina virtual armazena uma referência para a respectiva instância da classe `java.lang.Class` junto das informações do tipo na área de método.

2.1.4 Heap

Uma instância de uma classe ou objeto vetor sempre é armazenado na *heap*. Uma aplicação Java executa em sua própria instância da máquina virtual Java, e cada instância da JVM tem apenas uma *heap*. Portanto aplicações distintas não tem como acessar objetos uma das outras. Porém, dentro de uma mesma aplicação, a *heap* é compartilhada por todas as linhas de execução*threads*.

A JVM possui uma instrução que aloca memória na heap para um novo objeto a partir de uma classe, mas não possui instruções para liberar essa memória. A própria JVM é responsável por liberar a memória ocupada por um objeto não mais referenciado. O mecanismo geralmente utilizado para liberar a memória é o uso de um coletor de lixo[13].

Representação de Objetos A especificação da JVM não define como os objetos devem ser representados na *heap*.

De alguma forma, a representação de cada objeto deve conter as variáveis de instância declaradas na classe do objeto e em suas ancestrais, recursivamente. Dada uma referência

[12] O conceito de ligação dinâmica está relacionado ao conceito de carga sobre demanda, na qual as classes vão sendo carregadas de acordo com a necessidade

[13] Do inglês, ***garbage colector***

a um objeto, a JVM precisa localizar os dados de instância do objeto. Além disso, deve existir um meio de acessar as informações sobre a classe de um objeto (armazenada na área de método) a partir de uma referência para o objeto. Por este motivo, a memória alocada para um objeto geralmente inclui um apontador para a área de método.

Uma implementação de *heap* possível divide a *heap* em duas partes: conjunto de manipuladores[14] e um conjunto de objetos[15]. Uma referência a objeto é um apontador nativo para uma entrada do conjunto de manipuladores. Uma entrada do conjunto de manipuladores tem dois componentes: um apontador para os dados de instância no conjunto de objetos e um apontador para os dados da classe na área de método. A vantagem deste esquema é que facilita à máquina virtual combater a fragmentação da *heap*. Quando a máquina virtual move um objeto conjunto de objetos, é apenas necessário atualizar um apontador com o novo endereço do objeto. A desvantagem deste esquema é que todo o acesso aos dados de instância do objeto requer dereferenciar dois apontadores. A Figura 2.3 ilustra esta solução.

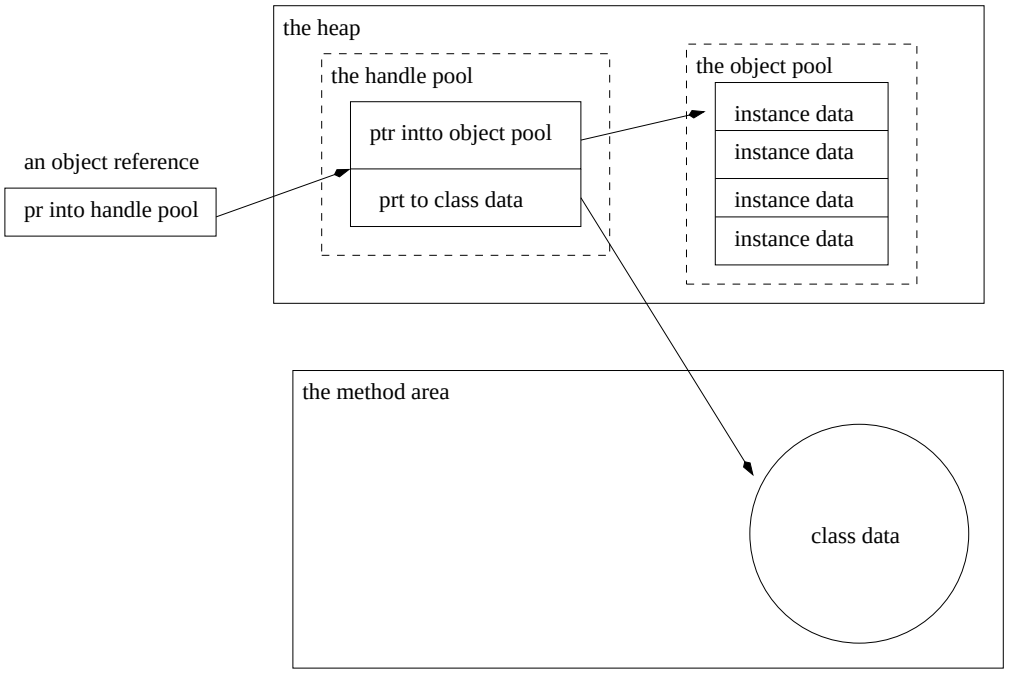

Figura 2.3 Heap com conjunto de manipuladores e conjunto de objetos

[14]Do inglês, ***handle pool***
[15]Do inglês, ***object pool***

Uma outra implementação possível da *heap* é a fazer com que uma referência a objeto seja um apontador nativo para um grupo de dados que contém dados de instância do objeto e um apontador para os dados da classe do objeto na área de método. Este esquema requer dereferenciar apenas um apontador para acessar um dado de instância de objeto, mas torna a movimentação de objetos mais complicada. Quando a máquina virtual move um objeto para combater a fragmentação neste tipo de heap, ela precisa atualizar toda referência que aponte para o grupo de de dados. A Figura 2.4 ilustra esta solução.

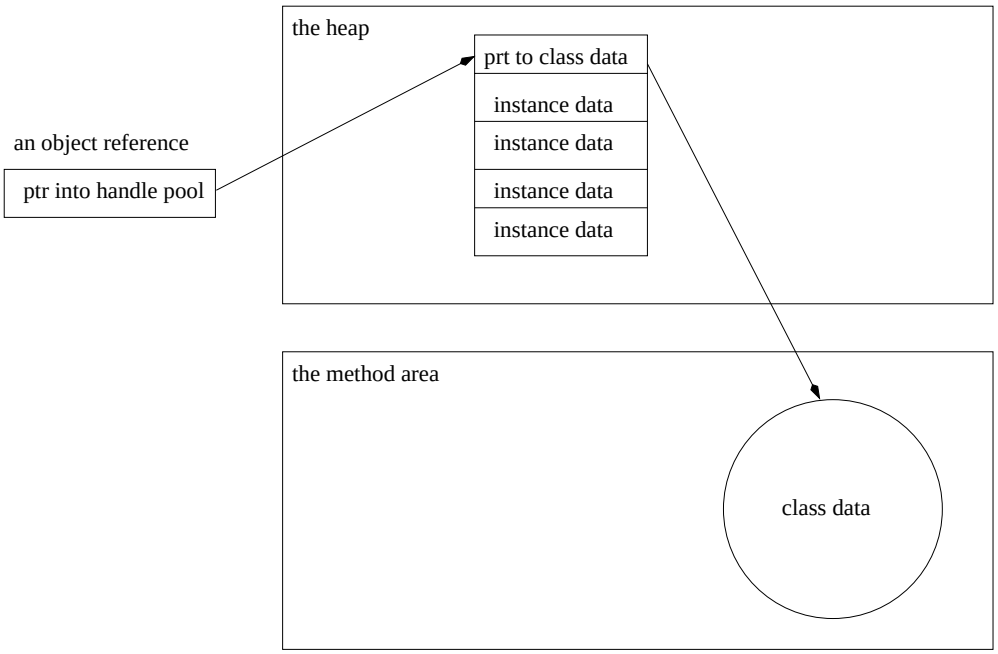

Figura 2.4 Heap com um grupo de dados contendo um apontador para a área de método

2.1.5 Contador de Programa

Cada linha de execução tem seu próprio registro contador de programa, criado no momento em que a linha de execução executa sua primeira instrução. Enquanto uma linha de execução executa um método Java, o contador de programa armazena o endereço da instrução sendo executada.

2.1.6 A Pilha Java

Quando uma linha de execução é criada, a JVM cria um nova pilha Java para a linha de execução. A pilha Java armazena o estado de uma linha de execução em ítens de pilha separados. A JVM executa dois tipos de operações diretamente sobre as pilhas Java: empilhar e desempilhar ítens.

O método que está sendo executado em determinado momento pela linha de execução é o método corrente da linha de execução. O ítem de pilha do método corrente é o ítem corrente. A classe que define o método corrente é a classe corrente, a tabela de símbolos da classe corrente é a tabela de símbolos corrente. Durante a execução de um método, a JVM mantém referência para a classe corrente e para a tabela de símbolos corrente. Quando a máquina virtual encontra instruções que operam sobre os dados armazenados no ítem da pilha, ela executa tais operações no ítem corrente.

Quando uma linha de execução invoca um método Java, a máquina virtual cria um novo ítem e o empilha no topo da pilha Java da linha de execução. Este novo ítem passa a ser o ítem corrente. Durante a execução do método, a máquina virtual utiliza o ítem para armazenar parâmetros, variáveis locais, computações intermediárias e outros dados.

Um método pode completar de duas formas. Se um método termina por retornar, ele tem uma finalização normal. Se um método termina por lançar uma exceção, ele tem uma finalização repentina. Quando um método termina, normalmente ou repentinamente, o JVM retira do topo da pilha e descarta o ítem da pilha correspondente. Dessa forma o ítem imediatamente anterior torna-se o ítem corrente.

Todos os dados em uma pilha Java de uma linha de execução são privados para esta linha de execução. Não é permitido que uma linha de execução altere o estado de outra linha de execução. Quando uma linha de execução invoca um método, as variáveis locais do método são armazenadas em um ítem no topo da pilha Java da linha de execução que invocou o método. Somente uma linha de execução pode alterar as variáveis locais deste ítem: a linha de execução que invocou o método.

Item de Pilha

Um ítem de pilha tem três seções, a saber:

(1) variáveis locais;

(2) pilha de operandos;

(3) dados do ítem.

Variáveis Locais A seção de variáveis locais é organizada como um vetor de índice inicial zero. Instruções que utilizam as variáveis locais precisam informar o índice específico de cada variável.

A seção de variáveis locais contém os parâmetros do método e suas variáveis locais. Os compiladores colocam os parâmetros dentro do vetor na mesma ordem em que foram declarados e antes das variáveis locais. A Figura 2.5 mostra o código fonte de dois métodos de uma classe, um método de classe e outro método de instância. A Figura 2.6 ilustra a seção de variáveis locais correspondente para ambos os métodos.

```
class Example {

   public static int runClassMethod( int i, long l, float f,
       double d, Object o, byte b){
       return 0;
   }

   public static int runInstanceMethod( char c, double d, short s){
       return 0;
   }
}
```

Figura 2.5 Código fonte de dois métodos Java

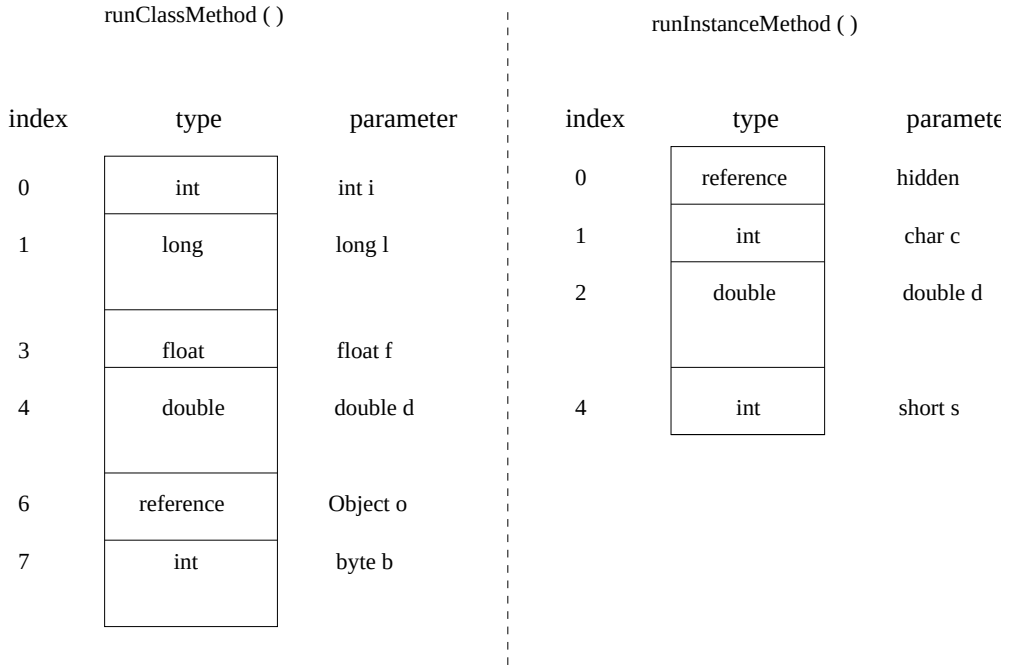

Figura 2.6 Ilustração da seção de variáveis locais dos métodos descritos na Figura 2.5

A Figura 2.6 mostra que o primeiro parâmetro na seção de variáveis locais do método `runInstanceMethod()` é do tipo `reference`, nota-se que tal parâmetro não ocorre no código fonte correspondente mostrado na Figura 2.5. Essa é a referência `this` passada de forma implícita a todo método de instância. Um método de instância utiliza a referência `this` para acessar os dados de instância do objeto que o invocou. Observando-se novamente a Figura 2.6, nota-se que um método de classe não recebe um parâmetro implícito `this`. Métodos de classe não são invocados a partir de objetos. Não é possível acessar diretamente variáveis de instância de uma classe a partir de um método de classe, isto ocorre porque não há instância associada a invocação de um método de classe.

Observa-se também que o parâmetro `Object o` – passado como parâmetro para o método de classe `runClassMethod()` – é recebido no método como uma referência. Todos os objetos são armazenados na heap, de forma que, os parâmetros de um método, bem como variáveis locais sempre são referências a um objeto, e nunca o objeto em si.

As variáveis locais devem ser colocadas no vetor de variáveis locais na ordem em que foram declaradas no código fonte correspondente.

Pilha de Operandos Da mesma forma que a seção de variáveis locais a pilha de operandos é organizada como um vetor de índice inicial zero. Mas diferentemente das variáveis locais, que são acessadas via índice, a pilha de operandos é acessada por empilhar e desempilhar valores. Quando uma instrução empilha um valor no topo da pilha, um outra instrução podem desempilha-lo e utilizá-lo.

Pode-se dizer que a JVM é baseada em pilha ao invés de registradores, uma vez que as instruções pegam seus operandos da pilha de operandos ao invés de registradores. Instruções podem também pegar operandos de outros lugares: imediatamente após o *opcode* (o *byte* representando a instrução) na seqüência de instruções e da tabela de símbolos.

A JVM utiliza a pilha de operandos como uma área de trabalho. Muitas instruções desempilham valores do topo da pilha de operando, operam sobre eles, e empilham o resultado. Por exemplo, a instrução `iadd` (instrução utilizada para adicionar dois valores do tipo *Integer*), desempilha dois valores do topo da pilha de operandos, adiciona-os e empilha o resultado novamente na pilha de operandos. Existem instruções responsáveis por copiar valores da pilha de operando para o vetor de variáveis locais e vice-versa.

Dados do Item de Pilha Além da variáveis locais e da pilha de operandos, o item de pilha Java possui dados para dar suporte a resolução de tabela de símbolos, retorno de método normal e envio exceções. Estes dados são armazenados na seção dados do item de pilha pertencente ao item pilha Java.

Muitas instruções da JVM utilizam entradas na tabela de símbolos. Algumas instruções simplesmente empilham constantes vindas da tabelas na pilha de operandos. Algumas instruções utilizam as entradas na tabela de símbolo para se referir a classes ou vetores os quais devem ser feitas novas instâncias, atributos que devem ser acessados e até métodos a ser invocados. Outras instruções determinam se um objeto particular é instância de determinada hierarquia de classes ou implementa alguma interface, tanto as classes como as interfaces são entradas na tabela de símbolos.

Sempre que a JVM encontra uma instrução que referencia uma entrada na tabela de símbolos, ela utiliza o apontador para a tabela de símbolos existente na seção dados do item da pilha. Quando a entrada na tabela de símbolos ainda é simbólica, a JVM precisa resolver

a referência naquele instante.

A JVM utiliza as informações da seção de dados do item da pilha para processar o término de métodos. Se um método termina normalmente, a máquina virtual precisa restaurar os dados de item do método que fez a invocação. Ela precisa fazer o contador de programa apontar para a instrução seguinte à instrução que invocou o método terminado. Caso o método tenha retornado um valor, este precisa ser empilhado no topo da pilha de operandos do método que fez a invocação.

Os dados de item de pilha também contém um tipo de referência para uma tabela de exceções do método, utilizada pela máquina virtual para processar as exceções durante a execução de um método.

2.1.7 Máquina de Execução

A máquina de execução geralmente é o ponto principal em uma implementação da JVM. A especificação da JVM define o comportamento da execução de um programa em termos de um conjunto bem definido de instruções que operam sobre um conjunto bem definido de tipos de dados. Para cada instrução, a especificação detalha o que a implementação deve fazer, mas especifica muito pouco sobre como a instrução deve ser implementada.

Cada linha de execução em uma aplicação Java sendo executada é uma instância distinta da máquina de execução da máquina virtual. Todo o tempo uma linha de execução está executando instruções ou métodos nativos.

O Conjunto de Instruções Cada uma das instruções contidas nos métodos das classes ou interfaces consiste de um *opcode* seguido por zero ou mais operandos. O *opcode* indica qual operação deve ser executada. Os operandos podem também ser obtidos a partir de outras áreas além dos operandos localizados ao final do *opcode*. Quando executa uma instrução, a máquina virtual pode usar entradas da tabela de símbolos, entradas na seção de variáveis locais dos dados de item de pilha corrente ou ainda valores armazenados no topo da pilha de operandos.

A JVM executa uma instrução por vez e continua a executar as instruções até que o

método inicial retorne ou ocorra uma exceção não tratada.

2.2 Protocolo de Metaobjetos de CLOS

A idéia básica do arquitetura de CLOS (Common Lisp Object System) é especificar um modelo para a implementação da linguagem e padronizá-lo. Fazendo com que o comportamento interno da implementação torne-se manipulável e controlável [Paepcke, 1993].

Além de ser um poderosa linguagem, CLOS tem dois objetivos adicionais:

- permitir que usuários e programas externos inspecionem o funcionamento interno do ambiente CLOS – softwares para visualização de hierarquia de classes, ferramentas de análise como depuradores, etc. Inspecionar o funcionamento interno inclui por exemplo, a habilidade de determinar a estrutura de um programa sem estudar o código fonte ou realizar engenharia reversa ou descobrir quais métodos são especializados por uma determinada classe;
- permitir que programas externos estendam a linguagem CLOS sem modificar o código de implementação existente e sem afetar outros programas já existentes – A habilidade de estender ou modificar a linguagem é necessária para permitir a experimentação e ajustes do comportamento da linguagem CLOS. Isto pode incluir por exemplo controlar como os *slots* são acessados ou como instâncias são criadas.

Para se fazer uma análise do funcionamento interno de CLOS pode-se separar a linguagem em dois aspectos: aspecto estático e o aspecto dinâmico.

O aspecto estático da arquitetura CLOS tem haver com a *arquitetura de meta-nível*, descrevendo os componentes do sistema, sua estrutura, os blocos de construção procedurais (classes definidas por usuários, métodos, etc) e como estes estão relacionados.

O aspecto dinâmico está relacionado ao *protocolos* que descrevem a manipulação dos blocos de construção procedurais que precisam ocorrer para realizar o comportamento da linguagem em tempo de execução.

Esta abordagem permite que através da própria linguagem seja possível realizar alterações na própria linguagem, produzindo assim um alto grau de reflexão computacional. É para

isto que o Protocolo de Metaobjetos do CLOS foi desenvolvido.

Para se poder alterar o comportamento interno ou estender o metamodelo é preciso manipular as meta-classes corretamente. A Figura 2.7 mostra as principais meta-classes que representam os principais elementos estáticos da linguagem: *Classes*, *Slots*, *Methods*, *Generic Functions*, *Method Combination*.

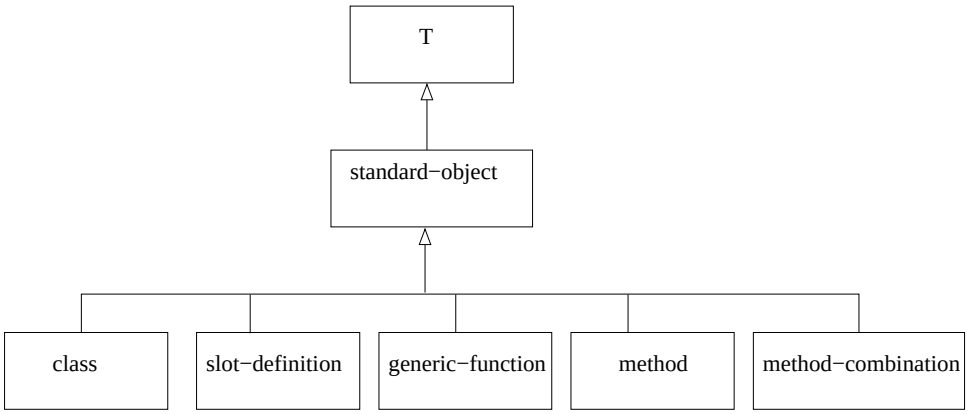

Figura 2.7 A principal hierarquia de classes do MOP

As meta-classes `class`, `slot-definition` e `method` também têm uma hierarquia de meta-classes abaixo delas.

2.3 A Arquitetura de uma Máquina Virtual UML

O uso de linguagens de modelagem, tal qual a UML (Unified Modelling Language), tem se mostrado útil para a descrição de softwares orientados a objeto [OMG, 2001]. Tradicionalmente, o modelo de uma aplicação é construído com uma ferramenta de modelagem e a implementação da solução é construída em uma linguagem de programação específica. Esta abordagem pode causar problemas tais como a perda de sincronismo entre o modelo da solução e código fonte, demora no tempo de implementação, falta de documentação e sistemas de difícil manutenção. Para amenizar este problema surgiram ferramentas que geram código diretamente a partir de modelos. Embora a utilização de tais ferramentas ofereça

vantagens como a diminuição no tempo de implementação da solução modelada, diminuição no número de erros, aumento da reutilização de classes, facilidade de entendimento e melhoria da documentação, o tempo gasto para que uma alteração no modelo tenha efeito sobre a implementação ainda é grande (geração de código a partir do modelo, compilar o código, desligar o sistema rodando, instalar, configurar a novo sistema e ligar o novo sistema). Esse problema faz com que a exploração e simulação de novos modelos com retorno do usuário imediato seja quase impossível, prejudicando assim a exploração de novas soluções de modelagem e tornando o modelo não otimizado.

A arquitetura da máquina virtual UML (MVUML) representa a linguagem de modelagem (UML), o modelo descrito em UML e as instâncias das classes definidas pelos modelo, ou seja, ela junta o ambiente de modelagem ao ambiente de execução [Riehle et al., 2001].

A MVUML é baseada na especificação da UML que define quatro níveis lógicos para modelagem [OMG, 2001].

- nível M0 – contém objetos que representam as instâncias das classes de aplicação que são executadas. Estes objetos são chamados de objetos de usuário e são instâncias das classes modeladas pelo usuário;
- nível M1 – contém objetos que representam as classes que modelam o sistema. Estes objetos são chamados de classes de usuário;
- nível M2 – contém objetos que representam a linguagem de modelagem, no caso a UML. Este nível é chamado de metamodelo;
- nível M3 – contém objetes que representam a linguagem na qual a UML é representada. Estes nível é chamado de meta-metamodelo;

Dessa forma, cada nível provê o meio para descrever o nível inferior.

Para que o sistema possa oferecer uma exploração e simulação de novos modelos com retorno do usuário imediato é preciso que os níveis de modelagem estejam conectados de forma causal. A conexão causal pode ser definida como: Um nível de modelagem tem uma relação de causalidade com o nível de modelagem imediatamente inferior quando o nível inferior estiver de acordo com o especificado pelo nível superior e qualquer alteração no nível superior cause mudanças correspondentes no nível inferior.

O que normalmente se encontra são os objetos do nível M1, ou seja, as classes que representam a modelagem da solução no domínio do problema, duplicadas em ambientes distintos. Modela-se utilizando uma ferramenta de modelagem e gera-se código de classes a partir destes modelos.

A MVUML define uma arquitetura lógica e uma arquitetura física. Na arquitetura lógica tudo é objeto, incluindo as classes e seu comportamento. Objetos representam classes UML, objetos representam classes de usuário e objetos representam objetos de usuário. Até os relacionamentos entre classes de usuário são objetos, por exemplo: objetos que representam associação e generalização. A classe raiz da arquitetura lógica é chamada de *element*, uma importante sub-classe de *element* é a classes *modelElement* que é a raiz de todas as classes UML.

A arquitetura física é responsável por implementar a lógica provendo classes físicas para a representação dos objetos lógicos. Todo o objeto está associado a uma classe lógica e a uma classe física. A classe lógica de um objeto define as propriedades do objeto a partir de uma perspectiva de modelagem lógica: ela define os atributos, associações, modelo de estados e comportamento em tempo de execução. As duas principais classes da arquitetura física são: classe *Element* – classe física de qualquer objeto no sistema e *Class* classes de todos os objetos lógicos que representam classes.

2.3.1 Modelo de Execução

A MVUML é limitada em relação ao que pode ser executado e na verdade precisa mais do que um modelo UML para executar. Embora diagramas de colaboração e diagramas de seqüência em UML capturem o comportamento de interação entre objetos, ambos têm uma visão atemporal e específica para um cenário particular. O que melhor funciona para obter o comportamento dinâmico é o diagrama de estado. Toda a classe tem um diagrama de estado que descreve o os possíveis estados das instâncias e as possíveis transições que podem ocorrer. Cada instância executa o diagrama de estados sempre que ela recebe um evento. A interpretação da máquina de estados é parte da implementação da classe *Element*. Porém, isto não é suficiente. A MVUML descreve as dependências inter-objetos e restrições entre objeto com OCL (Object Constraint Language). Isto faz com que mudanças de estado

em um objeto causem eventos importantes para outros objetos do sistema não conectados como o objeto original através do diagrama de estados. Além disso, como a UML (ainda) não é uma linguagem de programação, detalhes de algoritmos e classes são programados manualmente em uma arquitetura de extensão da UML.

2.4 Smalltalk

A linguagem Smalltalk-80 foi concebida por Alan Kay durante o projeto *Dynabook* (XEROX PARC, 1970 a 1980 – um ambiente novo de computação para adultos e crianças com características altamente revolucionárias para a época). Kay se inspirou nas linguagens LOGO (interface gráfica e destinada a crianças) e LISP.

Smalltalk pode ser considerada uma linguagem rigorosamente orientada a objeto, uma vez que não possui tipos de dados primitivos, tudo é objeto. Toda a computação é realizada através de envio de mensagens entre objetos. Isto inclui até operações aritméticas tais como soma, subtração, operações de controle de fluxo sobre objetos representando blocos de código. Não existem operadores nem comandos tais como *if* ou *while* disponibilizados pela linguagem como no caso de Java [Arnold and Gosling, 1996] ou C++ [Stroustrup, 1986].

Smalltalk tem sido tradicionalmente implementada sobre uma máquina virtual baseada em pilha junto com um conjunto de bibliotecas padrão [Harris, 2001]. Estas bibliotecas incluem implementação de estruturas de dados tais como *sets*, *sequences*, *dictionaries* bem como interfaces para dispositivos periféricos. A maioria destas bibliotecas é implementada com a própria linguagem Smalltalk. Porém, cerca de uma centena de rotinas primitivas são implementadas pela máquina virtual e tratadas de forma especial durante as operações de envio de mensagem. Estas rotinas primitivas implementam operações aritméticas, alocação de espaço em memória para objetos, funções de entrada e saída. As instruções contidas no *bytecode* geralmente trabalham manipulando a pilha e executando mensagens de envio. O estado inicial de uma máquina virtual é obtido a partir da imagem virtual que descreve o conteúdo da *heap*, os programas iniciais e os processos disponíveis dentro do sistema. Estes podem incluir compiladores, carregadores e ferramentas de desenvolvimento.

2.4.1 Meta-classes

Smalltalk-80 também define um sistema de meta-classes relativamente simples. O relacionamento entre meta-classes e classes normais é similar ao relacionamento entre classes normais e suas instâncias. Uma meta-classe é simplesmente uma classe cujas instâncias são classes. Dentro da máquina virtual Smalltalk a hierarquia de meta-classes é definida para espelhar a hierarquia de classes normais. Por exemplo, se a classe `A` especializa a classe `B` então a meta-classe `A Class` especializa a meta-classe `B Class`. A classe `Object` é a raiz (origem, pai de todas) da hierarquia de classes normais. A meta-classe `Class` é a raiz (origem, pai de todas) da hierarquia das meta-classes. As próprias meta-classes são instâncias da `Meta-Class`, que por sua vez é instância de `Class`.

O sistema de meta-classes provido por Smalltalk pode ser considerado limitado. As Meta-classes seriam utilizadas na inicialização de instâncias e na representação de informações sobre a classe como um todo e não informações sobre cada uma das instâncias [Goldberg and Robson, 1983]. Na linguagem Java este tipo de funcionalidade é provida por construtores e membros estáticos de classes normais [Arnold and Gosling, 1996].

2.4.2 Arquitetura da Máquina Virtual Smalltalk

O sistema Smalltalk-80 possui dois componentes principais:

- A imagem virtual – constituída de todos os objetos do sistema;
- A máquina virtual.

Uma implementação da máquina virtual deve ser capaz de carregar uma imagem virtual.

O código fonte de métodos escritos por programadores deve ser compilado em seqüências de instruções chamadas de *bytecodes*. Estes *bytecodes* são então interpretados por um *interpretador* orientado a pilha. O *interpretador* manipula os objetos que existem em uma área de memória chamada *memória de objetos*. Existem também requisitos de *hardware* para que o *interpretador* e a *memória de objetos* sejam implementados [Goldberg and Robson, 1983].

Compilador

Um programador não interage diretamente com o compilador. Quando um novo método é adicionado a uma classe, a classe solicita ao compilador uma instância do *método compilado* que contém os *bytecodes* referentes ao novo método. A classe provê ao compilador algumas informações necessárias não contidas no código fonte, tais como nomes das variáveis de instância dos receptores (um receptor deve ser entendido como o objeto que recebeu a mensagem) e dicionários contendo variáveis compartilhadas acessáveis.

Métodos Compilados Uma instância de *método compilado* além de armazenar os *bytecodes* de um método também contém um conjunto de objetos chamado de *literal frame*. Um *literal frame* contém todos os objetos que não podem ser referenciados diretamente pelos *bytecodes*.

Bytecodes O interpretador entende 256 *bytecodes* que podem ser divididos em cinco categorias: *pushes*, *stores*, *sends*, *returns* e *jumps*.

Interpretador

O interpretador executa as instruções *bytecode* encontradas dentro de objetos *métodos compilados*. O interpretador tem seu estado definido pelos seguintes elementos:

- o *método compilado* cujos *bytecodes* estão sendo executados;
- a localização do próximo *bytecode* a ser executado no *método compilado – o apontador de instrução*;
- o receptor e argumentos da mensagem que invocou o *método compilado*;
- qualquer variável temporária que o *método compilado* precise;
- a pilha.

A execução da maioria das instruções *bytecode* involve a pilha do interpretador. *Bytecodes* do tipo *push* indicam onde encontrar objetos para adicionar à pilha. *Bytecodes* do tipo *Store*

indicam onde colocar objetos encontrados da pilha. *Bytecodes* do tipo *send* removem o receptor e argumentos da mensagem vindas da pilha. Quando o resultado de uma mensagem é computado, ele é empilhado no topo da pilha.

O interpretador funciona em um ciclo das seguintes ações:

(1) carregar o *bytecode* a partir do *método compilado*;

(2) incrementar o apontador de instrução;

(3) executar a função especificada pelo *bytecode*.

Contexto *Bytecodes* do tipo *push*, *store* causam poucas mudanças no estado do interpretador. Objetos podem ser movidos de/ou para a pilha e o apontador de instrução sempre está mudando; porém a maioria do estado do interpretador permanece inalterado. Instruções do tipo *send* ou *return* podem causar mudanças mais significativas no estado do interpretador. Quando uma mensagem é enviada, todos os cinco elementos que compõem o estado do interpretador precisam ser alterados para executar as instruções contidas em outro *método compilado* resultante da execução da nova mensagem. O estado anterior do interpretador precisa ser salvo porque os *bytecodes* localizados após o *bytecode* que causou o envio da nova mensagem precisam ser executados após o retorno da mensagem.

O interpretador salva seu estado em objetos chamados *contextos*. Irão existir muitos contextos em um sistema em um mesmo momento. O contexto que representa o estado corrente do intrepretador é chamado de *contexto ativo*. Quando um bytecode do tipo send no *método compilado* do contexto ativo requer que um novo *método compilado* seja executado, o contexto ativo fica *suspenso* e o novo contexto é criado e torna-se o contexto ativo. O contexto suspenso guarda o estado associado ao *método compilado* original até que o contexto se tornar o ativo novamente. O contexto precisa lembrar do contexto que ele suspendeu para que o contexto suspenso possa ser continuado quando um resultado for retornado.

Mensagens Quanto um *bytecode* do tipo *send* é encontrado, o interpretador localiza o respectivo *método compilado* da seguinte forma:

(1) encontra o receptor da mensagem. O receptor está abaixo dos argumentos na pilha do interpretador. O número de argumentos é indicado pelo *bytecode* do tipo *send*;

(2) acessa um dicionário de mensagem. O dicionário de mensagem original é encontrado na classe do receptor;

(3) procura o identificador da mensagem no dicionário de mensagem. O identificador é indicado pelo *bytecode* do tipo *send*;

(4) Se o identificador é encontrado, então o *método compilado* associado descreve a resposta para a mensagem;

(5) Se o identificador não é encontrado, é preciso procurar em um novo dicionário de mensagens (retornando ao passo 3). O novo dicionário de mensagens será encontrado na superclasse da última classe cujo dicionário de mensagens for procurado. Este ciclo pode ser repetido diversas vezes.

Caso o identificador não seja encontrado na classe do receptor nem em nenhuma de suas superclasses, um erro é reportado e a execução dos próximos *bytecodes* é suspensa.

Métodos Primitivos As ações do interpretador após encontrar um *método compilado* depende se o mesmo é um *método primitivo* ou não. Caso o *método compilado* não seja primitivo, um novo objeto *Contexto de Método* é criado e torna-se ativo. Caso seja um método primitivo, o interpretador pode ser capaz de responder a mensagem sem realmente executar os *bytecodes*.

As principais funções dos métodos primitivos são:

- funções aritméticas;
- gerenciamento de armazenamento;
- controle;
- entrada e saída.

A Memória de Objeto

A memória de objeto provê ao interpretador uma interface para os objetos que formam a imagem virtual do Smalltalk-80. Cada objeto está associado a um identificador único

chamado de *ponteiro de objeto*. A memória de objeto e o interpretador comunicam-se através de ponteiros de objetos.

A memória de objeto associa cada um dos ponteiros de objeto com um conjunto de outros ponteiros de objeto. Todo ponteiro de objeto está associado com o ponteiro de objeto de uma classe. Se um objeto tem variáveis de instância, seu ponteiro de objetos também está associado com os ponteiros de objeto de seus valores. O valor de uma variável de instância pode ser alterado, mas a classes associada com um objeto não pode. A memória de objeto provê cinco funções fundamentais para o interpretador:

(1) acessar o valor de uma variável de instância de um objeto. O ponteiro de objeto e o índice da variável de instância precisam ser fornecidos. O ponteiro de objeto do valor da variável de instância é retornado;

(2) alterar o valor de uma variável de instância de um objeto. O ponteiro de objeto e o índice da variável de instância precisam ser fornecidos. O ponteiro de objeto do novo valor também precisa ser fornecido;

(3) acessar uma classe de um objeto. O ponteiro de objeto da instância precisa ser fornecido. O ponteiro de objeto da classe da instância é retornado;

(4) criar um novo objeto. O ponteiro de objeto da classe do novo objeto e o número de variáveis de instância que o objeto deve possuir precisam ser fornecidos;

(5) recuperar o número de variáveis de instância de um objeto. O ponteiro do objeto precisa ser fornecido. O número de variáveis de instância é retornado.

Representação de números *Small Integers* Uma instância de um *SmallInteger* não precisa de armazenamento uma vez que tanto sua classe como seu valor podem ser determinados a partir de seu ponteiro de objeto.

2.5 Programação Orientada a Aspectos

A ciência da computação tem experimentado um grande número de linguagens de programação e sistemas desde o puro *assembly* e códigos de máquina passando pela programação funcional, programação procedural, programação estruturada, programação lógica,

programação orientada a objetos. Cada uma destas tecnologias tenta trabalhar a habilidade do programador em atingir uma separação de preocupações [16] no código fonte. Embora a abordagem de construção de software orientado a objetos consiga abstrair objetos do domínio do problema de forma modular, existem algumas preocupações que estão espalhadas por todas as classes, por exemplo: segurança, rastreabilidade, controle da acesso, persistência, etc (*crosscutting concerns*).

A proposta da programação orientada a aspectos é explicitar a separação das preocupações de forma que elas sejam programadas separadamente e posteriormente integradas ao código funcional [Kiczales et al., 1997]. Existem duas abordagens para a utilização da programação orientada a aspectos, a abordagem estática e a abordagem dinâmica.

A abordagem estática utiliza uma linguagem para a definição dos aspectos, uma linguagem para a construção do código funcional (aplicação em si) e um *weaver*[17] para realizar a composição dos aspectos sobre o código funcional.

A abordagem dinâmica permite a composição de aspectos durante o tempo de execução.

2.6 Meta Object Facility - MOF

O MOF[Santos, 2003] define uma linguagem abstrata e um *framework* para especificação, construção e gerenciamento de metamodelos independentes de tecnologia de implementação. Alguns exemplos incluem o metamodelo UML[Group, 2001], CWM[POOLE, 2001] e o próprio MOF. O MOF possui ainda um conjunto de regras para implementação de repositórios, que manipulam metadados descritos pelos metamodelos. Essas regras definem um padrão de mapeamento entre os metamodelos MOF e um conjunto de APIs para gerenciamento de metadados. Por exemplo, o mapeamento *MOF-IDL(Interface Definition Language)* é aplicado aos metamodelos MOF (UML, CWM) para produzir uma API CORBA que gerencie os metadados, instâncias desses metamodelos. O mapeamento *MOF-Java*, através do padrão JMI (Java Metadata Interface), define as mesmas regras de mapeamento para

[16] Do inglês, separation of concerns.
[17] Do inglês, tecelão

API Java.

A especificação MOF compreende o seguinte:

- Uma definição formal para o meta-metamodelo MOF, ou seja, uma linguagem abstrata para a definição dos metamodelos.
- As regras para o mapeamento dos metamodelos MOF para uma tecnologia de implementação, por exemplo, CORBA ou Java.
- Padrão XMI para intercâmbio, em XML, dos metadados e metamodelos entre as ferramentas. XMI define um conjunto de regras que mapeiam os metamodelos MOF e os metadados em documentos XML.

O MOF é um *framework* extensível, ou seja, novos padrões de metadados podem ser adicionados ao mesmo. Para isto, ele conta com uma arquitetura em quatro camadas, também chamada de *OMG Metadata Archicteture*, conforme mostra a Tabela 2.1.

Nível MOF	Termos Utilizados	Exemplos
M3	Meta-metamodelo	Modelo MOF
M2	Metamodelo, Meta-metadados	Metamodelo UML e CWM
M1	Modelos, Metadados	Modelos UMLs
M0	Dados, Objetos	Dados

Tabela 2.1 Arquitetura de metadados da OMG

A instância de uma camada é sempre modelada por uma instância de uma camada imediatamente superior. Desta forma, a camada *M0* onde estão os dados são modelados por modelos UML, como diagramas de classes que estão na camada *M1*. A camada *M1* por sua vez é modelada pelo metamodelo UML, camada *M2*, que utiliza os construtores básicos como classes, relacionamentos, entre outros. Este metamodelo é uma instância do modelo MOF, que é chamado também de meta-metamodelo. A extensibilidade se dá pelo fato de, em cada camada, podermos adicionar classes que são instâncias de outra classe de uma camada imediatamente superior.

Pode-se dizer que o MOF é um meta-metamodelo, uma vez que é utilizado para definir metamodelos, como por exemplo a UML.

O modelo MOF é orientado a objetos e suporta construtores de metamodelagem alinhados com os construtores da UML.

O MOF tem o propósito de permitir a definição e manipulação de metamodelos em vários domínios, com o foco inicial sendo em análise e projeto de metamodelos de objetos. As restrições em MOF são expressas pela linguagem OCL(*Object Constraint Language*).

De forma apenas ilustrativa, a Figura 2.8 mostra o modelo MOF contido no pacote "*model*", que é o único pacote do modelo.

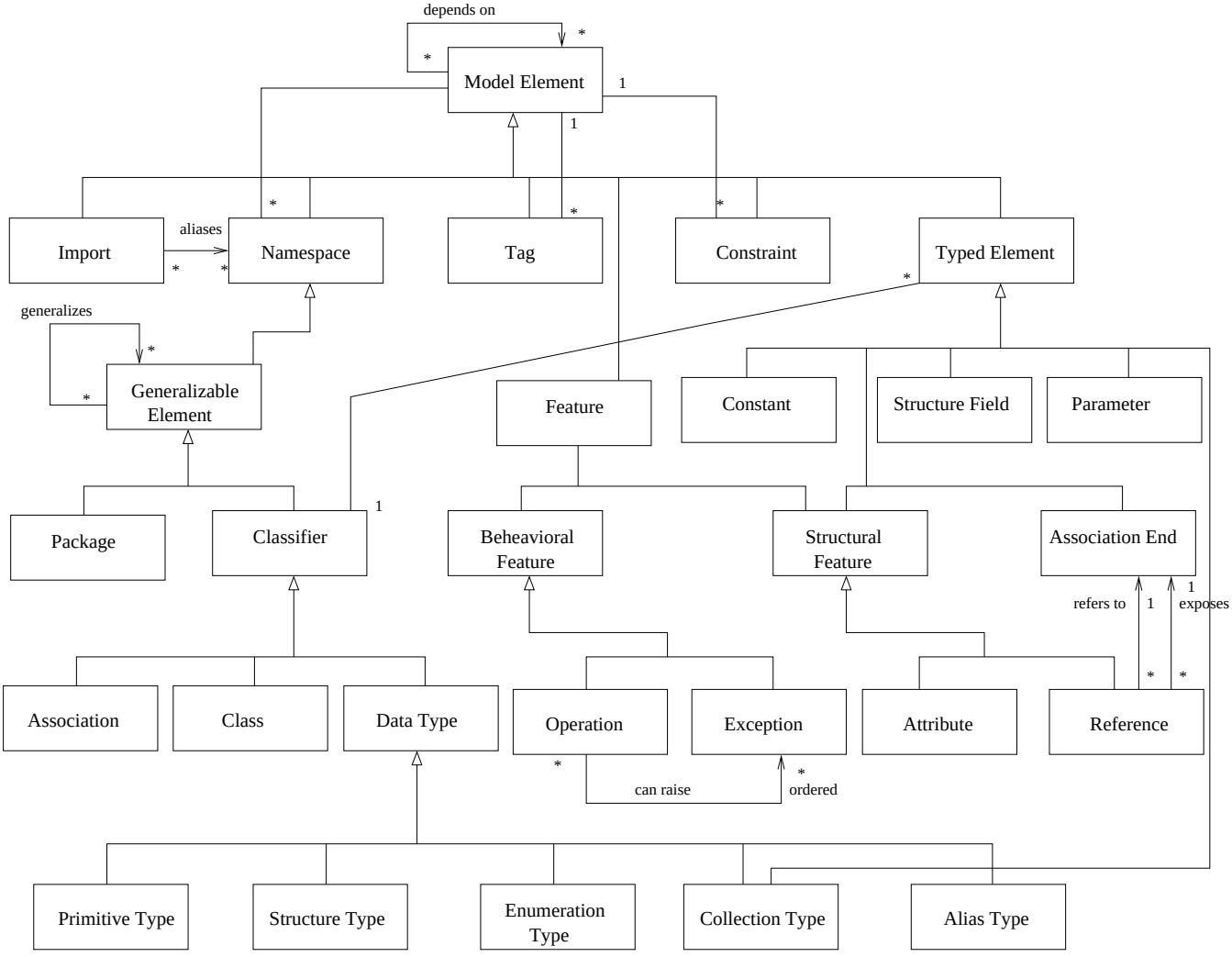

Figura 2.8 Estrutura do modelo MOF Model Package

Em MOF, um objeto é descrito como um elemento de um modelo, qualquer elemento é primeiramente um *model element*, e deve ser classificado de acordo com a hierarquia abaixo dele.

Em MOF, para a definição de um atributo, parâmetro ou constante, é necessário descrever o tipo (*type element*). *Type Element* é a classe base para classes como *Attribute*, *Parameter* e *Constant*. Essa obrigatoriedade está relacionada ao objetivo do MOF que é descrever metamodelos com o foco em análise e projeto de metamodelos de objetos, onde saber a informação de tipos é desejada.

Construtores de Metamodelagem A metamodelagem do MOF é primeiramente sobre definição de modelos de informação para meta-dados. O MOF usa um framework de modelagem de objetos que é essencialmente um subconjunto do núcleo UML. Os quatro principais construtores de modelagem são:

- Classes: modelam meta-objetos MOF;
- Associações: modelam relações binárias entre meta-objetos;
- Tipos de Dado[18]: modelam os outro dados (tipos primitivos, tipos externos, etc.);
- Pacotes: tornam os modelos modulares.

[18]Do inglês, *data types*

Capítulo 3

Conceitos de Orientação a Objeto

Os princípios da abordagem de construção de software orientada a objeto são implementados de diferentes formas por diferentes linguagens de programação orientadas a objeto. Esse Capítulo discute os principais conceitos de orientação a objeto implementados pelas linguagens Java [Flanagan, 1997] e Eiffel [Meyer, 1997] e que possuem um paralelo no ambiente Virtuosi. Esse Capítulo referencia constantemente o Capítulo 4 para explicar os conceitos de orientação a objeto implementados pela Virtuosi.

3.1 Justificativa

Existe uma grande lacuna referente à comunicação entre homens e computadores. Computadores operam, em geral, num nível muito atômico, manipulando dígitos binários, posições de memória, registradores, etc., enquanto que pessoas preferem expressar-se usando línguas naturais e às vezes com o auxílio de notação matemática [Kowaltowski, 1983]. Para superar esse obstáculo foram criadas sucessivas camadas de abstração sobre as instruções de máquina. A primeira camada de abstração foi a linguagem de montagem[1] que embora facilitasse a construção de programas ainda utilizava instruções muito próximas das instruções de máquina. Depois disso foram introduzidas as linguagens imperativas (tais como FORTRAN, BASIC, C), que eram abstrações sobre a linguagem de montagem mas que ainda obrigavam o programador a pensar em termos da estrutura do computador ao invés da estrutura do problema que estava tentando solucionar, produzindo assim uma lacuna entre os ditos domínio do problema e domínio da solução. Pode-se dizer que a capacidade de res-

[1] Do Inglês, ***assembly languages.***

olução de problemas está diretamente associada a qualidade e ao tipo de abstração utilizada [Eckel, 2003]. Sendo assim, foram introduzidas linguagens que abstraiam o mundo de forma específica, como LISP no qual tudo se resolve com listas ou PROLOG no qual tudo se resolve através de encadeamento de decisões. Cada uma dessas soluções apresenta-se ideal para um tipo específico de problema, porém torna-se inadequada para outros tipos de problema.

A abordagem de construção de software orientada a objeto propõe uma maneira diferente de representar os elementos dentro do domínio da solução. Essa representação é genérica o suficiente para não ficar restrita a um determinado tipo de problema e permite que a estrutura do domínio da solução seja definida em termos do domínio do problema. Em outras palavras, esta abordagem permite que uma solução seja descrita em termos do problema e não em termos do computador no qual a solução é desenvolvida. A abordagem de construção de software orientado a objeto tem como base o conceito de que o mundo real pode ser descrito em termos de uma coleção de tipos de objetos diferentes que interagem entre si. Essa interação ocorre através da troca de mensagens, ou seja, através da invocação de operações de outro objeto. Dessa forma, a computação realizada por um software ocorre através de objetos realizando tarefas em resposta a requisições de outros objetos. Esse tipo de relacionamento pode ser chamado de cliente-fornecedor uma vez que um objeto faz uso dos serviços disponibilizados por outros objetos no papel de cliente e ele mesmo disponibiliza serviços para outros objetos no papel de fornecedor.

Um objeto é composto de outros objetos, e por este motivo, uma linguagem orientada a objeto deve ser capaz de criar novos tipos de objetos utilizando os tipos já existentes.

3.2 Tipos Abstratos de Dado e Classes

Um grupo de objetos no mundo real com comportamento similar pode ser especificado pelo conjunto de operações relevantes que têm em comum. Quando tal especificação é independente de uma linguagem de programação em particular, é chamada de **tipo abstrato de dado** (TAD). O conceito de TAD pode ser visto como a base da abordagem de construção de software orientado a objeto [Rist and Terwinllinger, 1995].

A implementação de um TAD em uma linguagem de programação orientada a objeto

é geralmente feita através da criação de uma **classe** que implementa todo o conjunto das operações definidas pelo TAD correspondente.

3.2.1 Tipos Abstratos de Dado e Classes em Java

A Figura 3.1 mostra um exemplo simplificado (sem as operações) da implementação de uma classe na linguagem Java.

```
class Person
{
   /* corpo da classe Person, atributos e métodos */
   ...
}
```

Figura 3.1 Exemplo de uma classe em Java

Em Java, além de classes normais existem as classes abstratas e as interfaces. Tanto a classe abstrata quanto a interface são explicadas em detalhe na Seção 3.7.1.

3.2.2 Tipos Abstratos de Dado e Classes em Eiffel

A Figura 3.2 mostra um exemplo simplificado (sem as operações) da implementação de uma classe na linguagem Eiffel.

Em Eiffel, além de classes normais, existem as classes abstratas e as classes expandidas. O conceito de classe abstrata em Eiffel é explicado em detalhe na Seção 3.7.2, e o conceito de classe expandida na Seção 3.3.2.

```
class PERSON
creation
   --nome do(s) procedimento(s) utilizados na criação
   -- de novas instâncias da classe.
   ...
feature
   --declarações das features (operações e atributos)
   -- da classe
   ...
end--PERSON
```

Figura 3.2 Exemplo de uma classe em Eiffel

3.2.3 Tipos Abstratos de Dado e Classes na Virtuosi

A Figura 3.3 mostra um exemplo simplificado (sem as operações) de uma implementação de classe segundo os conceitos especificados pelo metamodelo da Virtuosi através da linguagem Aram[2].

```
class Person
{
   /* corpo da classe Person, atributos, métodos e ações */
   ...
}
```

Figura 3.3 Exemplo de uma implementação de classe segundo o metamodelo da Virtuosi

Uma classe segundo a Virtuosi é discutida em detalhe na Seção 4.1.3.

[2]Aram é a primeira linguagem de programação compatível aos conceitos de orientação a objeto definidos pelo metamodelo da Virtuosi e também foi desenvolvida dentro do Projeto Virtuosi [BORGES, 2004].

3.3 Atributos

Além das operações, existem outras características do grupo de objetos que são necessárias à implementação, mas não são operações em si. Tais características simplesmente armazenam informação e são chamadas de **atributos**. O conjunto dos atributos de uma classe é chamado de **representação da classe**.

Uma classe serve como modelo para a construção de objetos de um determinado TAD. Um objeto pode ser chamado de instância de uma classe. Um objeto possui duas partes principais, a saber:

- **estado de objeto**: conjunto particular de valores correspondente a cada um dos atributos definidos pela representação da classe;
- **comportamento**: conjunto das operações implementadas pela classe compartilhado por todas as instâncias da classe.

Cada objeto tem um estado próprio e particular. Um objeto pode possuir um estado totalmente diferente de outro objeto da mesma classe embora compartilhe da mesma representação e do mesmo comportamento.

3.3.1 Atributos em Java

Em Java, além dos atributos que fazem parte do estado de um objeto – atributo de instância – existem os atributos que fazem parte do estado de uma classe – atributos de classe. Um atributo de classe não pertence a uma instância em particular, ao invés disso, um atributo de classe pertence a classe que define os objetos, e portanto, é compartilhado por todas as instâncias.

Java é uma linguagem híbrida, e pode ser considerada como não rigorosamente orientada a objeto. Isso é explicado em parte pela composição de um objeto, ou seja, pelo seu estado. Um atributo de um objeto em Java pode ser um tipo referência ou um tipo primitivo. O valor de um atributo de um tipo primitivo não é um objeto, mas sim, um valor localizado em uma área de memória. A Figural 2.2 mostra os tipos primitivos que a linguagem Java disponibiliza.

Um tipo primitivo define somente quais os valores suportados pelo tipo. Um tipo primitivo não define um conjunto de operações para atuar sob os valores que pode armazenar, ao invés disso, um valor de um tipo primitivo sofre a ação de operações definidas pela linguagem.

A existência de tipos primitivos em Java visa principalmente melhorar o desempenho durante a execução do software.

Um atributo do tipo referência possui como valor o endereço de um objeto em memória.

A Figura 3.4 mostra a implementação em Java de uma classe com dois atributos, o primeiro sendo uma referência e o segundo um valor de tipo primitivo.

```
class Person
{
   String name;
   int age;
}
```

Figura 3.4 Atributos em uma classe em Java

3.3.2 Atributos em Eiffel

A linguagem Eiffel pode ser considerada rigorosamente orientada a objeto. Um dos motivos é a ausência de tipos primitivos. Em Eiffel existe uma distinção entre a linguagem utilizada para a definição de classes e as classes em si mesmas. A linguagem é utilizada para criar novas classes a partir das já existentes. Para tanto, a linguagem Eiffel assume a existência de uma biblioteca de classes pré-definidas (INTEGER, REAL, BOOLEAN, CHARACTER e DOUBLE).

Um atributo em Eiffel sempre está associado a um objeto, porém, tal associação pode ocorrer de forma direta ou indireta.

Um atributo associado indiretamente armazena uma **referência** ao objeto. O mecanismo

de referência permite que um mesmo objeto seja referenciado por mais de uma referência ao mesmo tempo.

Por outro lado, um atributo associado diretamente armazena o próprio objeto. Esse tipo de atributo é um exemplo de atributo de **tipo expandido**[3]. Um atributo de tipo expandido tem uma relação de composição exclusiva com o objeto que o possui, ou seja, o atributo de um objeto não pode ser referenciado por um objeto externo. Um objeto contido é também chamado de sub-objeto.

Existem duas maneiras de se utilizar um atributo de tipo expandido.

A primeira é por declará-lo como atributo expandido durante a definição da classe. Essa maneira é mais flexível, uma vez que permite a declaração de um atributo de qualquer classe existente como sendo expandido. A Figura 3.5 mostra o código fonte de uma classe X com a declaração de dois atributos da classe C: o atributo *ref* de tipo referência e o atributo *sub* de tipo expandido (sub-objeto).

```
class X feature
   ref: C
   sub: expanded C
end
```

Figura 3.5 Atributo de tipo expandido declarado por definição de classe

Uma instância chamada *objX* de um objeto da classe X pode ser ilustrada conforme a Figura 3.6, mostrando a diferença entre um atributo de tipo referência e um atributo de tipo expandido.

A segunda maneira de se utilizar um atributo expandido, menos flexível, é utilizada quando se tem certeza que um objeto de uma classe sempre será utilizado como atributo expandido, neste caso a classe do atributo deve ser definida como uma classe expandida. A Figura 3.7 mostra o código fonte de uma classe expandida de nome E.

[3]Do Inglês, ***expanded type.***

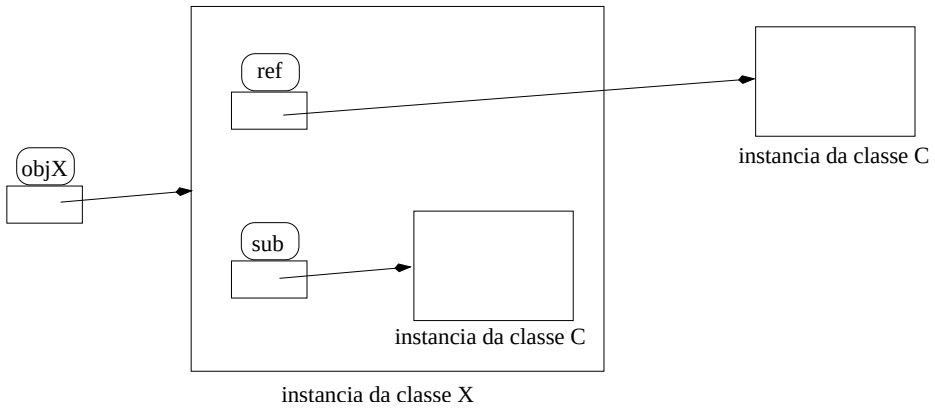

Figura 3.6 Ilustração de um relacionamento de composição exclusiva através do uso de atributos do tipo expandido.

```
expanded class E feature
   -- resto da definição da classe
end
```

Figura 3.7 Atributo de tipo expandido declarado por definição de classe

Um exemplo importante de classes expandidas são as classes pré-definidas (INTEGER, REAL, BOOLEAN, CHARACTER e DOUBLE).

A Figura 3.8 mostra uma classe implementada em Eiffel com dois atributos. O primeiro sendo uma referência a objeto de uma classe normal e o segundo sendo um atributo de tipo expandido, definido por uma classe expandida (no caso *INTEGER*).

A existência de tipos expandidos em Eiffel é justificada por quatro questões principais, a saber:

(1) desempenho do software;

(2) espaço poupado pela ausência da referência;

(3) possibilidade de importar softwares em outras linguagens que fazem uso de tipos primitivos;

(4) permitir o relacionamento de composição exclusiva.

```
class PERSON
creation
   --nome do(s) procedimento(s) utilizados na criação
   -- de novas instâncias da classe.
feature
   name: STRING
   idade: INTEGER
end--PERSON
```

Figura 3.8 Atributos em uma classe em Eiffel

Deve-se observar que os três primeiros motivos – 1, 2 e 3 – podem ser relacionados a desempenho ou compatibilidade, enquanto que o quarto motivo é puramente conceitual, contradizendo assim, o exposto em [Kolling, 1999].

3.3.3 Atributos na Virtuosi

A exemplo de Eiffel, o ambiente Virtuosi disponibiliza uma biblioteca de classes básicas (*Integer*, *Real*, *Boolean*, *Character* e *String*) para a construção de novas classes chamadas de 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.

Os atributos segundo a Virtuosi são discutidos em detalhe na Seção 4.1.4.

3.4 Referências

Uma referência pode assumir papéis diferentes de acordo com o contexto no qual se encontra. Além do papel de atributo, uma referência pode ser uma variável local, um parâmetro formal ou ainda um parâmetro real. Todos os papéis desempenhados por uma referência serão discutidos ao longo deste Capítulo.

3.4.1 Referências em Java

Conforme explicado na Seção 3.3.1 uma referência na linguagem Java referencia um objeto em memória, ou seja, a referência a objeto possui como valor o endereço de um objeto em memória.

3.4.2 Referências em Eiffel

Conforme explicado na Seção 3.3.2 uma referência na linguagem Eiffel também referencia um objeto em memória. Portanto, uma referência a objeto também possui como valor o endereço de um objeto em memória.

3.4.3 Referências na Virtuosi

O metamodelo da Virtuosi define quatro tipos de referência, a saber:

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

Todos esses tipos de referência são discutidos em detalhe no Capítulo 4, respectivamente nas Seções 4.1.4,4.1.2,4.1.2 e 4.1.1.

3.5 Operações

3.5.1 Operações em Java

A implementação de uma operação na linguagem Java é feita através de um método. As partes fundamentais de um método são o nome, uma lista de parâmetros, o tipo de retorno e o corpo do método. A Figura 3.9 ilustra a declaração de um método.

```
class ClassName
{
   returnType methodName( /* lista de parâmetros */ )
   {
       /* corpo do método */
   }
}
```

Figura 3.9 Ilustração da composição de um método em Java

O valor retornado por um método é um valor de um tipo primitivo ou uma referência para um objeto. Para tanto, na declaração de um método, a palavra que vem antes do nome do método – o tipo de retorno – é um dos tipos primitivos disponibilizados pela linguagem Java, o nome de uma classe ou ainda o nome de uma *interface*. O conceito de *interface* em Java é explicado em detalhes na seção 3.7.1.

Um método em Java pode não retornar um valor. Para isto, a palavra que vem antes do nome do método deve ser a palavra *void*.

Cada um dos itens que compõem a lista de parâmetros tem um tipo e um nome. O tipo do argumento pode ser o nome de um tipo primitivo , o nome de uma classe ou ainda o nome de uma *interface*. Quando um argumento é uma referência a objeto o que é passado para o método é uma cópia da referência a objeto passada como parâmetro. No caso de um argumento de tipo primitivo o que é passado para o método é uma cópia do valor primitivo passado como parâmetro. O nome de um método mais sua lista de parâmetros formam

a assinatura de um método. Uma classe não pode possuir dois métodos com a mesma assinatura.

Uma classe em Java precisa também de métodos especiais, denominados construtores, para criar novas instâncias da classe. A utilização de um construtor em Java é explicado na Seção 3.7.1.

3.5.2 Operações em Eiffel

A implementação de uma operação na linguagem Eiffel é feita através de uma rotina, existem dois tipos de rotina, a saber:

- um procedimento;
- uma função;

Uma rotina, seja um procedimento ou uma função, tem como parte de sua definição um uma lista de parâmetros formais na qual cada um dos parâmetros formais tem um nome e um tipo. O tipo do parâmetro formal pode ser um tipo expandido ou uma classe. No primeiro caso, o que é passado para o procedimento, no momento de sua invocação, é uma cópia do valor passado como parâmetro real. No segundo caso, quando o parâmetro formal é uma referência a um objeto de uma determinada classe, o que é passado para o método – no momento de sua invocação – é uma cópia da referência passada como parâmetro real.

Um procedimento é uma rotina que não retorna valor. Em Eiffel recomenda-se que um procedimento seja empregado quando a rotina causar um efeito no estado do objeto. Recomenda-se também que uma função seja utilizada para as rotinas que retornam um valor sem alterar o estado do objeto.

Em Eiffel, as partes fundamentais de um procedimento são o nome, uma lista de parâmetros formais e o corpo do método. A Figura 3.10 ilustra a declaração de um procedimento em Eiffel. A assinatura de um procedimento é definida pelo nome do procedimento mais sua lista de parâmetros. Uma classe não pode possuir dois procedimentos com a mesma assinatura.

Em Eiffel, as partes fundamentais de uma função são o nome, uma lista de parâmetros formais, o corpo do método e o tipo de retorno da função. A Figura 3.11 ilustra a declaração

```
class PERSON creation

feature
   procedureName(
                  -- lista de parâmetros formais
              ) is
    do
       -- corpo do procedimento
    end
end
```

Figura 3.10 Ilustração da composição de um procedimento em Eiffel

de uma função em Eiffel. A assinatura de uma função é definida pelo nome da função, sua lista de parâmetros e seu tipo de retorno. A exemplo de um parâmetro formal, o tipo de retorno pode ser um tipo expandido ou uma referência a objeto. Uma classe não pode possuir duas funções com a mesma assinatura.

```
class PERSON creation

feature
    functionName(
                  -- lista de parâmetros formais
              ): returnType is
    do
       -- corpo da função
    end
end
```

Figura 3.11 Ilustração da composição de uma função em Eiffel

Eiffel utiliza um procedimento para a criação de novas instâncias de uma classe. Para indicar que um procedimento é um procedimento de criação é preciso listar o seu nome logo

em seguida da palavra *creation*. A Figura 3.12 mostra um exemplo de um procedimento de criação em Eiffel que recebe dois parâmetros do tipo referência a um objeto (n e a)e um parâmetro de tipo expandido (y).

```
class PERSON
creation
    make
feature
   make(n:STRING; a:STRING; y:INTEGER) is
    do
      name := n
      address := a
      year_of_birth := y
    end
end
```

Figura 3.12 Um procedimento de criação de um objeto em Eiffel

3.5.3 Operações na Virtuosi

Segundo o metamodelo da Virtuosi, uma operação de TAD 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.

As operações na Virtuosi são discutidas em detalhe na Seção 4.1.6.

3.6 Encapsulamento e Visibilidade

Encapsulamento em Relação a Objetos Um dos princípios encontrados na abordagem de construção de software orientado a objeto é **encapsulamento**. Encapsulamento pode ser entendido no sentido de recomendação para que o estado de um objeto somente possa ser alterado por operações da classe que o define e pela própria instância que possui tal estado. Considere o seguinte exemplo: Uma carro A possui um motor M. O motor de um carro só pode ser acessado por um carro, além disso não é qualquer carro que tem acesso ao motor M, somente o carro A tem tal acesso.

Encapsulamento em Relação a Classes A obtenção de encapsulamento é, em parte, alcançada através da **visibilidade** aplicada tanto em atributos como em operações de uma classe. A visibilidade define se um atributo ou operação é disponibilizado e ainda para quais classes. Uma operação definida por uma classe pode invocar qualquer outra operação definida dentro da mesma classe e tem acesso irrestrito a todos os atributos definidos dentro da mesma classe. Acesso irrestrito significa, nesse contexto, a capacidade de leitura e/ou gravação. Isso significa que dentro de uma mesma classe, todas as operações e atributos podem ser utilizados livremente na construção de uma nova operação. Além disso, normalmente uma classe utiliza referências a objetos de outras classes durante a definição de suas operações (relacionamento cliente-fornecedor). Uma classe fornecedora pode especificar quais operações e atributos devem ser disponíveis às classes cliente. O conjunto das operações e atributos disponibilizados por uma classe constitui sua ***interface***. A *interface*[4] de uma classe é o meio pelo qual uma instância da classe interage com outros objetos. Considere o seguinte exemplo: Existem duas classes, uma cliente A e uma fornecedora B: existe uma operação X fornecida pela classe B. Do ponto de vista da classe A que é cliente da classe B, isto é tudo o que importa. Por outro lado, para a classe B a operação X é muito complexa e talvez seja melhor dividir a operação complexa em operações menores *X1* e *X2*. A classe A não precisa, nem deve, ter conhecimento das operações menores, o que importa para ela é que a operação X proceda da forma esperada. Caso a classe fornecedora B decida reimplementar a operação X com três operações menores ao invés de duas, a classe cliente A não sofre

[4] A técnica de programação através de *interfaces* não é exclusividade da orientação a objeto, porém seu uso é facilitado pelas linguagens orientadas a objeto.

nenhuma alteração.

3.6.1 Encapsulamento e Visibilidade em Java

Em Java tanto atributos como métodos podem ter sua visibilidade alterada pelo uso de especificadores de acesso. Segue uma lista dos quatro níveis de acesso e seus respectivos especificadores:

(1) acesso público – denotado pela palavra *public*;

(2) acesso privado – denotado pela palavra *private*;

(3) acesso de pacote – denotado pela ausência de um especificador.

(4) acesso protegido – denotado pela palavra *protected*.

O acesso público indica que o método ou atributo em questão pode ser acessado sem restrições, por métodos definidos em qualquer classe. O termo "sem restrições" em se tratando de atributos significa que o atributo pode ser lido ou alterado, enquanto que em se tratando de método, significa que o método pode ser invocado a partir de métodos dentro ou fora da classe que o define.

O acesso privado indica que o método ou atributo em questão pode ser acessado somente por métodos definidos dentro da própria classe. Ou seja, um atributo com acesso privado somente pode ser lido ou alterado a partir de uma referência que esteja dentro de um dos métodos da própria classe. No entanto, isso não impede que um objeto altere o estado de outro objeto da mesma classe, quebrando a recomendação de encapsulamento em relação a objetos. Desta forma, pode-se dizer que a linguagem Java fornece encapsulamento somente em relação a classes. Para ilustrar, considere a Figura 3.13. A Figura mostra o código fonte de uma classe A em Java. A classe possui um atributo inteiro com acesso privado chamado k e um método f que recebe como parâmetro um objeto t da própria classe. Considere a execução de uma invocação do método f a partir de um objeto *alfa*. Deve-se notar que os comandos definidos dentro do método f alteram o valor de k tanto para o objeto corrente, *alfa*, quanto para o objeto t.

```
class A {
   private int k;
   void f ( A t ){
       k = 2;
       k = t.k;
       t.k = 10;   //Quebra de encapsulamento em relação a objetos.
   }
}
```

Figura 3.13 Quebra de encapsulamento em relação a objetos em Java

O acesso de pacote (também chamado de acesso amigável[5] ou acesso padrão) e o acesso protegido tem seus comportamentos descritos em termos da existência de pacotes e do relacionamento de herança.

Um único arquivo fonte em Java pode possuir uma ou mais classes. Dentro de um arquivo fonte somente uma classe pode ser pública, ou seja, somente uma classe pode ser utilizada por outras classes. As demais classes definidas dentro do arquivo fonte, se existirem, são escondidas do mundo exterior. Essas classes escondidas servem somente de suporte para a construção da classe pública principal do arquivo.

Um arquivo fonte em Java sempre faz parte de um pacote, mesmo que esse não seja declarado explicitamente. Recomenda-se que um pacote agrupe classes que tenham um relacionamento forte entre si. Um pacote em Java pode ser entendido como uma biblioteca de classes. Entre os fatores que servem de motivação para o uso da técnica de pacotes está a obtenção de nomes únicos para as classes. Isto se justifica no caso de Java principalmente pela possibilidade de desenvolver aplicativos na *internet* na qual um conflito de nomes pode facilmente ocorrer. Com o uso de pacotes, o nome de uma classe é composto pelo nome de seu pacote mais o nome da própria classe. Um pacote agrupa o conjunto das classes públicas contidas nos arquivos fonte que o compõe.

O acesso de pacote indica que o método ou atributo em questão pode ser acessado por métodos definidos por qualquer classe que pertença ao mesmo pacote.

[5]Do Inglês, ***friendly.***

O acesso protegido indica que o método ou atributo em questão pode ser acessado por métodos definidos por qualquer classe que pertença ao mesmo pacote e por métodos definidos por subclasses pertencentes à qualquer outro pacote. Deve-se notar que no caso da tentativa de acessar um atributo ou método protegido a partir de uma subclasse, o acesso somente pode ser realizado através de uma referência do tipo da subclasse em questão ou de alguma subclasse da própria subclasse, nunca através de uma referência da superclasse.

3.6.2 Encapsulamento e Visibilidade em Eiffel

Em Eiffel atributos e rotinas (funções ou procedimentos) são chamadas de características[6]. Uma característica em Eiffel em relação à visibilidade pode ser:

- escondida: quando somente pode ser acessada por rotinas internas à classe e não pode ser acessada por classes cliente;
- exportada: quando pode ser acessada por rotinas internas à classe e pode ser acessada por classes cliente.

Uma classe em Eiffel define quais dentre suas características são exportadas e também define quais são as classes cliente para as características exportadas. Isto ocorre através do conceito de lista de exportação. São definidas seções dentro da classe identificadas pela palavra *feature*. Após a palavra *feature* de cada uma das seções encontra-se uma lista de exportação (denotada pelo uso de chaves) contendo o nome das classes que podem acessar as features da seção em questão. Uma classe pode possuir uma ou muitas seções *feature*. Existem duas classes internas de Eiffel que facilitam o uso da lista de exportação, a saber: *ANY* e *NONE*. Quando uma lista de exportação contém a classe *ANY* isso significa que as características pertencentes à seção são exportadas para toda e qualquer classe. Quando uma lista de exportação contém a classe *NONE* isso significa que as características pertencentes à seção não são exportadas para nenhuma classe, ou seja, não existem classes clientes dessas características. A Figural 3.14 mostra uma classe em Eiffel com um procedimento de construção e uma função complexa exportadas para toda e qualquer classe e uma função simples exportada para *NONE*, ou seja, a função simples somente pode ser utilizada pelas rotinas definidas dentro da classe *PERSON*.

[6]Do Inglês, ***feature.***

```
class PERSON
creation
    make
feature {ANY}
   make(n:STRING; a:STRING; y:INTEGER) is
    do
       -- ...
    end
   complexFunction():STRING is
    do
       -- ...
    end
feature {NONE}
   simpleFunction():CHARACTER is
    do
       -- ...
    end
   -- ... demais características não exportadas
end
```

Figura 3.14 Duas seções de características em Eiffel, uma exportada e outra não exportada

Em Eiffel, embora um atributo possa ser exportado, seu valor somente é alterado por rotinas da própria classe. Isso significa que um atributo quando exportado é somente de leitura, ou seja, não pode sofrer atribuição. Isso garante o encapsulamento em relação a objetos, enquanto que a lista de exportação garante o encapsulamento em relação a classes.

3.6.3 Encapsulamento e Visibilidade na Virtuosi

Segundo o metamodelo da Virtuosi, somente operações e construtores podem ter sua visibilidade alterada no que tange o acesso originado em outras classes. Uma operação ou um construtor define explicitamente quais as outras classes cujas operações ou construtores podem realizar invocações sobre a operação ou construtor em questão. Para tanto, o meta-

modelo da Virtuosi define o conceito de lista de exportação.

As questões de encapsulamento e visibilidade na Virtuosi são detalhadas nas Seções 4.1.6 e 4.1.7.

3.7 Herança entre Classes

Considere a seguinte situação: Um TAD chamado A possui cinco operações que definem seu comportamento. Um outro TAD chamado B possui as mesmas cinco operações de A e mais duas operações particulares. Uma linguagem de programação orientada a objeto que implemente esses tipos abstratos de dado deve prover um meio pelo qual somente as diferenças entre A e B precisem ser definidas na classe B. Isso é obtido através de um relacionamento de herança. Pode-se dizer que quando duas classes estão relacionadas por herança, uma delas é a classe ancestral e outra é a classe herdeira. Uma classe herdeira herda o comportamento e a representação de sua classe ancestral.

O mecanismo de herança causa alterações no comportamento normal de uma classe com relação à visibilidade dos atributos, invocação de operações, passagem de parâmetros, comandos de atribuição e até na questão do encapsulamento conforme as Seções 3.7.1, 3.7.2 e 3.7.3 detalham.

O mecanismo de herança auxilia o programador a evitar a redundância de código fonte. Além disso, linguagens de programação orientadas a objeto geralmente permitem que uma referência aponte para objetos de diferentes classes, desde que essas classes sejam herdeiras da classe da referência. Essa propriedade é chamada de **atribuição dinâmica**. Uma conseqüência da atribuição dinâmica é que a invocação de operações pode variar de acordo com a classe do objeto que a referência aponta. Isso ocorre porque todo objeto de uma classe herdeira tem em seu estado os atributos definidos pela classe ancestral, por isso, um objeto de uma classe herdeira é um objeto da classe ancestral. Deve-se notar que o contrário não é verdadeiro.

Algumas linguagens de programação orientadas a objeto ainda permitem que para uma determinada classe, operações distintas tenham o mesmo nome, desde que difiram na assinatura(número e tipo de parâmetros). Esta abordagem combinada a atribuição dinâmica

define a propriedade de **polimorfismo**.

Existem linguagens de programação tais quais as definidas em [Stroustrup, 1986] e [Meyer, 1997] que permitem que uma classe possua duas ou mais classes ancestrais. Essa propriedade é chamada de herança múltipla. Embora a herança múltipla permita que uma classe herde características de mais de uma classe ancestral, sua utilização acarreta em uma maior complexidade para resolver as possíveis ambigüidades geradas. Por exemplo, deve haver um mecanismo para que dois métodos distintos, cada qual definido em uma classe ancestral, possam ser invocados a partir de um objeto instância de uma classe descendente dessas duas classes ancestrais.

3.7.1 Herança em Java

A linguagem Java dá suporte ao mecanismo de atribuição dinâmica e portanto suporta polimorfismo.

Em Java, a relação de herança entre classes é indicada pelo uso da palavra *extends* entre nome da classe e o nome da classe ancestral. Uma classe herdeira pode ter somente uma classe ancestral direta[7]. Porém, uma classe ancestral pode ter muitas classes herdeiras recursivamente. Uma classe não pode ser direta ou indiretamente ancestral de si própria, o que acarretaria em uma problema de referência circular. A Figura 3.15 mostra um exemplo de uma classe ancestral e uma classe herdeira.

O mecanismo de herança em Java faz com que uma classe herdeira possua, além dos seus próprios atributos, os atributos da representação da classe ancestral. Porém, devido ao mecanismo de visibilidade de Java, somente os atributos definidos com acesso protegido e acesso público na classe ancestral podem ser acessados pelos métodos definidos na classe herdeira. Caso a classe herdeira esteja no mesmo pacote da classe ancestral, os atributos definidos com acesso de pacote também podem ser acessados. Em suma, todos os atributos de uma classe ancestral fazem parte da representação de uma classe herdeira, mas devido ao mecanismo de visibilidade e o mecanismo de pacotes de Java, podem não estar disponíveis

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

para a manipulação a partir dos métodos da classe herdeira.

O comportamento de uma classe (o conjunto dos métodos com exceção dos construtores) também é herdado. Mas devido ao mecanismo de visibilidade de Java, somente são acessíveis a partir da classe herdeira os métodos com acesso protegido e acesso público. Da mesma forma que ocorre com os atributos, os métodos com acesso de pacote podem ser acessados somente se a classe herdeira estiver no mesmo pacote da classe ancestral.

Uma vez que os construtores não fazem parte do comportamento de um TAD, eles não são herdados. Uma classe define seus próprios construtores. Para facilitar o trabalho do programador, os construtores de classes herdeiras podem invocar os construtores da classe ancestral a fim de inicializar corretamente os atributos definidos pela classe ancestral. Esse acesso é feito através da palavra *super* seguida dos parâmetros do construtor em questão. A Figura 3.15 mostra a invocação de um construtor da classe ancestral a partir de um construtor de uma classe herdeira.

A linguagem Java dá suporte a classes abstratas. Um objeto não pode ser instância de uma classe abstrata. Isto ocorre porque uma classe abstrata é uma classe que possui ao menos um método abstrato – um método sem implementação. Uma classe abstrata é utilizada quando o programador deseja implementar alguns métodos na classe ancestral mas também deseja que um conjunto de métodos seja obrigatoriamente implementado de forma particular por cada uma das classes herdeiras da classe abstrata. Uma classe abstrata pode possuir atributos como qualquer outra classe.

3.7.2 Herança em Eiffel

A linguagem Eiffel dá suporte ao mecanismo de atribuição dinâmica e portanto suporta polimorfismo.

Em Eiffel, a relação de herança entre classes é indicada pelo uso da palavra *inherit* entre nome da classe e o nome da classe ancestral. Eiffel suporta múltipla herança. Uma classe ancestral pode ter muitas classes herdeiras recursivamente. Uma classe não pode ser direta ou indiretamente ancestral de si própria, o que acarretaria em uma problema de referência circular.

```
class Person
{
   protected String name;
   protected int age;
   public Person(String n, int a){
       name = n;
       age = a;
   }
}
class Employee extends Person
{
   private String idCompany;
   public Employee(String n, int a, String i){
       super(n, a);
       idCompany = i;
   }
}
```

Figura 3.15 Invocação de construtor de uma classe ancestral em Java

Em Eiffel todas as características de uma classe são herdadas, sem exceção, ou seja, todas as características da classe (atributos, funções e procedimentos) são herdados.

Os construtores, embora não façam parte do comportamento de um TAD, são herdados. A classe herdeira pode escolher fazer uso de um procedimento de criação herdado como seu construtor ou pode utiliza-lo como um procedimento normal.

Ao contrário de Java, na qual a classe ancestral define o que as classes herdeiras tem acesso, em Eiffel, a classe herdeira tem todas as características da classe ancestral a sua disposição e pode alterá-las de acordo com sua necessidade. Uma classe herdeira em Eiffel pode:

- renomear uma feature herdada;
- alterar o estado de exportação (visibilidade) de uma feature herdada;
- redefinir uma feature herdada na qual a redefinição pode envolver:

  - alterar o código que implementa a rotina;
  - alterar a assinatura da feature (os argumentos formais e tipo de retorno, caso existam);
  - alterar as pré-condições e as pós-condições de uma feature[8].

- implementar uma rotina abstrata;
- tornar uma rotina herdada abstrata;
- unir[9] características.

Além disso, a classe herdeira é responsável por adaptar as características herdadas com relação ao polimorfismo.

A linguagem Eiffel dá suporte a classes abstratas. Um objeto não pode ser instância de uma classe abstrata. Isto ocorre porque uma classe abstrata é uma classe que possui ao menos um método abstrato – um método sem implementação. Uma classe abstrata é utilizada quando o programador deseja implementar alguns métodos na classe ancestral mas também deseja que um conjunto de métodos seja obrigatoriamente implementado de forma particular por cada uma das classes herdeiras da classe abstrata.

3.7.3 Herança na Virtuosi

O metamodelo da Virtuosi oferece suporte para o relacionamento de herança simples entre classes e por conseqüência, também oferece suporte à propriedade de polimorfismo.

O relacionamento de herança entre classes na Virtuosi é detalhado na Seção 4.1.3.

[8]Eiffel trabalha com o conceito de programação por contrato. Isso permite que para cada uma das características da classe, sejam definidas pré-condições, pós-condições e asserções. A discussão deste assunto não está no escopo deste trabalho.

[9]Do Inglês, ***joining.***

3.8 Interface

Uma interface apenas declara um conjunto de operações sem implementá-las. Uma classe pode implementar uma interface, ou seja, uma classe pode estabelecer um contrato com a interface garantindo que irá implementar todas as suas operações. Uma classe geralmente pode implementar mais de uma interface ao mesmo tempo.

Interfaces geralmente podem estabelecer relacionamentos de herança entre si. As regras e efeitos deste tipo de relacionamento entre interfaces são similares as definidas para herança entre classes, conforme definida na Seção 3.7.

3.8.1 Interface em Java

A linguagem Java dá suporte ao uso de interfaces.

3.8.2 Interface em Eiffel

A linguagem Eiffel não dá suporte ao uso de interfaces.

3.8.3 Interface na Virtuosi

A Virtuosi ainda não suporta o conceito de interfaces, porém, deverá dar suporte no futuro.

3.9 Tratamento de Exceções

De maneira informal, pode-se definir uma exceção como um evento anormal que interrompe a execução de um sistema computacional. As fontes de tais anormalidades podem ser de natureza da própria aplicação ou geradas pelo sistema de execução (falta de memória, má aplicação de operações aritméticas, indisponibilidade de arquivos, etc).

Um dos objetivos em um projeto de desenvolvimento de software é minimizar as situações onde o sistema deixe de responder ou apresente um erro de execução. Para alcançar este objetivo algumas linguagens de programação oferecem mecanismos para o tratamento de exceções.

Esta seção discute como Java, Eiffel e a Virtuosi implementam seus mecanismos de tratamento de exceções.

3.9.1 Tratamento de Exceções em Java

Em Java o tratamento de exceções sempre ocorre dentro do corpo de um método. Um método pode ou não ter seções de código protegidos pelo mecanismo de tratamento de exceção.

Basicamente o mecanismo de tratamento de exceção da linguagem Java funciona da seguinte forma: a partir da declaração de uma palavra reservada *try* define-se um bloco de código protegido. Para este bloco define-se quais os tipos de exceção devem ser apanhados (através de cláusulas *catch TipoDaExeção* que definem as ações que devem ser executadas em cada caso.

No advento da captura de uma exceção é possível lançar esta mesma uma exceção para que ela seja apanhada em um *catch* de escopo mais abrangente (dentro do mesmo método ou em um bloco protegido no método invocador). Caso uma exceção não seja apanhada ela é repassada de forma recursiva aos métodos invocadores e se mesmo assim ela não for apanhada a linha de execução[10] corrente é finalizada. Ou seja, uma exceção lançada deve ser apanhada por uma cláusula *catch* ou a linha de execução é encerrada.

Para um bloco protegido por um *try* é possível também definir um conjunto de ações que deve ser executado com ou sem o advento de exceções. Essas ações devem estar contidas em um bloco definido pela cláusula *finally*. A somente uma cláusula *finally* para cada *try*.

A Figura 3.16 mostra um exemplo de código fonte em Java contendo dois métodos com tratamento de exceção. Uma exceção será gerada devido a tentativa de executar uma divisão por zero. Esta exceção é capturada dentro do próprio método *methodB* e em seguida é

[10]Do inglês, *thread*

lançada novamente para o método chamador *methodA*.

```
public class Teste {
    public void methodA() {
        try {
            this.methodB();
        } catch (Exception e) {
            System.out.println("Catching the ArithmeticException
            throwed by methodB");
        }
    }
    private void methodB() {
        try {
            int i = 8/0;
        } catch (ArithmeticException e) {
            System.out.println("An ArithmeticException was throwed");
            throw e; // Throws the exception to caller
            "methodA".
        }
    }
}
```

Figura 3.16 Invocação de construtor de uma classe ancestral em Java

A linguagem Java possui toda uma hierarquia de classes para representação de exceções, porém não faz parte do escopo deste trabalho explicá-las.

3.9.2 Tratamento de Exceções em Eiffel

A linguagem Eiffel tem como um de seus princípios a programação por contrato[11]. A partir deste conceito surgem os conceitos de pré-condição, pós-condição e invariante. Citando [Meyer, 1997]: "Uma chamada de rotina é bem sucedida se ela termina sua execução em um estado que satisfaça o contrato estabelecido para aquela rotina. Caso contrário a rotina falha". E a partir desta definição, pode-se dizer que uma exceção é um evento em tempo de execução que pode fazer uma chamada de rotina falhar, ou seja, uma chamada de rotina irá falhar se e somente se uma exceção ocorrer durante sua execução e a rotina não for capaz de se recuperar da exceção.

Meyer define em [Meyer, 1997] o principio de manipulação de exceção disciplinado que prega que existem somente duas respostas legitimas a ocorrência de uma exceção dentro de uma rotina:

- tentar novamente – tentar alterar as condições que causaram a exceção e executar novamente a rotina a partir do começo;
- falhar – organizar e limpar o ambiente, encerrar a execução da rotina corrente e reportar a falha a rotina chamadora.

Para tanto Eiffel define duas palavras reservadas: *rescue* e *retry*.

A palavra *rescue* indica que a cláusula descreve como tentar recuperar-se de uma exceção. Uma vez que a cláusula *rescue* descreve operações que devem ser executadas quando o comportamento da rotina está fora do modelo definido pela pré-condições (*require*), corpo da rotina (*do*) e pós-condições (*ensure*), ela deve aparecer (quando existir) após estas três cláusulas. A Figura 3.17 ilustra uma rotina qualquer em Eiffel e sua estrutura para o tratamento de exceções.

A palavra *retry* somente pode aparecer dentro de uma cláusula *rescue*. Sua execução consiste em executar novamente o corpo da rotina a partir do começo(as inicializações não são refeitas).

[11] Do inglês, *design by contract*

```
routine is
    require
        precondition
    local
        ...Local entity declarations...
    do
        body
    ensure
        postcondition
    rescue
        rescute_clause
    end
```

Figura 3.17 Estrutura de uma rotina em Eiffel com tratamento de exceção

Caso a execução de um bloco definido pelo *rescue* chegar ao seu fim e não encontrar nenhum *retry*, a rotina atual falha e uma exceção é lançada para a rotina invocadora.

3.9.3 Tratamento de Exceções na Virtuosi

A Virtuosi ainda não dá suporte ao tratamento de exceções, mas isso é um dos trabalhos futuros que devem ser implementados.

Capítulo 4

Arquitetura da Virtuosi

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

Para auxiliar o entendimento dos conceitos explicados ao longo desse Capítulo, são utilizados trechos de código fonte de uma aplicação de software orientado a objeto escritos na linguagem Aram[1]. O código completo desta aplicação, bem como suas versões nas linguagens Java e Eiffel estão descritos no Apêndice A.

4.1 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[2]. 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

[1] Aram é a primeira linguagem de programação compatível aos conceitos de orientação a objeto definidos pelo metamodelo da Virtuosi e também foi desenvolvida dentro do Projeto Virtuosi [BORGES, 2004].

[2] A primeira versão do metamodelo Virtuosi foi proposta em [NUNES, 2001] dentro do Projeto Virtuosi. Desde então, o metamodelo Virtuosi vem sendo refinado e estendido para contemplar novos conceitos de programação orientada a objetos.

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

4.1.1 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 4.1 mostra o uso de valores literais em cada uma das situações supracitadas.

Referência a Literal

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

(1) como parâmetro formal;

(2) como parâmetro real;

(3) como origem de atribuição em um comando de atribuição de variável enumerada;

(4) como uma das possibilidades para o segundo elemento de uma comparação de valor de variável enumerada.

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

Figura 4.1 Código fonte em Aram mostrando os possíveis usos de um valor literal

Deve-se notar que uma referência a literal no papel de parâmetro real não pode sofrer atribuição – uma vez que não existe uma meta-classe que represente o comando de atribuição para referência a literal. Também não é possível declarar uma variável local do tipo referência a literal, visto que não existe uma meta-classe que represente o comando para declarar variáveis do tipo referência a literal.

A Figura 4.2 mostra o relacionamento das meta-classes que representam valores literais e referências a literal com outros componentes do metamodelo da Virtuosi.

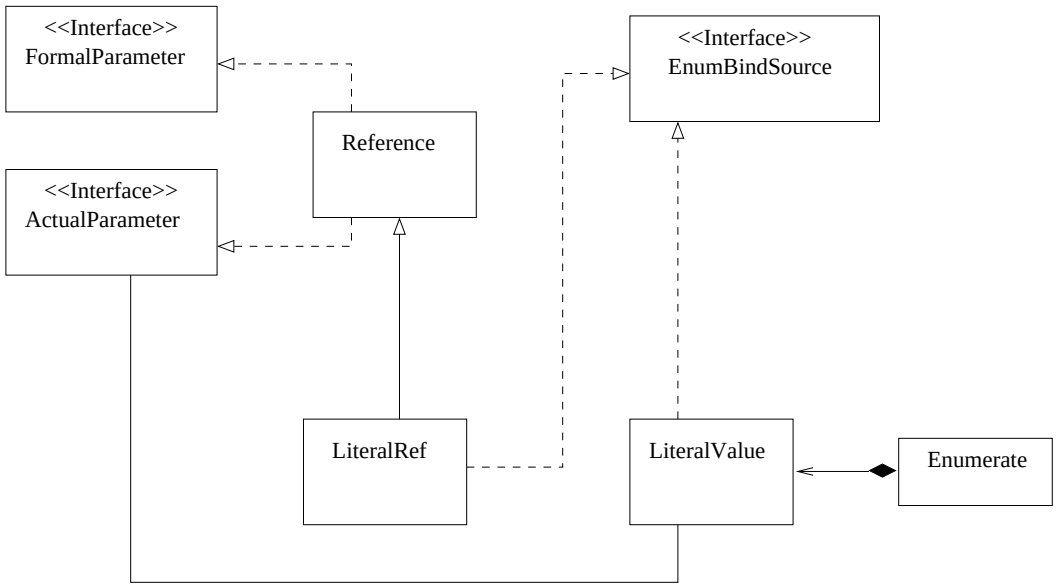

Figura 4.2 Relacionamento das meta-classes que representam valores literais e referências a literal com outros componentes do metamodelo da Virtuosi

O uso de um valor literal e de uma referência a literal é detalhado na Seção 4.1.7, especificamente na discussão sobre comandos de declaração de variáveis.

4.1.2 Bloco de Dados e Índice

Referência a Bloco de dados

Uma **referência a bloco de dados** consiste em uma referência para uma seqüência contígua de dados binários em memória. Esse tipo de referência é geralmente utilizado para a construção de classes pré-definidas (*Integer*, *Real*, *Boolean*, *Character* e *String*) utilizadas para compor outras classes. A referência a bloco de dados é discutida em detalhe na Seção 4.1.4. Um exemplo do uso de bloco de dados em uma classe que representa uma imagem é apresentado no Apêndice D.

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 4.1.7, 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 4.1.8;

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

A Figura 4.3 mostra o relacionamento da meta-classe que representa a referência a bloco de dado 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 bloco de dados existe, é preferível analisá-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

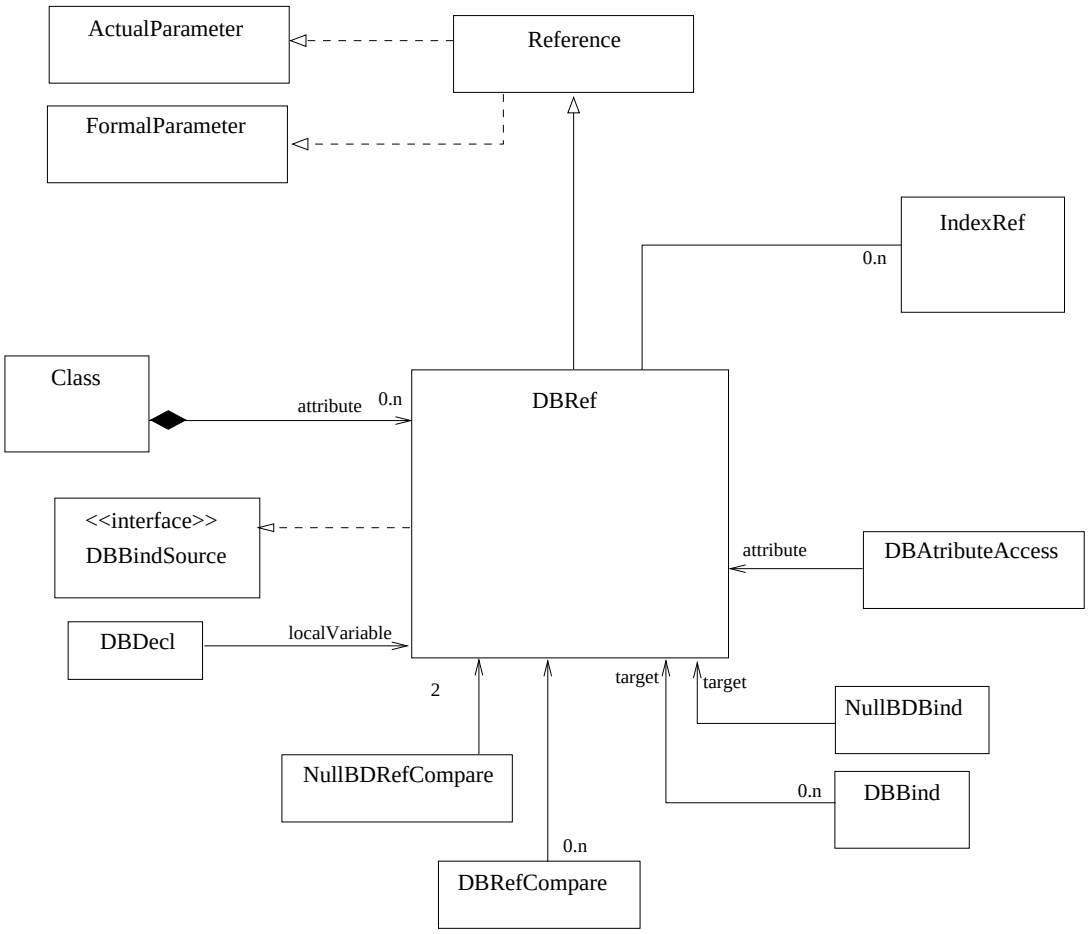

Figura 4.3 Relacionamento da meta-classe que representa a referência a bloco de dado com outros componentes do metamodelo da Virtuosi, excetuando-se os comandos e testáveis especiais

entre zero e o tamanho do bloco de dados menos um. Um índice possui comandos tais como ir para frente, ir para trás, ir para determinada posição, etc. Também possui métodos de adição, subtração, multiplicação, etc. Estes métodos permitem o índice navegar na seqüência de dados binários.

O metamodelo da Virtuosi formaliza a relação entre um índice e uma referência a bloco de dados, conforme mostra a Figura 4.4.

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;

Figura 4.4 Relação entre um índice e uma referência a bloco de dados

(2) como parâmetro real;

(3) como índice de uma referência a bloco de dados;

(4) na comparação de referência a índice;

(5) na comparação de referência a índice nula;

(6) variável local;

(7) como alvo da atribuição em um comando de atribuição de referência a índice;

(8) como 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 4.5 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 analisá-la separadamente em cada caso durante esta Seção.

4.1.3 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;

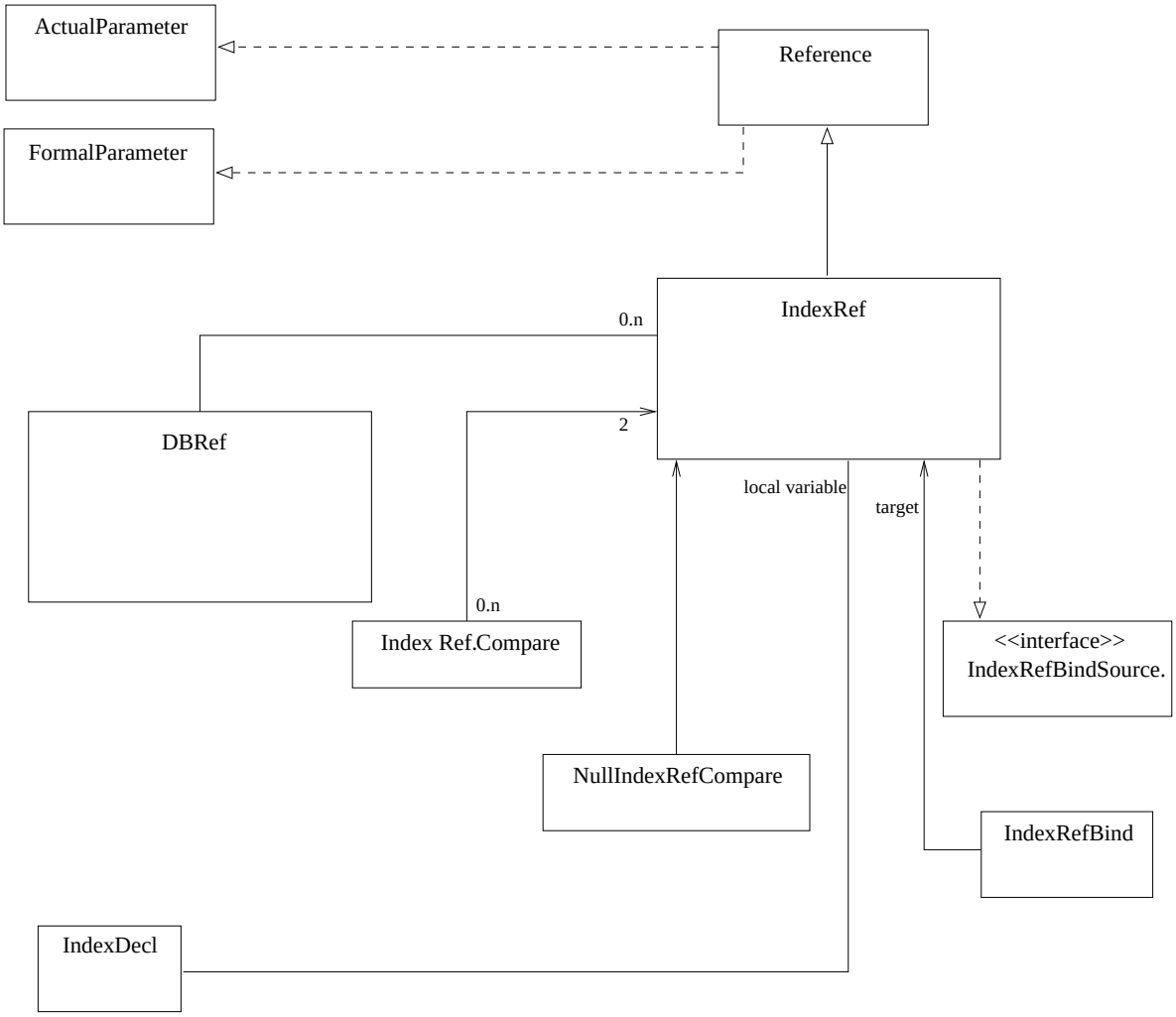

Figura 4.5 Relacionamento da meta-classe que representa a referência a índice com outros componentes do metamodelo da Virtuosi

(2) como possuidora de atributos referência a objeto em um relacionamento composição exclusiva;

(3) como possuidora de atributos referência a bloco de dados em um relacionamento composição exclusiva;

(4) como possuidora de atributos de variáveis enumeradas em um relacionamento composição exclusiva;

(5) como tipo de uma referência a objeto;

(6) como possuidora de invocáveis;

(7) como cliente da lista de exportação de um invocável;

A Figura 4.6 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.

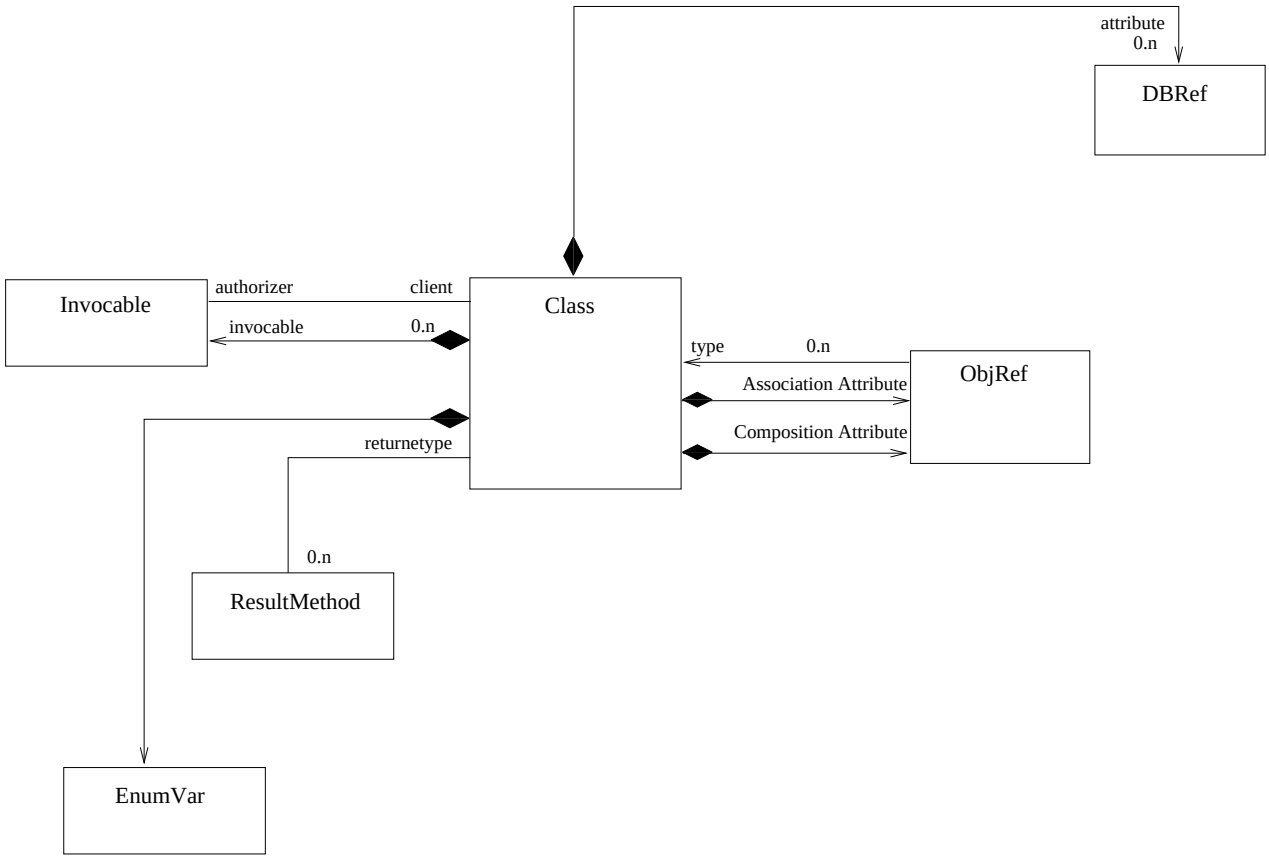

Figura 4.6 Relacionamento da meta-classe que representa uma classe de aplicação com outros componentes do metamodelo da Virtuosi

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 classes pode ser organizado como um grafo acíclico dirigido[4] 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 4.7.

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.

4.1.4 Atributos

Ao contrário de linguagens tais como C++ e Java onde são disponibilizados tipos primitivos à linguagem e respectivo sistema de execução (no caso de Java a *Java Vitual Machine*), o am-

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

[4]Um grafo acíclico dirigido pode ser entendido como uma árvore.

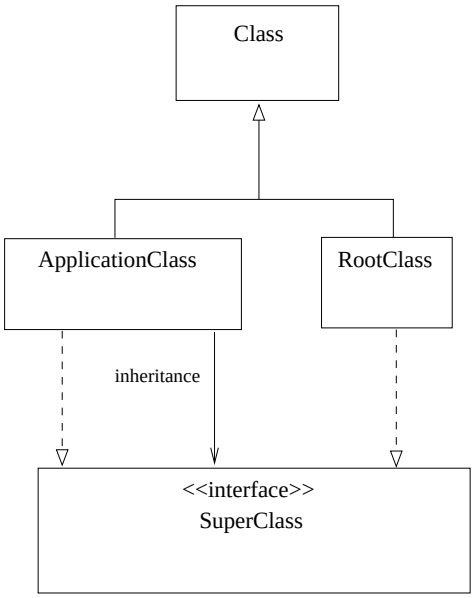

Figura 4.7 Relação de herança entre classes segundo o metamodelo da Virtuosi

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

Portanto, o ambiente Virtuosi não possui tipos primitivos exceto bloco de dados e índice que são discutidos em detalhe na Seção 4.1.2.

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 de uma classe conforme mostra a Figura 4.8.

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:

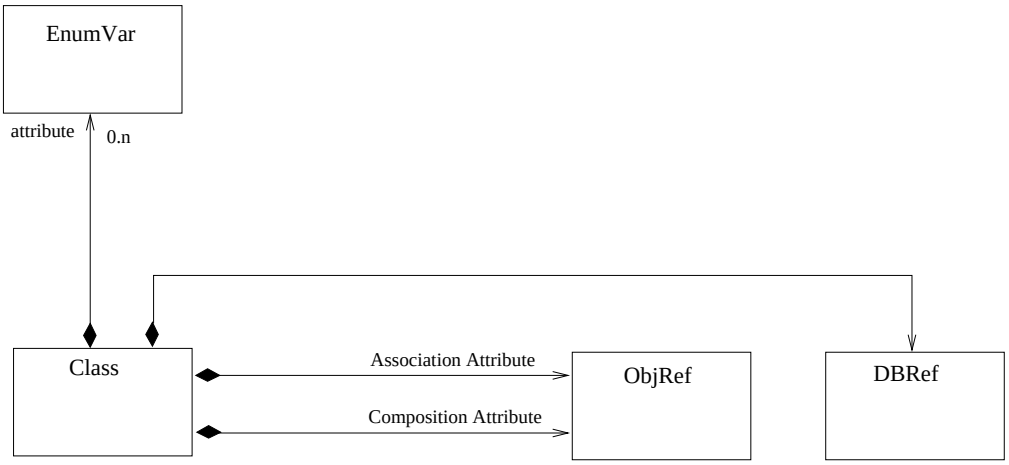

Figura 4.8 Os três tipos de atributos possíveis em uma classe segundo o metamodelo da Virtuosi

(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 4.1.7, 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 4.1.7, 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 4.1.7, 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 4.1.7, 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 analisá-la separadamente em cada caso durante esta Seção.

A Figura 4.9 mostra o relacionamento da meta-classe que representa a referência a objeto com outros componentes do metamodelo da Virtuosi.

A Figura 4.10 mostra uma classe implementada em Aram – segundo a definição do metamodelo da Virtuosi – com três atributos do tipo referência a objeto. O primeiro e o segundo atributo são instâncias de uma classe pré-definida (*String*). O terceiro atributo é instância de uma classe de aplicação, no caso `Pessoa`.

Um atributo do tipo referência a objeto pode estar associado a classe por **composição** ou **associação**.

Composição Observando a Figura 4.10 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 4.11 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.

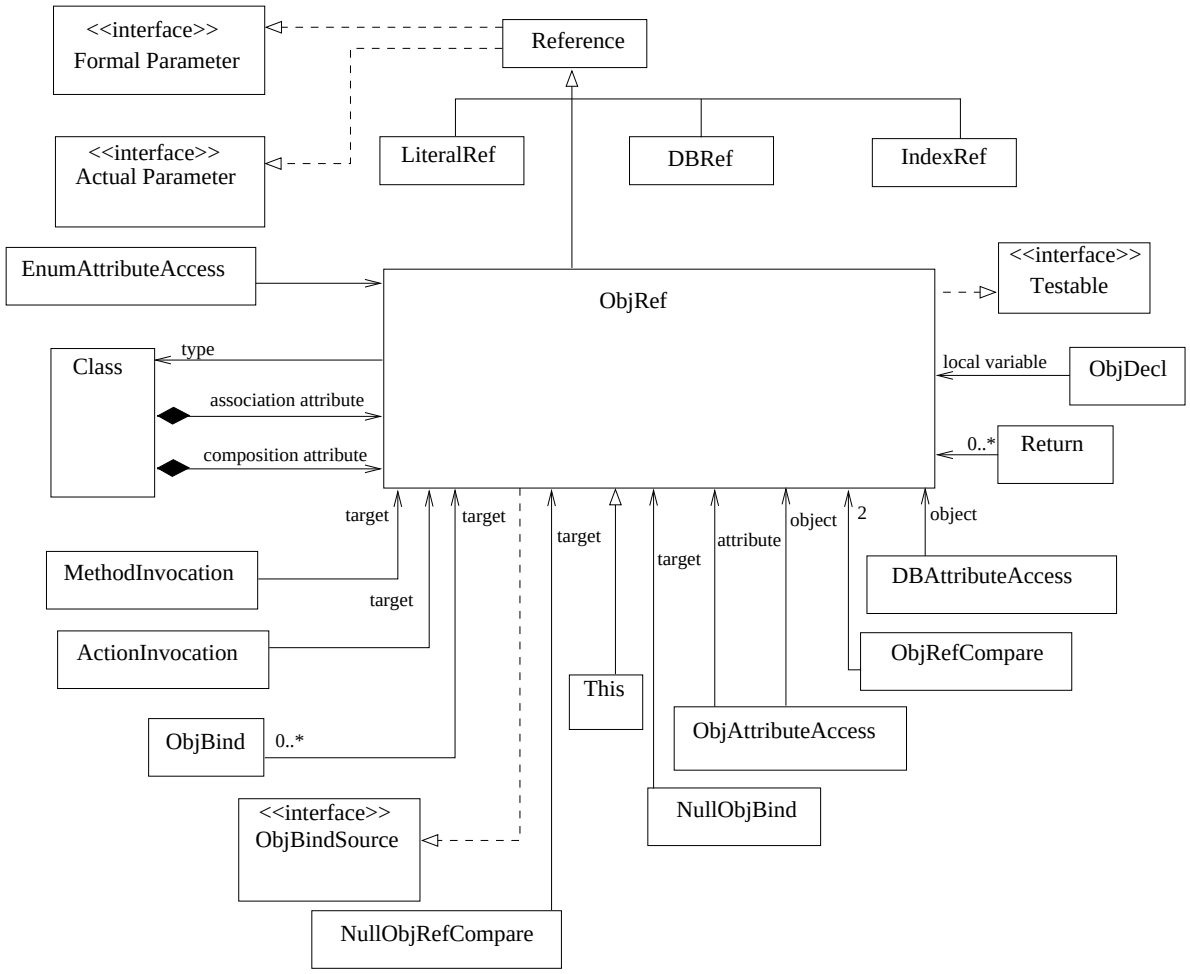

Figura 4.9 Relacionamento da meta-classe que representa a referência a objeto com outros componentes do metamodelo da Virtuosi

```
class Pessoa {
   composition String nome;      //classe pré-definida
   association String endereco;  //classe pré-definida
   association Person esposa;    //classe de aplicação
}
```

Figura 4.10 Atributos em uma classe segundo o metamodelo da Virtuosi

Associação Observando novamente a Figura 4.10 nota-se que o segundo atributo – uma *String* de nome *address* – possui como parte de sua declaração a palavra *association*. A

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

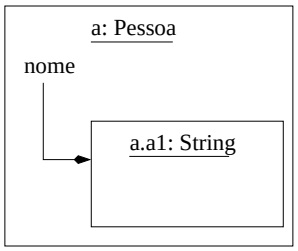

Figura 4.11 Código fonte em Aram e diagrama de objeto de uma relação de composição entre uma classe e um atributo

palavra *association* indica um relacionamento de **associação** entre uma classe e um atributo. A Figura 4.12 mostra um código fonte com uma relação de associação entre classe e atributo e ilustra a semântica correspondente utilizando os objetos envolvidos. Em uma relação de associação o objeto representado pelo atributo não faz parte do objeto representado pela classe, simplesmente está associado a ele. Como conseqüência do relacionamento de associação um objeto associado pode ser referenciado por qualquer outro objeto.

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

Figura 4.12 Código fonte em Aram e diagrama de objeto de uma relação de associação entre uma classe e um atributo

A Figura 4.13 mostra como o metamodelo da Virtuosi formaliza as relações de composição e associação entre uma classe e seus atributos.

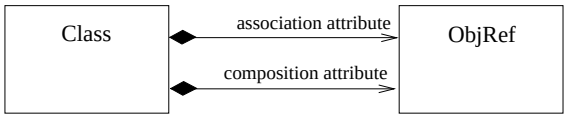

Figura 4.13 As duas maneiras em que uma referência a objeto representa o papel de atributo segundo o metamodelo da Virtuosi

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 4.14 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;
```

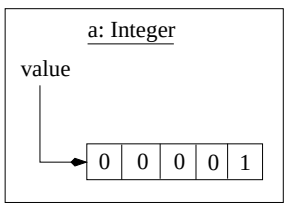

Figura 4.14 Um exemplo de atributo do tipo bloco de dados e uma representação de um objeto correspondente – código fonte em Aram

A manipulação de blocos de dados é o mais baixo nível em que a MVV opera e é feita sempre através de comandos e testáveis de sistema. O comandos e testáveis de sistema são definidos na Seção 4.1.8.

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 4.15 mostra o código fonte de uma classe que possui um atributo 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 Dice   {
   enum {1, 2, 3, 4, 5, 6 }    value = 1 ;
```

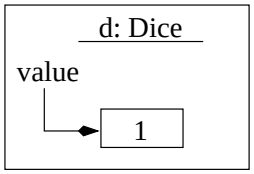

Figura 4.15 Um exemplo de atributo do tipo enumerado e uma representação de um objeto correspondente – código fonte em Aram

4.1.5 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 a literal.

A Figura 4.16 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.

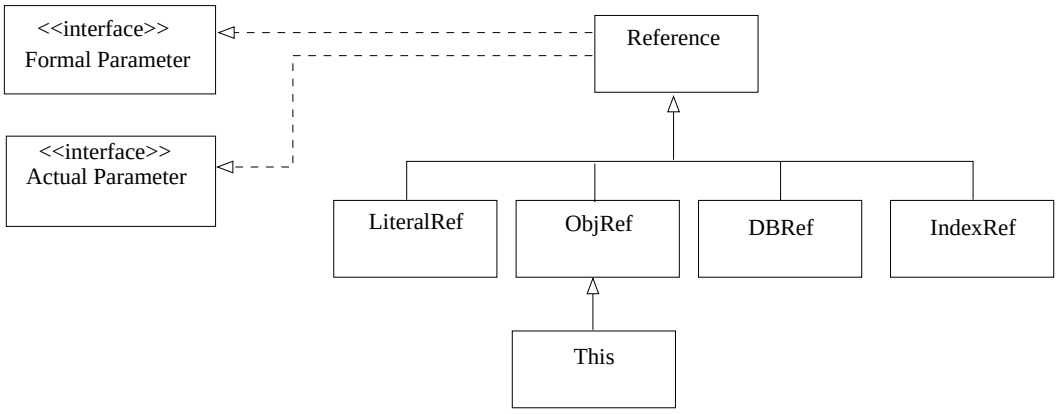

Figura 4.16 Relacionamento entre as meta-classe que definem os tipos de referência na Virtuosi

Observando-se a Figura 4.16 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.

4.1.6 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 4.17.

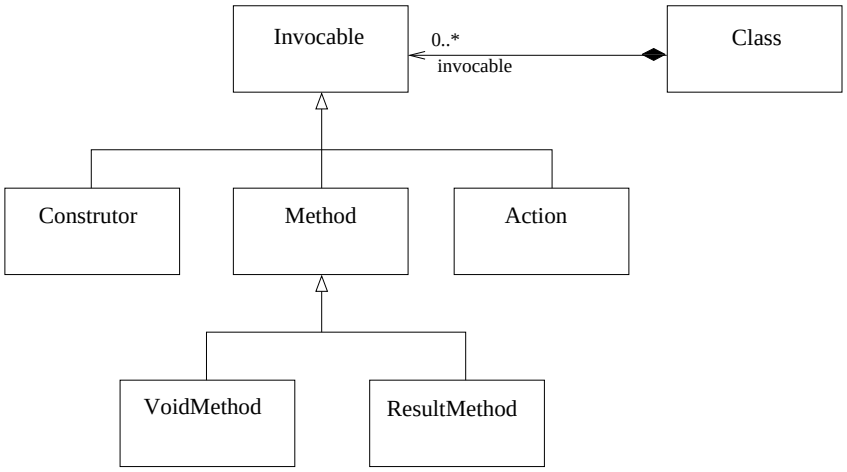

Figura 4.17 Relação entre uma classe e as possíveis implementações de suas operações

Um método, um construtor ou uma ação podem sofrer invocação e, por isso, o metamodelo da Virtuosi os formaliza como **invocáveis**[5].

A Figura 4.18 mostra o relacionamento da meta-classe Invocável com outros componentes do metamodelo da Virtuosi.

A meta-classe que representa um invocável – independente do seu tipo (construtor, método ou ação) – se relaciona através de associações com outros componentes do metamodelo da Virtuosi nas seguintes situações:

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

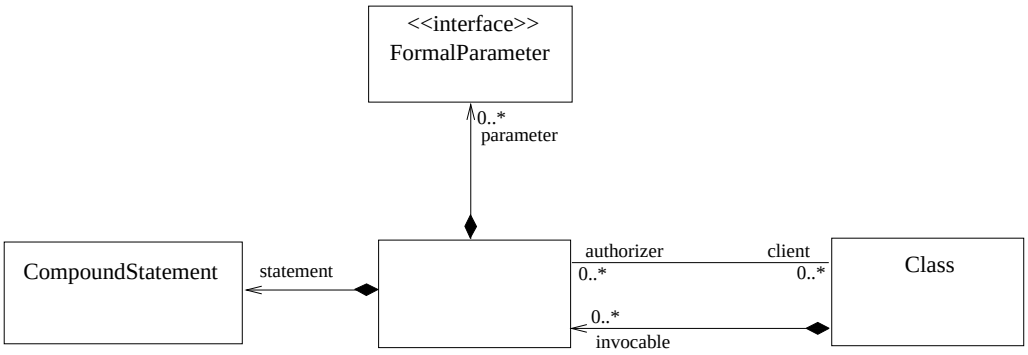

Figura 4.18 Relacionamento da meta-classe Invocável com outros componentes do metamodelo da Virtuosi

(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 no qual o parâmetro tem o papel de **parâmetro formal**[6], 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, no qual um parâmetro é uma referência a um objeto já existente, tendo assim, o papel de **parâmetro real**[7].

O conjunto de parâmetros formais de um invocável pode ser vazio ou composto de referências. Qualquer tipo de referência pode ser um parâmetro formal, inclusive uma referência a literal[8] utilizada para receber os parâmetros reais do tipo valor literal.

[6]Do inglês ***formal parameter***.

[7]Do inglês ***actual parameter***.

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

A Figura 4.19 mostra um exemplo de uma classe de aplicação `Taxi` com dois métodos, um com uma lista de parâmetros vazia (`sairPassageiro`) e outro com uma lista contendo um parâmetro formal do tipo referência a objeto (`entrarPassageiro`). A Figura mostra também a classe de aplicação `Principal`, que invoca os dois métodos da classe de aplicação `Taxi`.

O metamodelo da Virtuosi formaliza a relação entre um invocável e seus parâmetros, conforme mostra a Figura 4.18.

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 4.20. 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 4.21.

Construtor Um construtor não faz parte das operações definidas por um TAD pois não interfere no comportamento dos objetos representados pelo TAD. Porém, tem fundamental

comando de atribuição para referência a literal. Portanto, uma referência a literal sempre tem seu valor literal atribuído por um comando de invocação de invocável

```
class Principal
{
   constructor iniciar() exports { all }
   {
       Taxi corsa = Taxi.instanciar();
       ...
       Boolean entrou = corsa.entrarPassageiro(andrea);
       ...
       corsa.sairPassageiro();
       ...
   }
   ...
}
class Taxi {
   ...

   // método sem parâmetros
   method void sairPassageiro( ) exports { Principal }
   {
       ...
   }
   // método com parâmetros
   method Boolean entrarPassageiro( Pessoa p ) exports { Principal }
   {
       ...
   }
}
```

Figura 4.19 Código fonte em Aram contendo um método sem parâmetros e um método com parâmetros

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.

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

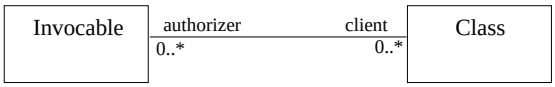

Figura 4.21 Relação de um Invocável com sua lista de exportação

Método Um método é a maneira mais comum de implementar uma operação definida por um TAD. Existem dois tipos de métodos, a saber: **método sem retorno** – muitas vezes chamado de procedimento – e **método com retorno** – muitas vezes chamado função – conforme formalizado pelo metamodelo da Virtuosi e mostrado na Figura 4.22.

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 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 4.23 mostra um exemplo de código fonte em Aram

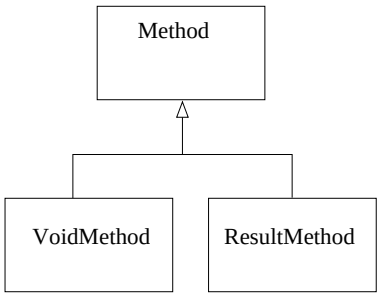

Figura 4.22 Métodos com retorno e métodos sem retorno

contendo um método sem retorno e um método com retorno. O comando retorno é um comando simples e é discutido na Seção 4.1.7.

```
class Taxi {
   ...

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

Figura 4.23 Diferenciação entre métodos com retorno e métodos sem 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 condicional. Em outras palavras, o retorno de uma ação permite ao comando de desvio condicional decidir qual dentre duas seqüências de comandos deve ser interpretada.

Essa abordagem é bem diferente da abordagem normalmente utilizada por linguagens de programação nas quais um desvio condicional sempre depende da avaliação de uma expressão que retorna um valor verdadeiro ou falso. Isto permite, por exemplo, que qualquer classe defina uma ou mais ações que podem ser utilizadas para a tomada de decisão em um desvio condicional. Além disso, toda classe pode definir uma ação chamada `default` ou ação padrão. Esta ação padrão não precisa ser explicitamente chamada, ou seja, se um comando de invocação tiver como teste simplesmente uma referência a objeto isto significa que a ação invocada deve ser à padrão permitindo por exemplo, que o programador possa utilizar um objeto da classe pré-definida *Boolean* da forma tradicionalmente utilizada.

Uma ação não retorna uma referência a objeto. Diferente de um método com retorno ou um construtor – no qual uma referência é retornada – o retorno de uma ação é um comando simples chamado: **comando resultado de teste**. O comando de desvio e o comando resultado de teste são detalhados na Seção 4.1.7.

A Figura 4.24 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 4.24 Código fonte em Aram contendo uma declaração de uma ação

4.1.7 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**[9] pode ser uma atribuição, uma invocação de operação, um desvio condicional, conforme detalhado no restante dessa Seção. Um **comando composto**[10] 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 4.25.

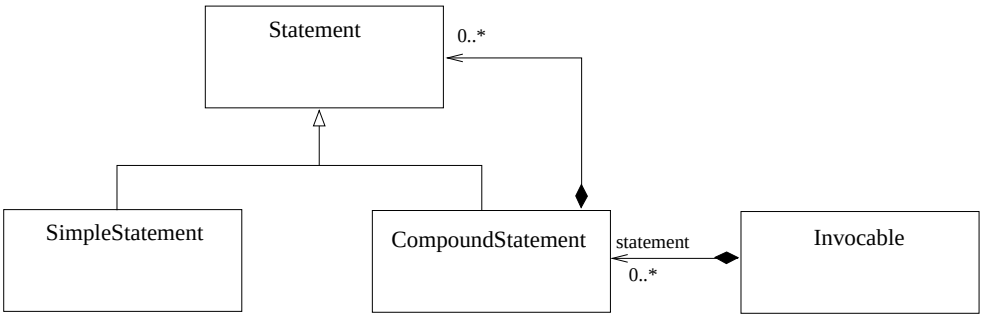

Figura 4.25 Relacionamento de herança entre as meta-classes comando, comando simples e comando composto

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

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 4.26 mostra a hierarquia de meta-classes que determina os comandos simples disponíveis para a declaração de variáveis locais, segundo o metamodelo da Virtuosi.

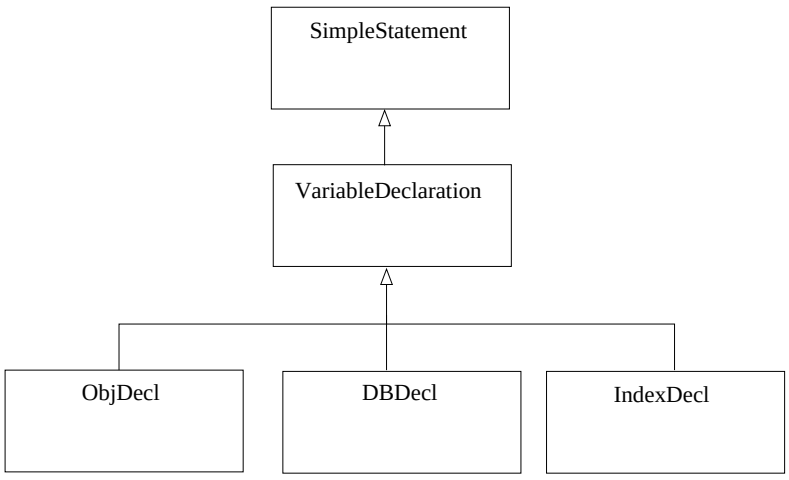

Figura 4.26 Tipos de declarações para variáveis locais

Conforme mostra a Figura 4.26, é possível declarar variáveis locais do tipo:

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

- referência a índice.

Observando novamente a Figura 4.26 nota-se que não existe declaração de variável local do tipo referência a literal. Isto ocorre porque uma referência a literal somente é permitida como parâmetro de um invocável, ou seja, no momento da invocação o que é passado como parâmetro é um valor literal, e não uma referência a literal. A Figura 4.27 mostra uma invocação de um método passando um valor literal e a declaração desse mesmo método contendo um parâmetro formal do tipo referência a literal.

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

Figura 4.27 Invocação de método passando como parâmetro um valor literal e a declaração do método invocado – código fonte em Aram

Declaração de Variáveis do Tipo Referência a Objeto A Figura 4.28 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.

```
class Person {
   constructor make() exports all
   {
       // declaração de variável local inicialmente nula.
       String name;
       ...
   }
   ...
}
```

Figura 4.28 Declaração de uma variável local do tipo referência a objeto – código fonte em Aram

Declaração de Variáveis do Tipo Referência a Bloco de Dados A Figura 4.29 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.

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

Figura 4.29 Declaração de uma variável local do tipo referência a bloco de dados – código fonte em Aram

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 4.30 mostra a declaração de uma variável local do tipo referência a índice. Após a interpretação desse comando a referência não possui um alvo, ou seja, não está apontando alguma seqüência de dados binários em memória, ou seja, é uma referência nula.

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

Figura 4.30 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 caracter '=' chamado de caracter de atribuição. O que estiver à esquerda do caracter de atribuição é chamado **alvo da atribuição**, e o que estiver à direita do caracter de atribuição é chamado **origem da atribuição**.

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

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

Atribuição de Referência a Objeto No caso da atribuição de referência a objeto o alvo da atribuição sempre é uma referência a objeto, enquanto que a origem da atribuição pode ser:

- uma referência a objeto;
- um **acesso a atributo objeto**, ou seja, um acesso – somente de leitura – a um atributo do tipo referência a objeto de outro objeto instância da mesma classe que implementa o invocável no qual 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 4.31 mostra o relacionamento das meta-classes que representam os dois comandos de atribuição de referência a objeto com outros componentes do metamodelo da Virtuosi.

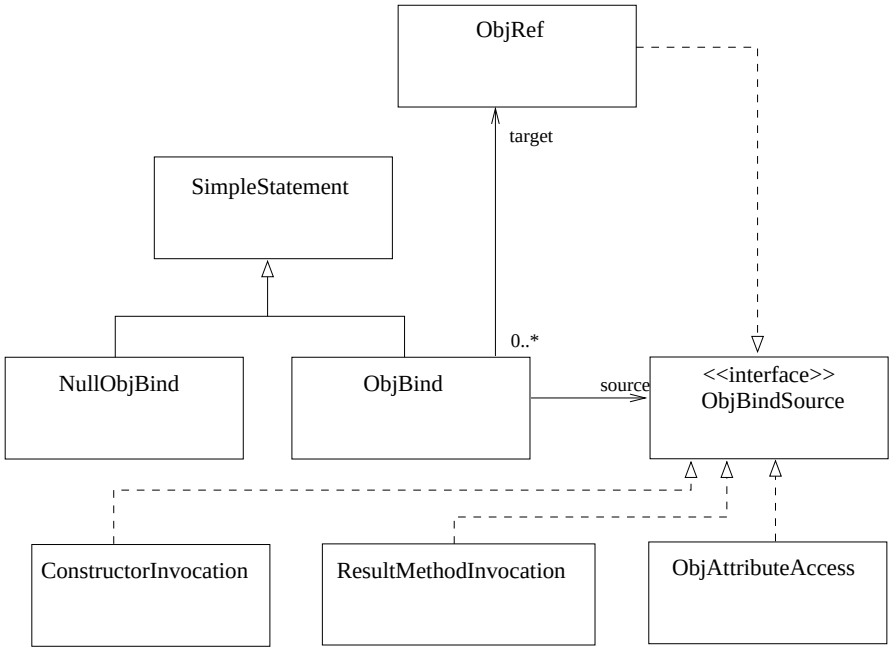

Figura 4.31 Relacionamento das meta-classes que representam os comandos de atribuição de referência a objeto com outros componentes do metamodelo da Virtuosi

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 no qual 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 4.32 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.

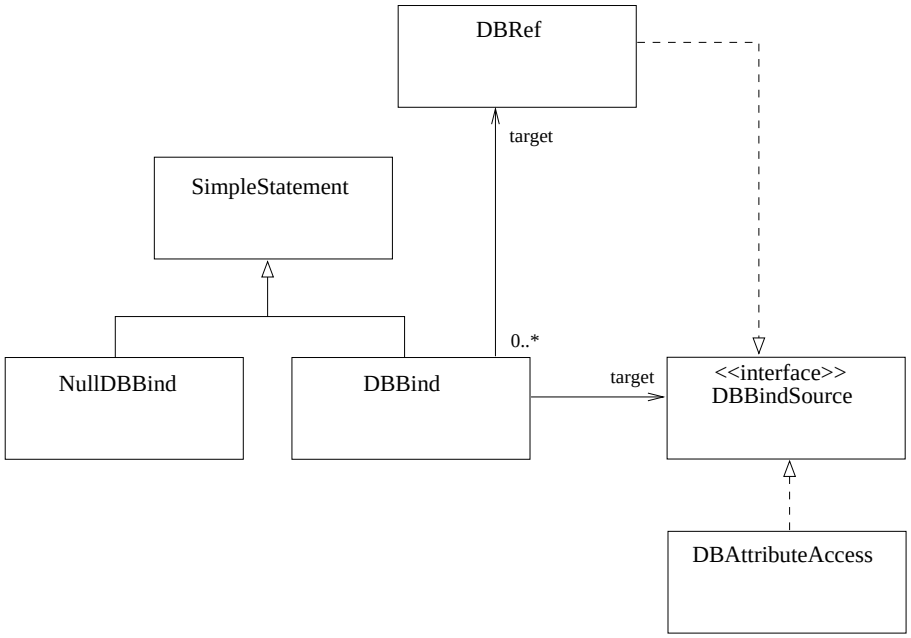

Figura 4.32 Relacionamento das meta-classes que representam os dois comandos de atribuição de referência a bloco de dados com outros componentes do metamodelo da Virtuosi

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 4.33 mostra o 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;

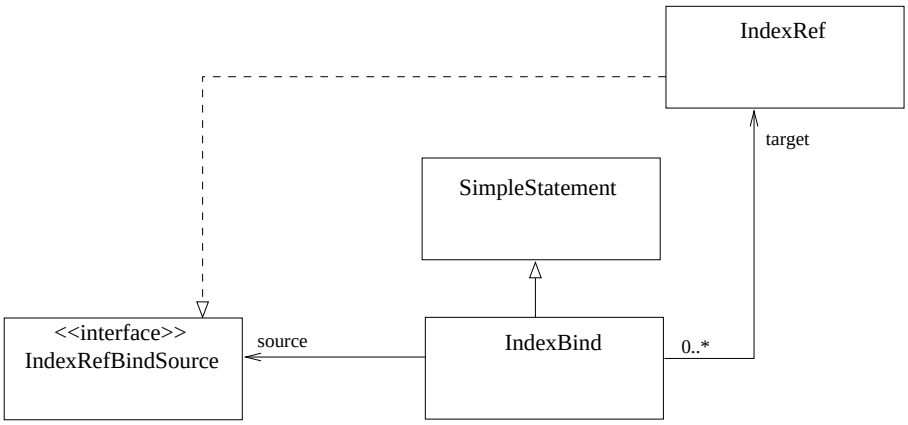

Figura 4.33 Relacionamento da meta-classe comando de atribuição de referência a índice com outros componentes do metamodelo da Virtuosi

- uma referência a literal;
- um acesso a atributo variável enumerada, 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 no qual 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 4.34 mostra o 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 4.35.

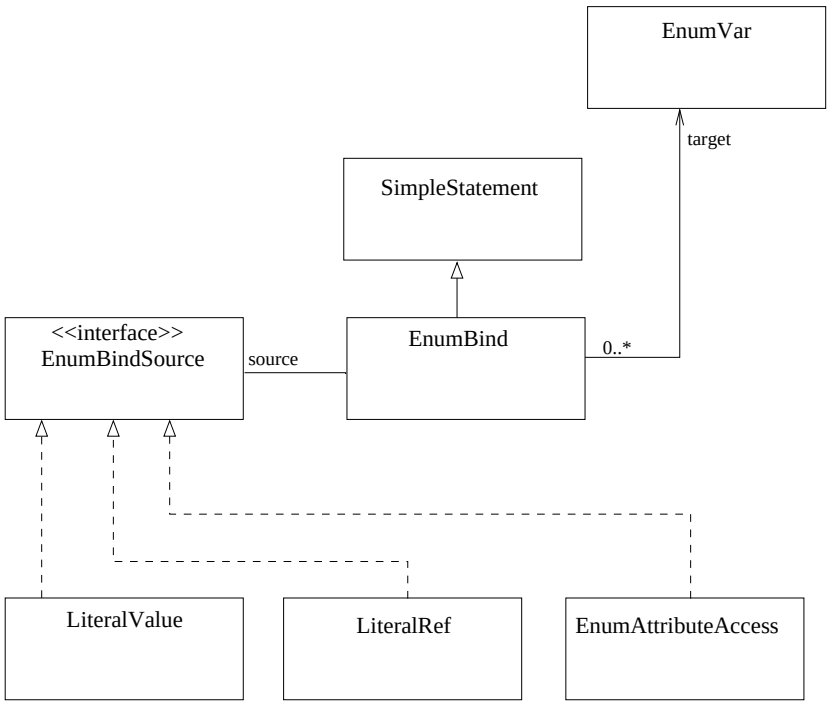

Figura 4.34 Relacionamento da meta-classe comando de atribuição variável enumerada com outros componentes do metamodelo da Virtuosi

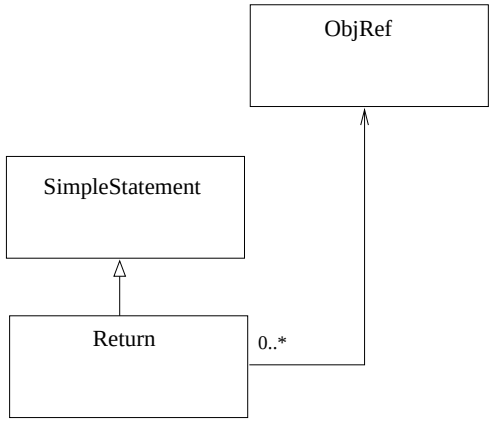

Figura 4.35 Relacionamento entre um comando de retorno e uma referência a objeto

Retorno de Ação

Conforme explicado na Seção 4.1.6, uma ação não retorna uma referência, ao invés disso, tem como retorno um comando *desvie* ou um comando *execute*. Um comando resultado de teste

é o comando retornado por todos os componentes do metamodelo que são testáveis. Tanto o comando resultado de teste quanto os comandos testáveis são detalhados na discussão do comando de desvio condicional nessa Seção.

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 4.36 mostra o relacionamento da meta-classe que representa um comando de invocação com sua hierarquia e outros componentes do metamodelo da Virtuosi.

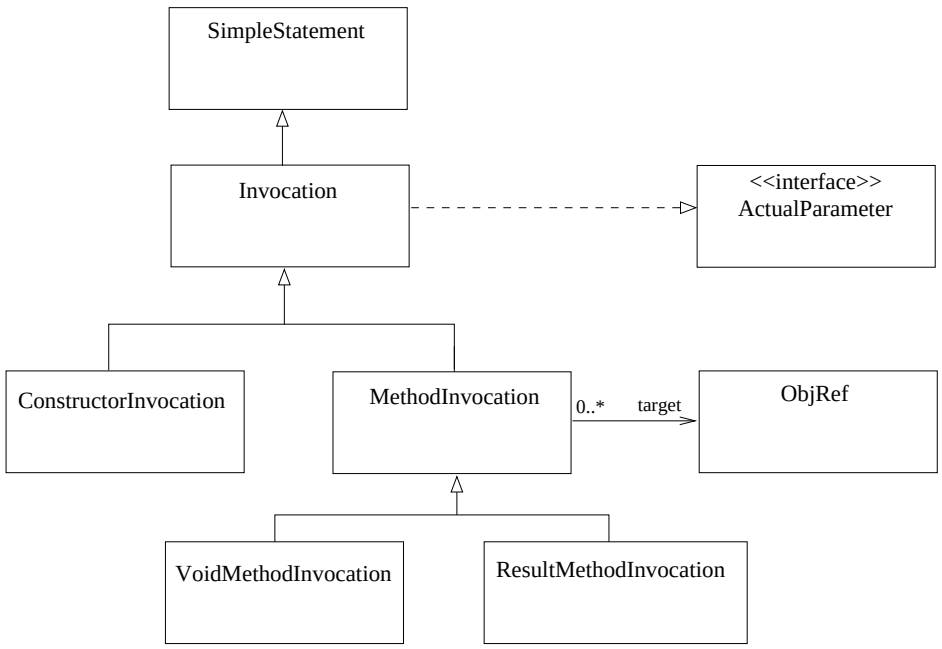

Figura 4.36 Relacionamento da meta-classe que representa um comando de invocação com sua hierarquia e outros componentes do metamodelo da Virtuosi

O metamodelo da Virtuosi formaliza a relação entre os comandos de invocação e os respectivos invocáveis, conforme mostra a Figura 4.37.

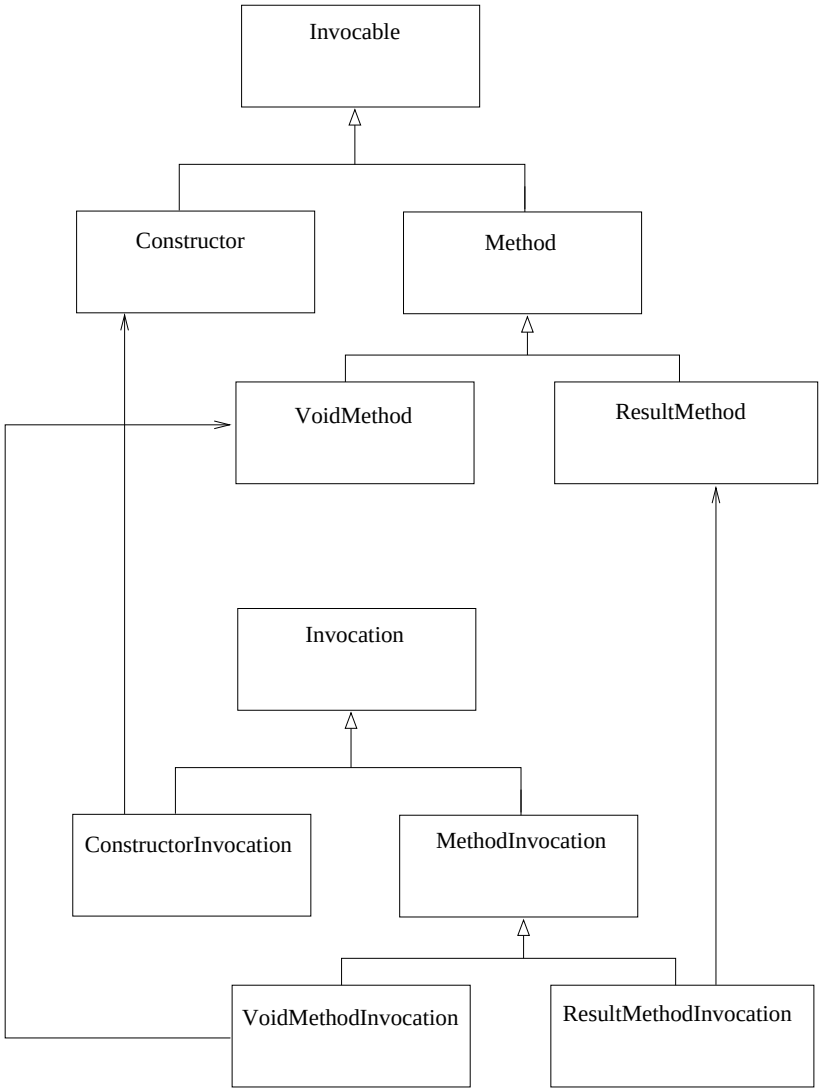

Figura 4.37 Relação entre os comandos de invocação e os invocáveis

Conforme a Figura 4.37 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.

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 explicitamente no código fonte, ele sempre é utilizado em conjunto de um comando de desvio condicional.

Desvio Condicional Um desvio condicional tem duas seqüências de comandos que podem ser interpretados (exclusivamente). A primeira seqüência é chamada de **caminho destino** e a segunda é chamada **caminho alternativo**.

Testável Na Virtuosi, para se interpretar um comando de desvio, não é necessário avaliar o valor de um objeto da classe *Boolean*, embora isso possa ser feito.

Um comando de desvio condicional está associado a algo que pode ser testado, um **testável**[11]. Um testável, quando interpretado, retorna um dentre dois comandos possíveis, a saber:

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

- 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 4.38.

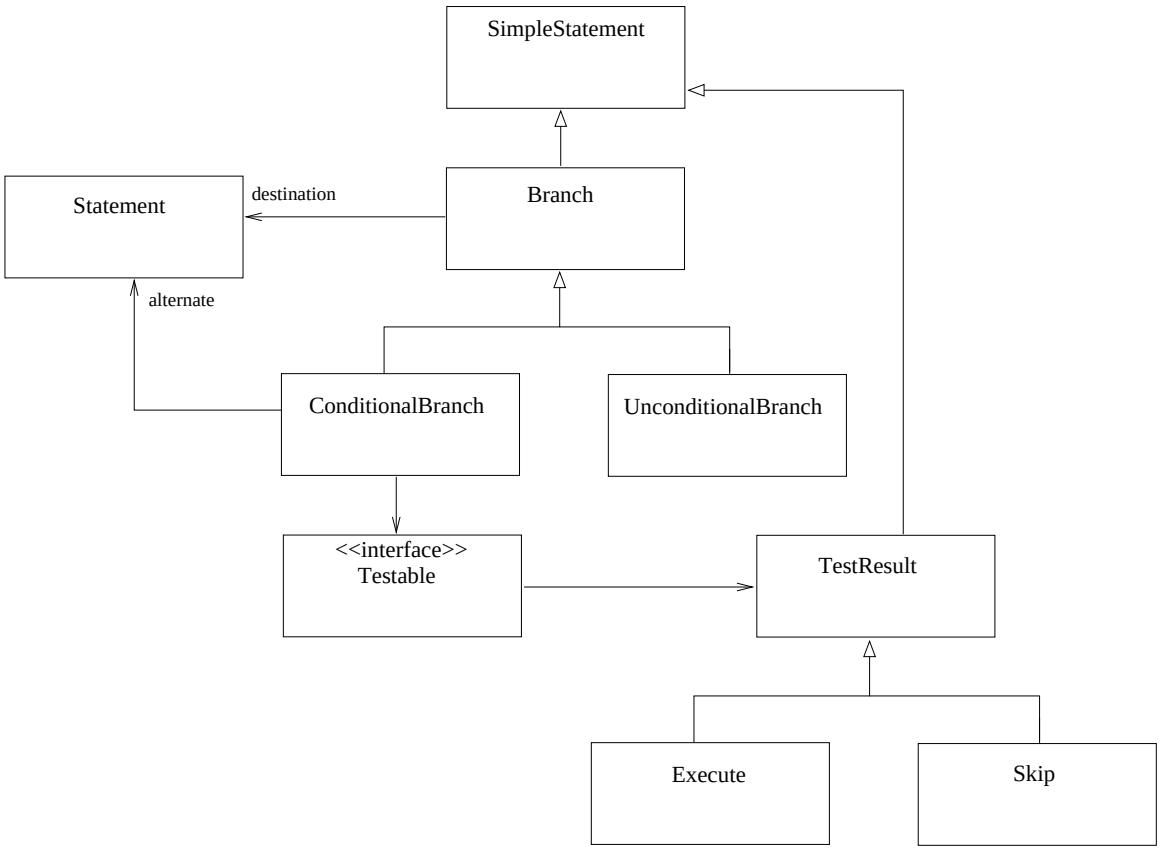

Figura 4.38 Comandos de desvio e como se relacionam com as seqüências de comandos

Entre os componentes definidos como testáveis pelo metamodelo da Virtuosi, está a invocação de uma ação. Uma ação, embora seja um componente invocável, não possui um comando de invocação correspondente. Uma ação também é invocada a partir de uma referência a objeto, contudo, sua utilização *sempre* está associada a um comando de desvio condicional, ou seja, a invocação de uma ação é realizada a partir de uma referência a objeto somente quando esta invocação for o elemento testável de um comando de desvio condicional.

Deve-se notar a diferença entre uma ação e uma invocação de ação. Uma invocação de ação causa a interpretação da seqüência de comandos definidos por uma ação. Um comando de desvio condicional interpreta um testável, para que este retorne um resultado de teste. O testável em questão *pode* ser uma invocação de ação. A invocação de ação interpreta cada um dos comandos definidos pela ação até encontrar um comando resultado de teste. Esse resultado de teste é utilizado pelo comando de desvio condicional.

O metamodelo da Virtuosi formaliza a relação de um desvio condicional, um testável, uma invocação de ação e uma ação, conforme mostra a Figura 4.39.

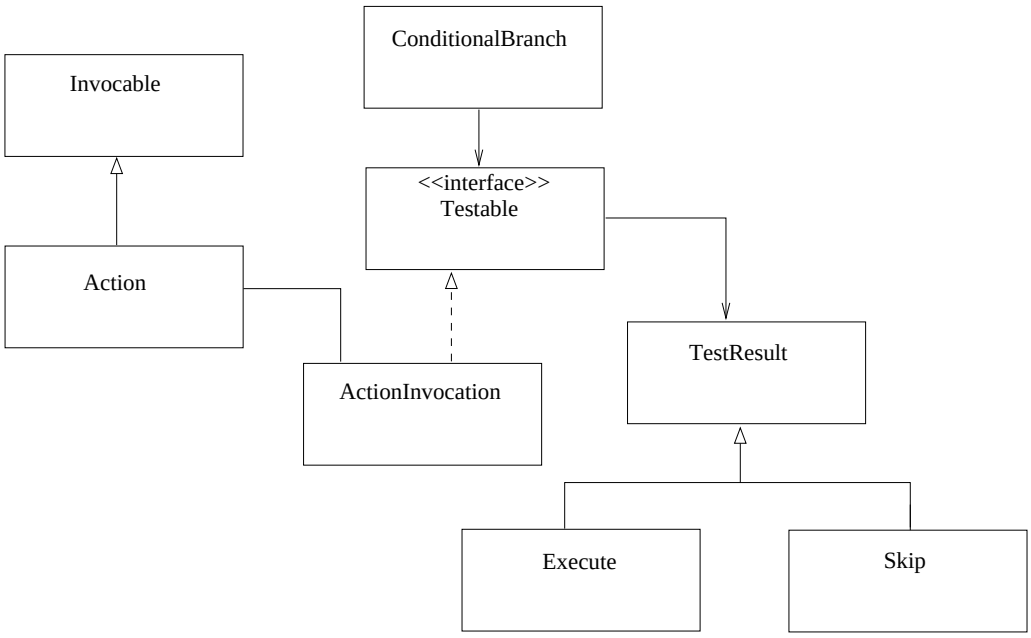

Figura 4.39 Relação de um desvio condicional, um testável, uma invocação de ação e uma ação

A Figura 4.40 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*.

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

Figura 4.40 Desvio condicional com um testável que é uma invocação de uma ação – código fonte em Aram

A Figura 4.41 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 nas quais um desvio condicional sempre depende da avaliação de uma expressão que retorna um valor verdadeiro ou falso. Isto permite, por exemplo, que qualquer classe defina uma ou mais ações que podem ser utilizadas para a tomada de decisão em um desvio condicional. Além disso, toda classe pode definir uma ação chamada `default` ou ação padrão. Esta ação padrão não precisa ser explicitamente chamada, ou seja, se um comando de invocação tiver como teste simplesmente uma referência a objeto, isto significa que a ação invocada deve ser à padrão. A Figura 4.42 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

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


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

Figura 4.41 Desvio condicional com um testável que é uma comparação de valor entre uma variável enumerada e um valor literal – código fonte em Aram

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 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;

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

- uma comparação entre uma referência a bloco de dados e uma referência nula a bloco de dados, chamada **comparação de referência nula a bloco de dados** – utilizada para verificar se uma referência é nula;
- uma comparação entre uma referência a índice e uma referência nula, chamada **comparação de referência nula a índice** – utilizada para verificar se uma referência é nula;
- uma comparação de **valor** entre uma variável enumerada e um segundo elemento do tipo variável enumerada, valor literal, referência a literal ou ainda um acesso a atributo variável enumerada, chamada **comparação de valor de variável enumerada** – utilizada para comparar valores;

O metamodelo da Virtuosi formaliza todos elementos testáveis que podem ser utilizados por um desvio condicional, conforme mostra a Figura 4.43.

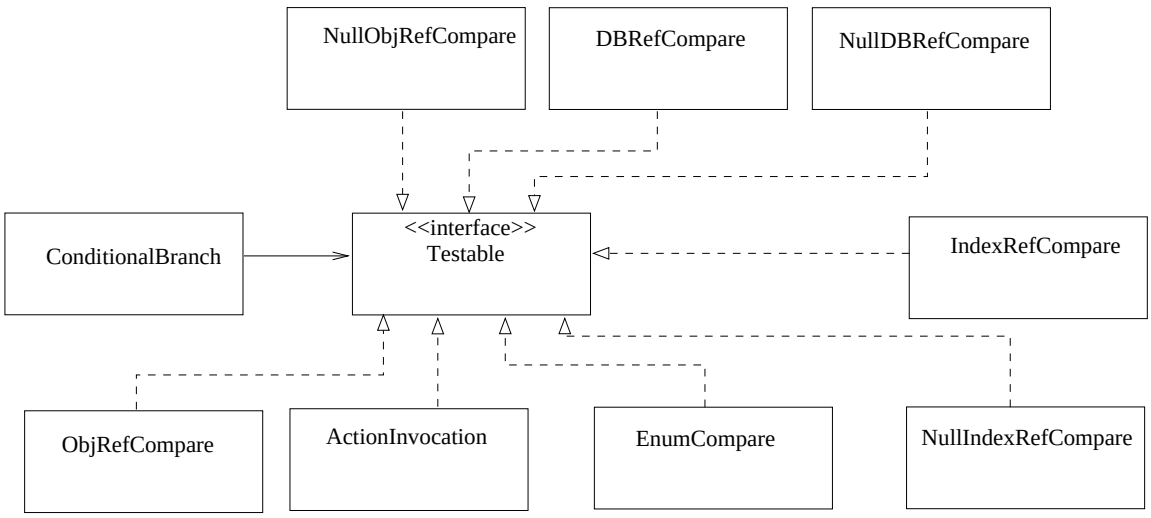

Figura 4.43 Relação de um desvio condicional com todos os testáveis possíveis

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 4.44 mostra.

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

Figura 4.44 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.

4.1.8 Comandos de Sistema e Testáveis de Sistema

Conforme citado na Seção 4.1.4, o ambiente Virtuosi disponibiliza uma biblioteca de classes pré-definidas (*Integer*, *Real*, *Boolean*, *Character* e *String*) para a construção de novas classes chamadas de classes de aplicação.

Contudo, o ambiente Virtuosi também disponibiliza comandos simples e testáveis de sistema para a construção de classes que utilizem referências a bloco de dados. As próprias classes pré-definidas disponibilizadas pelo ambiente Virtuosi são construídas com estes co-

mandos e testáveis de sistema.

Para cada uma das classes pré-definidas existe um conjunto definido de comandos de sistema e um conjunto definido de testáveis de sistema. A lista completa dos comandos e testáveis de sistema é apresentada no Apêndice C.

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 4.45.

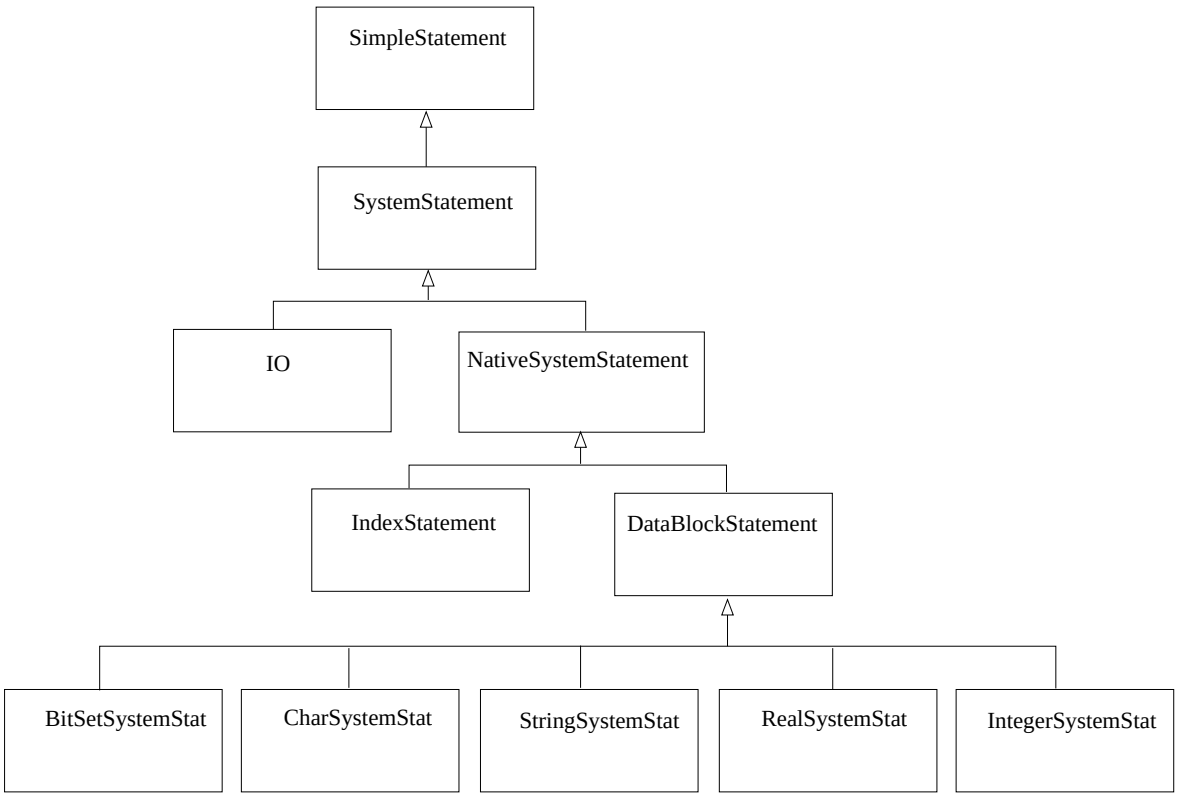

Figura 4.45 Comandos de sistema básicos

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 4.46 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 bloco de dados com a representação de valores inteiros podem receber como parâmetros um valor literal indicando o tamanho do bloco de dados que deve ser criado.

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

Testáveis de Sistema

Os testáveis disponibilizados pelo ambiente Virtuosi são chamados de testáveis de sistema. Com o intuito de organizar a hierarquia das meta-classes que representam estes testáveis, o metamodelo da Virtuosi define uma hierarquia de testáveis de sistema, conforme mostra a Figura 4.47.

Observa-se que abaixo da meta-classe representando os testáveis de sistema, existem 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;

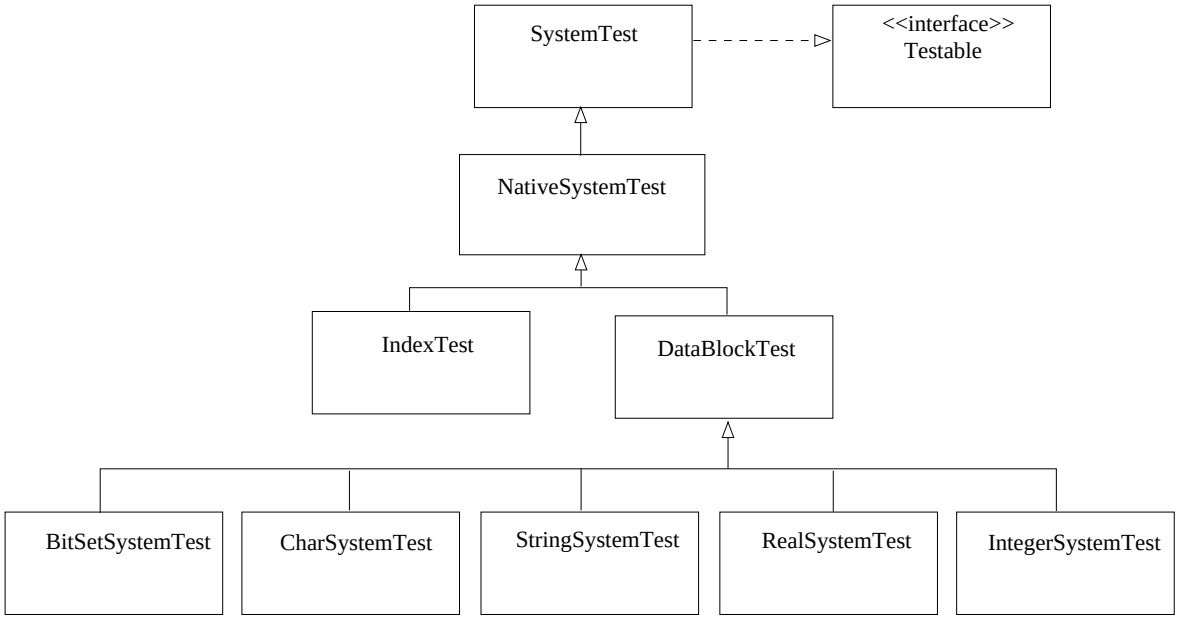

Figura 4.47 Testáveis de sistema disponibilizados pelo sistema

- testáveis de sistema para valores inteiros;
- testáveis de sistema para valores reais;
- testáveis de sistema para valores caracter;
- testáveis de sistema para valores conjunto de caracter;

A Figura 4.48 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.

4.2 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

```
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 4.48 Uso de testáveis especiais nos comandos de desvio utilizados na construção de uma classe pré-definida *Integer* – código fonte em Aram

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 nas quais é implementada. Essa estratégia permite que uma mesma aplicação seja interpretada em diferentes plataformas. Uma máquina virtual, como o próprio nome diz, é um programa computacional capaz de interpretar outros programas. Para tanto, os programas devem ser escritos em termos do conjunto de operações e dos tipos de dado suportados pela máquina virtual.

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

A tecnologia Java é o principal exemplo atual de utilização da estratégia de máquinas virtuais. Java tornou-se uma plataforma padrão de desenvolvimento e execução de aplicações portáveis, principalmente em sistemas embarcados[12], em *applets* na Internet ou ainda em aplicações multi-plataforma. A portabilidade de Java é realizada através da compilação de um programa fonte escrito em Java para uma seqüência de instruções da máquina virtual Java. Essa seqüência de instruções é uma representação intermediária do programa fonte, e, no caso de Java, é chamada de *bytecodes* do Java. Os bytecodes do Java são independentes da arquitetura computacional na qual 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[13] 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

[12]Do inglês, **embedded systems**

[13]Embora a palavra grafo seja mais apropriada, por questões históricas foi mantida a palavra árvore.

são instâncias das meta-classes definidas pelo metamodelo da Virtuosi e as ligações entre os nós são as relações entre tais meta-classes.

O uso da representação intermediária no formato de árvore de programa adotada neste trabalho justifica-se pelos seguintes fatos:

- permite depurar o metamodelo da Virtuosi. Um dos objetivos deste trabalho;
- permite extender facilmente o metamodelo (por exemplo: controle de concorrência e migração de objetos);
- mantém, ou até mesmo aumenta, os seguintes benefícios da representação no formato de *bytecode*:
  - possibilidade de reflexão computacional. O formato de árvore facilita os processos de inspeção, alteração dinâmica e introdução de aspectos [Kiczales et al., 1997] de forma dinâmica.
  - riqueza de informação disponível em tempo de execução que pode ser utilizada para fins de depuração do código e otimizações, permitindo até a geração de código binário otimizado específico para a plataforma alvo.

Um aspecto negativo da utilização de árvores de programa é a aparente perda de desempenho.

4.2.1 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. Por exemplo, na aplicação cujo código fonte é transcrito na Seção A.3 do Apêndice A, existe uma

árvore para cada uma das classes da aplicação: *Principal*, *Taxi*, *Pessoa* e ainda existem as árvores referentes às classes pré-definidas: *Integer*, *String* e *Boolean*.

A Figura 4.49 ilustra o relacionamento entre as árvores de programa que compõem uma aplicação. Neste caso, as classes correspondentes são chamadas *A*, *B*, *C* e *D*.

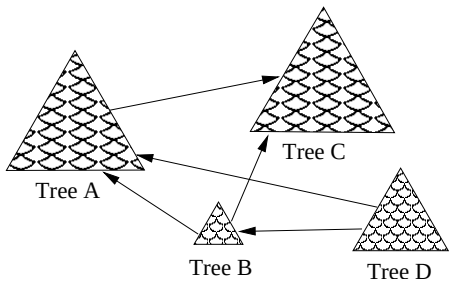

Figura 4.49 Conjunto de árvores de programa que compõem uma aplicação de software

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

Exemplo Básico

A Figura 4.50 apresenta o fragmento de código fonte da classe `Pessoa`.

A Figura 4.51 mostra a árvore de programa correspondente ao código fonte da Figura 4.50 através de um diagrama de colaboração UML.

Com base na Figura 4.51, 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 4.50 Fragmento de código fonte da classe `Pessoa`

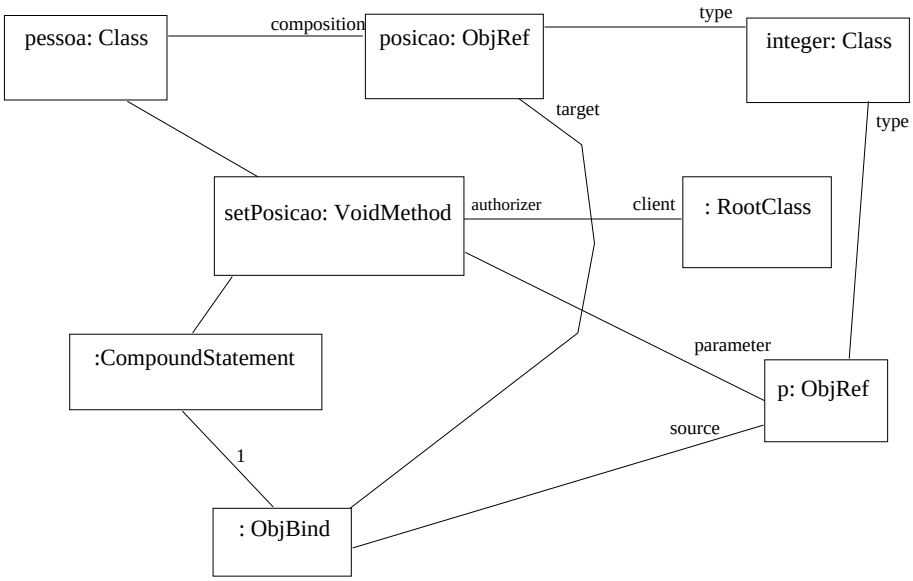

Figura 4.51 Árvore de programa parcial referente à classe Pessoa

(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 instância 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 4.52 apresenta o fragmento de código fonte da classe `Pessoa`.

A Figura 4.53 mostra a árvore de programa correspondente ao código fonte da Figura 4.52 através de um diagrama de colaboração UML.

Com base na Figura 4.53, 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 4.52 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

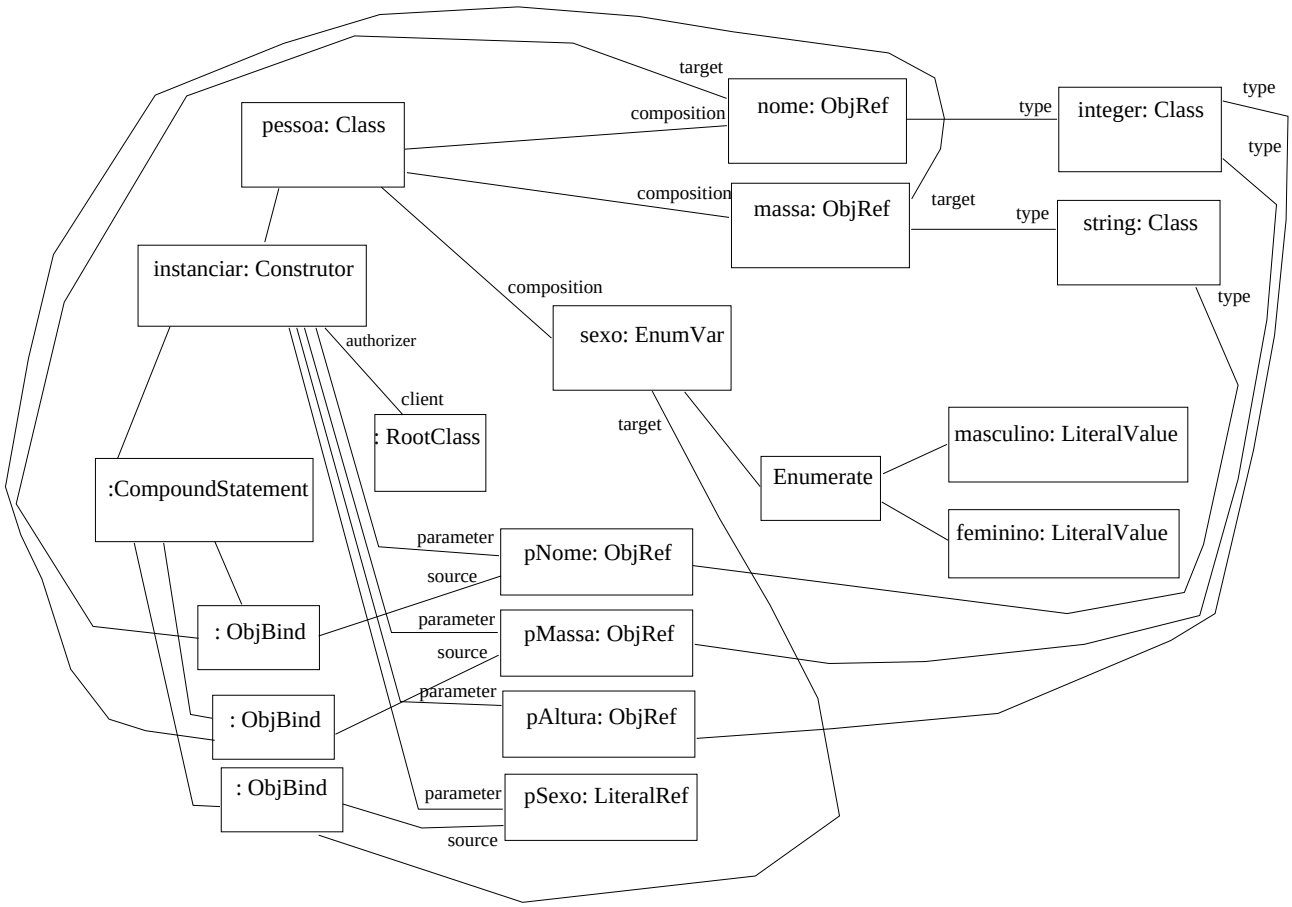

Figura 4.53 Árvore de programa parcial referente à classe Pessoa

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

representa um comando simples de atribuição de referência a objeto. Cada um dos objetos que representam os comandos de atribuição possui duas associações com objetos instância da meta-classe que representa referência a objeto, um representando o alvo da atribuição e o outro representando a origem da atribuição (no caso os pares: `nome` – `pNome`, `massa` – `pMassa` e `sexo` – `pSexo`).

Pontos de Ligação entre Árvores de Programa

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 a 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, ou seja, cada classe da aplicação tem sua representação no formato de árvore de programa armazenada em um arquivo físico separado. 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 no caso desta pertencer 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 4.54 ilustra a ocorrência dos **pontos de ligação** (referências simbólicas) entre árvores de programa de uma aplicação.

Para facilitar o trabalho do carregador de árvores (detalhado na Seção 4.3.2) na tarefa de resolver as referências simbólicas entre árvores, cada árvore de programa possui duas listas que contém todos os seus pontos de ligação, a saber:

(1) lista de referência a classes;

(2) lista de referência a invocáveis.

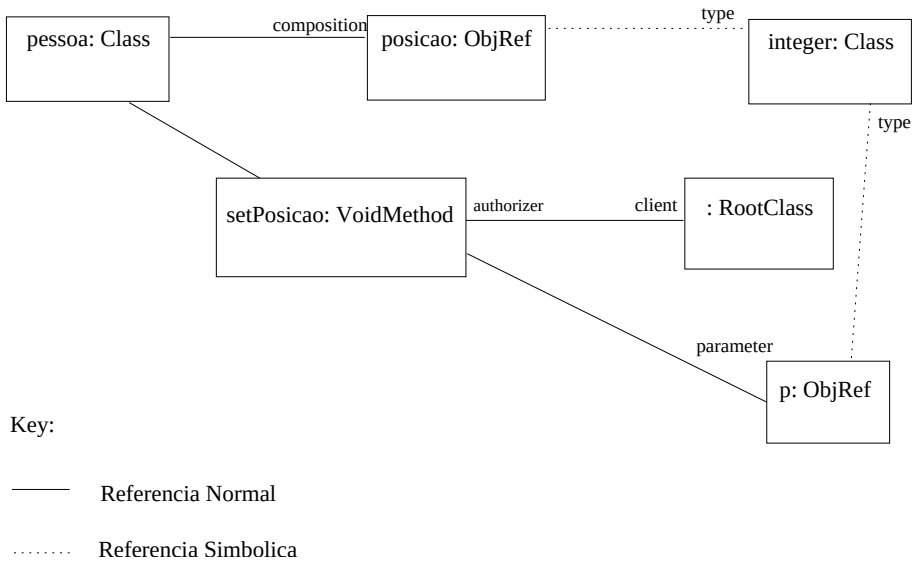

Figura 4.54 Referências simbólicas entre árvores de programa

4.2.2 Lista de Referências a Classe

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

4.2.3 Lista de Referências a Invocáveis

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

4.3 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.

4.3.1 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 árvores**;

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

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

(4) A máquina virtual empilha a atividade inicial na **pilha de atividades**, o que faz com que a atividade inicial seja interpretada. Novas atividades são empilhadas – recursivamente – na pilha de atividades em resposta 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 existam outras pilhas de atividades com atividades empilhadas, a aplicação é encerrada. A existência de outras pilhas de atividade é discutida na Seção 4.3.4.

4.3.2 Área de Árvores

A área de árvores é a região de memória na qual as árvores de programa de cada uma das classes de aplicação são armazenadas[14].

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

No caso dos objetos instâncias da meta-classe que representa uma referência a objeto, a ligação com o seu respectivo tipo é feita de forma indireta através de uma tabela chamada **tabela de árvores**. Esse tipo de estratégia foi utilizado por [HU et al., 2003] e é utilizada devido à natureza distribuída do ambiente Virtuosi, na qual 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 Árvores

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

- Entrada de Árvore Local (EAL);
- Entrada de Árvore Remota (EAR) – neste caso a entrada também armazena o nome da classe de aplicação, mas ao invés de possuir um apontador para uma área de memória,

[14]Embora a arquitetura MVV ainda não preveja a utilização variáveis de classe, se este fosse o caso estas seriam armazenadas na área de árvores.

a entrada possui a identificação da MVV remota e a posição da entrada da tabela de árvores remota.

A Figura 4.55 ilustra a tabela de árvores contendo entradas locais e entradas remotas.

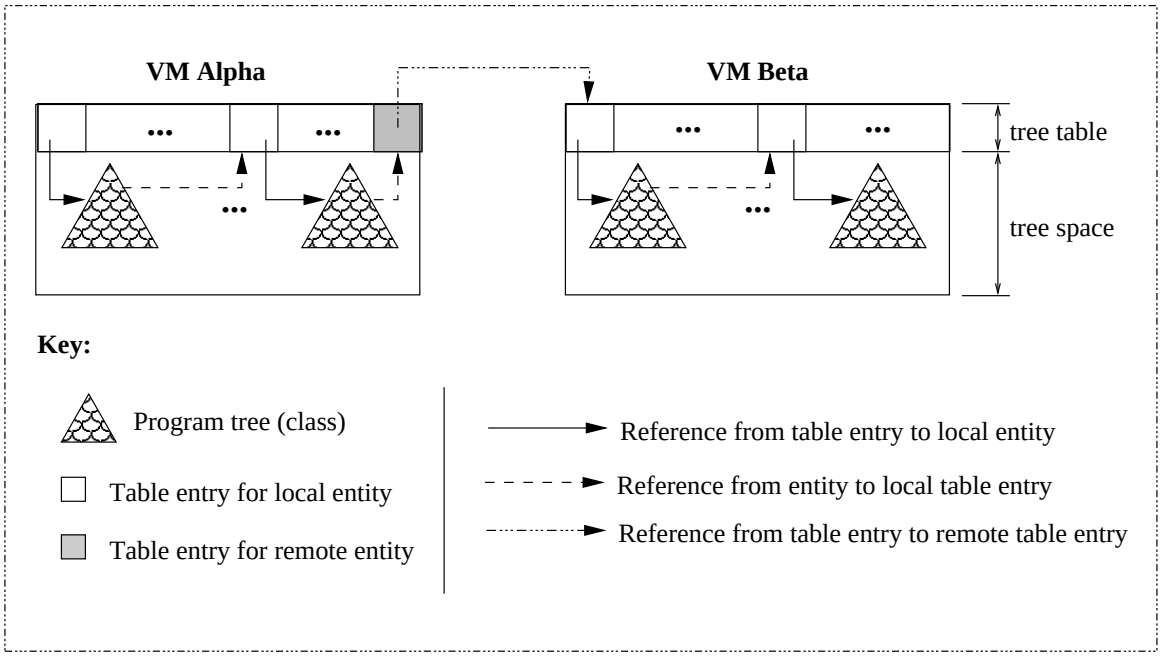

Figura 4.55 Tabelas de árvore com referências locais e remotas

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

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 na qual o invocável está localizado. Pode-se invocar um invocável de um objeto local ou de um objeto remoto. Portanto, existem dois tipos de entrada na tabela de

invocáveis, a saber:

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

A Figura 4.56 ilustra a tabela de invocáveis contendo entradas locais e entradas remotas.

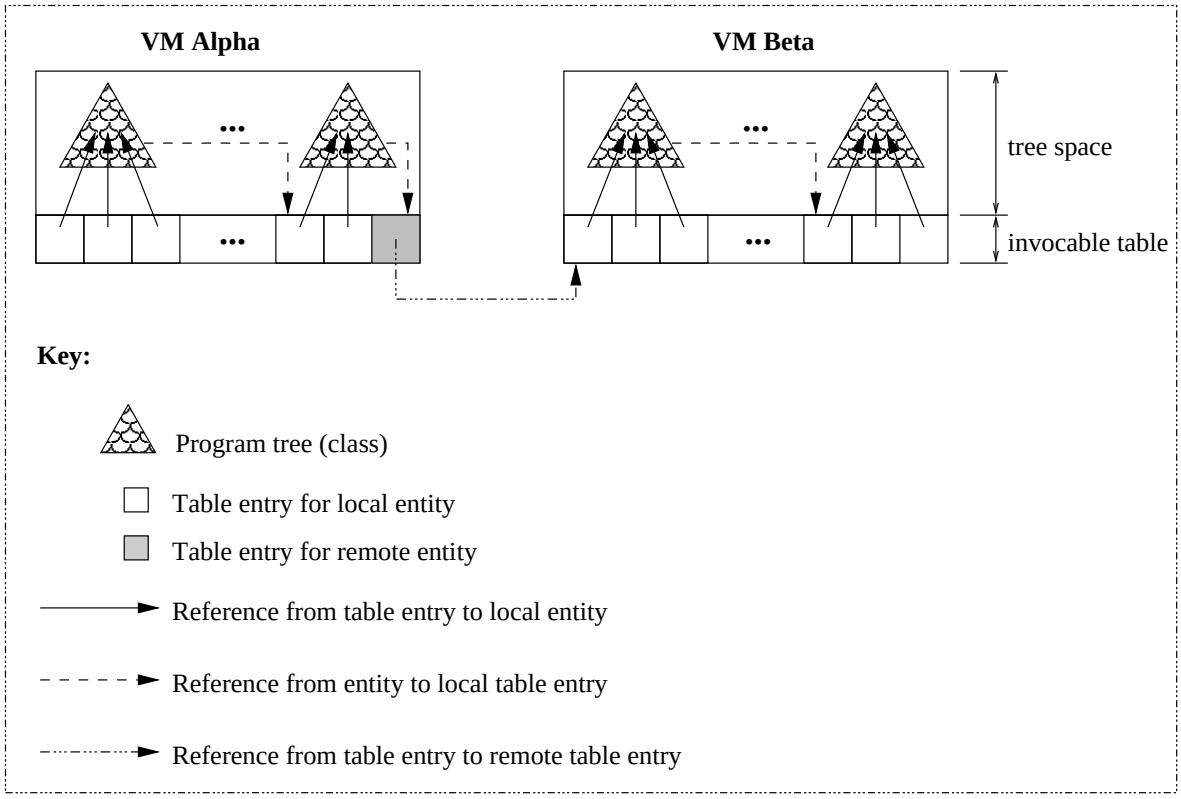

Figura 4.56 Tabelas de invocáveis com referências locais e remotas

Carregador de Árvores

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

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

(1) localizar e carregar uma árvore de programa.

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

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

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

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

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

A Figura 4.57 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` faz uma invocação de um método sobre o atributo da classe `Integer` (soma, por exemplo). Observa-se na Figura 4.57.a a situação da classe `Pessoa` ainda não carregada e com os pontos de ligação ainda como referências simbólicas. A Figura 4.57.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 4.57.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 4.57.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`.

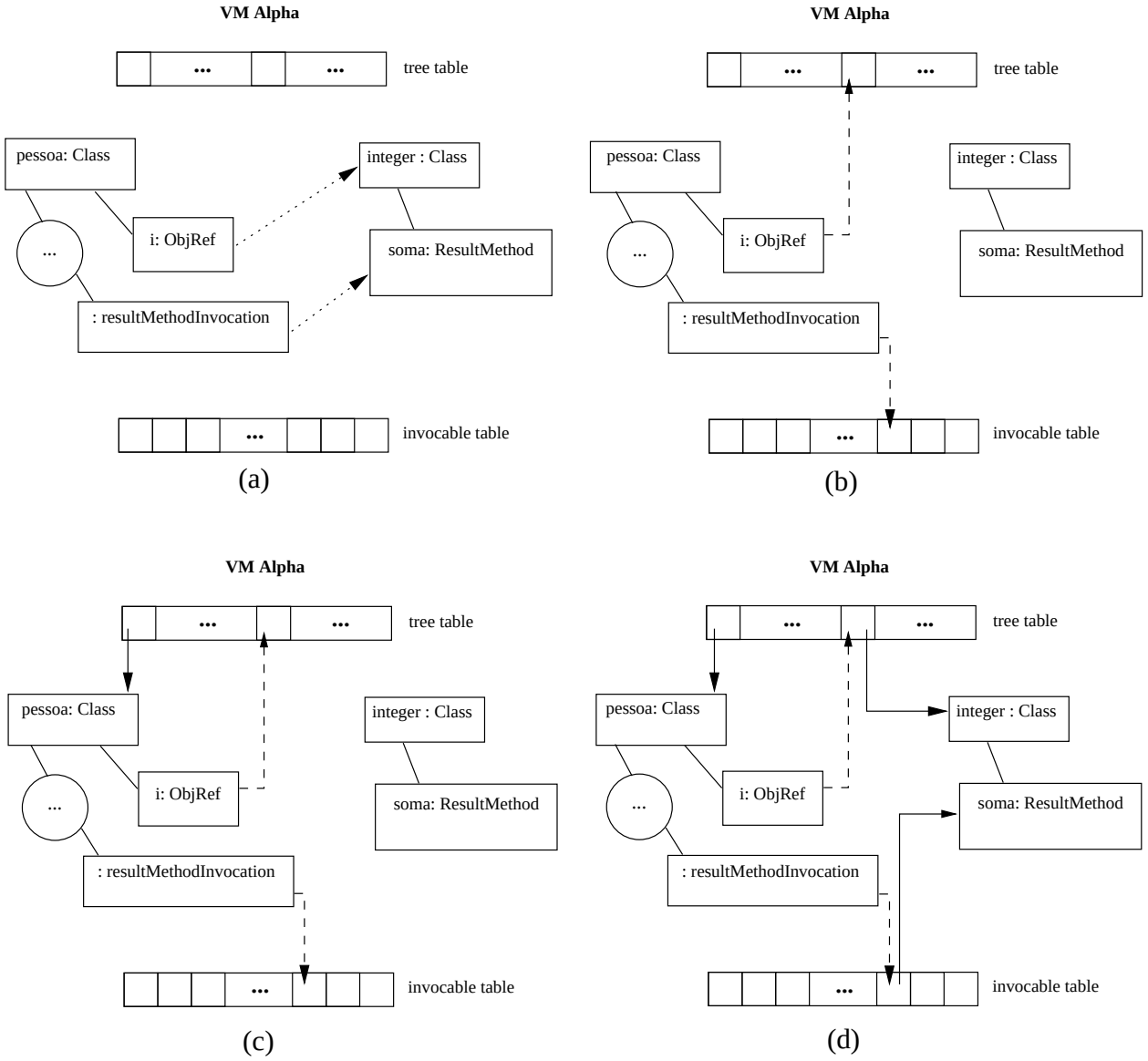

Figura 4.57 Ilustração da carga de duas árvores de programa ligadas entre si

4.3.3 Área de Objetos

A área de objetos é a região de memória na qual 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[15]. A ligação entre os objetos é feita sempre através da **tabela de objetos**. Essa estratégia é utilizada devido à natureza distribuída do ambiente Virtuosi, no qual um objeto pode estar 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 o qual aponta, o nome do objeto contentor (no caso do objeto estar contido em outro objeto) e um apontador para a área de memória na qual o objeto está localizado.

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

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

A Figura 4.58 ilustra a tabela de objetos contendo entradas locais e entradas remotas entre duas MVV.

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

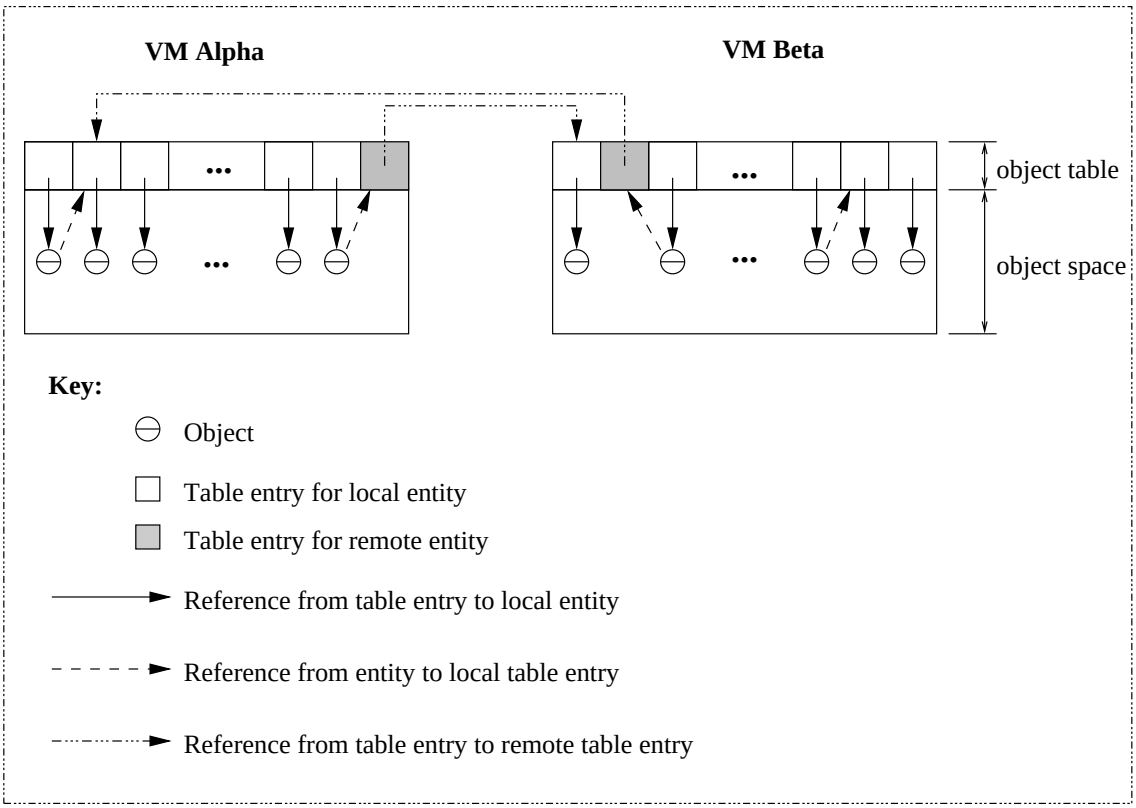

Figura 4.58 Tabelas de objetos com referências locais e remotas

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)[16];

(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.

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

4.3.4 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 na qual 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[17]). 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**.

[17] 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[18].

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:

(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.

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 4.59 mostra a notação gráfica para a representação de uma atividade de objeto.

[18] A linguagem Aram ainda não da suporte à diferenciação de invocações síncronas e assíncronas.

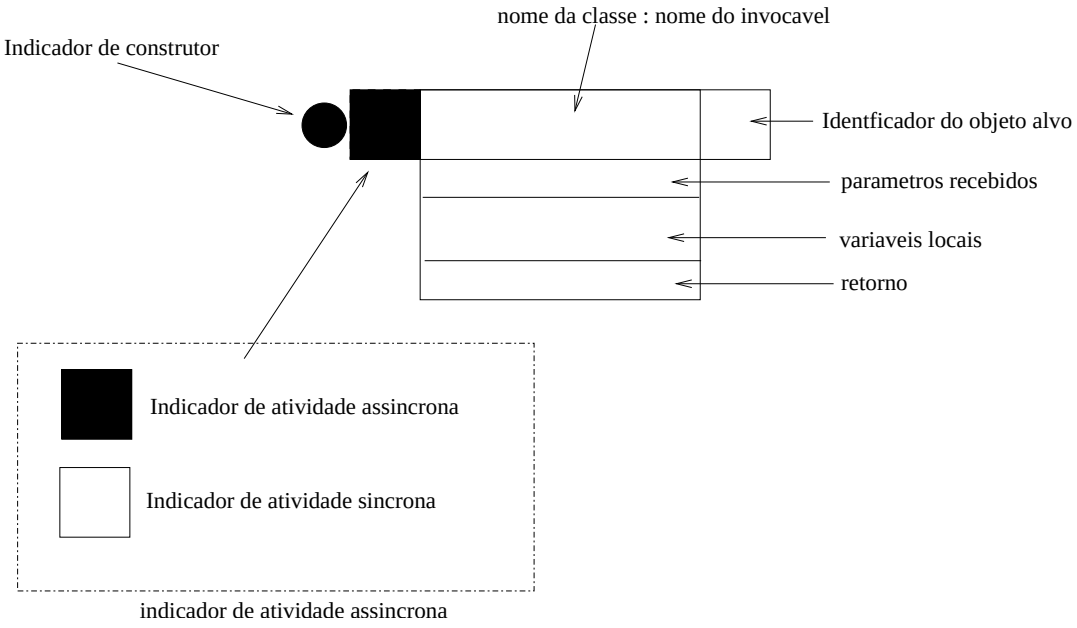

Figura 4.59 Notação gráfica para a representação de uma atividade de objeto

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 com que todas as atividades síncronas sejam empilhadas na mesma pilha de atividades[19] na qual a atividade invocadora existe. Esta estratégia é semelhante a definida pela MEPA em [Kowaltowski, 1983].

A Figural 4.60 ilustra uma série de visões momentâneas[20] da área de atividade de uma instância da MVV interpretando uma atividade síncrona referente à invocação de um construtor de uma classe de aplicação. No corpo do construtor existem dois comandos de

[19]Uma pilha de atividade pode ser comparada a uma linha de execução, do inglês ***thread***

[20]Do inglês, **snapshots**

invocação:

(1) uma invocação do construtor de uma classe que, por sua vez, contém:

(a) uma invocação de método.

(2) uma invocação de método que, por sua vez, contém uma invocação de método.

Deve-se notar que a ilustração está em termos de atividade, sendo que quaisquer outros comandos são omitidos.

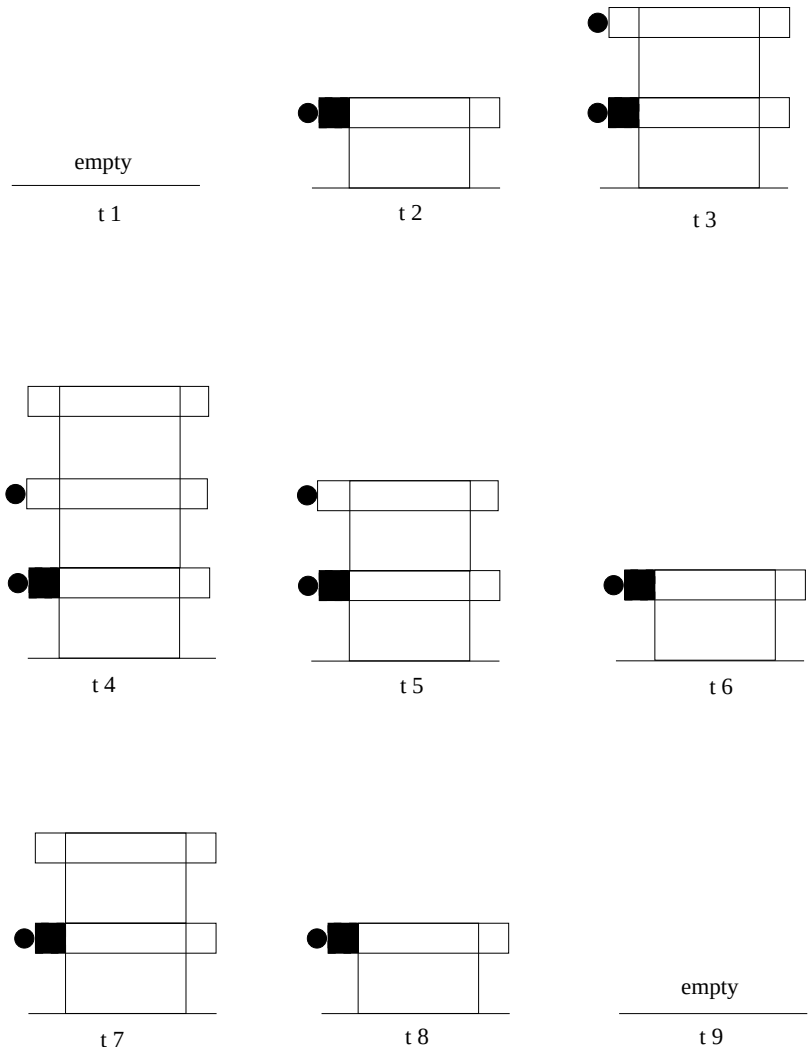

Figura 4.60 Visões momentâneas de uma pilha de atividade síncrona

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.

A Figural 4.61 ilustra uma série de visões momentâneas da área de atividade de uma instância da MVV interpretando uma atividade assíncrona referente à invocação de um construtor de uma classe de aplicação. No corpo do construtor existe somente um comando de invocação de construtor de forma assíncrona. Isto faz com que uma nova pilha de atividades seja criada na área de atividade.

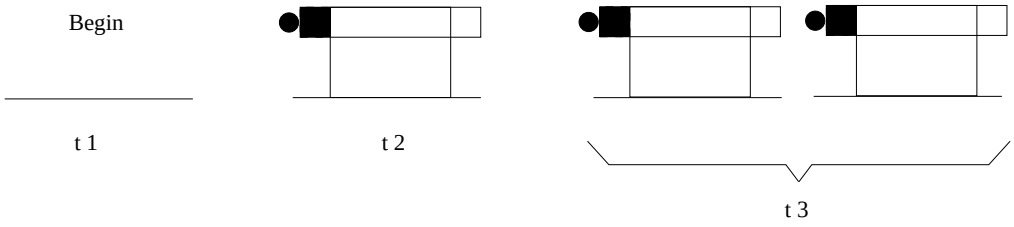

Figura 4.61 Visões momentâneas da criação de uma nova pilha de atividade assíncrona

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[21].

A Figura 4.62 ilustra uma situação de invocação de atividade assíncrona remota.

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

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

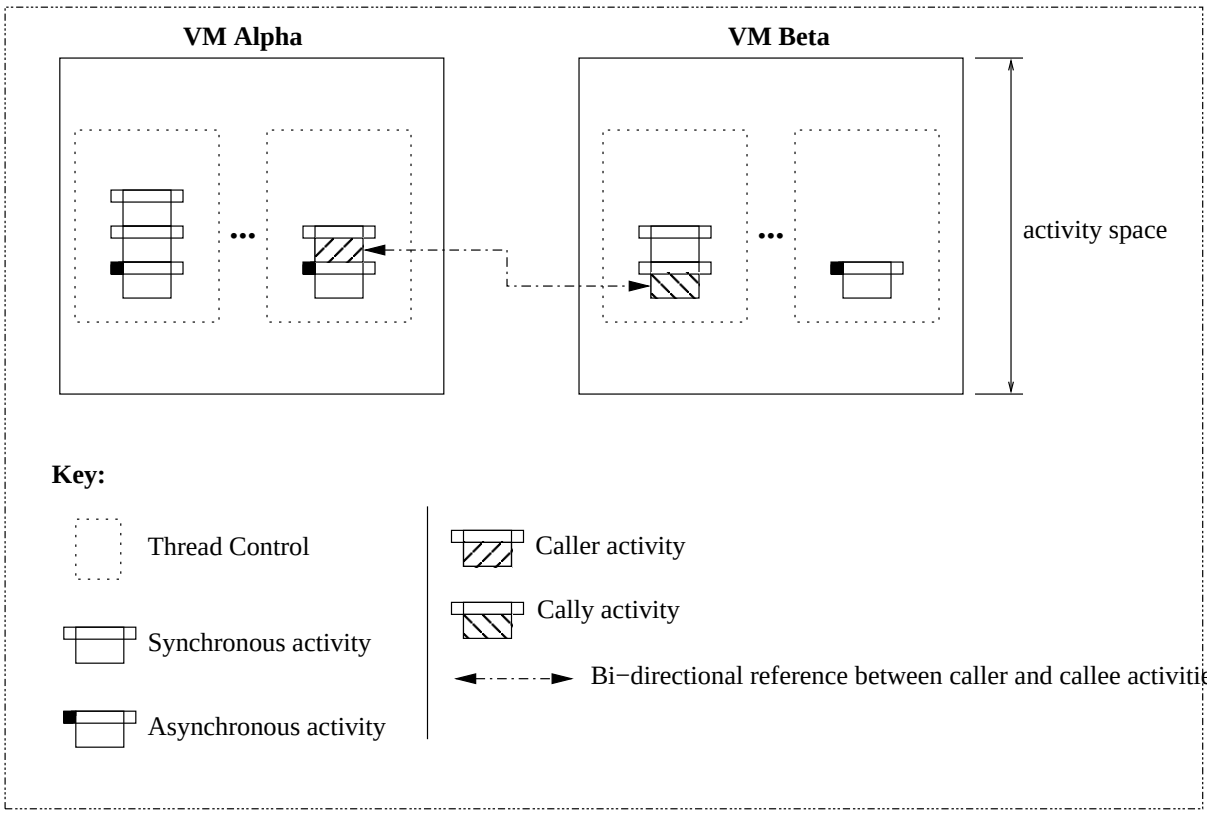

Figura 4.62 Uma atividade assíncrona remota

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

No caso dos construtores, após o término da atividade uma referência para o novo objeto criado é adicionado no topo da pilha de atividades. No caso de um método com retorno uma referência a objeto é adicionada no topo da pilha em resultado da interpretação de um comando de retorno. No caso de uma ação, o que é adicionado no topo da pilha de atividades é um dos comandos resultado de teste. Portanto, uma pilha de atividades pode empilhar elementos que são atividades, referências a objeto ou até mesmo comandos resultado de teste.

4.3.5 Resumo da Arquitetura da Máquina Virtual Virtuosi

Uma visão geral de todas as áreas que compõem a arquitetura da MVV é fornecida na Figura 4.63.

4.3.6 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 árvores.

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

Criação de um Objeto

Para criar um objeto – uma instância de uma classe de aplicação – na MVV é preciso primeiramente obter um nome único para o objeto[22]. 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 a sua classe de aplicação;

(3) para cada um dos atributos do tipo 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 do tipo blocos de dados – definidos na árvore de programa correspondente à classe de aplicação que o objeto é instância – declarar um apontador para uma seqüência de dados binários em memória (área de objetos);

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

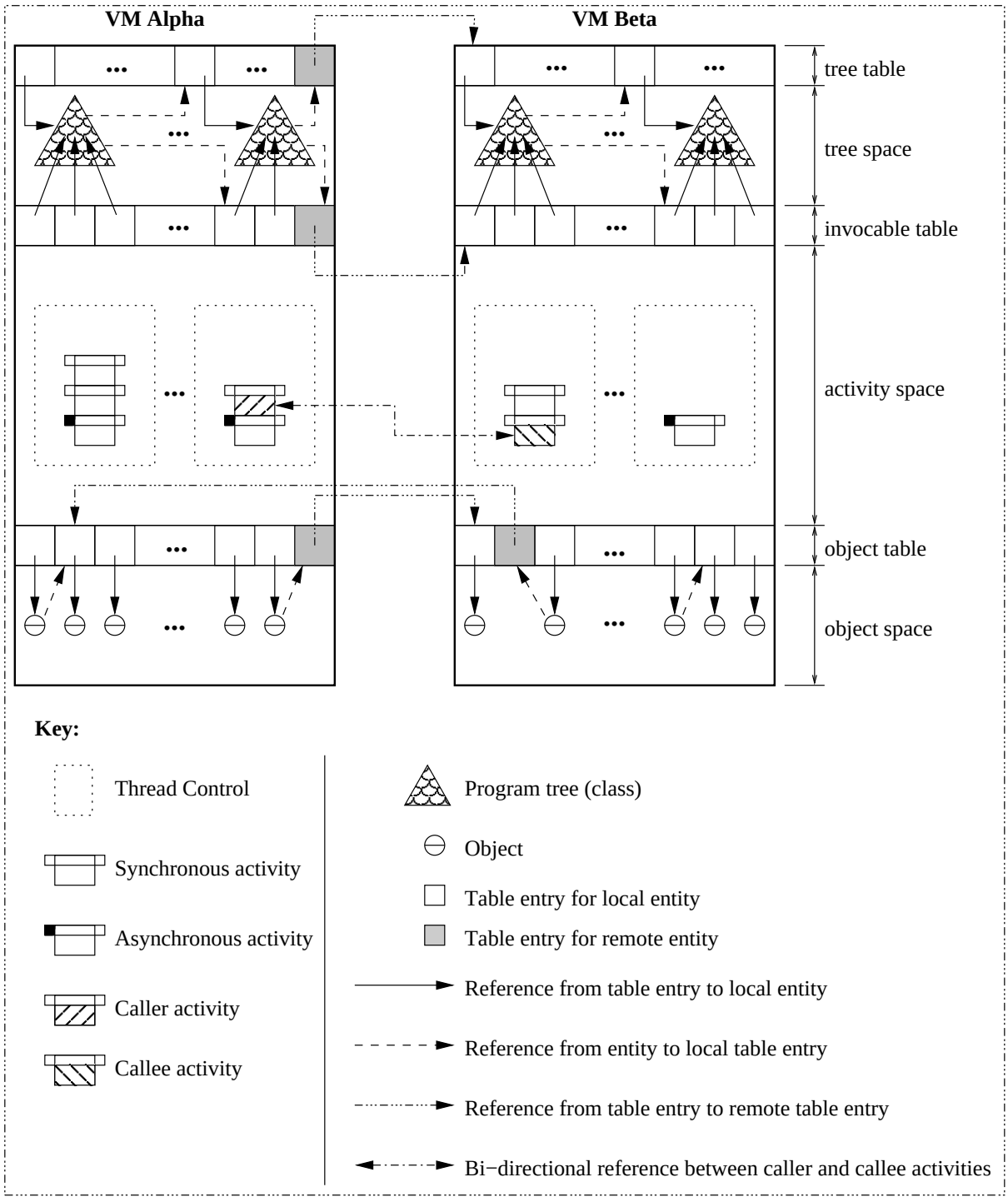

Figura 4.63 Arquitetura da Máquina Virtual Virtuosi

(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.

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

Criação de uma Atividade

A criação de uma atividade é um processo simples. Para criar uma atividade na MVV é preciso 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 – através de suas associações.

Após a criação de uma atividade, a MVV a adiciona no topo da pilha de atividades e ordena a atividade que comece a se interpretar. A atividade então é passada ao comando composto definido pelo invocável para que o comando seja interpretado. A interpretação

do comando composto, por sua vez, é interpretar cada um dos comandos que ele possui. A atividade é então passada para cada um dos comandos na seqüência em que são interpretados, isto permite ao comando corrente ter acesso às variáveis locais e aos parâmetros da atividade.

Cada comando "sabe" o que deve ser feito, por exemplo, um comando de atribuição de referência a objeto tem associado a ele uma origem e um alvo, e o resultado de sua interpretação é que a referência alvo passe a apontar para o mesmo objeto apontado pela referência origem. Quando um comando de invocação é interpretado, isto faz com que uma nova atividade seja criada e comece a ser interpretada em um processo recursivo. Quando uma atividade termina de interpretar seu comando composto, isto faz com que a atividade seja retirada da pilha de atividades.

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

Cada um dos comandos definidos pelo metamodelo da Virtuosi – já descritos na Seção 4.1.7 – define as ações realizadas em decorrência de sua interpretação. Abaixo segue uma lista contendo o nome e uma descrição sucinta do resultado da interpretação de cada um dos comandos:

- **Comando Composto** – sua interpretação implica a interpretação de cada um dos comandos que o compõem, de forma recursiva;
- **Comando de Declaração de Variáveis do Tipo Referência a Objeto** – sua interpretação adiciona à atividade invocada uma variável local que referencia um objeto;
- **Comando de Declaração de Variáveis do Tipo Referência a Bloco de Dados** – sua interpretação adiciona à atividade invocada uma variável local que referencia um

bloco de dados;

- **Comando de Declaração de Variáveis do Tipo Referência a Índice** – sua interpretação adiciona à atividade invocada uma variável local que referencia um índice;
- **Comando de Atribuição de Referência a Objeto** – sua interpretação faz com que a referência a objeto alvo passe a apontar para o mesmo objeto em memória apontado pela origem da atribuição;
- **Comando de Atribuição de Referência Nula a Objeto** – sua interpretação faz com que a referência a objeto alvo passe a não apontar para nenhum objeto em memória;
- **Comando de Atribuição de Referência a Bloco de Dados** – sua interpretação faz com que a referência a bloco de dados alvo passe a apontar para a mesma seqüência contígua de dados binários em memória apontada pela origem da atribuição;
- **Comando de Atribuição de Referência Nula a Bloco de Dados** – sua interpretação faz com que a referência a bloco de dados alvo passe a não apontar para nenhuma seqüência contígua de dados binários em memória;
- **Comando de Atribuição de Referência a Índice** – sua interpretação faz com que a referência a índice alvo passe a apontar para a mesma posição de uma seqüência contígua de dados binários em memória apontada pela origem da atribuição;
- **Comando de Atribuição de Variável Enumerada** – sua interpretação faz com que a variável enumerada receba o valor enumerado existente na origem da atribuição;
- **Comando de Retorno de Método** – sua interpretação faz com que a atividade invocada seja retirada do topo da pilha de atividades e em seguida faz com que a referência a objeto – o alvo – seja colocada no topo da pilha de atividades. Dessa forma a referência a objeto retornada poderá ser a origem de uma atribuição na atividade invocadora;
- **Comando de Invocação de Construtor** – sua interpretação faz com que um novo objeto seja criado (um objeto instância da classe de aplicação dona do construtor associado) e uma nova atividade – atividade invocada – seja criada, adicionada no topo da pilha de atividades e por fim interpretada (recursivamente). É possível que existam parâmetros associados ao comando de invocação do construtor, estes parâmetros então são adicionados à atividade invocada;

- **Comando de Invocação de Método sem Retorno** – sua interpretação faz com que uma nova atividade – atividade invocada – seja criada, empilhada no topo da pilha de atividades e por fim interpretada (recursivamente). Ao final da interpretação, a atividade invocada é retirada do topo da pilha de atividades e o próximo comando pertencente à atividade invocadora é interpretado;
- **Comando de Invocação de Método com Retorno** – sua interpretação faz com que uma nova atividade – atividade invocada – seja criada, empilhada no topo da pilha de atividades e por fim interpretada (recursivamente). O fim da interpretação de um método com retorno ocorre pela interpretação de um comando de retorno de método. Dessa forma, o próximo comando da atividade invocadora, que é obrigatoriamente um comando de atribuição de referência a objeto, utiliza a referência a objeto empilhada como a origem da atribuição;
- **Comando Execute** – sua interpretação faz com que a atividade invocada pelo comando de desvio seja retirada do topo da pilha de atividades e em seguida faz com que o próprio comando *execute* seja empilhado no topo da pilha de atividades;
- **Comando Desvie** – sua interpretação faz com que a atividade invocada pelo comando de desvio seja retirada do topo da pilha de atividades e em seguida faz com que o próprio comando *desvie* seja empilhado no topo da pilha de atividades;
- **Comando de Desvio Condicional** – sua interpretação faz com que uma nova atividade seja invocada. Após a atividade ser executada o comando de desvio condicional desempilha o resultado de teste empilhado no topo da pilha de atividades. Caso o resultado de teste seja um comando de *execute*, então, o caminho destino (que, por sua vez, é um comando simples ou composto) é interpretado recursivamente. Caso o resultado de teste seja um comando *desvie*, então, o caminho alternativo (que, por sua vez, é um comando simples ou composto) é interpretado recursivamente;
- **Comando de Desvio Incondicional** – sua interpretação faz com que o caminho destino (que, por sua vez, é um comando simples ou composto) seja interpretado recursivamente;
- **Comando de Vazio** – sua interpretação não possui nenhum efeito, seu uso está geralmente associado à simulação de um comando de repetição que não realiza nenhuma computação;

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

Deve-se notar que dentre os items da lista supracitada não existe um comando para invocação de ação. Isso ocorre porque a invocação de uma ação sempre está (de forma implícita) associada a um comando de desvio. Ou seja, quem invoca uma ação é o comando de desvio, conforme mostra a Seção 4.1.6. Dessa forma uma ação não pode ser invocada de forma independente de um comando de desvio e por isso não existe um comando de invocação de ação.

Capítulo 5

Implementação da Máquina Virtual

O primeiro protótipo da MVV foi implementado com o objetivo de validar o metamodelo da Virtuosi e o uso de árvores de programa como representação intermediária para máquinas virtuais. Este Capítulo tem por objetivo mostrar como a arquitetura definida na Seção 4.3.5 foi implementada, bem como as limitações do protótipo.

5.1 Ambiente de Desenvolvimento

Para a construção do protótipo foi utilizada a linguagem de programação Java. Poderia ter-se escolhido qualquer linguagem, a escolha de Java foi devido ao fato de ser uma boa linguagem, moderna, bem aceita no meio acadêmico e industrial, com conceitos de orientação a objetos implementados de forma adequada para as necessidades do protótipo e não introduz qualquer restrição na implementação da MVV.

5.2 Limitações

A atual implementação do protótipo possui algumas limitações em relação à arquitetura definida no Capítulo 4, a saber:

(1) dá suporte somente à interpretação de aplicações centralizadas na mesma máquina virtual, embora as estruturas básicas de distribuição (tabela de árvores, tabela de invocáveis e tabela de objetos) tenham sido implementadas.

(2) não dá suporte a mobilidade de objetos nem a invocação remota de invocáveis[1].

(3) não dá suporte a invocação remota de invocáveis[2].

(4) dá suporte somente à interpretação de uma pilha de atividade por instância da MVV, de forma que somente invocações síncronas podem ser interpretadas[3];

(5) da suporte a apenas um bloco de dados por objeto;

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

(7) dentre todos os comandos de sistema listados no Apêndice C, foram implementados apenas:

- CreateDB;
- StoreInteger;
- StoreChar;
- MakeString;
- AddInteger;
- SubtractInteger;

(8) dentre todos os testáveis de sistema definidos no Apêndice C, foram implementados apenas:

- SameBits;
- GreaterThanInteger;
- LowerThanInteger;
- GreaterOrEqualInteger;
- LowerOrEqualInteger;
- GreaterThanChar;
- LowerThanChar;

[1]Este trabalho é estudo de outra dissertação de mestrado [CESAR FILHO, 2004].

[2]Este trabalho é estudo de outra dissertação de mestrado.

[3]Concorrência é objeto de estudo de outra dissertação de mestrado em andamento na atual data.

- GreaterOrEqualChar;
- LowerOrEqualChar;
- GreaterThanString;
- LowerThanString;
- GreaterOrEqualString;
- LowerOrEqualString;

5.3 Diagrama de Classes da MVV

A Figura 5.1 mostra um diagrama de classes com as principais classes que compõem a MVV.

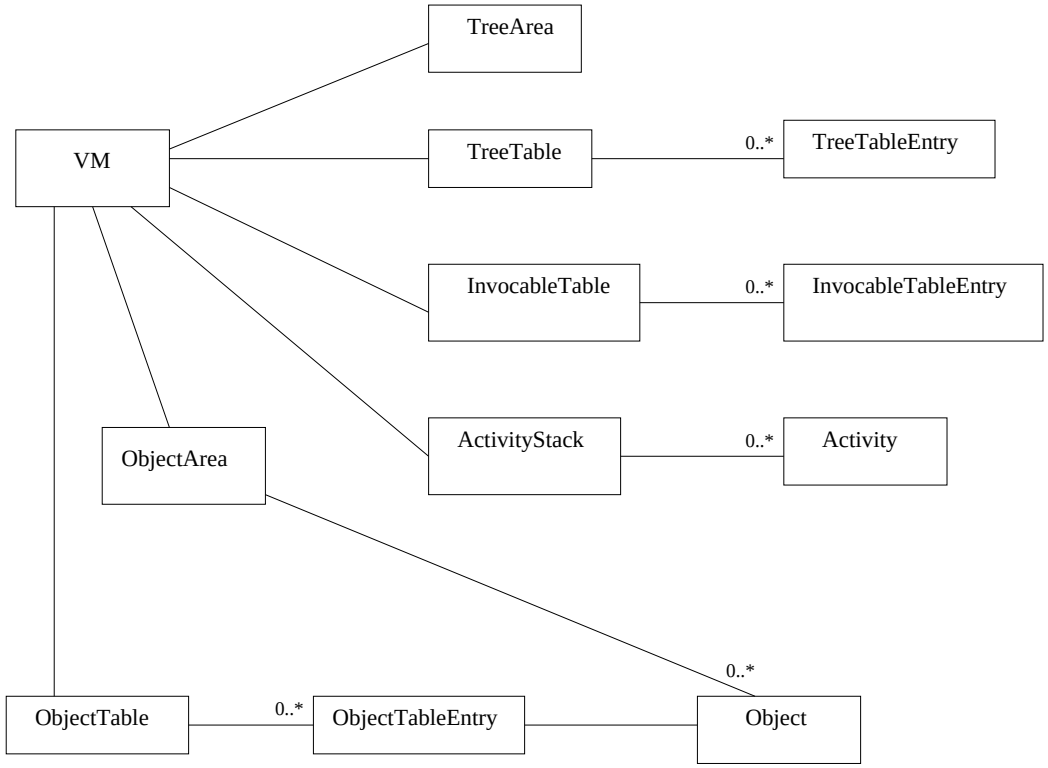

Figura 5.1 Diagrama da principais classes da Máquina Virtual Virtuosi

5.4 Formato das Árvores de Programa

As árvores de programa de cada uma das classes de aplicação são armazenadas em um arquivo físico separado. Cada arquivo contém a árvore de programa mais a lista de referência a classes e a lista de referência a invocáveis. Tanto a árvore de programa quanto as listas são armazenadas na forma de objetos Java serializados. A extensão de um arquivo que contém uma árvore de programa com suas respectivas listas de referências é `.tree`.

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

Essa Seção discute o ciclo de vida de uma aplicação Virtuosi em termos das classes que compõem a MVV.

O método estático `start` da classe `VM` é invocado. 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 4.3.2 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 da 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 da 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 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 no 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;

5.6 Diagrama de Objetos de Uma Instância da MVV

A Figura 5.3 mostra um diagrama de objetos – simplificado e "congelado" em um determinado momento da interpretação – que contém algumas instâncias das principais classes da MVV. Deve-se notar que o diagrama mostra apenas um fragmento da árvore de programa referente à classe de aplicação `A`, não mostrando, por exemplo, o corpo do método `ma` e do construtor `ca`. A Figura 5.2 mostra parte do código fonte da classe de aplicação `A` sendo interpretado. A Figura 5.4 ilustra o relacionamento entre os objetos contidos no diagrama de objetos mostrados na Figura 5.3 sob uma perspectiva arquitetural da MVV.

Observando-se a Figura 5.3 nota-se que existe uma instância da classe `TreeArea` (única em toda a MVV) que possui uma instância da árvore de programa correspondente à classe de aplicação `A`, que, por sua vez, possui uma instância da meta-classe `Method` chamada `ma` e uma instância da meta-classe `Constructor` chamada `ca`. Existe uma instância da classe `TreeTable` (única em toda a MVV) que possui uma instância da classe `TreeTableEntry` associada à árvore de programa da classe de aplicação `A`. Existe uma instância da classe

```
class A
{
   constructor ca( ) exports all {
       ma();
       ...
   }
   method void ma( ) exports all {
       ...
   }
   ...
}
```

Figura 5.2 Código fonte em Aram da classe A

`ObjectArea` (única em toda a MVV) que armazena uma instância da classe `Object` chamado `1` que representa um objeto criado dinamicamente cuja classe de aplicação é a classe `A`. Existe uma instância da classe `ObjectTable` que possui uma instância da classe `ObjectTableEntry` que está associada ao objeto da classe `Object` chamado `1`. Existe uma instância da classe `ActivityStack` que possui duas instâncias da classe `Activity`, a primeira associada ao construtor `ca` da classe de aplicação `A`, e a outra associada ao método `ma` da classe de aplicação `A` invocado pelo construtor `ca`. Ambos os invocáveis estão sendo interpretados sobre o objeto da classe `Object` chamado `1`.

5.7 Metodologia de Desenvolvimento

A metodologia de desenvolvimento adotada foi baseada em algumas boas práticas encontradas em *Extreme Programming* (XP). Um dos motivos desta escolha foi a ênfase dada por XP aos testes automatizados. Segundo XP, antes de se implementar uma funcionalidade é necessário escrever o teste correspondente. Com o teste em mãos, o teste é executado. O teste irá falhar em um primeiro momento, uma vez que a funcionalidade a ser testada ainda não foi construída. Daí então, constrói-se a funcionalidade para que a mesma passe a

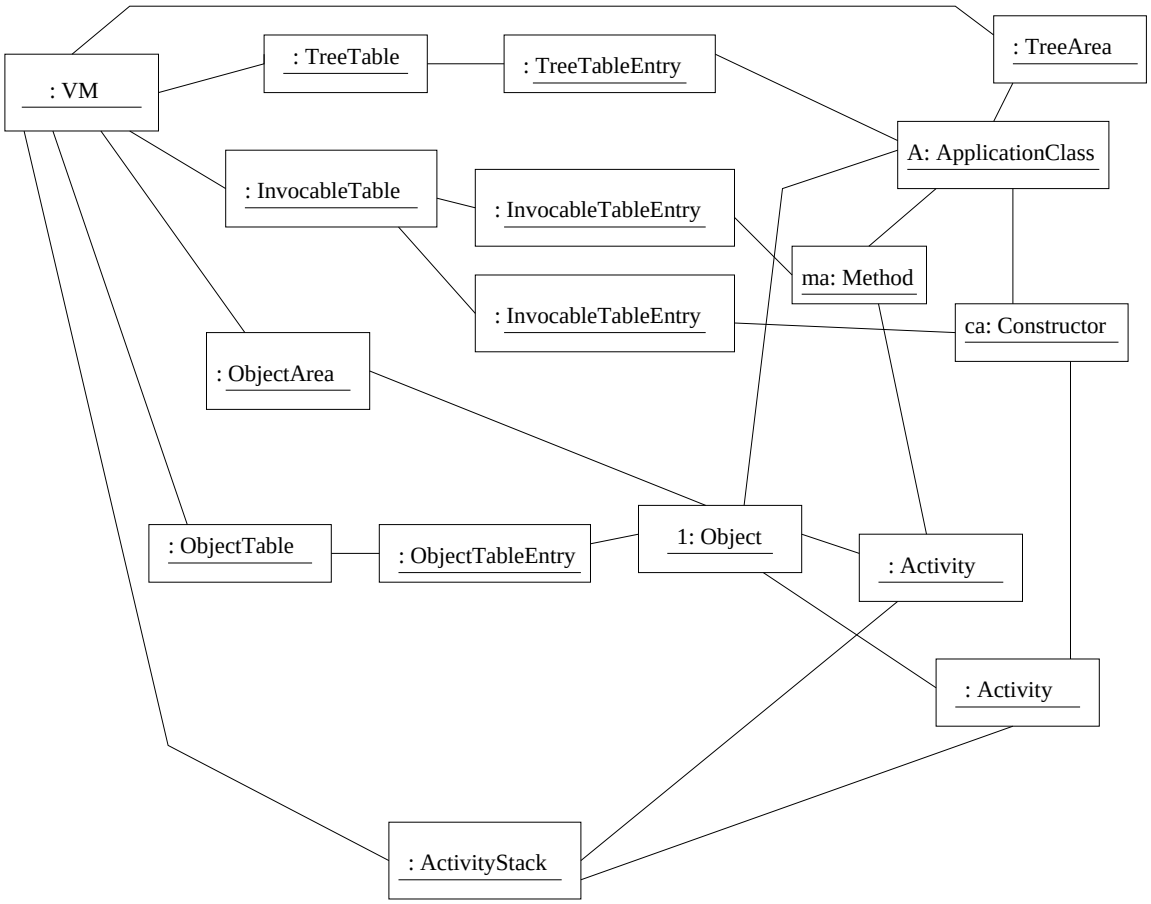

Figura 5.3 Diagrama de objetos da Máquina Virtual Virtuosi

atender o teste. Sempre antes de uma nova funcionalidade ser adicionada a MVV um teste era construído e adicionado ao grupo de testes anteriores. Isto fez com que todo o momento em que um teste era executado, todos os testes existentes fossem avaliados novamente. Isso fez com que, ao final do projeto, os testes sobre a aplicação tenham sido testados centenas, milhares de vezes. Dessa forma, o protótipo foi desenvolvido tendo como base o metamodelo da Virtuosi e através de um desenvolvimento orientado a testes.

5.7.1 Testes de Construção de Árvores de Programa

Os primeiros testes construídos foram os testes responsáveis por construir árvores de programa. Isto foi necessário uma vez que ainda não existia um compilador para geração das

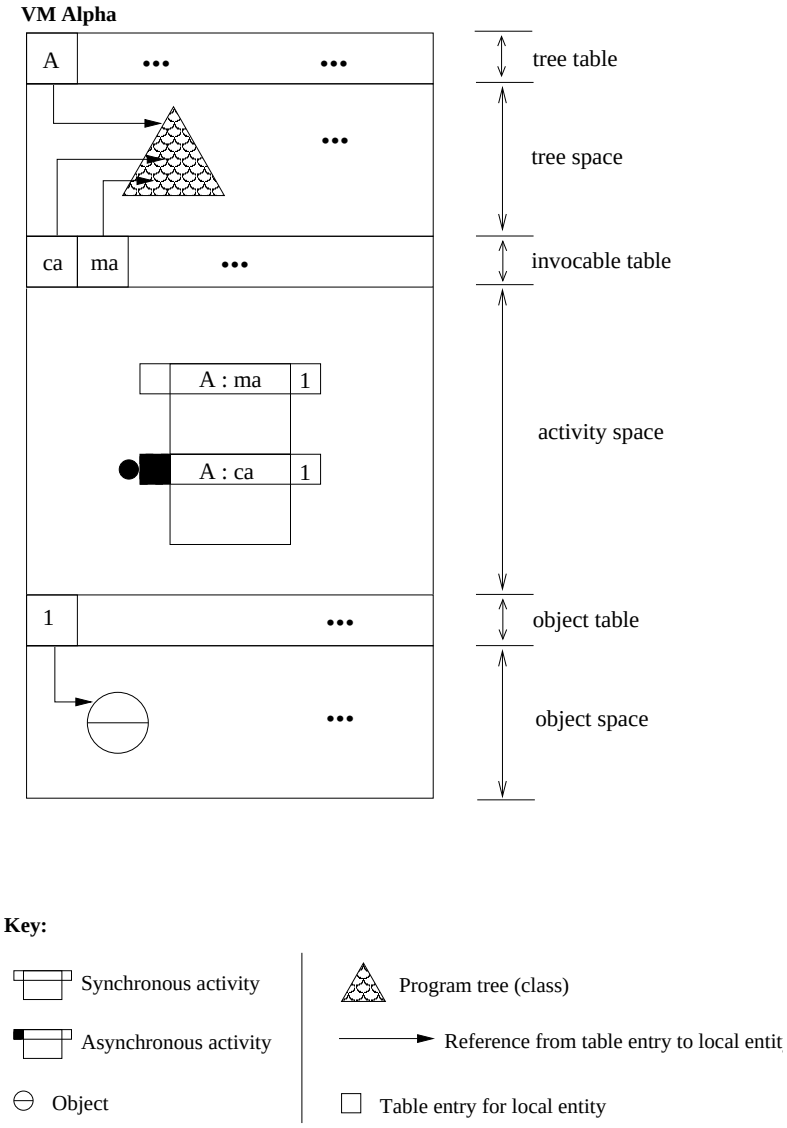

Figura 5.4 Ilustração do relacionamento entre os objetos em uma instância da MVV

árvores de programa[4]. Deve-se notar que, para que estes testes passassem a ser executados com sucesso, foi preciso implementar um formato da árvore de programa bem como a lista de referência a classe e a lista de referência a invocáveis. Estas três estruturas foram armazenadas em arquivos com a extensão `.tree`.

[4]Atualmente, já está disponível um compilador para a linguagem Aram que gera árvores de programa como código objeto.

5.7.2 Testes de Carga de Árvore de Programa

Em seguida, foram construídos os testes responsáveis por validar o carregador de classes da MVV. Dado um arquivo `.tree` contendo uma certa árvore de programa, o teste verificava se:

- a árvore foi corretamente carregada para a área de classes;
- as referências a objeto da árvore estavam apontando para as entradas corretas na tabela de árvores;
- as invocações de invocáveis estavam apontando para as entradas corretas na tabela de invocáveis.

Deve-se notar que para que estes testes passassem a ser executados com sucesso, foi preciso implementar o subsistema carregador de árvores, a área de árvores com suas tabelas de árvores e de invocáveis.

5.7.3 Testes de Interpretação de Árvore de Programa

Em seguida foram construídos os testes responsáveis por avaliar a interpretação dos comandos definidos pelos invocáveis presentes nas árvores de programa. A entrada para esses testes era um arquivo `.tree` contendo uma certa árvore de programa e a avaliação do teste era a impressão de todas as visões momentâneas da pilha de atividade. A criação dos testes ocorreu de forma incremental, ou seja, cada novo teste era adicionado ao conjunto de testes existentes.

A complexidade das árvores de programa utilizadas como insumo para os testes foi aumentada a cada teste. Isso fez também com que o desenvolvimento do protótipo fosse incremental. A maior parte das meta-classes do metamodelo da Virtuosi foi implementada em decorrência da necessidade de executar os testes com sucesso.

Abaixo segue uma lista com os principais testes realizados. Para cada um dos testes são apresentados os principais elementos envolvidos.

- CriaArvoreParaTestarComandosDesvio

  - testa o comando de desvio condicional invocando uma ação que retorna um comando desvie;
  - testa o comando de desvio condicional invocando uma ação que retorna um comando execute;
  - testa o comando de desvio condicional invocando uma ação cujo retorno depende do parâmetro passado.

- CriaArvoreParaTestarInvocacaoDeAcaoCujoResultadoDependoDoParametro
  - testa a invocação de um método que possui uma ação que retorna "execute" ou "desvie" de acordo com o parâmetro passado.

- CriaArvoreParaTestarInvocacaoDeAcaoQueSempreDesvia
  - testa a invocação de um método que possui uma ação que sempre retorna o comando "desvie".

- CriaArvoreParaTestarInvocacaoDeAcaoQueSempreExecuta
  - testa a invocação de um método que possui uma ação que sempre retorna o comando "execute".

- CriaArvoreParaTestarInvocacaoDeMetodoComRetorno
  - testa a invocação de método com retorno;
  - testa o comando de atribuição de referência a objeto com a origem sendo uma invocação de método com retorno;
  - testa a passagem de mais de um parâmetro para um método com retorno.

- CriaArvoreParaTestarRecebimentoDeVariosParametrosEComandoRetorno
  - testa o recebimento de mais de um parâmetro em um método;
  - testa o comando de retorno de referência a objeto.

- CriaArvoreParaTestarSomaDeObjetosInteiros
  - testa a declaração de variáveis locais do tipo referência a objeto;

  - testa a invocação de construtor passando um parâmetro do tipo valor literal;
  - testa o comando de atribuição de referência a objeto com a origem sendo uma invocação de construtor;
  - testa a invocação de método sem retorno passando um parâmetro do tipo valor literal;
  - testa a invocação de método sem retorno passando um parâmetro do tipo referência a objeto.

- CriaArvoreParaTestarVariosParametrosNoConstrutor
- CriarArvoreParaManipularValoresInteiros
  - testa o comando de sistema CreateDB;
  - testa o comando de sistema StoreInteger;
  - testa o testável de sistema GreaterOrEqualInteger
  - testa o comando de sistema AddInteger;
  - testa o comando de atribuição de referência a bloco de dados com a origem sendo um acesso a atributo bloco de dados;
  - testa o comando de atribuição de referência a bloco de dados;
  - testa o comando de declaração de variável local do tipo referência a bloco de dados;
  - testa o uso da referência This como alvo em uma invocação de método.
- CriarArvoreParaTestarTestáveisEPassagemDeParametrosNulos
  - testa um comando de desvio com um testável do tipo comparação de referências a objeto;
  - testa um comando de desvio com um testável do tipo comparação de referências a bloco de dados;
  - testa um comando de desvio com um testável do tipo comparação de referências a bloco de dados nulo;
  - testa a passagem de parâmetro nulo para um método.
- Por último foi avaliado um teste da interpretação das árvores de programa referentes a aplicação contida na Seção A.3.

5.8 Estatísticas da Implementação

A implementação do protótipo da MVV gerou os seguintes números:

- Cento e quarenta e uma (141) classes dividas em dois pacotes:
  - core (47) – Pacote que contém as classes internas à MVV. Exemplos das classes contidas neste pacote são as classes que representam o carregador de árvores, atividades, pilha de atividade, exceções da MVV, etc;
  - meta (94)– Pacote que contém as meta-classes do metamodelo da Virtuosi. Exemplos das classes contidas neste pacote são as classes Referência, Referência a Objeto, Classe, Declaração de Variável, Parâmetro Formal, Invocável, Invocação, Desvio, etc.
- Trinta e uma classes de teste;
- Cerca de dez mil linhas de código.

Capítulo 6

Conclusão

De um modo geral, pode-se dizer que os objetivos específicos desta dissertação, conforme descrito na Seção 1.4, foram alcançados de forma satisfatória, a saber:

- definir um modelo de classes (metamodelo) que represente os elementos existentes em uma linguagem de programação orientada a objetos;
- projetar a arquitetura de uma máquina de interpretação (máquina virtual) para um sistema computacional baseado em objetos que facilite a computação distribuída;
- validar o metamodelo e arquitetura definidas (através da implementação da máquina virtual Virtuosi).

Além disso, foi dado o primeiro passo para o cumprimento do objetivo do projeto Virtuosi que é desenvolver uma ferramenta com as perspectivas pedagógica e experimental para a construção (edição, compilação, e depuração) e a execução de sistemas de software distribuídos.

6.1 Contribuição Científica do Trabalho

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

- serve como base para um ambiente distribuído de execução de software orientado a objetos – o ambiente Virtuosi;
- formaliza o metamodelo da Virtuosi que define e limita os conceitos de orientação a objetos implementados por qualquer linguagem de programação que deseje ser compatível com o ambiente Virtuosi;

- valida o uso de árvores de programa como representação intermediária para software orientado a objeto;
- valida o uso de tabelas de manipulação para manter referências entre árvores de programa carregadas em memória (relacionamentos inter e intra-classes);
- valida o uso de tabelas de manipulação para manter referências entre objetos instância de classes de aplicação;
- valida o uso de um carregador de árvore que carrega árvores de programa a partir de arquivos em disco para a memória.

6.2 Trabalhos Futuros

Através do desenvolvimento desta dissertação é possível prever os seguintes trabalhos futuros:

- Melhorias na implementação da Máquina Virtual Virtuosi:
  - Melhorar a estrutura interna com o objetivo de simplificar a implementação;
  - Construir uma implementação em uma linguagem com melhor desempenho do que Java;
  - Melhorar a ligação entre o compilador existente e a MVV;
  - Construir uma interface homem-máquina amigável(editor, depurador, etc);
  - Permitir mais de um bloco de dados por objeto.
- Conceitos essenciais que devem ser implementados:
  - Herança;
  - Polimorfismo;
  - Entrada e Saída;
  - Manipulação de índices de bloco de dados;
  - Lista de Exportação.
- Extensões do metamodelo para contemplação de:

  - Tratamento de Exceções;
  - Chamada de procedimento remoto (RPC);
  - Migração de objetos;
  - Mais de uma linha de execução (*multi-threads*);
  - Controle de concorrência;
  - Pré e pós-condições bem como invariantes;
  - Classes abstratas;
  - Interfaces;
  - Expressões;
  - Coleções.

- Construção de bibliotecas para usuários finais (programadores).
- Definir um padrão de *bytecode* correspondente ao metamodelo da Virtuosi que fosse um subconjunto do *bytecode* do Java. Esta abordagem atenderia às aplicações para dispositivos móveis.

Portanto, o trabalho realizado permitiu a criação de uma definição formal e correspondente implementação para uma máquina virtual orientada a distribuição, a qual pode servir como base para diversos outros trabalhos mais avançados.

Apêndice A

Estudo de Caso

Esse apêndice contém o código fonte referente de uma pequena aplicação de software orientada a objeto implementada nas linguagens Java, Eiffel e Aram. O intuito é comparar e fornecer uma visão geral da construção de uma aplicação segundo os conceitos formalizados pelo metamodelo da Virtuosi.

A aplicação possui três classes de aplicação: A classe `Principal` (reponsável por definir a seqüência da computação realizada pela aplicação), a classe `Taxi` e a classe `Pessoa`. Para a aplicação implementada em Aram são adicionadas as classes pré-definidas *Integer, Boolean* e *String*.

Abaixo segue uma descrição não sistemática de cada uma das classes da aplicação:

A classe `Principal` é responsável por:

- criar um táxi;
- criar uma pessoa;
- tentar fazer com que a pessoa entre no táxi, caso isso seja possível o táxi deve mover-se uma certa distância e em seguida a pessoa deve sair do táxi;
- caso a pessoa seja obesa ela deve começar um regime até que não seja mais obesa.

A classe `Taxi` pode ser definida pelos seguintes requisitos:

- um táxi pode transportar somente uma pessoa em um determinado momento;
- um táxi pode estar trabalhando ou não;
- um táxi possui uma posição;
- um táxi quando criado está na posição zero, trabalhando e não possui passageiro;

- um táxi pode mover-se um a determinada distância;
- um táxi quando em movimento faz com que a pessoa também esteja se movendo;
- um táxi pode pegar um passageiro – para que um passageiro entre no táxi é preciso que o táxi esteja trabalhando e a pessoa seja obesa;
- um táxi pode entregar um passageiro.

A classe `Pessoa` pode ser definida pelos seguintes requisitos:

- uma pessoa tem um nome;
- uma pessoa tem uma massa;
- uma pessoa tem uma altura;
- uma pessoa tem uma posição;
- uma pessoa é do sexo masculino ou feminino exclusivamente;
- uma pessoa quando criada precisa receber um nome, uma massa, uma altura, uma posição e uma determinação do sexo;
- uma pessoa do sexo masculino é considerada obesa[1] quando sua altura (em centímetros) menos cem é menor ou igual sua massa, enquanto que uma pessoa do sexo feminino é considerada obesa quando sua altura menos oitenta é menor ou igual a sua massa;
- uma pessoa pode emagrecer uma quantia de massa determinada.

A.1 Aplicação em Java

```
public class Principal {
   public static void main (String[] args){
       Principal principal = new Principal();
   }
   public Principal()
```

[1]Estes valores são meramente ilustrativos e não tem fundamento teórico

```
    {
        Taxi corsa = new Taxi();
        String nome = new String("Andrea Barbieri");
        int massa = 70;
        int altura = 180;

        Pessoa andrea = new Pessoa(nome, massa, altura, Pessoa.feminino);

        boolean entrou = corsa.entrarPassageiro(andrea);
        if (entrou) {
            int distancia = 10;
            corsa.mover(distancia);
            corsa.sairPassageiro();
        }

        int t = 2;
        while (andrea.obesa()) {
            andrea.emagreca(t);
        }
    }
}


class Taxi {
    private Pessoa passageiro;
    private int posicao;
    private boolean trabalhando;

    public Taxi()
    {
        posicao = 0;
        trabalhando = true;
    }
```

```
    boolean entrarPassageiro(Pessoa p)
    {
        boolean resultado;
        if (trabalhando) {
            if (p.obesa())  {
                resultado = false;
            } else {
                passageiro = p;
                resultado = true;
            }
        } else {
            resultado = false;
        }
        return resultado;
    }

    void sairPassageiro()
    {
        passageiro = null;
    }

    void mover(int distancia)
    {
        posicao = posicao + distancia;
        if (passageiro != null) {
            passageiro.setPosicao(posicao);
        }
    }
}

class Pessoa {
    private String nome;
    private int massa;
```

```
    private int altura;
    private int posicao;

    public static final String feminino = "feminino";
    public static final String masculino = "masculino";
    public static String sexo = masculino;

    public Pessoa(
        String pNome,
        int pMassa,
        int pAltura,
        String pSexo)
    {
        nome = pNome;
        massa = pMassa;
        altura = pAltura;
        sexo = pSexo;
        posicao = 0;
    }

    public void setPosicao(int p)
    {
        posicao = p;
    }

    public boolean obesa()
    {
        int v;
        if (sexo.equals("masculino")) {
            v = 100;
        } else {
            v = 80;
        }
```

```
        int h = altura;
        h = h - v;

        if (massa >= h)
            return true;
        else
            return false;
    }
    public void emagreca(int i)
    {
        massa = massa - i;
    }
 }
```

A.2 Aplicação em Eiffel

```
class PRINCIPAL creation
    make

feature {ANY }
    make is
        do
            corsa: TAXI
            nome: STRING
            massa: INTEGER
            altura: INTEGER

            !!corsa.make
            !!nome.make("Andrea Barbieri")
            massa := 70
            altura := 180
```

```
            andrea: PESSOA
            !!andrea.make(nome,massa,altura,1)// 1 feminino

            entrou: BOOLEAN

            entrou := corsa.entrarPassageiro(andrea)

            if (entrou = true) then
                distancia: INTEGER
                distancia := 10
                corsa.mover(distancia)
                corsa.sairPassageiro
            end

            from
                t: INTEGER
                t = 2
            until
                andrea.obesa = true
            loop
                andrea.emagreca(t);
            end
        end
end

class TAXI creation
    make
feature {NONE}
    passageiro: PESSOA
    posicao: expanded INTEGER
    trabalhando: expanded BOOLEAN
```

```
feature {ANY }
    make is
        do
            posicao := 0
            trabalhando := 0
        end

feature {PRINCIPAL }

    entrarPassageiro(p: PESSOA) is
        do
            resultado: BOOLEAN
            if (trabalhando = true)then
                if (p.obesa = true) then
                    resultado := false
                else
                    passageiro := p
                    resultado := true
                end
            else
                resultado := false;
            end
        Result :=
        end

    sairPassageiro is
        do
            passageiro := Void
        end

    mover(distancia: INTEGER) is
        do
            posicao := posicao + distancia
```

```
            if ( passageiro /= Void)then
                passageiro.setPosicao(posicao)
            end
        end
end


class PESSOA creation
    make
feature {NONE}
    nome: expanded STRING
    massa: expanded INTEGER
    altura: expanded INTEGER
    posicao: expanded INTEGER
    sexo: expanded INTEGER

    feminino, masculino : INTEGER is unique

feature {ANY }
    make(pNome: STRING, pMassa,pAltura,pSexo: INTEGER) is
        do
            posicao := pNome
            massa := pMassa
            altura := pAltura
            sexo := pSexo
            posicao := 0

        end

    setPosicao(p:INTEGER) is
        do
            posicao := p
        end
```

```
    obesa: BOOLEAN is
        do
            v: INTEGER
            if (sexo == masculino) then
                v := 100
            else
                v := 80
            end
            h: INTEGER
            h := altura
            h := h - v

            if (massa >= h)then
                Result := true
            else
                Result := false
            end
        end

    emagreca (i: INTEGER) is
        do
            massa := massa - i
        end
end
```

A.3 Aplicação em Aram

```
 class Principal
 {
    constructor iniciar() exports all
```

```
    {
        Taxi corsa = Taxi.instanciar();
        String nome = String.make("Andrea Barbieri");
        Integer massa = Integer.make(70);
        Integer altura = Integer.make(180);

        Pessoa andrea = Pessoa.instanciar(nome, massa, altura, feminino);

        Boolean entrou = corsa.entrarPassageiro(andrea);
        if (entrou) {
            Integer distancia = Integer.make(10);
            corsa.mover(distancia);
            corsa.sairPassageiro();
        }

        Integer t = Integer.make(2);
        while (andrea.obesa()) {
            andrea.emagreca(t);
        }
    }
}

class Taxi
{
    association Pessoa passageiro;
    composition Integer posicao;
    composition Boolean trabalhando;

    constructor instanciar() exports { Principal }
    {
        posicao = Integer.make(0);
        trabalhando = Boolean.make(true);
    }
```

```
    method Boolean entrarPassageiro(Pessoa p) exports { Principal }
    {
        Boolean resultado;
        if (trabalhando) {
            if (p.obesa())  {
                resultado = Boolean.make(false);
            } else {
                passageiro = p;
                resultado = Boolean.make(true);
            }
        } else {
            resultado = Boolean.make(false);
        }
        return resultado;
    }

    method void sairPassageiro() exports { Principal }
    {
        passageiro = null;
    }

    method void mover(Integer distancia) exports { Principal }
    {
        posicao.add(distancia);
        if (passageiro != null) {
            passageiro.setPosicao(posicao);
        }
    }
}

class Pessoa
{
    composition String nome;
```

```
composition Integer massa;
composition Integer altura;
composition Integer posicao;

enum {masculino, feminino } sexo = masculino;

constructor instanciar(
    String pNome,
    Integer pMassa,
    Integer pAltura,
    literal pSexo) exports all
{
    nome = pNome;
    massa = pMassa;
    altura = pAltura;
    sexo = pSexo;
    posicao = Integer.make(0);
}

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

action obesa() exports all
{
    Integer v;
    if (sexo == masculino) {
        v = Integer.make(100);
    } else {
        v = Integer.make(80);
    }
    Integer h = Integer.make(altura);
```

```
        h.subtract(v);

        if (massa_.gt(h))
            execute;
        else
            skip;
    }
    method void emagreca(Integer i) exports all
    {
        massa.subtract(i);
    }
}

class Integer
{
    datablock value;

    constructor make( literal k ) exports all
    {
            value = datablock.make( 32 );
            value.storeInteger( k );
    }

    constructor make( Integer i ) exports all
    {
            this.set( i );
    }

    method void add( literal k ) exports all
    {
        Integer i = make( k );
        add( i );
    }
```

```
method void add( Integer i ) exports all
{
        datablock d = i.value;
        value.addInteger( d );
}

method void subtract( literal k ) exports all
{
    Integer i = make( k );
    subtract( i );
}

method void subtract( Integer i ) exports all
{
    datablock d = i.value;
    value.subtractInteger( d );
}


method void set( literal k ) exports all
{
        value.storeInteger( k );
}

method void set( Integer i ) exports all
{
        datablock k = i.value;
        value = k.clone( );
}

action equals( Integer i ) exports all
{
```

```
        datbalock k = i.value;
        if ( value.equals( k ) )
            execute;
        else
            skip;
}

action differs( Integer i ) exports all
{
        datbalock k = i.value;
        if ( value.equals( k ) )
            skip;
        else
            execute;
}

action geq( Integer i ) exports all // greater or equal
{
        datbalock k = i.value;
        if ( value.geqInteger( k ) )
            execute;
        else
            skip;
}

action leq( Integer i ) exports all // less or equal
{
        datbalock k = i.value;
        if ( value.leqInteger( k ) )
            execute;
        else
            skip;
}
```

```
   action gt( Integer i ) exports all // greater than
   {
           datbalock k = i.value;
           if ( value.gtInteger( k )
               execute;
           else
               skip;
   }

   action lt( Integer i ) exports all // less than
   {
           datbalock k = i.value;
           if ( value.ltInteger( k ) )
               execute;
           else
               skip;
   }
}

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

   constructor make() exports all
   {
   }

   constructor make( literal k ) exports all
   {
       value = k; // ocorrera erro em tempo de execução se
                  // k for inválido
   }
```

```
constructor make( Boolean b ) exports all
{
    value = b.value;
}

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

action default() exports all
{
    if (value==true)
        execute;
    else
        skip;
}

action equals( Boolean b ) exports all
{
    if ( value == b.value )
        execute;
    else
        skip;
}

action equals( literal k ) exports all
{
    if ( value == k )
```

```
            execute;
        else
            skip;
    }

    action differs( Boolean b ) exports all
    {
        if ( value == b.value )
            skip;
        else
            execute;
    }

    action differs( literal k ) exports all
    {
        if ( value == k )
            skip;
        else
            execute;
    }

}

class String
{
    datablock value;

    constructor make( literal k ) exports all
    {
         value = datablock.makeString( k );
    }

    constructor make( String s ) exports all
```

```
{
     datablock k = s.value;
     value = k.clone( );
}

action equals( String s ) exports all
{
     datablock k = s.value;
     if ( value.equals( k ) )
         execute;
     else
         skip;
}

action differs( String s ) exports all
{
     datablock k = s.value;
     if ( value.equals( k ) )
         skip;
     else
         execute;
}

action geq( String s ) exports all // greater or equal
{
     datablock k = s.value;
     if ( value.geqString( k ) )
         execute;
     else
         skip;
}

action leq( String s ) exports all // less or equal
```

```
    {
         datablock k = s.value;
         if ( value.leqString( k ) )
             execute;
         else
             skip;
    }

    action gt( String s ) exports all // greater than
    {
         datablock k = s.value;
         if ( value.gtString( k ) )
             execute;
         else
             skip;
    }

    action lt( String s ) exports all // less than
    {
         datablock k = s.value;
         if ( value.ltString( k ) )
             execute;
         else
             skip;
    }
}
```

Apêndice B

Principais classes do Metamodelo da Virtuosi

Esse apêndice contém um diagrama com uma visão unificada das principais meta-classes do metamodelo da Virtuosi definido na Seção 4.1.

.

Apêndice C

Comandos e Testáveis de Sistema

Esse apêndice contém uma lista dos comandos e testáveis de sistema disponibilizados pelo ambiente Virtuosi.

C.1 Comandos de Sistema

Existem dois tipos de comandos nativos:

- comandos para a manipulação de bloco de dados;
- comandos para manipulação de índices.

Os comandos para manipulação de bloco de dados são divididos em:

- 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;

C.1.1 Comandos para a Manipulação de Bloco de Dados

Comandos de sistema para seqüência de dados binários neutros em relação ao tipo

CreateDB – causa a alocação de espaço em memória para o bloco de dados. O tamanho do bloco de dados definido pelo valor literal (interpretado como inteiro) passado como parâmetro ao comando.

CreateIndex – causa a criação de um índice que referencia uma posição da seqüência contígua de dados binários apontada por uma referência a bloco de dados.

Clone – retorna uma cópia exata da seqüência contígua de dados binários.

Clear – faz com que toda a seqüência contígua de dados binários receba o valor zero;

Length – retorna uma referência a índice com o valor do número total de elementos da seqüência contígua de dados binários.

Cardinality – retorna uma referência a índice cujo valor corresponda ao número de elementos da seqüência contígua de dados binários cujo valor seja um.

GetRange – retorna um novo bloco de dados contendo a seqüência contígua de dados binários entre os dois índices informados como parâmetro.

SetRange – faz com que a seqüência contígua de dados binários entre os dois índices informados como parâmetro receba o valor um.

ClearRange – faz com que a seqüência contígua de dados binários entre os dois índices informados como parâmetro receba o valor zero.

FlipRange – faz com que a seqüência contígua de dados binários entre os dois índices informados como parâmetro receba o valor inverso ao valor atual.

And – faz com que a seqüência contígua de dados binários receba para cada um de seus elementos o valor resultado de uma operação lógica de conjunção com o elemento na mesma posição do bloco de dados passado como parâmetro.

Or – faz com que a seqüência contígua de dados binários receba para cada um de seus elementos o valor resultado de uma operação lógica de disjunção inclusiva com o elemento na mesma posição do bloco de dados passado como parâmetro.

Xor – faz com que a seqüência contígua de dados binários receba para cada um de seus elementos o valor resultado de uma operação lógica de disjunção exclusiva com o elemento na mesma posição do bloco de dados passado como parâmetro.

SetBit – faz com que o elemento da seqüência contígua de dados binários na posição definida pelo índice passado como parâmetro receba o valor um.

ClearBit – faz com que o elemento da seqüência contígua de dados binários na posição definida pelo índice passado como parâmetro receba o valor zero.

FlipBit – faz com que o elemento da seqüência contígua de dados binários na posição definida pelo índice passado como parâmetro receba o valor inverso ao valor atual.

NextSetBit – retorna uma referência a índice com o valor da posição do primeiro elemento da seqüência contígua de dados binários cujo valor é um.

NextClearBit – retorna uma referência a índice com o valor da posição do primeiro elemento da seqüência contígua de dados binários cujo valor é zero.

Comandos de sistema para valores inteiros

StoreInteger – armazena no bloco de dados um valor no formato de valor inteiro (com complemento de dois) correspondente ao valor literal passado como parâmetro.

AddInteger – faz com que o bloco de dados tenha seu valor adicionado ao valor do bloco de dados passado como parâmetro. Deve-se notar que ambos os valores estão no formato de valores inteiros (com complemento de dois).

SubtractInteger – faz com que o bloco de dados tenha seu valor subtraído do valor do bloco de dados passado como parâmetro. Deve-se notar que ambos os valores estão no formato de valores inteiros (com complemento de dois).

MultiplyInteger – faz com que o bloco de dados tenha seu valor multiplicado pelo valor do bloco de dados passado como parâmetro. Deve-se notar que ambos os valores estão no formato de valores inteiros (com complemento de dois).

DivideInteger – faz com que o bloco de dados tenha seu valor dividido pelo valor do bloco de dados passado como parâmetro. Deve-se notar que ambos os valores estão no formato de valores inteiros (com complemento de dois).

Comandos de sistema para valores reais

StoreReal – armazena no bloco de dados referenciado um valor no formato de valor real correspondente ao valor literal passado como parâmetro.

AddReal – faz com que o bloco de dados tenha seu valor adicionado ao valor do bloco de dados passado como parâmetro. Deve-se notar que ambos os valores estão no formato de valores reais.

SubtractReal – faz com que o bloco de dados tenha seu valor subtraído do valor do bloco de dados passado como parâmetro. Deve-se notar que ambos os valores estão no formato de valores reais.

MultiplyReal – faz com que o bloco de dados tenha seu valor multiplicado pelo valor do bloco de dados passado como parâmetro. Deve-se notar que ambos os valores estão no formato de valores reais.

DivideReal – faz com que o bloco de dados tenha seu valor dividido pelo valor do bloco de dados passado como parâmetro. Deve-se notar que ambos os valores estão no formato de valores reais.

Comandos de sistema para valores caracter

StoreChar – armazena um valor no formato UNICODE correspondente ao valor literal passado como parâmetro.

ToUpper – faz com que o valor armazenado no bloco de dados passe a representar o caracter com caixa alta.

ToLower – faz com que o valor armazenado no bloco de dados passe a representar o caracter com caixa baixa.

Comandos de sistema para valores conjunto de caracter

Um conjunto de caracteres é normalmente chamado de *string*. Uma string é composta por um ou mais blocos de dados que armazenam um valor caracter.

MakeString – cria um bloco de dados com o tamanho igual ao tamanho de um bloco de dados utilizado para armazenar um caracter vezes o número de dígitos do valor literal

passado como parâmetro. Este comando armazena o valor binário no formato de caracter para cada um dos dígitos do parâmetro.

Concat – faz com que o valor da *string* passada como parâmetro (como um valor literal) seja adicionada – como uma nova seqüência de blocos de dados – ao final do bloco de dados.

C.1.2 Comandos para a Manipulação de Índices

Forward – faz com que o índice passe a apontar para posição da seqüência contígua de dados binários imediatamente posterior à atual.

Backward – faz com que o índice passe a apontar para a posição da seqüência contígua de dados binários imediatamente anterior à atual.

First – faz com que o índice passe a apontar para a primeira posição da seqüência contígua de dados binários.

Last – faz com que o índice passe a apontar para a última posição da seqüência contígua de dados binários.

SetIndex – faz com que o índice passe a apontar para a posição determinada pelo índice recebido como parâmetro.

AddIndex – faz com que o índice passe a apontar para a posição determinada pelo resultado da soma dele mais o índice recebido como parâmetro.

SubtractIndex – faz com que o índice passe a apontar para a posição determinada pelo resultado da subtração dele menos o índice recebido como parâmetro.

MultiplyIndex – faz com que o índice passe a apontar para a posição determinada pelo resultado da multiplicação dele pelo índice recebido como parâmetro.

DivideIndex – faz com que o índice passe a apontar para a posição determinada pelo resultado da divisão dele pelo índice recebido como parâmetro.

C.2 Testáveis de Sistema

Existem dois tipos de testáveis nativos:

- testáveis para a manipulação de bloco de dados;
- testáveis para manipulação de índices.

Os testáveis para manipulação de bloco de dados são divididos em:

- 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;

C.2.1 Testáveis para a Manipulação de Bloco de Dados

Testáveis de sistema para seqüência de dados binários neutros em relação ao tipo

SameBits – retorna um comando execute caso o bloco de dados passado como parâmetro seja igual ao bloco de dados existente. Caso contrário retorna um comando desvie.

Intersect – retorna um comando execute caso o bloco de dados passado como parâmetro tenha ao menos um dado binário igual na mesma posição. Caso contrário retorna um comando desvie.

IsEmpty – retorna um comando execute caso o bloco de dados existente tenha em todas as posições o valor zero. Caso contrário retorna um comando desvie.

GetBit – retorna um comando execute caso o dado binário na posição definida pelo índice passado como parâmetro seja igual a um. Caso contrário retorna um comando desvie.

Testáveis de sistema para valores inteiros

GreaterThanInteger – retorna um comando execute caso o bloco de dados passado como parâmetro armazene um valor inteiro maior que o valor armazenado pelo bloco de dados existente. Caso contrário retorna um comando desvie.

LowerThanInteger – retorna um comando execute caso o bloco de dados passado como parâmetro armazene um valor inteiro menor que o valor armazenado pelo bloco de dados existente. Caso contrário retorna um comando desvie.

GreaterOrEqualInteger – retorna um comando execute caso o bloco de dados passado como parâmetro armazene um valor inteiro maior ou igual que o valor armazenado pelo bloco de dados existente. Caso contrário retorna um comando desvie.

LowerOrEqualInteger – retorna um comando execute caso o bloco de dados passado como parâmetro armazene um valor inteiro menor ou igual que o valor armazenado pelo bloco de dados existente. Caso contrário retorna um comando desvie.

Testáveis de sistema para valores reais

GreaterThanReal – retorna um comando execute caso o bloco de dados passado como parâmetro armazene um valor real maior que o valor armazenado pelo bloco de dados existente. Caso contrário retorna um comando desvie.

LowerThanReal – retorna um comando execute caso o bloco de dados passado como parâmetro armazene um valor real menor que o valor armazenado pelo bloco de dados existente. Caso contrário retorna um comando desvie.

GreaterOrEqualReal – retorna um comando execute caso o bloco de dados passado como parâmetro armazene um valor real maior ou igual que o valor armazenado pelo bloco de dados existente. Caso contrário retorna um comando desvie.

LowerOrEqualReal – retorna um comando execute caso o bloco de dados passado como parâmetro armazene um valor real menor ou igual que o valor armazenado pelo bloco de dados existente. Caso contrário retorna um comando desvie.

Testáveis de sistema para valores conjunto de caracter

GreaterThanChar – retorna um comando execute caso o bloco de dados passado como parâmetro armazene um valor caracter maior que o valor armazenado pelo bloco de dados existente. Caso contrário retorna um comando desvie.

LowerThanChar – retorna um comando execute caso o bloco de dados passado como parâmetro armazene um valor caracter menor que o valor armazenado pelo bloco de dados existente. Caso contrário retorna um comando desvie.

GreaterOrEqualChar – retorna um comando execute caso o bloco de dados passado como parâmetro armazene um valor caracter maior ou igual que o valor armazenado pelo bloco de dados existente. Caso contrário retorna um comando desvie.

LowerOrEqualChar – retorna um comando execute caso o bloco de dados passado como parâmetro armazene um valor caracter menor ou igual que o valor armazenado pelo bloco de dados existente. Caso contrário retorna um comando desvie.

C.2.2 Testáveis para a Manipulação de Índices

EqualsIndex – retorna um comando execute caso o índice ou literal passado como parâmetro seja igual ao índice existente. Caso contrário retorna um comando desvie.

GreaterIndex – retorna um comando execute caso o índice ou literal passado como parâmetro seja maior que o índice existente. Caso contrário retorna um comando desvie.

LowerIndex – retorna um comando execute caso o index ou literal passado como parâmetro seja menor que o índice existente. Caso contrário retorna um comando desvie.

GreaterOrEqualIndex – retorna um comando execute caso o index ou literal passado como parâmetro seja maior ou igual ao índice existente. Caso contrário retorna um comando desvie.

LowerOrEqualIndex – retorna um comando execute caso o index ou literal passado como parâmetro seja menor ou igual ao índice existente. Caso contrário retorna um comando desvie.

Apêndice D

Utilização de bloco de dados

Esse apêndice contém um código fonte escrito na linguagem Aram referente de uma pequena classe que faz uso de referência a bloco de dados e índices para representar uma imagem de duas dimensões. Os três últimos métodos ainda não foram implementados.

```
class Imagem2D {
   datablock bitmap;
   composition Integer largura;
   composition Integer altura;

   constructor make(Integer largura, Integer altura) exports { all }
   {
       Integer zero = Integer.make(0);
       if (largura.gt(zero) && altura.gt(zero)) {
           this.largura = largura;
           this.altura = altura;
           Integer pontos = largura.clone();
           pontos.multiply(altura);
           length comprimento = pontos.makeLength();
           bitmap = datablock.make(comprimento);
       }
   }

   method Boolean set(Integer x, Integer y) exports { all }
   {
```

```
        Integer zero = Integer.make(0);
        if (x.geq(zero) && x.lt(largura) && y.geq(zero) && y.lt(altura)) {
            Integer p = x.clone();
            p.multiply(altura);
            p.add(y);
            index i = p.makeIndex(); // erro de execucao se p for inválido
            i.bind(bitmap); // erro de execucao se i for inválido

            i.set(); // altera o bit desejado!

            return Boolean.make(true);
        } else
            return Boolean.make(false);
    }

    method Boolean reset(Integer x, Integer y) exports { all }
    {
    //...
    }

    method Boolean flip(Integer x, Integer y) exports { all }
    {
    //...
    }

    action get(Integer x, Integer y) exports { all }
    {
    //...
    }
}
```

Referências Bibliográficas

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

[Atkinson, 1998] Atkinson, M. (1998). Providing orthogonal persistence for java. *Lecture Notes in Computer Science*, (1445):383–395. ECOOP'98.

[Birman, 1985] Birman, K. P. (1985). Replication and fault-tolerance in the ISIS System. *ACM Operating System Review*, 19(5). Proceedings of the 10th ACM Symposium on Operating System Principles.

[Blair et al., 2001] Blair, G., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitspatrick, T., Johnston, L., Moreira, R., Parlavantzas, N., and Saikoski, K. (2001). The design and implementation of Open ORB. In *IEEE Distributed Systems Online*.

[BORGES, 2004] BORGES, A. (2004). Uma linguagem de programação orientada a objetos mínima. 50 f. Trabalho de Conclusão de Curso (Graduação em Bacharelado Em Ciência da Computação) - Pontifícia Universidade Católica do Paraná. Orientador: Alcides Calsavara.

[Boykin et al., 1993] Boykin, J., Kirschen, D., Langerman, A., and Loverso, S. (1993). *Programming under Mach*. Addison-Wesley, Reading, MA.

[Briot et al., 1998] Briot, J.-P., Guerraoui, R., and Lohr, K.-P. (1998). Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329.

[Calsavara, 2000] Calsavara, A. (2000). Virtuosi: Máquinas virtuais para objetos distribuídos. Technical report approved on internal examination for career ascencionx, Pontifícia Universidade Católica do Paraná, Curitiba, Brazil. 99 pages in Portuguese.

[CESAR FILHO, 2004] CESAR FILHO, J. d. C. (2004). Mecanismo de mobilidade de objetos para a virtuosi. Master's thesis, PUCPR. 163 f. Dissertação (Mestrado em Informática Aplicada) - Pontifícia Universidade Católica do Paraná. Orientador: Alcides Calsavara.

[Eckel, 2003] Eckel, B. (2003). *Thinking in Java*. Pearson Education.

[Flanagan, 1997] Flanagan, D. (1997). *Java in a Nutshell*. O'Reilly.

[Franz, 1994] Franz, M. (1994). *Code-Generation On-the-Fly: A Key to Portable Software*. PhD thesis, ETH Zurich, Verlag der Fachvereine, Zurich.

[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.

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language*. Addison-Wesley, Reading, MA.

[Group, 2001] Group, O. M. (2001). Unified modeling language specification, version 1.4. *http://cgi.omg.org/docs/formal/01-09-67.pdf*.

[Harris, 2001] Harris, T. L. (2001). Extensible virtual machines. Technical report, University of Cambridge.

[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*, 6:109–133.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J. (1997). Aspect-oriented programming. ECOOP'97.

[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.

[Kolling, 1999] Kolling, M. (1999). The problem of teaching object-oriented programming, part 1: Languages. *Journal of Object-Oriented Programming*.

[Kon et al., 2002] Kon, F., Costa, F., Blair, G., and Campbell, R. H. (2002). The case for reflective middleware. *Communications of the ACM*, 45(6):33–38.

[Kon et al., 2000] Kon, F., Roman, M., Liu, P., Mao, J., T., Y., Magalhães, L., and Campbell, R. (2000). Monitoring, security, and dynamic configuration with the DynamicTAO reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware2000)*, pages 121–143.

[Kowaltowski, 1983] Kowaltowski, T. (1983). *Implementação de Linguagens de Programação*. Guanabara Dois.

[Maes, 1987] Maes, P. (1987). Concepts and experiments in computational reflection. *ACM SIGPLAN Notices*, 22(12):147–155. OOPSLA'87.

[Meyer, 1997] Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice Hall PTR, second edition.

[Mullender et al., 1990] Mullender, S. J., Rossum, G. v., Tanenbaum, A. S., Renesse, R. v., and Staveren, H. v. (1990). Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23:44–53.

[NUNES, 2001] NUNES, L. R. (2001). Estudos sobre a concepção de uma linguagem de programação reflexiva e correspondente ambiente de execução. *Anais do V Simpósio Brasileiro de Linguagens de Programação SBLP2001.*

[Oliva, 1998] Oliva, A. (1998). Guaraná: Uma arquitetura de software para reflexão computacional implementada em java. Master's thesis, Universidade Estadual de Campinas, Instituto de Ciência da Computação.

[OMG, 2001] OMG (2001). Omg unified modeling language specification. Technical report, OMG. Disponível em www.omg.org.

[Paepcke, 1993] Paepcke, A. (1993). User-level language crafting: Introducing the clos metaobject protocol. *Object-Oriented Programming: The CLOS Perspective*, pages 65–100.

[Parrington et al., 1995] Parrington, G. D., Shrivastava, S. K., Wheater, S. M., and Little, M. C. (1995). The design and implementation of Arjuna. *USENIX Computing Systems Journal*, 8(3).

[POOLE, 2001] POOLE, John; CHANG, D. T. D. M. D. (2001). Common warehouse metamodel: An introduction to the standard for data warehouse integration. *John Willey & Sons, Inc.*

[Riehle et al., 2001] Riehle, D., Fraleigh, S., Bucka-Lassen, D., and Omorogbe, N. (2001). The architecture of a uml virtual machine. ACM Press.

[Rist and Terwinllinger, 1995] Rist, R. S. and Terwinllinger, R. (1995). *Object Oriented Programming in Eiffel*. Prentice Hall.

[Santos, 2003] Santos, Hélio; Barros, R. F. D. (2003). Uma proposta para gerenciamento de metadados nos padrões xml e dtd em repositórios mof. *Anais do XVIII Simpósio Brasileiro de Banco de Dados*.

[Schill, 1993] Schill, A. B., M. M. U. (1993). Dc++ distributed object-oriented system support on top of osf dce. *Distributed Systems Engineering*, 1:112–125.

[Silberchatz and Galvin, 1998] Silberchatz, A. and Galvin, P. B. (1998). *Operating System Concepts*. Addison-Wesley, fifth edition.

[Stroustrup, 1986] Stroustrup, B. (1986). *The C++ Programming Language*. Addison Wesley, Reading, Massachusetts.

[Venkatasubramanian, 2002] Venkatasubramanian, N. (2002). Safe composability of middleware services. *Communications of the ACM*, 45(6):49–52.

[Venners, 1999] Venners, B. (1999). *Inside the Java 2 Virtual Machine*. McGraw-Hill.