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

Virtuosi: Máquinas Virtuais para Objetos Distribuídos

Alcides Calsavara

Trabalho apresentado em
concurso para professor titular

Fevereiro de 2000

"I do not see the object-oriented method as a mere fad; I think it is not trivial [...]; I know it works; and I believe it is not only different from but even, to a certain extent, incompatible with the techniques that most people still use today – including some of the principles taught in many software engineering textbooks. I further believe that object technology holds the potential for fundamental changes in the software industry, and that it is here to stay."

Bertrand Meyer

Resumo

Os sistemas de computação distribuída têm ganhado mais importância na última década e muita pesquisa tem sido feita para o seu aperfeiçoamento. Nessa área, nota-se que entre os sistemas de mais sucesso predomina a abordagem de construção de software baseada em orientação a objetos. De fato, o acelaramento no desenvolvimento de sistemas distribuídos tem contribuído para a maior disseminação de orientação a objetos. Neste contexto, a linguagem Java e seu ambiente de execução têm se destacado. Esse ambiente de execução, baseado em máquina virtual, tem demonstrado virtudes, especialmente portabilidade de código, mas também tem demonstrado algumas restrições, como desempenho e segurança. Esse fato tem motivado muita pesquisa no sentido de encontrar soluções alternativas a Java como ambiente de execução distribuída sobre plataformas heterogêneas. Paralelamente, o conceito de reflexão computacional tem sido fortemente pesquisado e aplicado em sistemas. Mais recentemente, a combinação de orientação a objetos, máquinas virtuais e reflexão computacional começou a ser explorada. Este trabalho propõe e especifica a arquitetura de um ambiente de execução que faz a combinação desses três conceitos. Essa arquitetura é denominada *Virtuosi*.

A especificação da *Virtuosi* inclui um modelo básico de objetos, um modelo de comunicação baseado em invocação de atividades de objetos, um modelo de comunicação baseado em notificação de eventos e um modelo de reflexão computacional que visa permitir a implementação dos demais mecanismos necessários para computação distribuída. A especificação apresentada neste trabalho deve servir como base para futuros estudos, extensões e detalhamento a fim de se chegar a uma implementação. Deve servir, também, como base para a criação de um ambiente de desenvolvimento de software apropriado para a *Virtuosi*.

Palavras-chaves: orientação a objetos, sistemas distribuídos, objetos distribuídos, máquina virtual, reflexão computacional.

Resumo

Os sistemas de computação distribuída têm ganhado mais importância na última década e muita pesquisa tem sido feita para o seu aperfeiçoamento. Nessa área, nota-se que entre os sistemas de mais sucesso predomina a abordagem de construção de software baseada em orientação a objetos. De fato, o acelaramento no desenvolvimento de sistemas distribuídos tem contribuído para a maior disseminação de orientação a objetos. Neste contexto, a linguagem Java e seu ambiente de execução têm se destacado. Esse ambiente de execução, baseado em máquina virtual, tem demonstrado virtudes, especialmente portabilidade de código, mas também tem demonstrado algumas restrições, como desempenho e segurança. Esse fato tem motivado muita pesquisa no sentido de encontrar soluções alternativas a Java como ambiente de execução distribuída sobre plataformas heterogêneas. Paralelamente, o conceito de reflexão computacional tem sido fortemente pesquisado e aplicado em sistemas. Mais recentemente, a combinação de orientação a objetos, máquinas virtuais e reflexão computacional começou a ser explorada. Este trabalho propõe e especifica a arquitetura de um ambiente de execução que faz a combinação desses três conceitos. Essa arquitetura é denominada *Virtuosi*.

A especificação da *Virtuosi* inclui um modelo básico de objetos, um modelo de comunicação baseado em invocação de atividades de objetos, um modelo de comunicação baseado em notificação de eventos e um modelo de reflexão computacional que visa permitir a implementação dos demais mecanismos necessários para computação distribuída. A especificação apresentada neste trabalho deve servir como base para futuros estudos, extensões e detalhamento a fim de se chegar a uma implementação. Deve servir, também, como base para a criação de um ambiente de desenvolvimento de software apropriado para a *Virtuosi*.

Palavras-chaves: orientação a objetos, sistemas distribuídos, objetos distribuídos, máquina virtual, reflexão computacional.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Na última década, especialmente, a comunidade científica das áreas relacionadas a pesquisa sobre sistemas de software tem sido fortemente motivada pelo rápido avanço na utilização de redes de computadores – com destaque para a Internet – a procurar soluções que permitam o desenvolvimento desses sistemas de forma mais rápida e que ao mesmo tempo aumente a sua confiabilidade. Muitas propostas foram feitas e, de fato, várias delas rapidamente tornaram-se produtos comerciais. Essas propostas, de maneira geral, procuravam contemplar os princípios de *computação distribuída* até então estabelecidos, tornando-os concretos, inevitavelmente, de acordo com algum paradigma de computação ou algum ambiente específico. Pode-se constatar que há um paradigma de computação que, apesar de ainda emergente em alguns meios, é predominante dentre as propostas que mais se destacaram: o paradigma de *orientação a objetos*. O emprego amplo de orientação a objetos na construção de sistemas de software distribuídos é dificilmente justificado por um fator isolado ou mesmo pelas qualidades do paradigma puramente; um estudo mais criterioso deve apontar uma conjunção de fatos que levaram à adoção, praticamente irreversível, desse paradigma pela maioria significativa da comunidade científica nessas e em futuras soluções. Não faz parte do escopo deste trabalho, entretanto, realizar tal estudo. Uma simples observação das soluções atualmente disponíveis para auxiliar na construção de sistemas distribuídos – novamente com destaque para a Internet – deixa evidente o amplo emprego de orientação a objetos, a começar pelo número de aplicações desenvolvidas em C++ e Java, um produto comercial. Reiterando, o crescente uso de Java não deve-se apenas ao fato de ser orientada a objetos – haja visto que C++ é ainda mais completa que Java com relação ao conjunto de princípios de orientação a objetos e não obteve o mesmo sucesso –, mas principalmente ao fator portabilidade provido

pelo ambiente de execução baseado em *máquinas virtuais*. Novamente, é difícil apontar um único fator isolado como responsável pelo sucesso de Java; tanto a técnica de orientação a objetos quanto a própria técnica de máquinas virtuais são ambas conhecidas há longa data. O fato é que a simples combinação das duas técnicas em um produto de forma consistente, no momento em que havia grande demanda para aplicações executáveis em plataformas heterogêneas, provou ser extremamente adequada e, por outro lado, resgatou a técnica de máquinas virtuais como uma solução promissora na construção de sistemas distribuídos. Atualmente, alguns grupos de pesquisa trabalham no desenvolvimento de máquinas virtuais que dão suporte aos requisitos básicos de sistemas, tais como persistência de objetos e coleta de lixo. Em paralelo a esse avanço no emprego de orientação a objetos devido ao retorno das máquinas virtuais causado pelo meio industrial, a comunidade científica tem proposto soluções para a construção de sistemas de software distribuídos baseadas em um conceito mais recente: a *reflexão computacional*[1]. Coincidentemente, o grande interesse recentemente notado por este conceito teve início a partir do momento em que passou a ser combinado com o paradigma de orientação a objetos. Atualmente, muitos grupos de pesquisa trabalham no sentido de empregar reflexão computacional para isolar as atividades de sistema das atividades *funcionais*, propriamente ditas, das aplicações distribuídas. As abordagens utilizadas por esses grupos são diversas, mas destacam-se pelo seu relativo sucesso as que propõem extensões a linguagens de programação orientada a objetos ou que criam bibliotecas específicas para essas linguagens. Essas propostas, entretanto, introduzem um grau de complexidade ainda elevado para os atuais padrões de conhecimento e técnicas de programação e, por isso, ainda são pouco utilizadas. O mais importante dessas pesquisas, todavia, não são os ambientes criados em si, mas a constatação de que o emprego de reflexão computacional é factível e viável como alternativa às duas abordagens normalmente adotadas na concepção de sistemas distribuídos: a solução baseada em bibliotecas, na qual há um alto grau de flexibilidade mas requer programadores altamente especializados e qualificados, e a solução integrativa, na qual os programadores ficam isentos dos aspectos de sistema mas as aplicações ficam atadas ao comportamento embutido na plataforma. Mais recentemente ainda, tem-se notado o surgimento de propostas que combinam orientação a objetos, máquinas virtuais e reflexão computacional, indicando uma nova tendência em pesquisas nesta área. Com essa abordagem espera-se conseguir uma solução que dê a mesma transparência provi-

[1] *Do inglês: computational reflection*

da pela abordagem integrativa e a mesma flexibilidade provida pela abordagem baseada em bibliotecas, sem requerer programadores com conhecimentos específicos sobre os sistemas de execução, contando ainda com a portabilidade das aplicações e o emprego de um paradigma de programação já estabelecido. Essas novas propostas, entretanto, encontram-se em estágio incipiente e a comunidade científica ainda deve experimentar e aprimorar a nova abordagem nos próximos anos.

1.2 Objetivo

Este trabalho define a arquitetura de um ambiente de execução distribuída de sistemas de software orientados a objetos. Neste ambiente, uma coleção de máquinas virtuais distribuídas por computadores heterogêneos cooperam na provisão de mecanismos que dão suporte aos sistemas e às correspondentes ferramentas de desenvolvimento. Cada máquina virtual hospeda uma parcela do universo de objetos e assegura que o comportamento de cada objeto esteja de acordo com a definição do correspondente tipo ou classe. O ambiente de execução permite que os sistemas distribuídos sejam concebidos como se fossem simples sistemas centralizados, isto é, como se todos os objetos residissem em um único espaço de endereçamento, em um único sistema operacional, e como se cada sistema fosse o único em execução a cada momento, sem qualquer concorrência entre sistemas ou entre partes de um mesmo sistema que executam em paralelo. Além de arranjar adequadamente os conceitos normalmente encontrados em sistemas orientados a objetos e em sistemas distribuídos, a arquitetura inclui o conceito de reflexão computacional, que serve como meio para a concretização de diversos mecanismos do ambiente de execução e que permite as aplicações redefinirem o comportamento desse ambiente, assim como o seu próprio comportamento.

Este trabalho restringe-se à especificação básica da arquitetura através de diversos modelos que formalizam os conceitos de forma integrada, introduzindo uma notação gráfica e uma terminologia para tal. Esses modelos, assim como a notação gráfica e a terminologia, devem servir de base para uma série de trabalhos complementares de detalhamento e implementação. Em uma segunda etapa, inclusive, devem servir de base para a criação de um ambiente de desenvolvimento de aplicações distribuídas.

1.3 Sinopse

A arquitetura especificada neste trabalho para um ambiente de execução distribuída de sistemas de software orientados a objetos é denominada *Virtuosi*, dando a conotação de harmonia e cooperação entre diversos componentes que, individualmente, executam suas tarefas específicas de maneira extremamente apropriada. Esse ambiente é composto de máquinas virtuais cooperantes que executam em computadores com sistemas operacionais e hardware padrão, apesar de heterogêneos, e conectados entre si por uma rede. As principais características da *Virtuosi* são: *(i)* suporte direto aos princípios de orientação a objetos, *(ii)* comunicação entre objetos distribuídos e *(iii)* reflexão computacional.

Uma máquina virtual é equipada com mecanismos para suportar todo o ciclo de vida de objetos que sejam instâncias de classes definidas como tipos de dados abstratos[2]. Assim, uma máquina virtual armazena um conjunto de definições de tipos – as classes – e um conjunto de instâncias desses tipos – os objetos – tal que a estrutura e o comportamento de um objeto sejam consistentes com o seu correspondente tipo. Além de preservar a semântica de encapsulamento de um tipo de dados abstrato, uma máquina virtual permite que tipos estabeleçam uma relação de hierarquia, com suporte ao mecanismo de herança entre tipos e à conseqüente propriedade de polimorfismo entre objetos. Uma máquina virtual, ainda, garante que as asserções, as pré-condições, as pós-condições e as invariantes definidas para um tipo nunca sejam violadas.

A *Virtuosi* permite que objetos se comuniquem através da invocação de métodos – tanto de modo síncrono quanto de modo assíncrono – e através do envio de eventos. O mecanismo de eventos provê uma abstração diferente da invocação de métodos, pois a comunicação passa a ser de muitos para muitos, ao invés de simplesmente um para um, e ocorre de forma assíncrona tal que cada objeto que recebe um evento pode acionar seu próprio método para tratamento. Além disso, a invocação assíncrona de métodos e o envio de eventos de forma assíncrona fornecem o princípio para a exploração de paralelismo em aplicações. Como os objetos residem em máquinas virtuais que executam em uma rede de computadores, a comunicação entre objetos distribuídos ocorre implicitamente com a resolução das referências entre estes na invocação de métodos e no envio de eventos através de máquinas distintas.

[2] *Do inglês: Abstract Data Types (ADT)*

Finalmente, a *Virtuosi* suporta o mecanismo de reflexão computacional: é possível descrever, controlar e adaptar o comportamento do sistema computacional através da associação de *metacomponentes* aos objetos. Pode-se definir metacomponentes para diversos fins, tanto para refletir sobre o comportamento dos aspectos funcionais de uma aplicação em si quanto para refletir sobre os seus aspectos de sistema, como persistência, concorrência, falhas, etc. A *Virtuosi* é recursiva pelo fato de todo metacomponente também poder ser um objeto. A primeira implicação disso é que os metacomponentes podem cooperar utilizando os próprios mecanismos de comunicação disponíveis para os objetos de aplicação. Dessa forma, pode-se definir um metacomponente que, para cumprir sua missão, coopere com outros metacomponentes, possivelmente associados a outros objetos e residentes em outras máquinas virtuais. Um exemplo típico é o metacomponente responsável pelo gerenciamento de uma transação atômica distribuída, que pode cooperar com metacomponentes de persistência, controle de concorrência e de recuperação associados aos diversos objetos envolvidos em uma transação. A segunda implicação é que é possível associar um metacomponente X a um metacomponente Y, tal que X reflita sobre o comportamento de Y e, da mesma forma, é possível associar um metacomponente Z ao metacomponente Y, tal que Z reflita sobre o comportamento de Y, e assim sucessivamente, caracterizando uma estrutura recursiva na qual cada nível de recursão representa um nível de reflexão sobre o nível que o antecede.

A *Virtuosi* diferencia-se de outras arquiteturas que também almejam o suporte à execução de sistemas de software orientados a objetos devido à forte integração dos princípios de orientação a objetos, dos mecanismos de comunicação entre objetos e do mecanismo de reflexão computacional. Os principais benefícios decorrentes dessa integração são a simplificação na compilação de programas escritos em linguagens orientadas a objetos, a otimização na geração de código executável, a simplificação na escrita desses programas com relação à comunicação entre objetos distribuídos, a portabilidade do código gerado pelos compiladores em função de executar em uma máquina virtual, a possibilidade de se construir um mesmo sistema de software utilizando-se diferentes linguagens, a possibilidade de se executar um sistema de software sobre plataformas heterogêneas de hardware e sistema operacional, a escalabilidade do sistema computacional, a simplificação na construção de software pela possibilidade de separação entre os aspectos funcionais de uma aplicação e os aspectos de sistema envolvidos, a possibilidade de se definir metacomponentes que especializem o comportamento padrão do sistema computacional e a conseqüente flexibilidade na configuração desse sistema computacional.

1.4 Organização do Texto

O restante deste texto está estruturado da seguinte forma:

Capítulo 2 sumariza os principais trabalhos sobre computação distribuída, orientação a objetos e reflexão computacional que serviram de base para a definição da *Virtuosi*, incluindo sistemas operacionais distribuídos, bibliotecas especializadas para sistemas distribuídos, *middleware* para sistemas distribuídos, máquinas virtuais e sistemas reflexivos.

Capítulo 3 especifica como objetos são organizados em máquinas virtuais distribuídas, como podem ser compostos para formar objetos mais complexos e como mantêm vínculos através de referências.

Capítulo 4 especifica o modelo básico de objetos, definindo os conceitos de estado e de métodos, sua representação em classes, a organização de classes em hierarquia, as propriedades de métodos que controlam o seu comportamento e os construtores básicos para a implementação de métodos.

Capítulo 5 especifica mais detalhadamente como os métodos podem ser especificados e as diferentes semânticas que se pode obter, principalmente considerando-se a hierarquia de classes e as formas de redefinição de métodos.

Capítulo 6 especifica o modelo de comunicação entre objetos distribuídos baseado na invocação de métodos, definindo o conceito de atividade, seus tipos e como estruturá-las para construir aplicações.

Capítulo 7 especifica o modelo de comunicação entre objetos distribuídos baseado no envio e tratamento de eventos.

Capítulo 8 especifica como os objetos das máquinas virtuais podem interagir com os dispositivos externos ao sistema, definindo os chamados objetos de fronteira.

Capítulo 9 especifica o modelo de reflexão computacional e discute as suas implicações nos modelos de objetos e de comunicação.

Capítulo 10 discute as principais conclusões do trabalho de definição da *Virtuosi* e apresenta uma série de possíveis trabalhos futuros, discutindo sobre os aspectos críticos de implementação da *Virtuosi* e como essa implementação pode ser realizada sobre plataformas de hardware e sistemas operacionais padrão.

Capítulo 2

Trabalhos Relacionados

Este Capítulo descreve resumidamente os principais trabalhos e conceitos que tiveram influência na concepção da *Virtuosi*.

2.1 Conceitos e Mecanismos

RPC O mecanismo de RPC (Remote Procedure Call), introduzido em [Nelson, 1981] e [Birrel and Nelson, 1984], é um marco importante em sistemas distribuídos, pois mostrou que é possível se construir aplicações com uma certa abstração com relação aos aspectos de sistema. O mecanismo serve de base para muitos dos sistemas em uso, tais como CORBA, DCE e Java RMI.

Orientação a Objetos A linguagem Simula-67 [Dahl and Nygaard, 1970] introduziu o paradigma de orientação a objetos não como fim, mas como meio de se representar entidades do mundo real que precisam ser simuladas. Essas entidades eram programadas através de classes que implementavam tipos de dados abstratos.

Máquina Virtual Segundo [Silberchatz and Galvin, 1998, pp. 75], o conceito de máquina virtual foi introduzido no sistema operacional VM da IBM. Basicamente, um conjunto de máquinas virtuais executam em um computador, dando a ilusão a cada usuário de ter o seu próprio processador. Uma das vantagens dessa abordagem é a segurança obtida no sistema com o isolamento de cada usuário. Por outro lado, não há qualquer suporte direto a compartilhamento entre aplicações. Isso somente é conseguido através de regiões de disco compartilhadas ou através de uma *rede virtual* – implementada em sofware sobre a rede física

– que conecta as máquinas virtuais e fornece um serviço de mensagens.

Reflexão Computacional O conceito de reflexão computacional, disseminado por [Maes, 1987], tem sido empregado nas mais diversas área de computação, com várias propostas de extensões a linguagens (principalmente a Smalltalk, C++ e Java). Esse conceito evidenciou a possibilidade de se construir sistemas utilizando-se uma abordagem alternativa às duas abordagens então mais conhecidas e até os dias de hoje predominantes: a abordagem de bibliotecas e a abordagem integrativa [Briot et al., 1998]. A reflexão computacional surgiu como uma esperança de se conseguir a mesma flexibilidade propiciada pela abordagem de bibliotecas e, simultaneamente, o mesmo nível de abstração e transparência provido pela abordagem integrativa. Outra vantagem ainda, seria a não necessidade de programadores altamente especializados em sistemas para o desenvolvimento de aplicações distribuídas.

2.2 Modelos de Objetos

Smalltalk-80 A linguagem Smalltalk-80 [Goldberg and Robson, 1983] deu um grande impulso no uso de orientação a objetos devido, principalmente, ao seu sofisticado suporte gráfico. O modelo de objetos é rigorosamente seguido nesta linguagem: todo e qualquer elemento de informação é um objeto, inclusive um simples valor inteiro. Uma das críticas mais freqüentes à linguagem, entretanto, é falta de tipos explícitos. Outra crítica desfavorável é o desempenho dos programas, pois são interpretados. Uma característica peculiar da linguagem é a sua reflexão computacional intrínseca através da introdução do conceito de metaclasses.

C++ A linguagem C++ [Stroustrup, 1986] é outro marco importante na evolução de orientação a objetos pois popularizou o paradigma tanto no meio científico quanto no meio industrial. Sua abordagem é evolutiva: simplesmente estende a linguagem C com a inclusão dos conceitos de orientação a objetos. É bastante utilizada até os dias de hoje devido à cultura criada e, principalmente, pela eficiência do código obtido.

Java A linguagem Java [Arnold and Gosling, 1996, Flanagan, 1997] surgiu como uma proposta de simplificação de C++ com o propósito de uso no chamado *software embarcado*, isto é, software que executa em equipamentos quaisquer – não necessariamente computadores – e que, por isso, em geral, dispõem de pouco poder de processamento e memória. Entretanto, com o início do uso intensivo da Internet, a linguagem passou a ser utilizada para fins de transporte de código através de plataformas heterogêneas e, com isso, ganhou uma nova dimensão. Atualmente, destaca-se pela sua simplicidade relativa de aprendizado, pela segurança propiciada pela inexistência de ponteiros, pela coleta de lixo automática, pela existência de muitas bibliotecas auxiliares, etc. Muito do trabalho sendo feito visa otimizar o código gerado *just in time* a fim de aumentar o domínio de aplicação da linguagem.

Eiffel A linguagem Eiffel [Meyer, 1997] procura ser rigorosa quanto ao conceito de tipo de dados abstrato, provendo meios de controlar asserções, pré-condição, pós-condição e invariantes. O modelo de objetos especifica, como Smalltalk, que todo elemento de informação seja um objeto. Entretanto, os compiladores e o ambiente de execução da linguagem estão otimizados para que isso não prejudique o desempenho das aplicações.

OMT O método de modelagem [Rumbaugh et al., 1994] foi muito importante na última década na popularização dos conceitos de orientação a objetos nas fases de análise e projeto de sistemas. O modelo de objetos utilizado é evolutivo, pois inclui muitos dos conceitos de modelos relacionais.

UML A linguagem UML [Rumbaugh et al., 1997, Eriksson and Penker, 1998] surgiu da união de três outras notações para sistemas orientados a objetos – a notação da OMT sendo uma delas. Os recursos da linguagem tem sido muito importantes para a consolidação do paradigma na construção de sistemas de software, muito embora haja muitas críticas com relação a falta de precisão semântica da linguagem.

2.3 Sistemas Operacionais Distribuídos

Amoeba O sistema operacional Amoeba [Mullender et al., 1990] foi desenvolvido com o objetivo de dar total transparência ao usuário final com relação à distribuição dos recursos do sistema pelos computadores da rede. Um usuário do sistema trabalha como se estive usando um simples sistema de *timesharing*, mas na realidade acessando recursos de diversos computadores. Esses recursos incluem servidores de processos, servidores de arquivos, servidores de diretório, etc. Amoeba utiliza o conceito de *microkernel* em sua arquitetura, o que significa que todos os recursos são configuráveis em cada computador. O ambiente de programação disponível segue o paradigma de orientação a objetos, disponibilizando uma linguagem própria (ORCA) para tal.

Mach O sistema operacional Mach [Accetta et al., 1986, Boykin et al., 1993] foi desenvolvido com os seguintes objetivos: fornecer uma base para a construção de outros sistemas operacionais, suportar espaço de endereçamento em larga escala, permitir acesso transparente aos recursos da rede, explorar paralelismo no sistema e nas aplicações e tornar o sistema Mach portável para uma grande coleção de máquinas. O sistema utiliza o conceito de *microkernel*, o qual fornece as seguintes abstrações às aplicações: processos, *threads*, objetos de memória, portas e mensagens.

2.4 Bibliotecas Especializadas

ISIS ISIS [Birman, 1985] implementa um conjunto de primitivas de comunicação entre processos. A semântica da comunicação é de grupo, com atomicidade no envio de mensagens e com ordenação de eventos. Foi um dos primeiros sistemas de suporte a distribuição a se destacar e é um dois mais importantes até os dias atuais. Entretanto, o seu uso requer o conhecimento de conceitos relativamente avançdos, isto é, os programadores de sistemas são indispensáveis no auxílio à construção de aplicações. Além disso, o paradigma de comunicação é a troca de mensagens entre processos, o que requer diversos se utilizado o paradigma de orientação a objetos nas aplicações.

Arjuna Arjuna [Parrington et al., 1995] é uma biblioteca que provê os mecanismos de comunicação, persistência, recuperação, controle de concorrência, transação e replicação de objetos distribuídos. O domínio principal de aplicações inclui as que necessitam ter alto grau de confiabilidade e tolerância a falhas. A versão atual está implementada em C++ – toda a interface de uso é orientada a objetos. Embora não haja qualquer extensão à linguagem, exige bom nível de conhecimento de sistemas distribuídos dos programadores de aplicações.

2.5 Máquinas Virtuais

Java Virtual Machine A máquina virtual de Java tem sido um dos elementos chaves no rápido crescimento da Internet, pois permite fácil transporte de código entre plataformas heterogêneas. O código gerado, entretanto, tem desempenho relativamente abaixo, mostrando-se inadequado para aplicações de maior porte. Uma solução muitas vezes adotada é o uso de compiladores *just in time*. A comunicação entre máquinas virtuais é facilitada por uma biblioteca de classes que implementa o protocolo RMI (Remote Method Invocation), uma versão modificada de RPC.

Jini O ambiente de execução [Jini, 1999] visa integrar componentes quaisquer que executem algum software nele *embarcados*. Esse ambiente faz uso da máquina virtual de Java: cada componente executa a máquina virtual e um conjunto de serviços provê os mecanismos neceseários para os componentes encontrem-se e comuniquem-se.

PJama O projeto PJama [Atkinson, 1998] visa fornecer persistência de objetos para aplicações Java de maneira ortogonal, isto é, para todos os tipos de dados e de maneira transparente. A abordagem adotada é basicamente modificar o interpretador de bytecode tal que qualquer alteração de estado seja detectada e o novo estado seja oportunamente salvo em memória estável. Nesse esquema, tudo o que a aplicação precisa acrescentar ao código normal é uma sinalização (através de *interfaces* apropriadas) de quais classes devem ter instâncias persistentes. Este projeto, embora restrito ao requisito persistência, mostra a viabilidade de se implementar mecanismos de sistema diretamente em máquinas

virtuais.

VVM O projeto VVM – Virtual Virtual Machines – tem por objetivo o desenvolvimento de um ambiente de execução único que suporte aplicações com diferentes tipos de *bytecode* [Folliot et al., 1997] [Folliot et al., 1998]. O seu objetivo principal é minimizar os problemas causados por heterogeneidade de plataformas nas aplicações distribuídas cujas prioridades são tratar adequadamente a distribuição de usuários e recursos, incluindo os conseqüentes requisitos de segurança.

Juice Juice [Franz, 1994, Kistler and Franz, 1997, Franz and Kistler, 1997] oferece um esquema alternativo a Java para a mobilidade de código sobre plataformas hetorêneas. Ao invés de transportar *bytecode*, Juice transporta a própria árvore de programa (representação obtida como resultado da primeira fase de compilação). Quando uma árvore de programa chega ao seu destino, um gerador de código apropriado para a plataforma gera o correspondente código nativo e o coloca em execução. Esse esquema oferece as seguintes vantagens em relação a Java: árvores de programa são menores que os correspodentes arquivos de *bytecodes*, o que torna o transporte mais rápido (o tempo gasto na geração de código nativo é compensado pelo tempo economizado no transporte); árvores de programa carregam informação semântica de maneira explícita, permitindo otimizações na geração de código e controle de segurança mais refinado; a execução final é mais rápida, mesmo quando comparado com compiladores *just in time* de Java (o motivo está na otimização conseguida com a informação semântica disponível na árvore de programa).

Guaraná O projeto Guaraná [Oliva, 1998] faz modificações na máquina virtual Java a fim de prover o mecanismo de reflexão computacional. Não há qualquer extensão à linguagem ou ao conjunto de instruções de bytecode: reflexão é disponibilizada aos programadores através de bibliotecas de classes apropriadas. Atualmente, há um trabalho em andamento [Oliva and Buzato, 1998] para a criação de metacomponentes básicos para a construção de aplicações distribuídas.

Alexandria Alexandria Digital Library Project, um projeto da UC Santa Barbara, utiliza uma plataforma para objetos distribuídos desenvolvida no próprio projeto

[Wu et al., 1997]. Uma dois principais requisitos da aplicação alvo do projeto é a disponibilidade de um mecanismo de migração de objetos para fins de distribuição de processamento. A plataforma para objetos distribuídos foi desenvolvida satisfaz esse requisito utilizando a Máquina Virtual Java como meio de distribuição de processamento, devido à independência de plataforma.

2.6 Middleware

CORBA CORBA [Soley and Kent, 1995] dá transparência às aplicações com relação a plataforma e linguagem. Entretanto, não permite migração de objetos, propriamente dito, pois, embora o estado possa ser transferido de uma plataforma para outra, a implementação do objeto é específica de cada plaforma [Wu et al., 1997]. A impossibilidade de migração de código, infelizmente, reduz severamente o nível de reuso. Além disso, o desenvolvimento de aplicações mais complexas exige bastante conhecimento dos programadores para o correto uso dos serviços especificados.

OSF DCE OSF DCE [DCE, 1992] é uma plataforma para sistemas distribuídas construída em cima de sistemas operacionais e hardware padrão. Os principais componentes da plataforma são o serviço de arquivos, o serviço de tempo, o serviço de diretório, o serviço de segurança, o suporte para RPC e autenticação e o suporte a *threads*. A plataforma segue uma abordagem mais evolutiva que revolucionária, pois preserva os sistemas operacionais existentes. Além disso, a própria interface de uso é mais conservadora, pois não faz uso de orientação a objetos.

CAPÍTULO 3

Organização de Objetos

Este Capítulo discute como objetos são alocados em um sistema distribuído constituído de máquinas virtuais e como estes objetos podem estabelecer vínculos (referências) entre si para fins de interação. Inicialmente define-se o conceito de identidade e, na seqüência, como se compõe e como se distribui objetos.

3.1 Identidade

Todo objeto possui uma identidade própria, isto é, alguma característica peculiar que nenhum outro objeto possui, o que permite sua identificação de maneira única no sistema. Neste trabalho, a identidade de um objeto é representada por um número inteiro para fins de legibilidade e facilidade de explicação dos exemplos. A Figura 3.1 ilustra a notação gráfica para representar um objeto com sua respectiva identidade. No exemplo, o objeto tem identidade 12. A representação de um objeto por um círulo divido em duas partes simboliza o fato de todo objeto conter um estado e um comportamento, isto é, um conjunto de métodos, conforme discutido no Capítulo 4. A implementação da noção de identidade pode ser feita de muitas formas, já bem divulgadas na literatura, como por exemplo a concatenação da identificação de local com uma marca de tempo.

Figura 3.1 Representação de um objeto com sua identidade

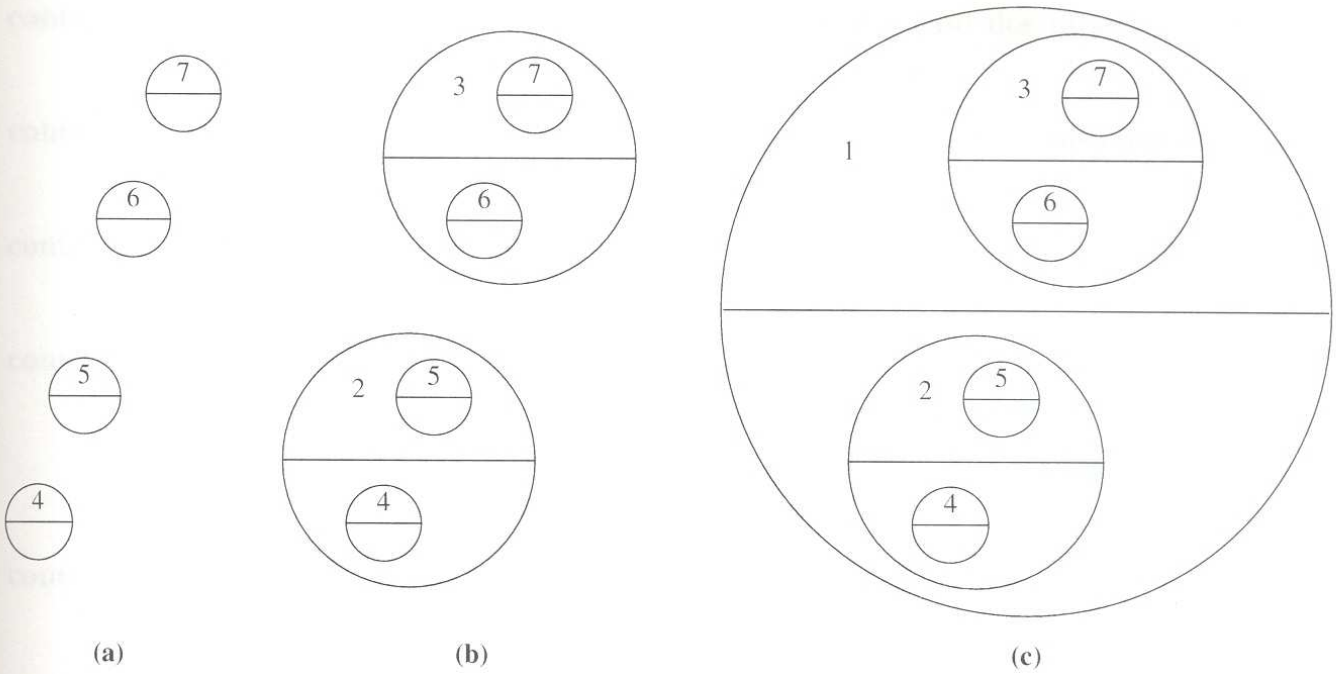


Figura 3.2 Composição de objetos

3.2 Composição

Objetos podem ser organizados de modo que, logicamente, uns contenham outros. Essa propriedade é comumente utilizada em técnicas de modelagem de sistemas para representar os chamados *objetos complexos* ou *objetos compostos*; nessas técnicas, a associação entre um objeto composto e seus componentes é denominada *agregação*. Seguem abaixo algumas definições de tipos de objetos que objetivam tornar explícitas todas as possibilidades de composição de objetos e estabelecer uma nomenclatura independente de técnica de modelagem. As definições são exemplificadas com os objetos ilustrados na Figura 3.2. Os tipos de objetos, com relação à propriedade de composição, são os seguintes:

atômico Um objeto que não contém outro objeto. Esse é o caso dos objetos 4, 5, 6 e 7, nas Figuras 3.2.a, 3.2.b e 3.2.c.

isolado Um objeto atômico não contido em outro objeto. Esse é o caso dos objetos 4, 5, 6 e 7 na Figura 3.2.a, somente.

contido Um objeto contido em outro objeto. Esse é o caso dos objetos 4, 5, 6, e 7 nas Figura 3.2.b e 3.2.c, e dos objetos 2 e 3 em na Figura 3.2.c.

contentor Um objeto que contém outro objeto. Esse é o caso dos objetos 2 e 3 nas Figuras 3.2.b e 3.2.c e do objeto 1 na Figura 3.2.c.

contentor global Um objeto contentor não contido em outro. Esse é o caso dos objetos 2 e 3 na Figura 3.2.b e do objeto 1 na Figura 3.2.c.

contentor local Um objeto contentor contido em outro. Esse é o caso dos objetos 2 e 3 na Figura 3.2.c.

contentor direto O objeto contentor direto de um objeto contido x é o objeto que contém x diretamente. Nas Figuras 3.2.b e 3.2.c, o contentor direto dos objetos 4 e 5 é o objeto 2 e o contentor direjto dos objetos 6 e 7 é o objeto 3. Na Figura 3.2.c, o contentor direto dos objetos 2 e 3 é o objeto 1.

contentor indireto Um objeto contentor indireto de um objeto contido x é um objeto que contém x indiretamente, isto é, contém direta ou indiretamente o objeto contentor direto de x. Na Figura 3.2.c, o objeto 1 é um contentor indireto dos objetos 4, 5, 6 e 7.

3.3 Distribuição

Todo objeto existe dentro do espaço de endereçamento de uma máquina virtual em particular, a qual situa-se em um computador real em particular. Um mesmo computador, por sua vez, pode abrigar diversas máquinas virtuais. Assim, os objetos podem estar distribuídos pelos diversos computadores de uma rede e, ainda, em um mesmo computador distribuídos por diversas máquinas virtuais. A alocação de objetos nas máquinas virtuais pode ocorrer de maneira totalmente arbitrária, com apenas uma restrição: um objeto contentor e todos os objetos nele contidos residem em um mesma máquina virtual. Essa restrição visa somente otimização, uma vez que há maior probabilidade de alta interação entre os objetos que formam um objeto contentor. A Figura 3.3 ilustra uma situação na qual uma coleção de objetos estão distribuídos por três computadores C_1, C_2 e C_3, conectados por uma rede[1]. Os computadores C_1 e C_2 abrigam as máquinas virtuais V_1 e V_2, respectivamente, enquanto que o computador C_3 abriga as máquinas virtuais V_3 e V_4. Cada máquina virtual abriga uma

[1] *A topologia de rede utilizada na representação é de barramento, mas isso não é relevante para a Virtuosi.*

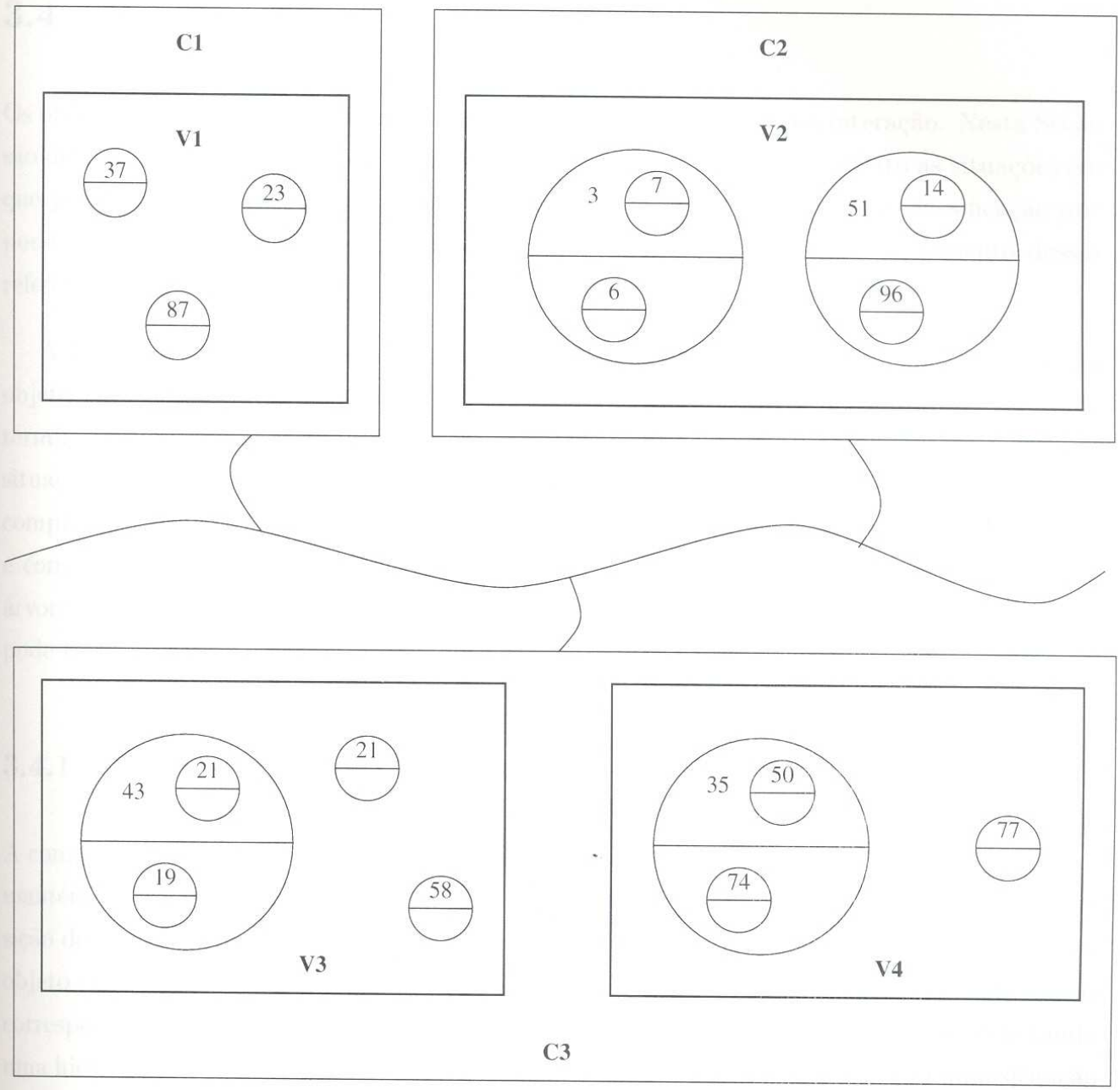


Figura 3.3 Objetos distribuídos na *Virtuosi*

coleção de objetos, dos diversos tipos definidos na Seção 3.2. Apesar da distribuição física dos objetos, potencialmente, todos podem se comunicar entre si, desde que as máquinas virtuais que os abrigam se conheçam. As formas de comunicação entre objetos são apresentadas nos Capítulos 6 e 7.

3.4 Referências para Objetos

Os objetos devem manter referências entre si a fim de possibilitar sua interação. Nesta Seção são definidos diferentes tipos de referências com o intuito de tornar explícito as situações em que pode ocorrer e em que situações são ilegais, além de estabelecer uma classificação que pode ser utilizada para fins de otimização no armazenamento e para o tratamento dessas referências.

A Figura 3.4 apresenta a notação gráfica utilizada para representar um referência de um objeto para outro: uma referência é simplesmente anotada através de uma linha tracejada terminada com uma seta indicando o objeto referenciado. Além disso, ilustra uma série de situações nas quais dois ou mais objetos mantêm referência entre si, de mais triviais a mais complexas, chegando até a grafos completos de referências. Um caso particular de grafo e comumente presente em aplicações é mostrado na Figura 3.4.h: objetos organizados em árvore ou hierarquia. Pode-se notar pelos exemplos (Figuras 3.4.d e 3.4.g), que um objeto pode também conter uma referência para si próprio.

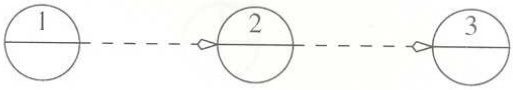
3.4.1 Referências em Composição

A composição lógica de objetos é materializada através de referências: um objeto contentor mantém referências para todos os objetos nele contidos. A Figura 3.5 ilustra uma composição de objetos representada primeiramente (3.5.a) utilizando a notação gráfica na qual um objeto contido aparece dentro de seu contentor e, em seguida (3.5.b), tornando explícitas as correspondentes referências. Pode-se observar a formação em árvore dos objetos, denotando uma hierarquia de composição de objetos. A composição de objetos caracteriza uma situação especial de referência. Por isso, há uma distinção entre dois tipos de referências e adota-se uma notação gráfica específica para cada tipo, a saber:

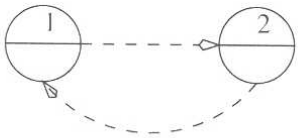
referência de associação Ocorre entre um objeto e outro desde que o primeiro não seja contentor do segundo. A correspondente notação gráfica diferencia-se pela seta vazia.

referência de composição Ocorre entre um objeto contentor e um objeto nele contido. A correspondente notação gráfica diferencia-se pela seta cheia.

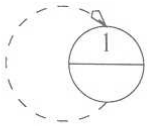
(a)



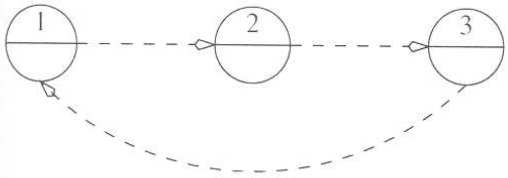
(b)



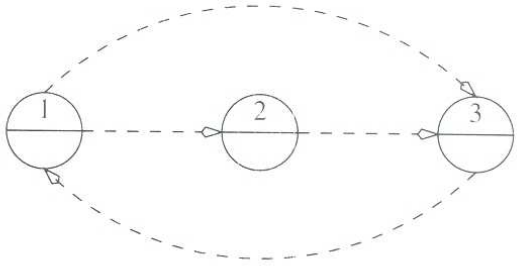
(c)



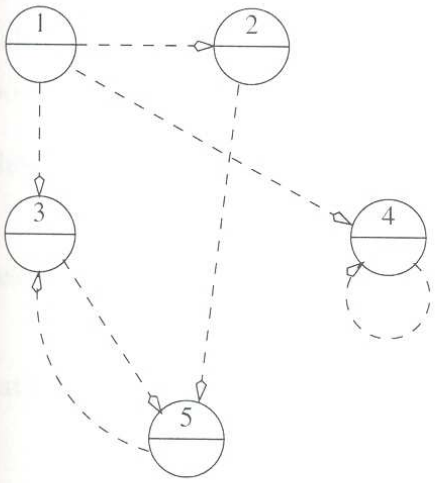
(d)



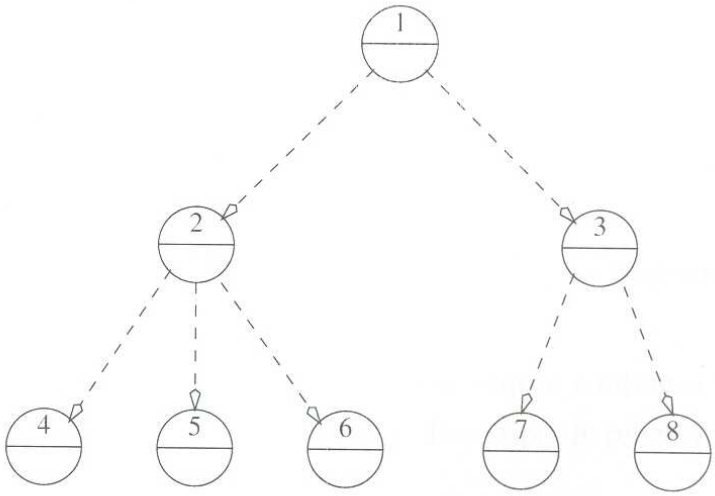
(e)



(f)



(g)



(h)

Figura 3.4 Representação de referências de associação de objetos

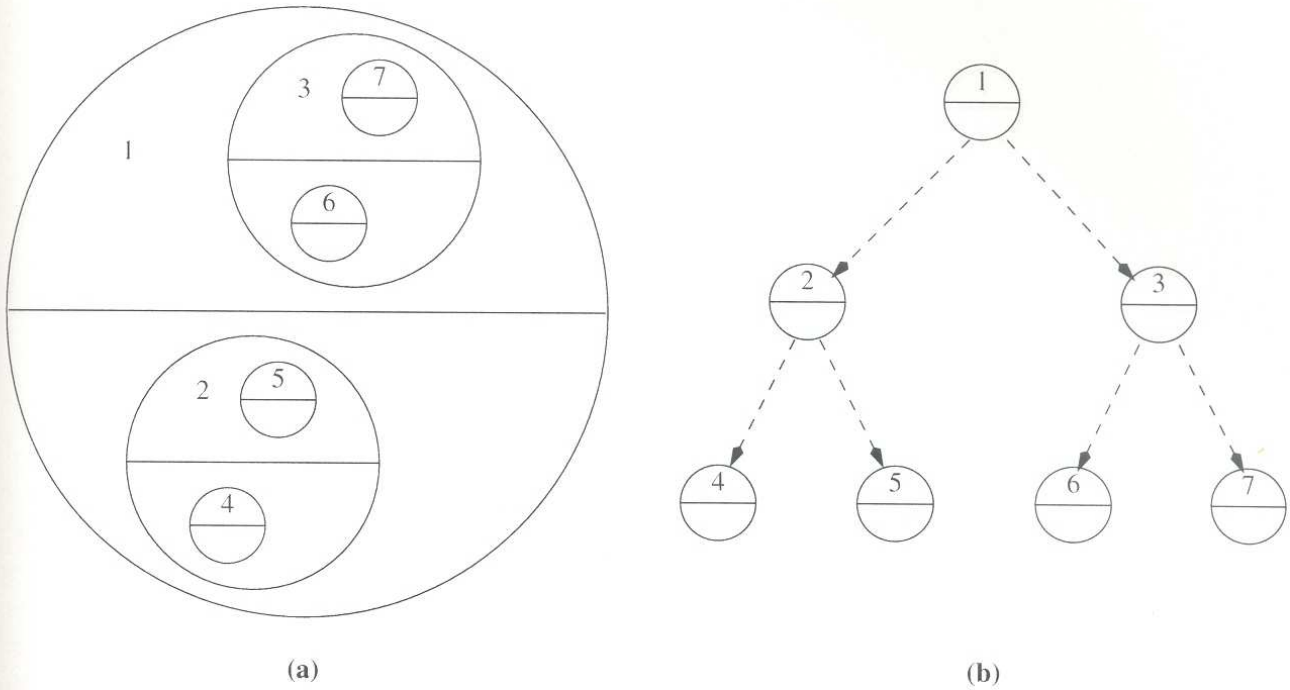


Figura 3.5 Representações alternativas para composição de objetos

A Figura 3.6 ilustra os dois tipos de referências: deve-se notar que os objetos 4 e 6 não fazem parte da hierarquia de composição e, por isso, referências para estes são de associação.

As referências em uma hierarquia de composição podem ocorrer não somente do objeto contentor para o objeto contido, mas opcionalmente no sentido inverso e entre objetos contidos no mesmo objeto contentor. Assim, as referências em uma hierarquia de composição podem ser classificadas da seguinte forma:

descensional Referência de um objeto contentor w para um objeto p (diretamente) contido em w. Esse tipo de referência é necessariamente de composição.

ascensional Referência de um objeto contido p para o seu correspondente objeto contentor direto ou indireto. Esse tipo de referência é de associação.

lateral Referência de um objeto p (diretamente) contido em um objeto contentor w para um objeto q também (diretamente) contido em w. Esse tipo de referência é de associação.

A Figura 3.7 ilustra um conjunto de objetos entre os quais os diferentes tipos de referência com relação à sua composição ocorrem. Todas as referências de composição (anotadas graficamente por uma seta cheia) são descensionais. Há três referências ascensionais: do objeto 4

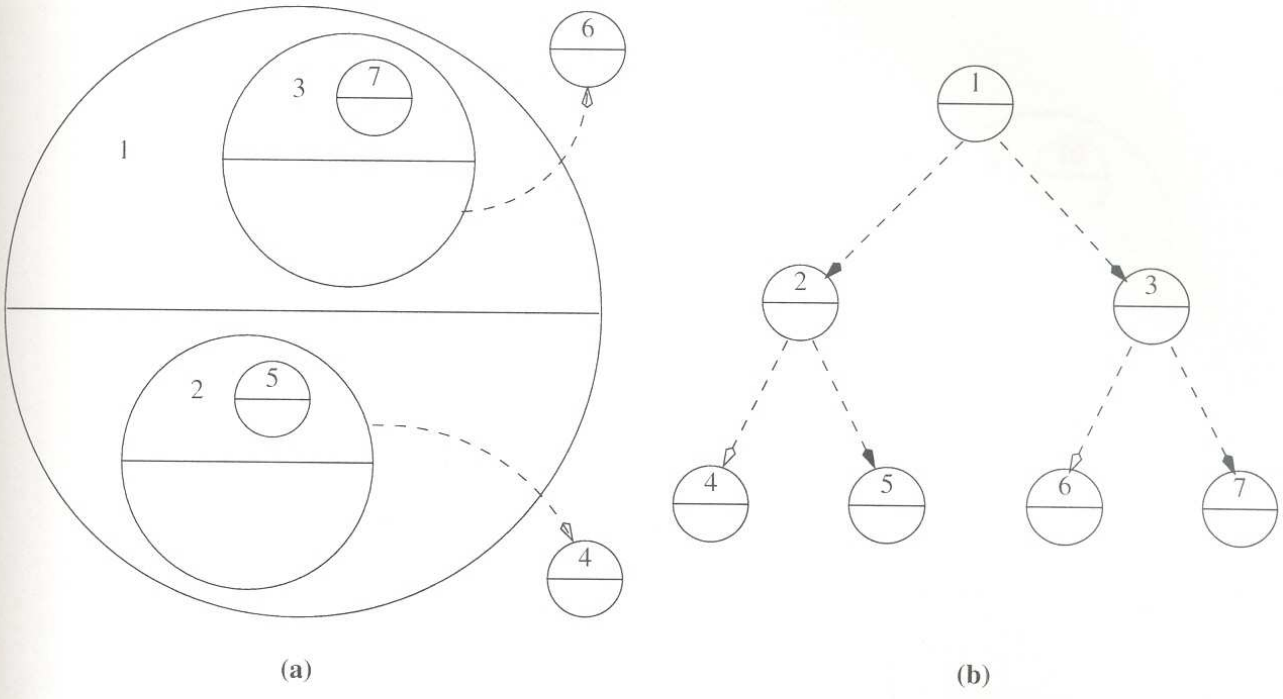
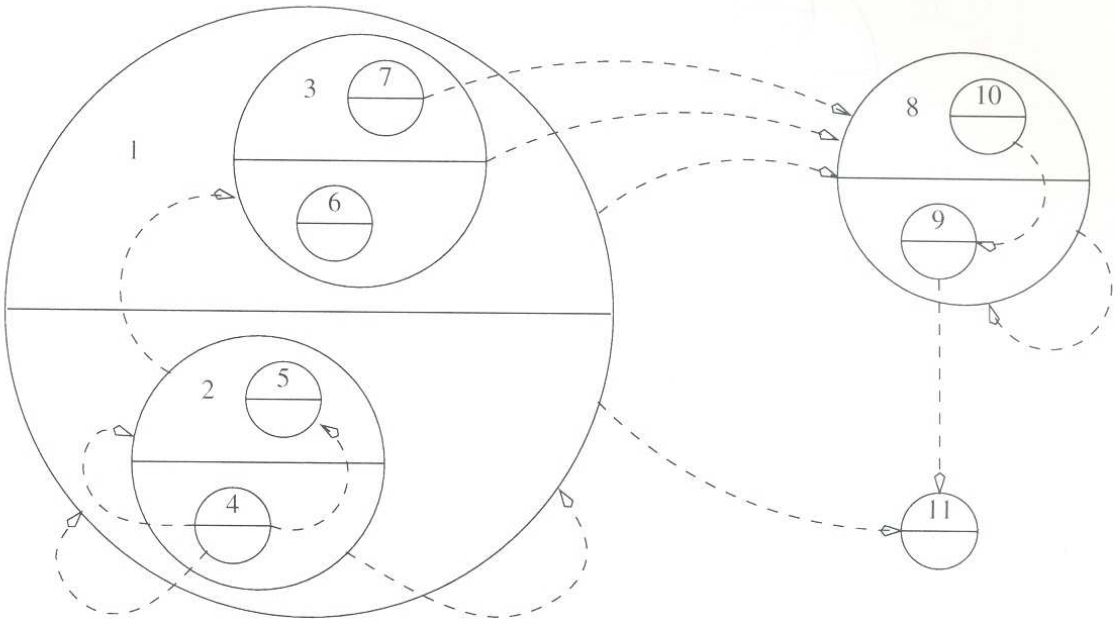


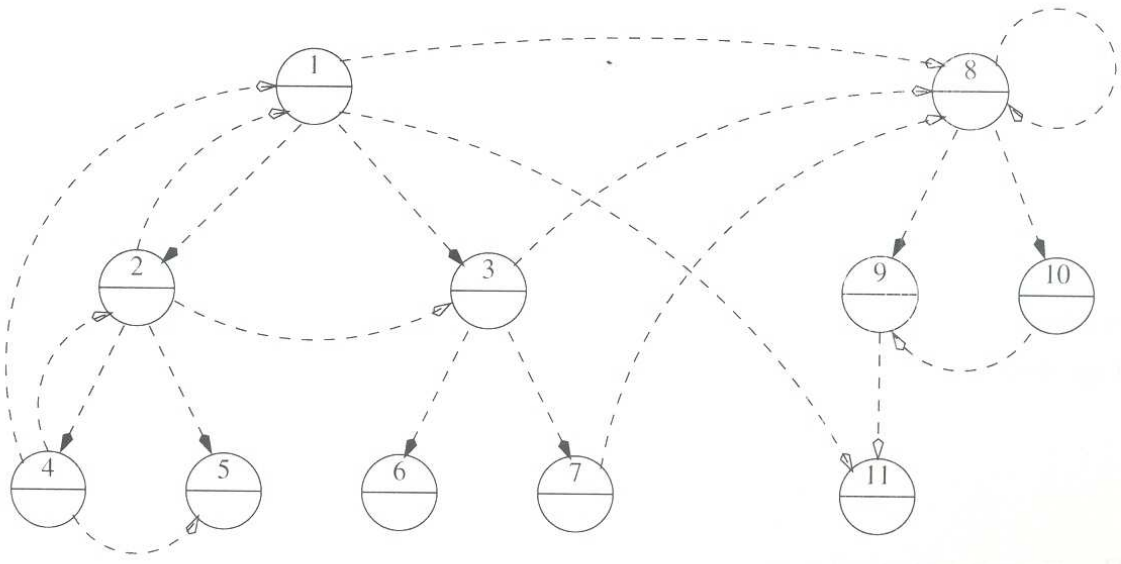
Figura 3.6 Referência de composição e referência de associação

para o objeto 2, do objeto 2 para o objeto 1 e do objeto 4 para o objeto 1. Há três referências laterais: do objeto 2 para o objeto 3, do objeto 4 para o objeto 5 e do objeto 10 para o objeto 9. As demais referências (tendo os objetos 8 e 11 como referenciados) ocorrem entre hierarquias de composição distintas e, por isso, não pertencem à classificação acima; essas referências são, entretanto, de associação e devem ser classificadas com relação à distribuição dos objetos, conforme discutido na Seção 3.4.2.

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 direto. A Figura 3.8 mostra um exemplo de referência ilegal para um objeto: o objeto 6, contido no objeto 3, é referenciado pelo objeto 2. Se tal referência fosse permitida, o princípio de encapsulamento de informação seria desrespeitado para o objeto 3, que contém o objeto 6, isto é, o objeto 6, embora contido no objeto 3, seria acessível por um objeto externo.



(a)



(b)

Figura 3.7 Referências em objetos compostos

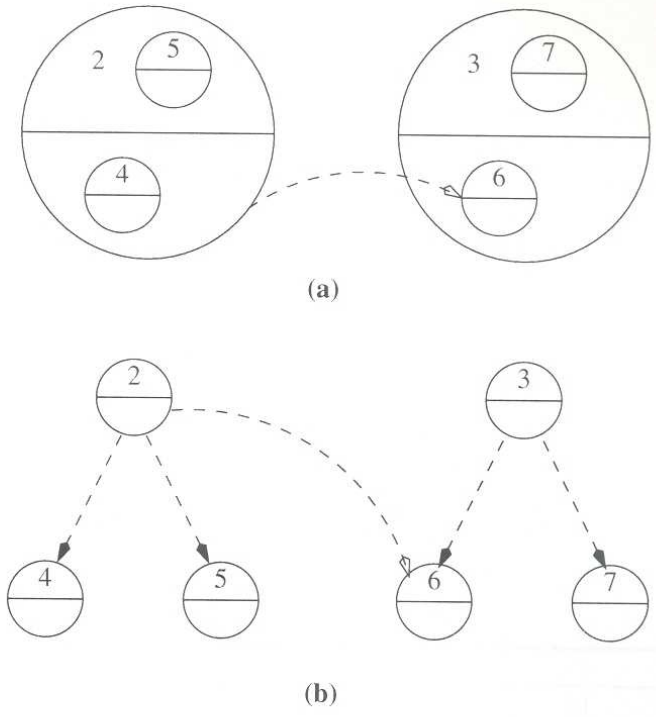


Figura 3.8 Referência ilegal para um objeto

3.4.2 Referências em Distribuição

A possibilidade de alocação de objetos em máquinas virtuais situadas em computadores distribuídos conduz a distintos tipos de referências entre objetos, a saber:

vicinal Referência de um objeto x para um objeto y isolado ou contentor global que não contém (direta ou indiretamente) x, quando x e y residem na mesma máquina virtual.

remota Referência de um objeto x para um objeto y isolado ou contentor global que não contém (direta ou indiretamente) x, quando x e y não residem na mesma máquina virtual.

circular Referência de um objeto contentor para si próprio.

A Figura 3.9 ilustra uma situação na qual esses tipos de referência ocorrem. Há três referências vicinais: do objeto 3 para o objeto 51, do objeto 43 para o objeto 58 e do objeto 35 para o objeto 77. Há sete referências remotas: do objeto 3 para os objetos 23, 87, 21 e 35, do 43 para os objetos 37, 87 e 35. Há uma referência circular: no objeto 35.

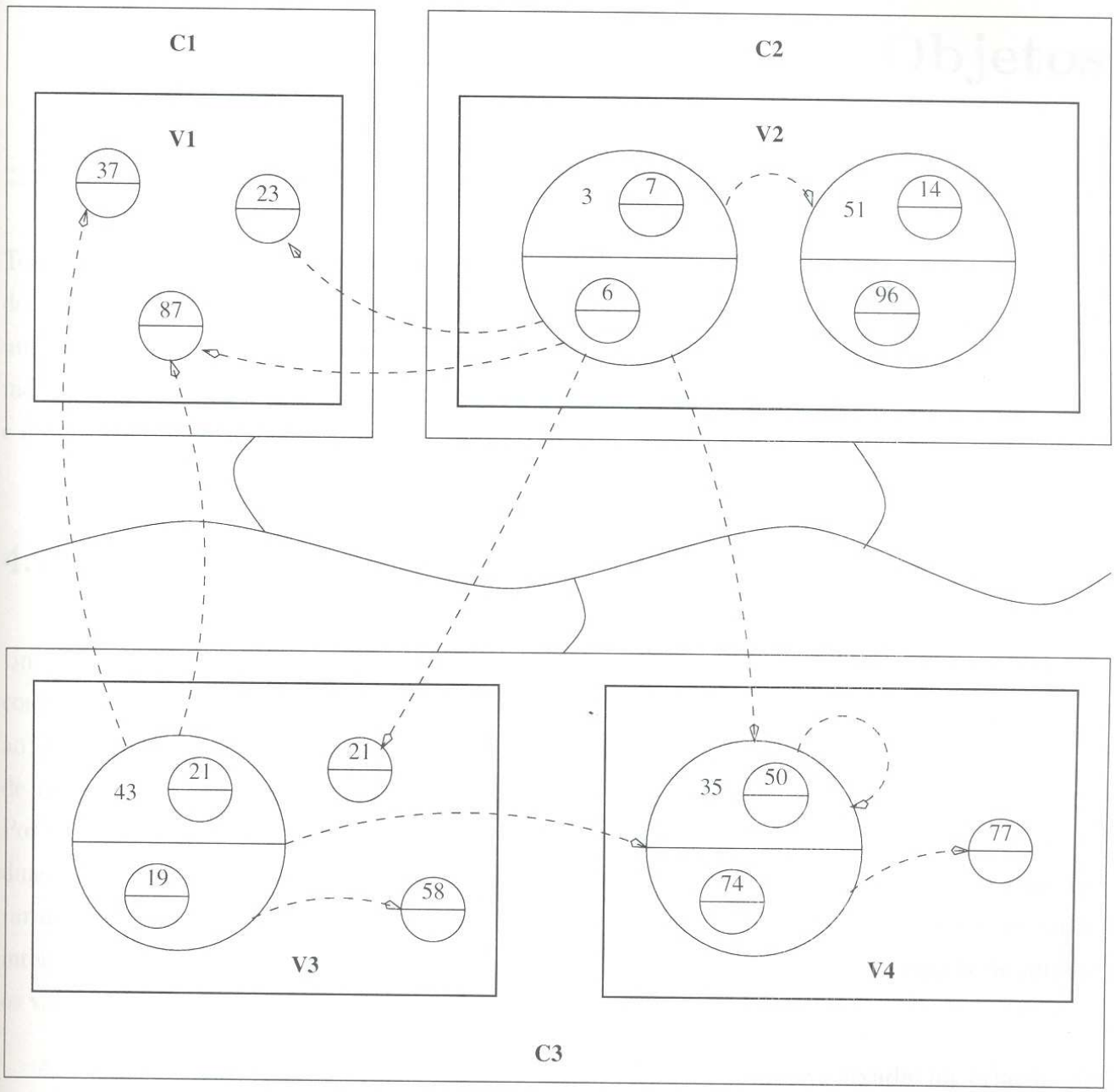


Figura 3.9 Referências em objetos distribuídos

Capítulo 4

Modelo Básico de Objetos

Todo objeto possui um estado e um conjunto de métodos que determinam a semântica do estado. O modelo básico de objetos apresentado neste Capítulo define o conteúdo de um estado e as como métodos são implementados; o Capítulo 5 discute em detalhe como os métodos são especificados. Este modelo de objetos baseia-se nas definições de objeto atômico e objeto contentor apresentadas no Capítulo 3.

4.1 Objeto Atômico

Um objeto atômico é constituído por um conjunto de n blocos de dados $b_1, ..., b_n$ e um conjunto de k métodos $m_1, ..., m_k$ ((Figura 4.1.a) ¬o conjunto de blocos de dados corresponde ao estado do objeto. Cada bloco de dados é uma série logicamente contígua de palavras de memória. O significado de cada bloco de dados é definido pelos métodos do objeto. Por exemplo, um objeto atômico pode armazenar um valor inteiro utilizando um bloco de duas palavras de memória. Nesse caso, um método para adicionar um outro valor inteiro (armazenado em outro objeto atômico, também utilizando um bloco de dados) ao valor inteiro deste objeto deve conhecer a convenção utilizada na representação binária de ambos os valores.

A quantidade de palavras em um bloco de dados é inicialmente fixada na criação do objeto, mas pode se alterar durante a vida do objeto, dependendo da sua semântica. Por exemplo, um valor inteiro pode ser inicialmente representado utilizando-se um bloco de duas palavras, mas se este vier a ter um valor que exija mais palavras para sua representação, digamos quatro palavras, então o bloco deverá ser expandido de acordo. Outro exemplo

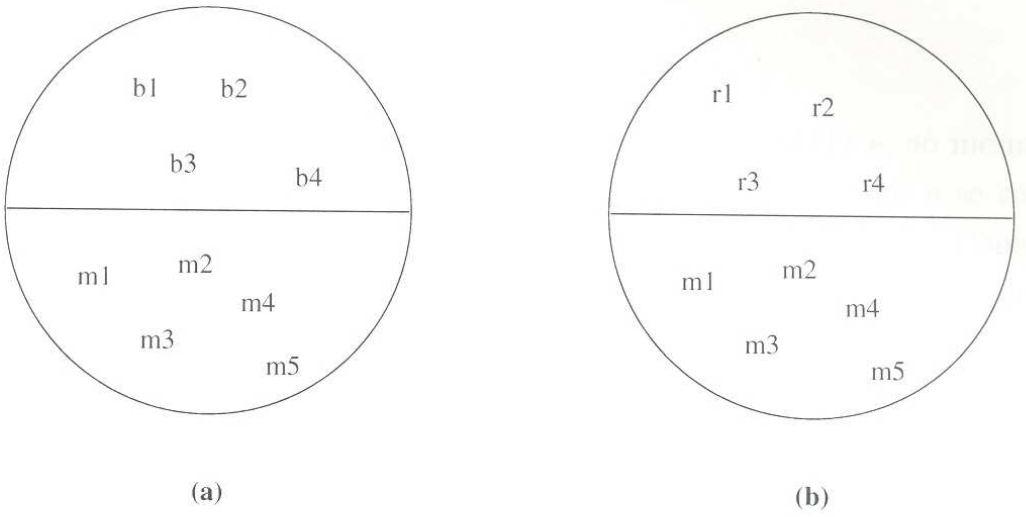


Figura 4.1 Constituição de um objeto atômico e de um objeto contentor

é a criação de um objeto da classe STRING que inicialmente armazene uma seqüência de n caracteres, mas que no decorrer de sua vida passe a ter $m > n$ caracteres – a diferença $m - n$ de caracteres implicará em uma expansão no correspondente bloco de dados usado para armazenamento. De forma análoga, o tamanho de um bloco de dados pode diminuir.

4.2 Objeto Contentor

Um objeto contentor é constituído por um conjunto de n referências $r_1, ..., r_n$ e um conjunto de k métodos $m_1, ..., m_k$ (Figura 4.1.b) – o conjunto de referências corresponde ao estado do objeto. A quantidade de referências nesse conjunto é fixada na criação do objeto e permanece a mesma durante toda a vida do objeto. Um acesso a uma referência pode ter os seguintes propósitos:

- modificar a referência: o objeto referenciado passa a ser outro
- copiar a referência: outra referência passa a existir para o objeto referenciado
- invocar um método do objeto referenciado

4.3 Classes

Uma classe é a implementação de um tipo de dados abstrato (ADT) e, ao mesmo tempo, designa o conjunto de todos os objetos que possuem estrutura (estado) e se comportam (métodos) segundo esse tipo, isto é, o conjunto de *instâncias* da classe. Como existem objetos atômicos e objetos contentores, analogamente as classes também podem ser atômicas ou contentoras, de forma exclusiva, tal que as instâncias de uma classe atômica são objetos atômicos e as instâncias de uma classe contentora são objetos contentores.

4.4 Herança

Duas classes podem estabelecer um relacionamento de herança entre si, tal que os métodos da classe *herdeira* ou *descendente* podem acessar tanto o estado quanto os métodos definidos pela classe *ancestral*, sem qualquer restrição. Em outras palavras, todas as definições de estado e de métodos constantes na ancestral são válidas para a herdeira. Como um objeto é atômico ou contentor, não é permitido estabelecer um relacionamento de herança entre uma classe contentora e uma classe atômica. Assim, um objeto atômico que seja instância de uma classe herdeira terá como estado o conjunto de blocos de dados definido pela ancestral unido com o conjunto de blocos de dados definido pela própria herdeira, enquanto que um objeto contentor que seja instância de uma classe herdeira terá como estado o conjunto de referências definido pela ancestral unido com o conjunto de referências definido pela própria herdeira. Com relação aos métodos, não importa se o objeto é atômico ou contentor – um objeto que seja instância de uma classe herdeira terá como métodos o conjunto definido pela ancestral unido com o conjunto definido pela herdeira.

Uma classe pode ter apenas uma ancestral direta[1], mas pode ter muitas descendentes, 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 (DAG) – uma árvore – de classes, no qual a propriedade de herança é transitiva, isto é, uma

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

classe herdeira assimila as definições de estado e de métodos de ancestrais diretas e indiretas também. Como em um relacionamento de herança podem ocorrer duas classes atômicas ou duas classes contentoras, mas nunca uma classe atômica com uma classe contentora, em uma dada hierarquia de classes existem somente classes atômicas ou somente classes contentoras. O conjunto de todas as classes, portanto, fica organizado como um conjunto de árvores ou de *hierarquias* disjuntas, sendo que algumas hierarquias são de classes atômicas e outras de classes contentoras.

A transitividade da propriedade de herança faz com que todo objeto tenha seu estado e conjunto de métodos definidos por uma seqüência de classes relacionadas através de herança. Pode-se dizer que cada classe da seqüência acrescenta uma nova *camada* de definições ao objeto. Assim, o estado e o conjunto de métodos de um objeto são estratificados numa quantidade n de camadas, numeradas seqüencialmente de 0 a $n-1$, tal que a classe correspondente à camada de nível k é herdeira direta da classe correspondente à camada de nível $k-1$. O número de seqüência de cada camada é dito ser o *nível* da camada. Da mesma forma, o nível da camada à qual pertence um método (referência ou bloco de dados) é dito ser o nível deste método (referência ou bloco de dados). Um método (referência ou bloco de dados) de nível k é acessível pelos métodos de nível $n \geq k$, mas não é acessível pelos métodos de nível $j < k$. A Figura 4.2.a ilustra um objeto atômico estratificado em duas camadas, contendo os métodos $m1, ..., m3$ e blocos de dados $b1, ..., b4$ nas respectivas camadas: o nível de $m2$ é 0, o nível de $m3$ é 1, o nível de $b1$ é 0, o nível de $b3$ é 1, etc. A Figura 4.2.b ilustra um objeto contentor estratificado em quatro camadas, contendo os métodos $m1, ..., m7$ e referências $r1, ..., r7$ nas respectivas camadas: o nível de $m2$ é 0, o nível de $m5$ é 2, o nível de $r4$ é 1, o nível de $r6$ é 3, etc.

4.5 Polimorfismo e Compatibilidade

Outra conseqüência da transitividade da propriedade de herança é que um objeto que pertence a uma classe também pertence a todas as suas ancestrais, pois seu estado e métodos implementam os tipos abstratos de dados correspondentes a estas classes. Por outro lado, como uma classe pode ter várias herdeiras, quando se diz que um objeto é instância de uma classe pode, na verdade, ser instância de qualquer uma de suas herdeiras. Ou seja, uma

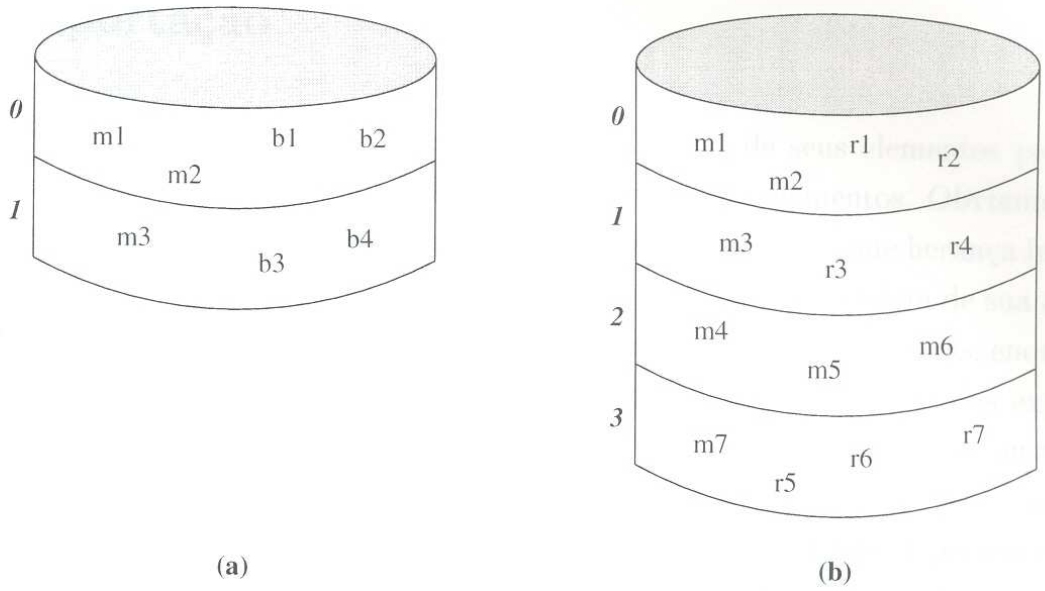


Figura 4.2 Estratificação de um objeto atômico e de um objeto contentor

referência de uma certa classe ou tipo pode ter como alvo instâncias de distintas classes, desde que sejam herdeiras (diretas ou indiretas) da classe à qual pertence a referência. Essa propriedade de uma referência poder ter como alvo objetos de classes distintas denomina-se *polimorfismo*. E, pelo fato de se poder assinalar a uma referência de uma certa classe um objeto que seja instância de qualquer uma de suas herdeiras, diz-se que uma classe herdeira é *compatível* com suas ancestrais diretas e indiretas. Equivalentemente, diz-se que o tipo correspondente à classe herdeira é compatível com o tipo correspondente à classe ancestral.

A compatibilidade entre classes permite ainda que sejam realizadas certas operações entre objetos de duas classes compatíveis. Por exemplo, pode-se copiar o estado de um objeto O_1 da classe C_1 para outro objeto O_2 da classe C_2, desde que C_1 seja compatível com C_2 – nesse caso, é possível que parte do estado de O_1 seja ignorado na operação de cópia (essa parte corresponde correspondente às classes herdeiras de C_2). O mesmo princípio pode ser empregado em operações de comparação entre estados de objetos e em operações aritméticas envolvendo objetos cujos estados representam valores numéricos inteiros e reais.

4.6 Exportação

Uma classe, denominada *fornecedora* pode exportar alguns de seus elementos para outras classes, denominadas *clientes*, tal que estas tenham acesso aos elementos. Obviamente, uma classe somente precisa ser cliente de outra se não for herdeira dela, já que herança implica em acesso total e irrestrito. Os elementos exportáveis de uma classe dependem de sua categoria: uma classe contentora pode exportar tanto seus métodos como suas referências, enquanto que uma classe atômica somente pode exportar seus métodos. As semânticas das exportações são definidas na seqüência e exemplificadas através do conjunto de objetos ilustrados na Figura 4.3, onde os objetos 2 e 5 pertencem à classe atômica A, o objeto 6 pertence à classe contentora D, o objeto 8 pertence à classe contentora C e o objeto 4 pertence à classe contentora B. Somente os elementos exportados pela respectiva classe são mostrados: a classe A exporta o método m_2 para as classes C e D, a classe D exporta o método m_3 e a referência r_3 para a classe C e a classe C exporta o método m_1 e as referências r_1 e r_2 para a classe B.

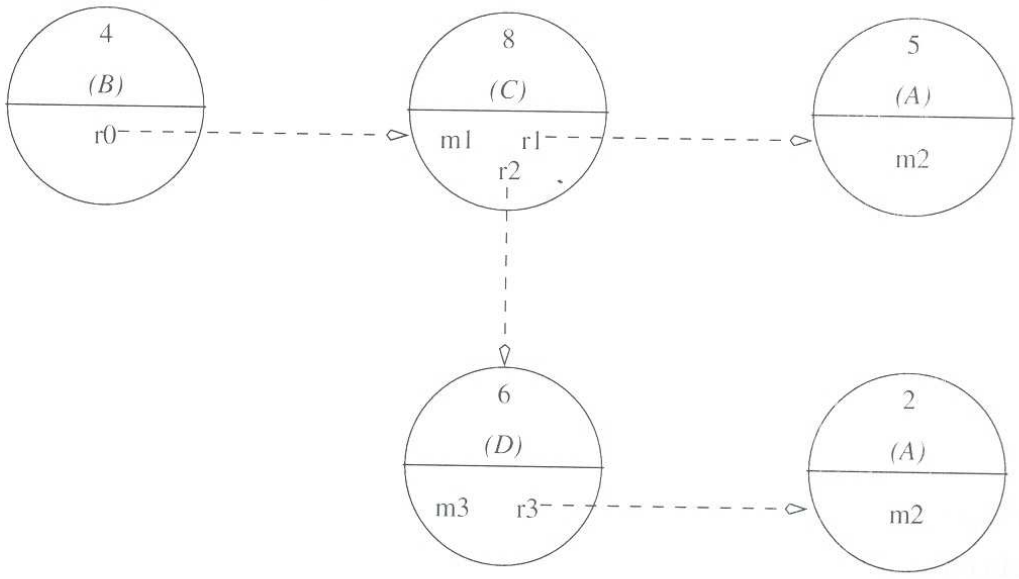


Figura 4.3 Exemplo de exportação de referências e métodos

- A exportação de um método permite que uma instância da classe cliente invoque o método de uma instância da classe fornecedora. Por exemplo, na Figura 4.3, o objeto 4 pode invocar o método m_1 do objeto 8 através da referência r_0, o objeto 8 pode

invocar o método m_2 do objeto 5 através da referência r_1 e pode invocar o método m_3 do objeto 6 através da referência r_2 e o objeto 6 pode invocar o método m_2 do objeto 2 através da referência r_3.

- A exportação de uma referência permite que uma instância da classe cliente acesse qualquer elemento exportado pela classe do objeto alvo da correspondente referência em uma instância da classe fornecedora. Por exemplo, na Figura 4.3, o objeto 4 pode invocar o método m_2 através das referências r_0 (do objeto 4 para o objeto 8) e r_1 (do objeto 8 para o objeto 5), o objeto 4 pode invocar o método m_2 do objeto 2 através das referências r_0 (do objeto 4 para o objeto 8), r_2 (do objeto 8 para o objeto 6) e r_3 (do objeto 6 para o objeto 2).

4.7 Constantes

Uma classe pode definir partes do correspondente estado como constante, isto é, com valor imutável depois que for feita a primeira atribuição. Uma classe atômica pode definir um bloco de dados como constante, enquanto que uma classe contentora pode definir uma referência como constante. Essa primeira atribuição pode ser feita na própria definição da classe, o que implica que toda instância da classe terá o mesmo valor para o elemento constante, ou durante o ciclo de vida de cada objeto, o que permite que cada objeto tenha um valor distinto para o elemento constante.

4.8 Invariantes

Uma classe define um conjunto de invariantes, isto é, um conjunto de condições que devem ser satisfeitas durante o ciclo de vida de todo e qualquer objeto que seja instância da classe. Uma invariante é uma expressão *booleana* cujos termos envolvem referências, blocos de dados e métodos acessíveis pela classe, operadores de comparação, operador de negação e conjunções *E* e *OU*.

4.9 Pré-condição e Pós-condição

Um método tem associada uma condição (expressão booleana construída da mesma forma que uma invariante) que deve ser satisfeita antes de sua execução – a pré-condição – e outra que deve ser satisfeita ao término de sua execução – a pós-condição.

4.10 Atividades

Uma atividade de um objeto corresponde à execução de um de seus métodos. O Capítulo 6 discute em detalhe como atividades são iniciadas, quais os tipos de atividades com relação a sincronismo e como atividades concorrentes são tratadas em um objeto.

4.11 Notificação de Eventos

Todo objeto possui uma tabela dos eventos que pode gerar em seus métodos e, para cada evento, uma correspondente lista de atividades de objetos interessadas em sua ocorrência. Assim, sempre que ocorre um evento (durante alguma atividade do objeto) o objeto deve notificar todas as atividades de objetos que registraram seu interesse pelo evento. Esse procedimento é discutido em detalhe no Capítulo 7.

4.12 Implementação de Métodos

A implementação de um método corresponde a um conjunto de referências locais e a uma seqüência de operações executáveis. Essas referências locais são acessíveis somente pelas operações da implementação, isto é, não podem ser exportadas para outros objetos. Os tipos de operações e correspondente semântica são descritos nas seções subseqüentes.

4.12.1 Acesso a Blocos de Dados

Uma operação pode fazer acesso a um bloco de dados contido no estado do objeto. Uma operação desse tipo pode tanto simplesmente ler o bloco como alterá-lo.

4.12.2 Invocação de Métodos

Uma operação pode invocar uma método do próprio objeto ou de outro objeto para o qual possua uma referência. Essa referência pode estar disponível em diferentes situações:

- a referência está contida no estado do objeto,
- a referência é local à implementação do método,
- a referência foi passada como parâmetro na invocação do método.

4.12.3 Expressões

Uma operação pode consistir em avaliar toda uma expressão aritmética ou booleana. Uma operação desse tipo é efetuada, isto é, uma expressão é avaliada através de sua decomposição em operações mais básicas, tais como acesso a blocos de dados e invocação de métodos.

4.12.4 Desvios

Uma operação pode provocar um desvio na seqüência de operações em execução. Esse desvio pode ocorrer tanto de forma incondicional como pode ocorrer dependendo do resultado da avaliação de uma expressão.

4.12.5 Asserções

Uma asserção é uma condição (expressão booleana construída da mesma forma que uma invariante) que pode ocorrer em qualquer ponto da implementação de um método e que deve ser satisfeita naquele ponto.

4.12.6 Geração de Eventos

Uma operação pode ser simplesmente a geração de um evento. A sua execução deve ser seguida pela correspondente notificação do evento.

4.12.7 Retorno

Um método pode, opcionalmente, retornar para o método que o invocou uma referência para um objeto, ao seu término. A implementação de um método, portanto, pode conter uma ou mais operações que fazem esse retorno e causam o encerramento da atividade. Caso a implementação não contenha nenhuma operação desse tipo, a atividade encerra-se quando a última operação da implementação for executada e, nesse caso, não há retorno.

4.12.8 Tratamento de Exceções

Uma exceção pode ocorrer em diversas situações:

- quando uma invariante, pré-condição, pós-condição ou asserção não é satisfeita,
- quando ocorre uma tentativa de invocação de método utilizando-se uma referência indefinida,
- quando se invoca um método que falha durante a sua execução, isto é, que produz uma exceção,
- quando se executa uma operação que explicitamente produz uma exceção,
- quando uma operação produz uma situação anormal detectada somente pelo ambiente de execução, como por exemplo falta de memória.

Logo, uma exceção pode ocorrer antes, durante ou após uma atividade, mas sempre denota alguma anomalia. Cada método deve prever as exceções que pode gerar ou receber e optar entre dar um tratamento para a exceção ou repassar a exceção para o método que fez a invocação. O tratamento de exceção em um método, por sua vez, pode fazer uma computação complexa e, possivelmente ainda, gerar novas exceções para tratamento pelo método que fez a invocação. O tratamento de exceção pode, também, causar a interrupção da atividade.

Capítulo 5

Especificação de Métodos

Um método possui uma assinatura e, opcionalmente, uma implementação. Os métodos de um objeto podem ter relacionamentos entre si, dependendo de suas assinaturas e também de certos atributos. Esses relacionamentos determinam qual implementação de método selecionar em uma invocação. Este Capítulo descreve os conceitos pertinentes a métodos e especificam como estes são estruturados em um objeto.

5.1 Assinatura

A assinatura de um método é formada por um nome, uma lista de tipos correspondentes aos parâmetros formais e um tipo de retorno. A Figura 5.1 ilustra a representação gráfica para uma assinatura f de um método; nessa representação, f simboliza a assinatura completa.

Figura 5.1 Representação da assinatura de um método

Um método é identificado univocamente em uma camada através de sua assinatura, isto é, em uma mesma camada não podem haver dois métodos com assinaturas idênticas. Entretanto, em um mesmo objeto, podem haver dois métodos com assinaturas idênticas desde que estejam em camadas distintas. Assim, cada método é identificado univocamente em um objeto pela combinação do seu nível com a sua assinatura.

5.2 Implementação

Um método de um objeto é dito ser *implementado* na camada k quando esta camada inclui, além da assinatura do método, uma implementação para este. Nesse caso, o método também é dito ser *concreto*. A Figura 5.2 mostra a notação gráfica para a implementação a de um método m com assinatura f.

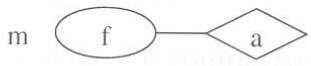


Figura 5.2 Notação para implementação de um método

5.3 Definição

Um método de um objeto é dito ser *definido* na camada k quando o método é declarado ou implementado na camada k.

5.4 Declaração

Um método de um objeto é dito ser *declarado* na camada k quando esta camada inclui a assinatura do método, mas não inclui uma implementação para o mesmo. Nesse caso, o método também é dito ser *abstrato*.

5.5 Invocação

Toda e qualquer invocação de método é feita contra um objeto para o qual se tem uma referência – o objeto *alvo* da invocação. A resolução de uma invocação inicia-se pela determinação do método do objeto alvo que não esteja protegido contra invocação e que seja *compatível* com a invocação, se existir. A verificação de compatibilidade entre uma invocação

e um método é feita através de uma comparação entre a invocação e a assinatura do método. Uma invocação de método é formada por um nome, uma seqüência de parâmetros reais e uma referência para o objeto de retorno da execução do método, quando essa terminar. Assim, uma invocação v de método é compatível com uma assinatura f quando as seguintes condições são simultaneamente satisfeitas:

(1) v e f têm o mesmo nome

(2) v e f têm a mesma quantidade de parâmetros formais

(3) o tipo do n-ésimo parâmetro real de v é compatível com o tipo do n-ésimo parâmetro formal de f

(4) o tipo do retorno de f é compatível com o tipo da referência para retorno especificada em v

5.6 Covariação

Uma assinatura g é uma covariante de outra f quando as seguintes condições são satisfeitas:

(1) f e g não são idênticas

(2) f e g têm o mesmo nome

(3) f e g têm a mesma quantidade de parâmetros formais

(4) o tipo do n-ésimo parâmetro formal de g é compatível com o tipo do n-ésimo parâmetro formal de f

(5) o tipo do retorno de g é compatível com o tipo do retorno de f

5.7 Covariação Lateral

Pode ocorrer covariação em uma mesma camada de um objeto, isto é, podem existir dois métodos com assinaturas f e g em uma mesma camada tal que g seja uma covariante de f, quando as seguintes condições adicionais são satisfeitas:

(1) a quantidade de parâmetros formais de f (que é a mesma para g) é maior ou igual a 1

(2) existe $k \geq 1$ tal que o tipo do k-ésimo parâmetro formal de g é diferente (embora seja compatível, isto é, mais específico) do tipo do k-ésimo parâmetro formal de f

Nesse caso, como a assinatura g estabelece tipos de parâmetros mais específicos que a assinatura f, o método com assinatura g (covariante) tem precedência sobre o método com assinatura f para fins de seleção do método a ser executado em uma invocação. Assim, o método com assinatura f somente pode ser selecionado para execução quando o tipo de algum dos parâmetros reais da invocação é mais geral que o correspondente tipo especificado em g.

As condições adicionais acima são necessárias pois, caso contrário, g seria uma covariante de f devido simplesmente à compatibilidade entre o tipo do retorno de g e o tipo do retorno de f. Se isso fosse permitido, na seleção do método a ser executado em uma invocação, o método com assinatura g sempre teria prioridade com relação ao método com assinatura f, não havendo, portanto, razão para a existência deste.

5.8 Ambigüidade

Não podem haver dois métodos com assinaturas f e g em uma mesma camada tal que as seguintes condições sejam simultaneamente satisfeitas:

(1) f e g têm o mesmo nome

(2) f e g têm a mesma quantidade de parâmetros formais

(3) existem n e k distintos tal que:

(a) o tipo do n-ésimo parâmetro formal de g é diferente o tipo do n-ésimo parâmetro formal de f

(b) o tipo do k-ésimo parâmetro formal de g é diferente o tipo do k-ésimo parâmetro formal de f

(c) o tipo do n-ésimo parâmetro formal de g é compatível com o tipo do n-ésimo parâmetro formal de f

(d) o tipo do k-ésimo parâmetro formal de f é compatível com o tipo do k-ésimo parâmetro formal de g

(4) o tipo do retorno de g é compatível com o tipo do retorno de f

5.9 Redefinição

Cada método possui um *atributo de redefinição* que determina se o método pode ou não ser definido novamente em outras camadas do mesmo objeto. Por redefinição de um método entende-se tanto a redefinição de sua assinatura quanto a de sua implementação, de maneira exclusiva ou inclusiva. Os valores para o atributo de redefinição de um método podem ser os seguintes:

REDEFINÍVEL Um método de nível k com atributo de redefinição REDEFINÍVEL pode ser novamente definido (implementado ou simplesmente declarado) em uma camada $n > k$. Esta nova definição ou *redefinição* do método pode preservar ou não os valores originais dos atributos de acesso e de redefinição. Logo, quando o atributo de redefinição é preservado, pode ocorrer uma segunda redefinição do método em uma camada $p > n$ e assim sucessivamente.

CONSTANTE Um método com atributo de redefinição CONSTANTE não pode ser novamente definido em outras camadas. Portanto, esse método deve necessariamente ser concreto.

A Figura 5.3 ilustra a notação gráfica para representação de redefinição de método: Figura 5.3.a mostra um método com assinatura f REDEFINÍVEL; Figura 5.3.b mostra um método com assinatura f CONSTANTE; Figura 5.3.c mostra uma redefinição de método.

5.10 Formas de Redefinição de Assinatura

Considerando uma redefinição de método na qual um método m_2, definido na camada n, redefine um método m_1, definido na camada $k < n$, há duas opções com relação às assinaturas de m_1 e m_2, a saber:

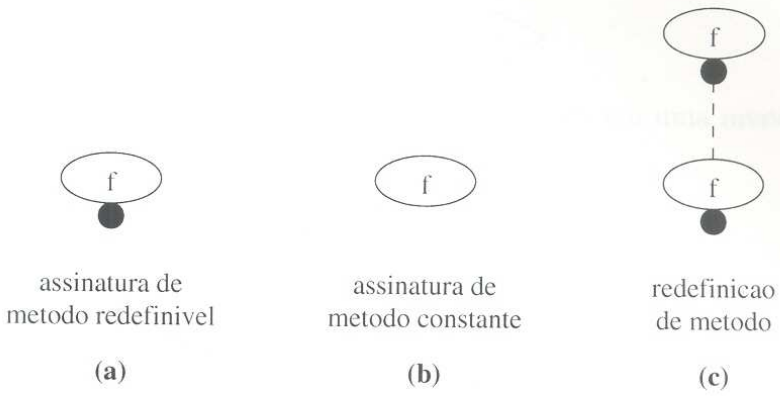


Figura 5.3 Notação para redefinição de método

(1) A assinatura de m_2 é idêntica à assinatura de m_1.

(2) A assinatura de m_2 é covariante da assinatura de m_1.

No caso de covariação, há ainda duas opções com relação à semântica, a saber:

(1) A assinatura de m_2 substitui *totalmente* a assinatura de m_1, significando que uma invocação sucede somente quando é compatível com a assinatura de m_2. Neste caso, somente a implementação correspondente a m_2 pode ser executada. Se a invocação não for compatível com m_2, nenhuma implementação é executada.

(2) A assinatura de m_2 substitui *parcialmente* a assinatura de m_1, significando que uma invocação sucede quando é compatível com a assinatura de m_2 ou com a assinatura de m_1. Neste caso, m_2 tem precedência sobre m_1 na verificação de compatibilidade e correspondente determinação da implementação a executar. Como há a possibilidade de m_1 ser escolhida na invocação, diz-se que a redefinição feita por m_2 é a *inclusão* de uma assinatura mais específica, sem eliminação da assinatura mais geral.

Assim, em uma série de redefinições de um método pelas camadas de um objeto, podem haver diferentes combinações, conforme detalhado nas seções subseqüentes, tal que cada assinatura da série pode ser ou não uma *candidata* a seleção em uma invocação do método, sendo que uma candidata deve, necessariamente possuir uma implementação vinculada. A Figura 5.4 mostra a notação gráfica para indicar quando uma assinatura é candidata a seleção em uma invocação.

A Figura 5.5 ilustra as formas de redefinição de assinatura possíveis, a saber:

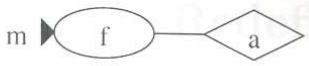


Figura 5.4 Notação para assinatura candidata em uma invocação

REPETIÇÃO Figura 5.5.a:

(1) a assinatura de m_1 é f

(2) a assinatura de m_2 é f

(3) a assinatura de m_2 substitui a assinatura de m_1 totalmente (somente a assinatura de m_2 é candidata)

COVARIAÇÃO-SUBSTITUIÇÃO Figura 5.5.b:

(1) a assinatura de m_1 é f

(2) a assinatura de m_2 é $g \neq f$

(3) g substitui f totalmente (somente g é candidata)

COVARIAÇÃO-INCLUSÃO Figura 5.5.c:

(1) a assinatura de m_1 é f

(2) a assinatura de m_2 é $g \neq f$

(3) g substitui f parcialmente (f e g são candidatas)

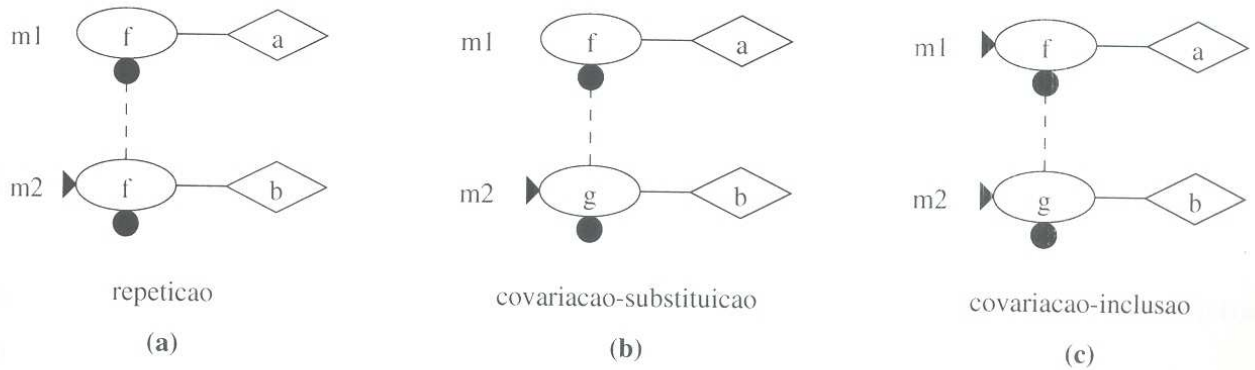


Figura 5.5 Formas de redefinição de assinatura

5.11 Formas Canônicas de Redefinição de Método

As formas canônicas de redefinição de método são as formas válidas de redefinição de assinatura e de implementação, combinadas[1]. A Tabela 5.11 sumariza todas as combinações possíveis que geram as formas canônicas. A Figura 5.6 ilustra as formas canônicas, utilizando a notação gráfica introduzida. As formas canônicas são explicadas nas seções subseqüentes, considerando inicialmente as diferentes formas de redefinição de implementação e, para cada uma destas, as diferentes formas de redefinição de assinatura. Essas formas podem ainda ser arranjadas duas a duas em série, criando uma ou mais seqüências de redefinições para um método, conforme discutido na Seção 5.18.

implementação	assinatura		
	repetição	covariação	
		substituição	inclusão
sobreposição de implementação	•	•	•
encadeamento de implementação	•	•	•
concretização de método	•	•	•
cancelamento de método	•		•
abstração introduzida	•	•	•
abstração mantida		•	•

Tabela 5.1 Formas canônicas de redefinição de método

5.12 Sobreposição de Implementação

Um método m_2, definido na camada n, é uma *sobreposição de implementação* de um método m_1, definido na camada $k < n$, quando:

(1) m_2 redefine m_1

[1] *Em uma forma canônica não é relevante a possibilidade de alteração do atributo de redefinição.*

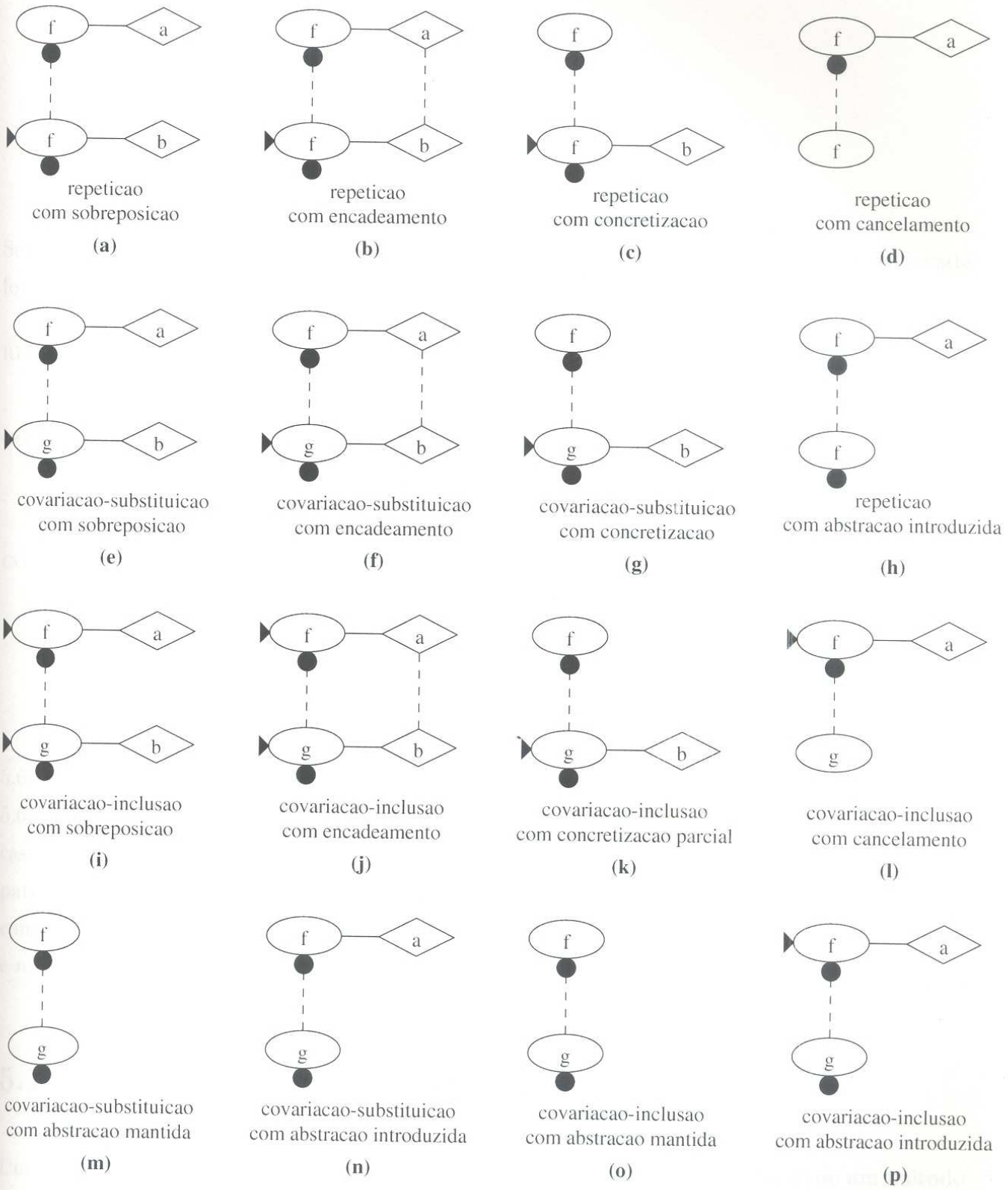


Figura 5.6 Formas canônicas de redefinição de método

(2) m_1 possui uma implementação i_1

(3) m_2 possui uma implementação i_2

(4) a execução de i_1 é independente da execução de i_2

(5) a execução de i_2 é independente da execução de i_1

A semântica de invocação de m_1 e m_2 depende da forma de redefinição de assinatura. Sendo a_1 a assinatura de m_1 e a_2 a assinatura de m_2, a semântica de invocação para cada forma de redefinição de assinatura é como segue.

REPETIÇÃO Qualquer invocação compatível com a_2 (ou a_1, pois são idênticas) causa a execução de i_2; a assinatura a_1 não é candidata.

COVARIAÇÃO-SUBSTITUIÇÃO Uma invocação compatível com a_2 causa a execução de i_2; caso contrário, nenhuma implementação é executada, pois a assinatura a_1 não é candidata.

COVARIAÇÃO-INCLUSÃO Uma invocação compatível com a_2 causa a execução de i_2; caso contrário, se a invocação for compatível com a_1 e se não houver um método m_3 de nível r, $k \leq r \leq n$ tal que a invocação seja compatível com a assinatura de m_3, então i_1 é executada. Portanto, nesse caso, a_1 e a_2 são candidatas.

Esses três casos são ilustrados nas figuras 5.6.a, 5.6.e e 5.6.i, respectivamente. Na Figura 5.6.a, uma invocação compatível com f causa a execução da implementação b. Na Figura 5.6.e, uma invocação compatível com a assinatura g causa a execução da implementação b; caso contrário não há execução de implementação. Na Figura 5.6.i, uma invocação compatível com a assinatura g causa a execução de b, enquanto que uma invocação compatível com f e incompatível com g ou com a assinatura de qualquer método nas camadas entre k e n, inclusive, causa a execução de a.

5.13 Encadeamento de Implementação

Um método m_2, definido na camada n, é um *encadeamento de implementação* de um método m_1, definido na camada $k < n$, quando:

(1) m_2 redefine m_1

(2) m_1 possui uma implementação i_1

(3) m_2 possui uma implementação i_2

(4) a execução de i_1 é independente da execução de i_2

(5) a execução de i_2 é imediatamente precedida pela execução de i_1

A Figura 5.7 mostra a notação gráfica para representar um encadeamento de implementação. No exemplo, somente a assinatura g é candidata. Assim, uma invocação compatível com g causa a execução da implementação a seguida pela execução da implementação b[2].

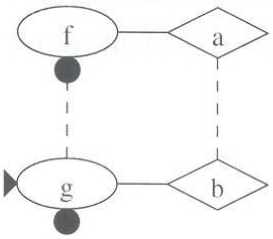


Figura 5.7 Notação para encadeamento de implementação

A semântica de invocação de m_1 e m_2 depende da forma de redefinição de assinatura. Sendo a_1 a assinatura de m_1 e a_2 a assinatura de m_2, a semântica de invocação para cada forma de redefinição de assinatura é como segue.

REPETIÇÃO Qualquer invocação compatível com a_2 (ou a_1, pois são idênticas) causa a execução de i_1 seguida pela imediata execução de i_2, pois somente a assinatura a_2 é candidata.

COVARIAÇÃO-SUBSTITUIÇÃO Uma invocação compatível com a_2 causa a execução de i_1 seguida pela imediata execução de i_2; caso contrário, nenhuma implementação é executada, pois somente a assinatura a_2 é candidata.

[2] *Uma questão a ser resolvida pela implementação da Virtuosi é o tratamento a ser dado por um possível retorno de uma implementação executada por encadeamento.*

COVARIAÇÃO-INCLUSÃO Uma invocação compatível com a_2 causa a execução de i_1 seguida pela imediata execução de i_2; caso contrário, se a invocação for compatível com a_1 e se não houver um método m_3 de nível r, $k \leq r \leq n$ tal que a invocação seja compatível com a assinatura de m_3, então i_1, e somente i_1, é executada. Portanto, nesse caso, a_1 e a_2 são candidatas.

Esses três casos são ilustrados nas figuras 5.6.b, 5.6.f e 5.6.j, respectivamente. Na Figura 5.6.b, uma invocação compatível com f causa a execução das implementações a e b, nesta ordem. Na Figura 5.6.f, uma invocação compatível com a assinatura g causa a execução das implementações a e b, nesta ordem; caso contrário não há execução de implementação. Na Figura 5.6.j, uma invocação compatível com a assinatura g causa a execução de a e b, nesta ordem, enquanto que uma invocação compatível com f e incompatível com g ou com qualquer método das camadas entre k e n, inclusive, causa a execução de a somente.

5.14 Concretização

Um método m_2, definido na camada n, é uma *concretização* de um método m_1, definido na camada $k < n$, quando:

(1) m_2 redefine m_1

(2) m_1 não possui uma implementação

(3) m_2 possui uma implementação i_2

A semântica de invocação de m_1 e m_2 depende da forma de redefinição de assinatura. Sendo a_1 a assinatura de m_1 e a_2 a assinatura de m_2, a semântica de invocação para cada forma de redefinição de assinatura é como segue.

REPETIÇÃO Qualquer invocação compatível com a_2 (ou a_1, pois são idênticas) causa a execução de i_2, pois somente a_2 é candidata.

COVARIAÇÃO-SUBSTITUIÇÃO Uma invocação compatível com a_2 causa a execução de i_2; caso contrário, se não houver um método m_3 de nível r, $k \leq r \leq n$ tal que a invocação seja compatível com a assinatura de m_3, então nenhuma implementação é executada, pois somente a_2 é candidata.

COVARIAÇÃO-INCLUSÃO Uma invocação compatível com a_2 causa a execução de i_2; caso contrário, se não houver um método m_3 de nível r, $k \leq r \leq n$ tal que a invocação seja compatível com a assinatura de m_3, então nenhuma implementação é executada, embora tanto a_1 quanto a_2 sejam candidatas. Por essa razão, a concretização é dita ser *parcial*.

Esses três casos são ilustrados nas figuras 5.6.c, 5.6.g e 5.6.k, respectivamente. Na Figura 5.6.c, uma invocação causa a execução da implementação b. Na Figura 5.6.g, uma invocação compatível com a assinatura g causa a execução da implementação b; caso contrário não há execução de implementação. Na Figura 5.6.k, uma invocação compatível com a assinatura g causa a execução de b, enquanto que uma invocação incompatível com g ou com qualquer método nas camadas entre k e n, inclusive, não causa a execução de qualquer implementação.

5.15 Cancelamento

Um método m_2, definido na camada n, é um *cancelamento* de um método m_1, definido na camada $k < n$, quando:

(1) m_2 redefine m_1

(2) m_2 não possui uma implementação

(3) m_2 possui atributo de redefinição CONSTANTE

A semântica de invocação de m_1 e m_2 depende da forma de redefinição de assinatura. Sendo a_1 a assinatura de m_1 e a_2 a assinatura de m_2, a semântica de invocação para cada forma de redefinição de assinatura é como segue.

REPETIÇÃO Qualquer invocação compatível com a_2 (ou a_1, pois são idênticas) não causa a execução de qualquer implementação, pois a_1 e a_2 não são candidatas.

COVARIAÇÃO-SUBSTITUIÇÃO Cancelamento não se aplica neste caso, pois se a_2 substitui totalmente a_1 e m_2 não possui uma implementação, então uma invocação compatível com a_1 ou a_2 não causa execução de qualquer implementação, o que é equivalente ao caso de REPETIÇÃO.

COVARIAÇÃO-INCLUSÃO Uma invocação compatível com a_2 não causa a execução de qualquer implementação; caso contrário, em uma invocação compatível com a_1, se não houver um método m_3 de nível r, $k \leq r \leq n$ tal que a invocação seja compatível com a assinatura de m_3, então i_1 é executada, pois a_1 é candidata.

Os casos de REPETIÇÃO e COVARIAÇÃO-INCLUSÃO são ilustrados nas figuras 5.6.d e 5.6.l, respectivamente. Na Figura 5.6.d, uma invocação compatível com f não causa a execução da implementação a ou de qualquer outra implementação. Na Figura 5.6.l, uma invocação compatível com a assinatura g não causa a execução de qualquer implementação, enquanto que uma invocação compatível com f e incompatível com g ou com qualquer método das camadas entre k e n, inclusive, causa a execução de a somente.

5.16 Abstração Mantida

Um método m_2, definido na camada n, é uma *abstração mantida* de um método m_1, definido na camada $k < n$, quando:

(1) m_2 redefine m_1

(2) m_1 não possui uma implementação (é abstrato)

(3) m_2 não possui uma implementação (é abstrato)

(4) m_2 possui atributo de redefinição REDEFINÍVEL

A semântica de invocação de m_1 e m_2 depende da forma de redefinição de assinatura. Sendo a_1 a assinatura de m_1 e a_2 a assinatura de m_2, a semântica de invocação para cada forma de redefinição de assinatura é como segue.

REPETIÇÃO Redefinição com abstração mantida não se aplica neste caso, pois não causaria qualquer modificação no comportamento do objeto.

COVARIAÇÃO-SUBSTITUIÇÃO Qualquer invocação compatível com a_1 ou a_2 não causa execução de qualquer implementação, pois a_1 e a_2 não são candidatas. (Entretanto, m_2 ainda pode ser redefinido por um método concreto.)

COVARIAÇÃO-INCLUSÃO Uma invocação compatível com a_2 não causa a execução de qualquer implementação; caso contrário, mesmo se a invocação for compatível com a_1 e não houver um método m_3 de nível r, $k \leq r \leq n$ tal que a invocação seja compatível com a assinatura de m_3, também nenhuma implementação é executada, pois a_1 e a_2 não são candidatas. (Também nesse caso, m_2 ainda pode ser redefinido por um método concreto.)

Esses casos são ilustrados nas figuras 5.6.m e 5.6.o, respectivamente. Na Figura 5.6.m, uma invocação compatível com g não causa a execução de qualquer implementação. Na Figura 5.6.o, uma invocação compatível com a assinatura g não causa a execução de qualquer implementação, enquanto que uma assinatura compatível com f e incompatível com g ou com qualquer método das camadas entre k e n, inclusive, não causa a execução de qualquer implementação.

5.17 Abstração Introduzida

Um método m_2, definido na camada n, é uma *abstração introduzida* de um método m_1, definido na camada $k < n$, quando:

(1) m_2 redefine m_1

(2) m_1 possui uma implementação i_1 (é concreto)

(3) m_2 não possui uma implementação (é abstrato)

(4) m_2 possui atributo de redefinição REDEFINÍVEL

A semântica de invocação de m_1 e m_2 depende da forma de redefinição de assinatura. Sendo a_1 a assinatura de m_1 e a_2 a assinatura de m_2, a semântica de invocação para cada forma de redefinição de assinatura é como segue.

REPETIÇÃO Qualquer invocação compatível com a_2 (ou a_1, pois são idênticas) não causa a execução de qualquer implementação, pois a_1 e a_2 não são candidatas.

COVARIAÇÃO-SUBSTITUIÇÃO Qualquer invocação compatível com a_1 ou a_2 não causa execução de qualquer implementação, pois a_1 e a_2 não são candidatas. (Entretanto, m_2 ainda pode ser redefinido por um método concreto.)

COVARIAÇÃO-INCLUSÃO Uma invocação compatível com a_2 não causa a execução de qualquer implementação; caso contrário, se a invocação for compatível com a_1 e não houver um método m_3 de nível r, $k \leq r \leq n$ tal que a invocação seja compatível com a assinatura de m_3, a implementação i_1 é executada, pois a_1 é candidata. (Também nesse caso, m_2 ainda pode ser redefinido por um método concreto.)

Esses casos são ilustrados nas figuras 5.6.n e 5.6.p, respectivamente. Na Figura 5.6.n, uma invocação compatível com g não causa a execução da implementação a ou qualquer outra. Na Figura 5.6.p, uma invocação compatível com g e incompatível com qualquer método das camadas entre k e n, inclusive, causa a execução de a.

5.18 Encadeamento de Redefinições de Métodos

As formas canônicas de redefinição podem ser arranjadas duas a duas em série, criando uma ou mais seqüências de redefinições para um certo método. Um arranjo de duas redefinições em série, entretanto, deve, a fim de respeitar a semântica de cada tipo de redefinição, obedecer as seguintes restrições:

(1) Um cancelamento de método não pode ser seguido de qualquer redefinição.

(2) Uma concretização de método pode seguir apenas uma abstração introduzida ou mantida.

(3) Uma abstração mantida pode seguir apenas uma abstração introduzida ou mantida.

(4) Uma abstração introduzida pode seguir apenas uma sobreposição de implementação, um encadeamento de implementação ou uma concretização de método.

(5) Uma sobreposição de implementação pode seguir apenas outra sobreposição, um encadeamento de implementação ou uma concretização de método.

(6) Um encadeamento de implementação pode seguir apenas uma sobreposição de implementação, um encadeamento de implementação ou uma concretização de método.

A Figura 5.8 ilustra seis arranjos de formas canônicas de redefinição, isto é, seis diferentes casos de seqüenciamento, sendo que há um conjunto distinto de assinaturas candidatas em uma invocação.

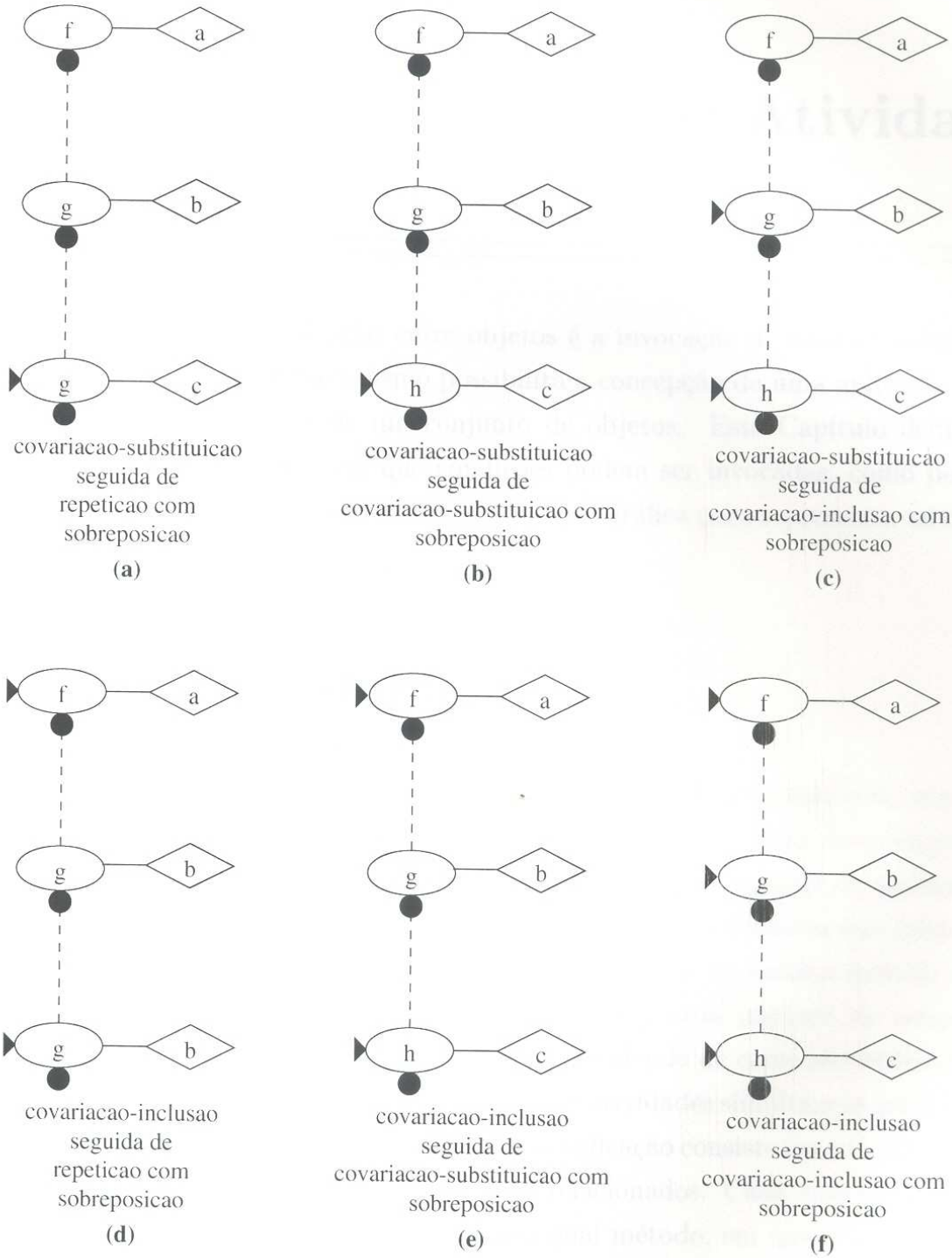


Figura 5.8 Casos de seqüenciamento de covariações

Capítulo 6

Comunicação por Atividades

A forma mais básica de comunicação entre objetos é a invocação de uma atividade de um objeto por outro objeto. Esse mecanismo possibilita a concepção de uma aplicação baseada no encadeamento de atividades de um conjunto de objetos. Este Capítulo define o que são atividades, quais seus tipos, em que condições podem ser invocadas, como podem ser estruturadas em uma aplicação e introduz uma notação gráfica para representar os conceitos descritos.

6.1 Atividades de Objetos

Uma atividade de um objeto corresponde à execução de um de seus métodos, isto é, cada invocação de método de um certo objeto dá início a uma nova atividade deste objeto. Uma atividade termina quando a execução do método termina, seja normal ou anormalmente (quando ocorre uma exceção). Duas invocações de dois métodos distintos dão início a duas atividades independentes. Da mesma forma, duas invocações do mesmo método também dão início a duas atividades independentes. Assim, para cada instante de tempo, cada objeto do sistema por 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 método invoca qual método, em que ordem e sob quais condições ocorrem as invocações e ainda, para cada par de *atividade invocadora* e *atividade invocada*, o modo de invocação com relação ao sincronismo.

6.2 Modos de Invocação de Atividades

Dependendo do modo de invocação com relação ao sincronismo, a atividade invocada pode ser classificada de três formas distintas, a saber:

atividade síncrona A atividade invocadora fica bloqueada até que a atividade invocada 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 uma exceção).

atividade assíncrona sem notificação de término 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 capturada pela atividade invocadora.

atividade assíncrona com notificação de término A atividade invocadora não fica bloqueada devido à invocação mas recebe uma notificação de término. Essa notificação pode ser *simples* ou pode trazer um retorno, quando a atividade invocada termina normalmente, ou uma exceção, caso contrário. Toda vez que uma atividade invoca uma atividade assíncrona com notificação de término, fica com uma nova notificação de término pendente. Uma atividade que invoca uma ou mais atividades assíncronas com notificação de término não pode encerrar-se até que todas as notificações de término pendentes sejam recebidas.

Deve-se observar que o modo de sincronismo para um certo método não precisa ser fixo, isto é, pode ser escolhido em tempo de execução pela atividade invocadora. Assim, é possível que um mesmo método seja executado como atividade síncrona em uma situação e como atividade assíncrona em outra, dentro da mesma aplicação ou não. Essa propriedade torna o sistema bastante flexível, permitindo que cada aplicação defina uma forma própria de estruturar suas atividades e possibilitando otimizações na utilização dos recursos do sistema a fim de explorar paralelismo na execução.

6.3 Representação de Atividades

A Figura 6.1 mostra a notação gráfica introduzida para representação de atividades de objetos. A Figura 6.1.a representa um objeto sem qualquer atividade. Uma atividade é representada por um quadrado anexado ao respectivo objeto. Esse quadrado pode ser vazio (Figura 6.1.b), cheio (6.1.c) ou semi-cheio (Figura 6.1.d), conforme a atividade seja invocada de forma síncrona, assíncrona sem notificação de término ou assíncrona com notificação de término, respectivamente. Além disso, um quadrado correspondente a uma atividade (síncrona ou assíncrona com notificação de término ou não) de um objeto pode ter anexado um pequeno círculo cheio (Figura 6.1.e) para indicar que se trata da atividade de construção do objeto, conforme discutido na Seção 6.5. A Figura 6.1.f mostra um objeto com duas atividades: uma atividade de construção invocada de modo assíncrono com notificação de término e outra atividade assíncrona; obviamente, a atividade de construção é a primeira a ser invocada. A Figura 6.1.g mostra um objeto com diversas atividades, sendo que a atividade de construção é assíncrona sem notificação de término.

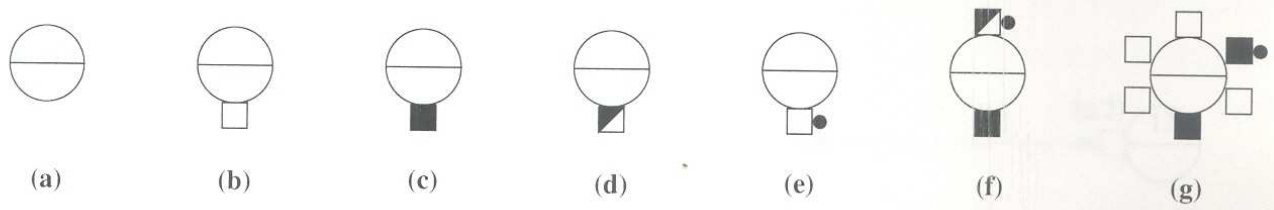


Figura 6.1 Representação de atividades de objetos

6.4 Representação de Invocação de Atividades

A Figura 6.2 introduz a notação gráfica para indicar que uma atividade invoca outra e ilustra a diferença semântica dos diversos modos de invocação. Primeiramente, cada atividade deve ser anotada com um rótulo. Na Figura 6.2.a, por exemplo, o objeto 4 tem a atividade a, o objeto 8 tem a atividade $a.1$ e o objeto 13 tem a a atividade $a.2$. Uma invocação de atividade é indicada por uma seta com origem na atividade invocadora e destino na atividade invocada. Na Figura 6.2.a, a atividade a invoca as atividades $a.1$ e $a.2$; o prefixo a indica

a atividade invocadora e os índices 1 e 2 indicam a ordem de invocação. A atividade $a.1$ é também designada como b por ser invocada no modo síncrono, indicando que um novo encadeamento de invocações inicia-se em paralelo, ou seja, a atividade b do objeto 8 equivale a uma nova *thread* de execução iniciada a partir da *thread* correspondente à atividade a. Nota-se, entretanto, que as atividades a e b, apesar de serem concorrentes, pertencem a objetos distintos e, portanto, não há concorrência interna em nenhum objeto.

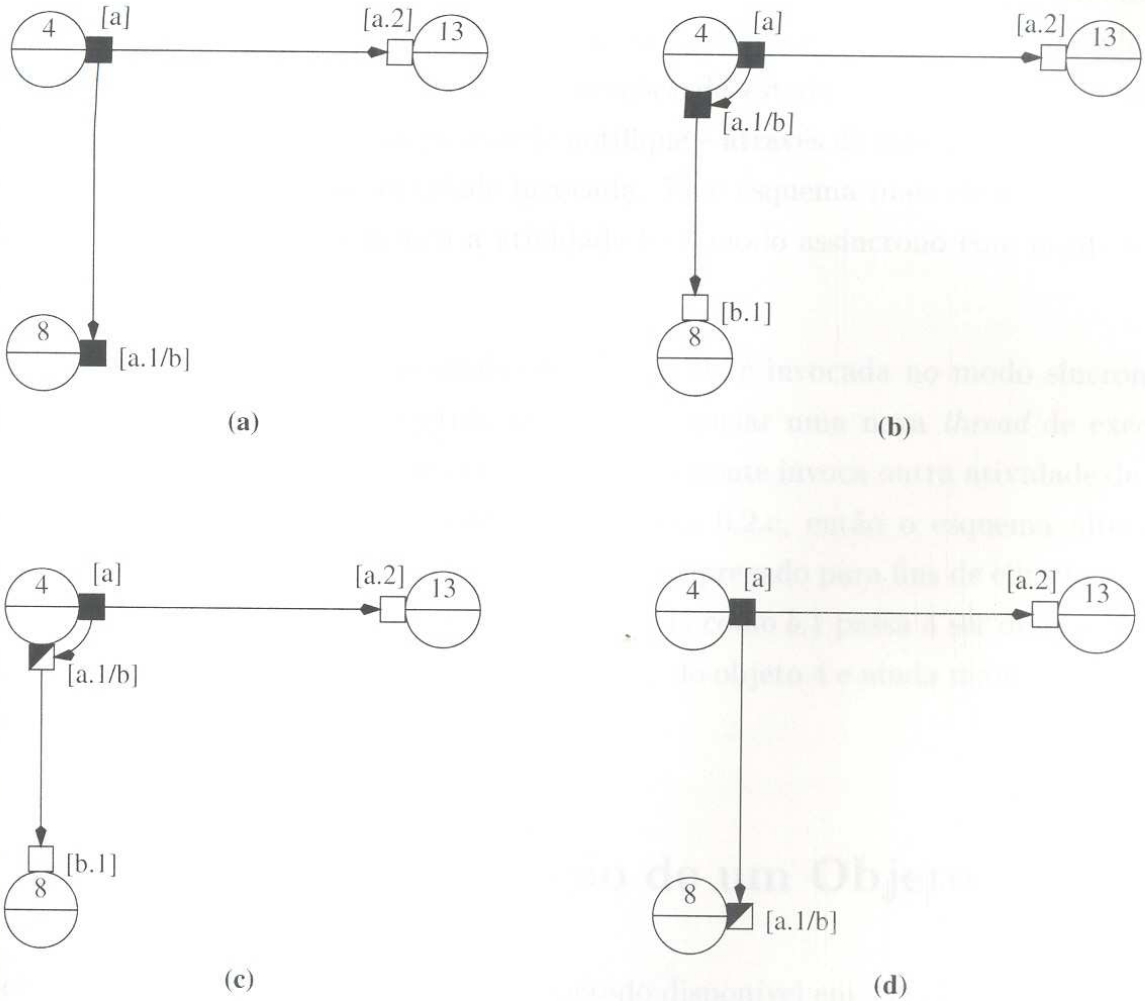


Figura 6.2 Modos de invocação de atividade

Na situação ilustrada na Figura 6.2.b, por outro lado, há concorrência interna no objeto 4, pois a atividade a do objeto 4 invoca de forma assíncrona a atividade b do próprio objeto 4. Neste exemplo, fica claro que o encadeamento de atividades b e $b.1$ ocorre em paralelo

com o encadeamento a e $a.2$. Esse tipo de organização de atividades é comumente utilizado em sistemas que dispõem de *threads* como mecanismo de se obter paralelismo: uma *thread* em execução em um certo objeto cria uma nova *thread* no próprio objeto. Nesse caso, as *threads* comunicam-se somente através do estado do objeto a que pertencem, pois é o único elemento que compartilham. Assim, no exemplo, a atividade a pode detectar o andamento e o término da atividade b através de freqüentes verificações (*polling*) no estado do objeto 4.

Esse esquema pode ser conveniente ou não, dependendo da aplicação. No caso em que a atividade a estiver interessada somente no término da atividade b, o retardo no andamento das atividades causado pelas freqüentes verificações de estado pode ser evitado com um emprego de um mecanismo que simplesmente notifique – através de interrupção – a atividade invocadora sobre o término da atividade invocada. Esse esquema mais eficiente é ilustrado na Figura 6.2.c: a atividade a invoca a atividade b de modo assíncrono com notificação de término.

Também ocorrem situações nas quais uma atividade é invocada no modo síncrono por outra do mesmo objeto com o objetivo único de se iniciar uma nova *thread* de execução, isto é, a implementação da atividade invocada simplesmente invoca outra atividade de outro objeto. Se esse for o caso da atividade b na Figura 6.2.c, então o esquema alternativo ilustrado na Figura 6.2.d deve ser preferencialmente empregado para fins de eficiência. Nesse novo esquema, a atividade do objeto 8 antes designada como $b.1$ passa a ser designada como atividade b, eliminando a atividade intermediária $a.1$ do objeto 4 e ainda mantendo a mesma funcionalidade.

6.5 Atividade de Construção de um Objeto

Cada objeto é criado com a execução de um método disponível em sua classe, especialmente implementado para esse fim – um *método construtor*. A execução de um método construtor para a criação de um novo objeto é denominada a *atividade de construção* do objeto. A Figura 6.3 ilustra a criação de um objeto: o objeto 45 é construído pela atividade $a.1$, invocada pela atividade a do objeto 27. Conforme discutido na Seção 6.3, a representação gráfica de uma atividade de construção inclui um pequeno círculo cheio anexado.

Figura 6.3 Atividade de construção de um objeto

6.6 Atividade Raiz

Todo encadeamento de atividades é iniciado a partir de uma atividade invocada de modo assíncrono (com ou sem notificação de término). Essa atividade inicial é denominada a *atividade raiz* do encadeamento. Dependendo do tipo de atividade que invoca uma atividade raiz, esta pode ser classificada em dois tipos, a saber:

atividade raiz por construção Atividade assíncrona iniciada pela atividade de construção do objeto ao qual a atividade está associada. A Figura 6.4 ilustra uma atividade raiz por construção: a atividade $a.1.1$ (ou simplesmente b) do objeto 45 é invocada pela atividade de construção $a.1$ deste objeto.

atividade raiz por invocação Atividade assíncrona iniciada por uma atividade distinta da atividade de construção do objeto ao qual a atividade está associada. A Figura 6.2 ilustra diversas atividades raizes por chamada. Na Figura 6.2.a, por exemplo, a atividade assíncrona b do objeto 8 é raiz por invocação, pois é invocada por uma atividade de outro objeto: a atividade a do objeto 4. Na Figura 6.2.b, a atividade assíncrona b do objeto 4 é raiz por invocação, pois é invocada por uma atividade do próprio objeto mas que não é a sua atividade de construção: a atividade a do objeto 4.

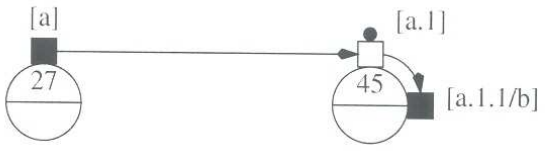


Figura 6.4 Atividade raiz por construção

6.7 Iniciação de Atividades usando Referências

Em uma invocação de atividade, o objeto que tem a atividade invocadora é denominado *fonte* da invocação, enquanto que o objeto que tem a atividade invocada é denominado *alvo* da invocação. O método correspondente à atividade invocadora determina qual é o objeto alvo da invocação através de uma referência para esse objeto. O objeto fonte obtém uma referência para o objeto alvo quando esta é passada como argumento na invoção de alguma de suas atividades ou quando uma atividade do objeto fonte invoca a atividade de construção do objeto alvo. Há o caso particular em que o objeto fonte é também o objeto alvo, isto é, uma atividade de um objeto dá início a outra atividade do próprio objeto. A Figura 6.5 ilustra diversos casos de invocação de atividade. Em todos estes casos, uma invocação ocorre desde que o objeto fonte tenha um referência para o objeto alvo (representada pela linha tracejada com uma seta indicando o objeto referenciado), ou desde que objeto fonte e alvo coincidam. Na Figura 6.5.a uma atividade do objeto 1 invoca uma atividade do objeto 2. Essa invocação ocorre de modo síncrono, visto que a atividade invocada está representada como um quadrado vazio. Na Figura 6.5.b estão representadas duas invocações em seqüência: uma atividade do objeto 1 invoca uma atividade do objeto 2 que, por sua vez, invoca uma atividade do objeto 3, sendo ambas as invocações no modo síncrono. Na Figura 6.5.c estão representadas duas invocações concorrentes: uma atividade do objeto 1 invoca uma atividade do objeto 2 enquanto que, simultaneamente, outra atividade do objeto 1 invoca outra atividade do objeto 2. Na Figura 6.5.d há duas seqüências concorrentes de invocação. Na Figura 6.5.e está representada uma invocação assíncrona sem notificação de término. Na Figura 6.5.f há duas invocações síncronas concorrentes, sendo que o objeto 2 é alvo em uma e, ao mesmo tempo, fonte em outra. As Figuras 6.5.g e 6.5.h representam duas invocações nas quais o objeto alvo é o próprio objeto fonte; na primeira a invocação é assíncrona sem notificação de término, o que permite o início de uma nova seqüência de invocações, enquanto que na segunda a invocação é síncrona, bloqueando a atividade invocadora. Na Figura 6.5.i uma atividade do objeto 1 invoca síncronamente uma atividade do objeto 2 que, por sua vez, invoca de modo assíncrono sem notificação de término outra atividade no próprio objeto 2, dando início a uma nova seqüência concorrente de invocações. As Figuras 6.5.j e 6.5.l mostram invocações síncronas e assíncronas entre atividades de um mesmo objeto. Finalmente, a Figura 6.5.m mostra uma seqüência de invocações síncronas entre diversas atividades de diversos objetos que se referenciam.

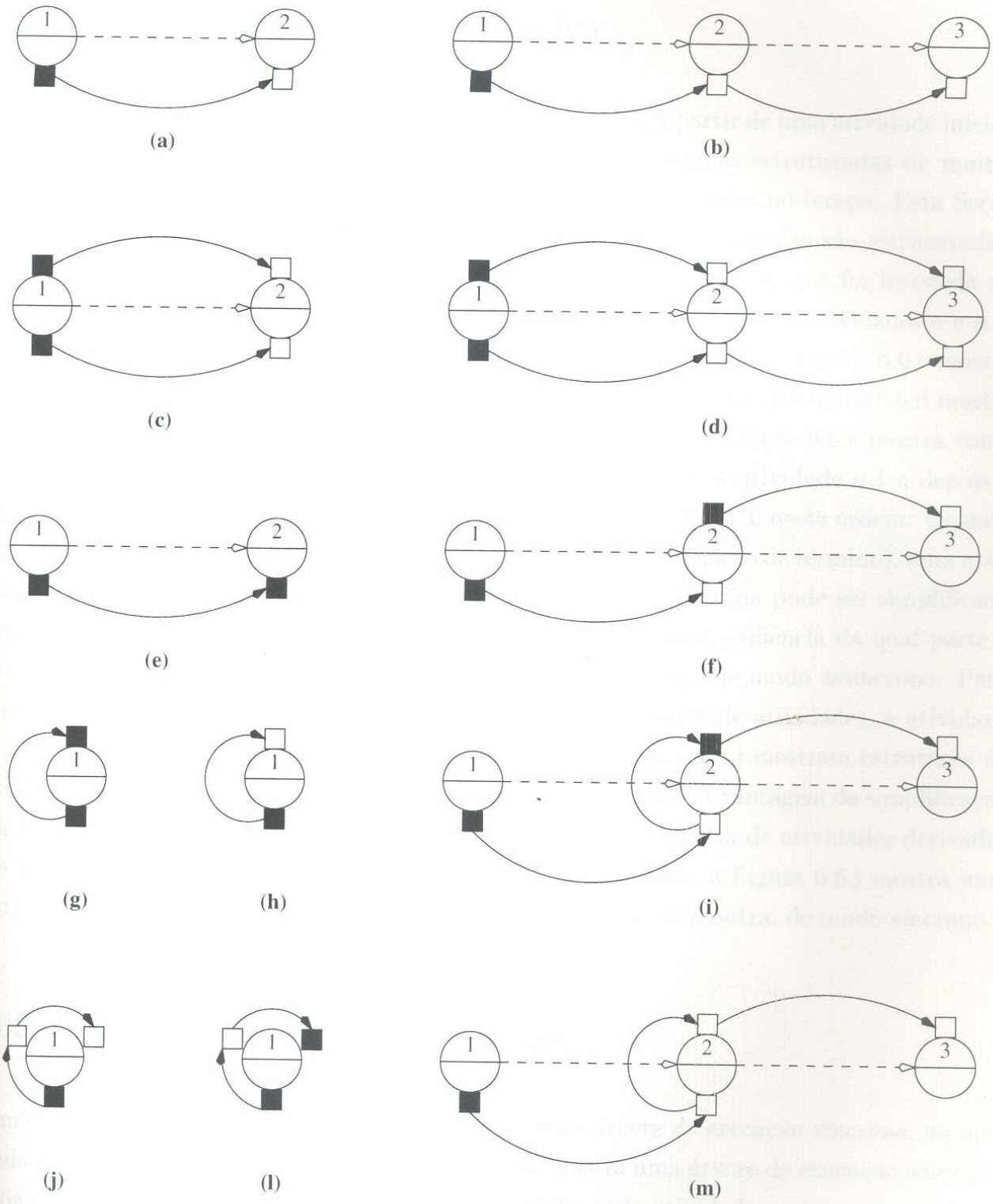


Figura 6.5 Cenários de iniciação de atividades usando referências

6.8 Estruturação de Atividades

Toda aplicação consiste em um encadeamento de atividades, a partir de uma atividade inicial. Tal encadeamento pode conter atividades síncronas e assíncronas estruturadas de muitas formas, definindo conjuntos de atividades seqüenciais e concorrentes no tempo. Esta Seção introduz uma notação para indicar como as atividades de uma aplicação são estruturadas, exemplificada pela Figura 6.6. A Figura 6.6.a mostra a atividade a, que foi invocada de modo assíncrono sem notificação de término. A Figura 6.6.b mostra as atividades a e $a.1$; o rótulo $a.1$ indica que é uma atividade invocada pela atividade a. A Figura 6.6.c mostra as atividades $a.1$ e $a.2$ invocadas pela atividade a. Da mesma forma, a Figura 6.6.d mostra três atividades invocadas seqüencialmente de modo síncrono. A Figura 6.6.e mostra como a notação é utilizada de forma recursiva: a atividade a invoca a atividade $a.1$ e depois a atividade $a.2$, que por sua vez invoca as atividades $a.2.1$, $a.2.2$ e $a.2.3$, nesta ordem. Quando uma atividade é invocada de forma assíncrona (com ou sem notificação de término), uma nova seqüência é iniciada. Por isso, o rótulo de uma atividade assíncrona pode ser simplificado para conotar o início de uma nova seqüência, concorrente com a seqüência da qual parte a invocação. Na Figura 6.6.f a atividade a invoca a atividade $a.1$ de modo assíncrono. Para simplificar a notação e enfatizar o início de uma nova seqüência de atividades, a atividade $a.1$ é simplesmente rotulada como b. As Figuras 6.6.g, 6.6.h e 6.6.i mostram estruturas de atividades nas quais ocorrem atividades síncronas e assíncronas. A vantagem da simplificação da notação fica evidente: na Figura 6.6.i, por exemplo, a seqüência de atividades derivadas de b é concorrente com a seqüência derivada de c. Finalmente, a Figura 6.6.j mostra uma estrutura particular na qual cada atividade invoca somente uma outra, de modo síncrono.

6.9 Sincronismo e Referências

Um encadeamento de atividades síncronas define uma *árvore de execução síncrona*, na qual cada nó corresponde a uma atividade. A Figura 6.7 ilustra uma árvore de execução síncrona, cuja raiz é a atividade a do objeto 12. Neste exemplo, cada atividade pertence a um objeto distinto. Por essa razão, há uma coincidência entre a árvore de execução e a própria árvore de objetos definida pelas referências que existem entre si, isto é, há um mapeamento direto entre a árvore de execução síncrona e a árvore de objetos.

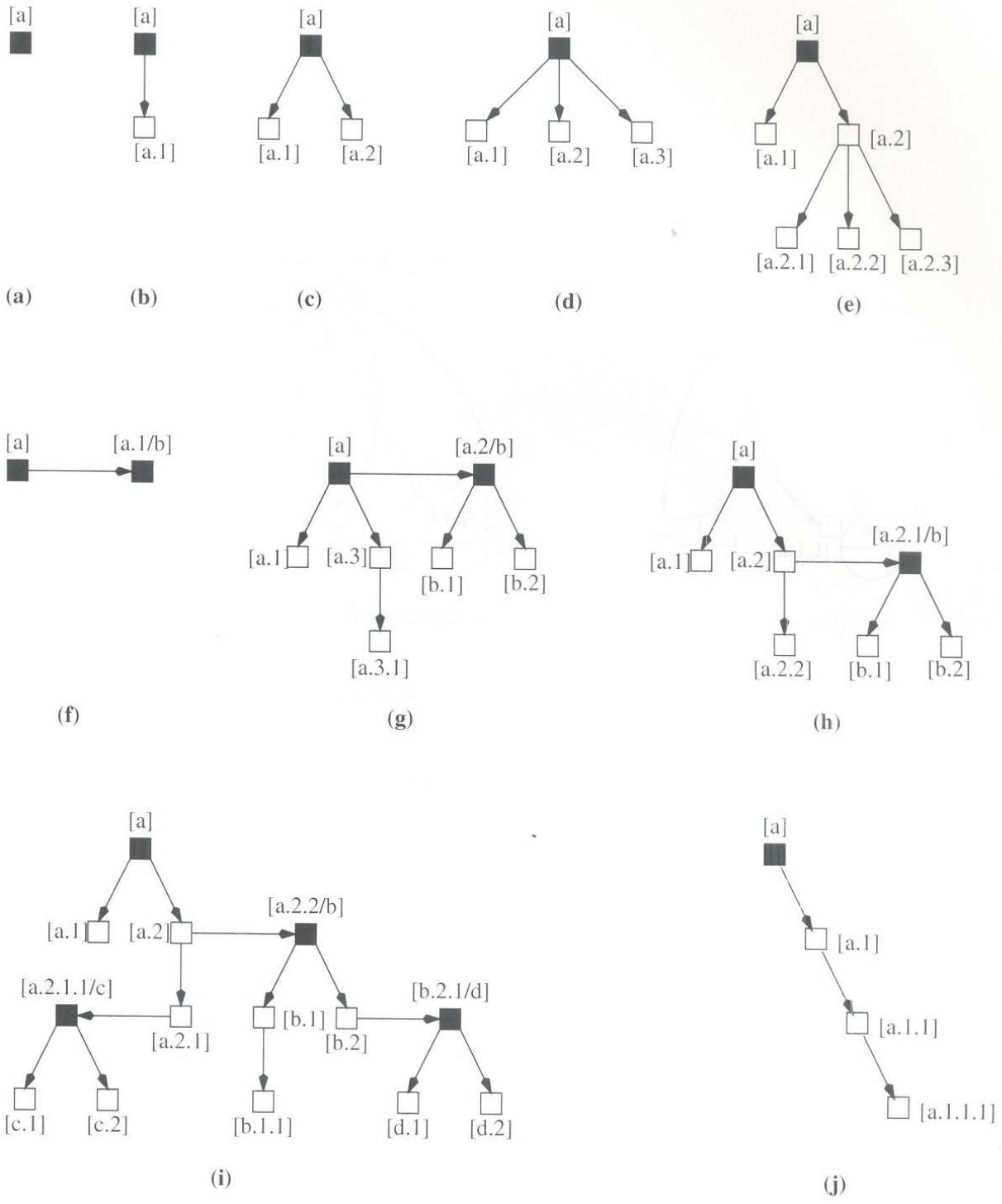


Figura 6.6 Representação de estruturação de atividades

Uma árvore de execução síncrona, entretanto, não precisa estar diretamente mapeada sobre uma árvore de objetos. A Figura 6.8 ilustra a mesma árvore de execução síncrona

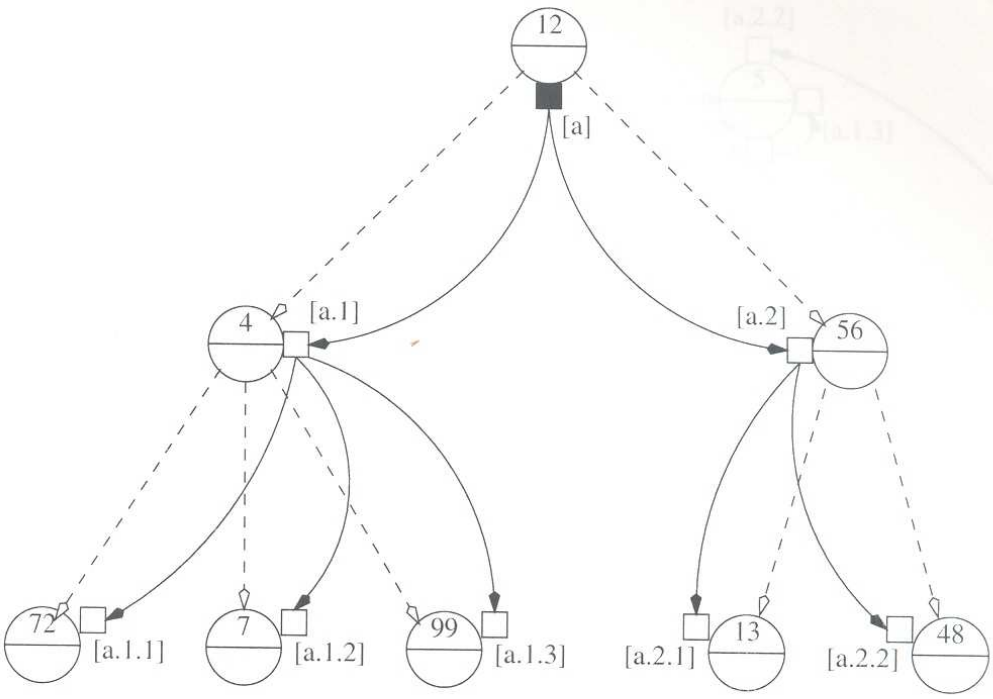


Figura 6.7 Árvore de execução síncrona sobre uma árvore de objetos

presente na Figura 6.7, porém mapeada sobre um conjunto de objetos cujas referências não definem uma árvore – definem um grafo genérico, contendo as referências necessárias para permitir as invocações de métodos. Neste caso, um mesmo objeto pode ter simultaneamente diversas atividades da mesma árvore síncrona (pela própria definição de sincronismo, somente uma dessas atividades por vez pode estar não bloqueada). O objeto 18, por exemplo, tem a atividade raiz a e também tem a atividade folha $a.2.1$.

Por outro lado, um encadeamento de atividades sobre uma árvore de objetos não precisa ser necessariamente uma árvore de execução síncrona. A Figura 6.9 mostra um encadeamento de atividades sobre uma árvore de objetos no qual ocorrem atividades assíncronas (a, b e c) e atividades síncronas ($b.1$, $b.2$, $b.3$, $c.1$, $c.2$). Neste exemplo, há duas árvores de execução síncrona – com raizes b e c – concorrentes, isto é, paralelas. Apesar de haver atividades assíncronas e haver paralelismo de execução, não há concorrência interna em nenhum dos objetos devido ao modo como estes se referenciam.

Invocações assíncronas de atividades podem, dependendo de como os objetos se referenciam, fazer com que um mesmo objeto tenha mais de uma atividade simultânea não bloqueada, isto é, tenha atividades concorrentes. A Figura 6.10 ilustra um encadeamento de atividades

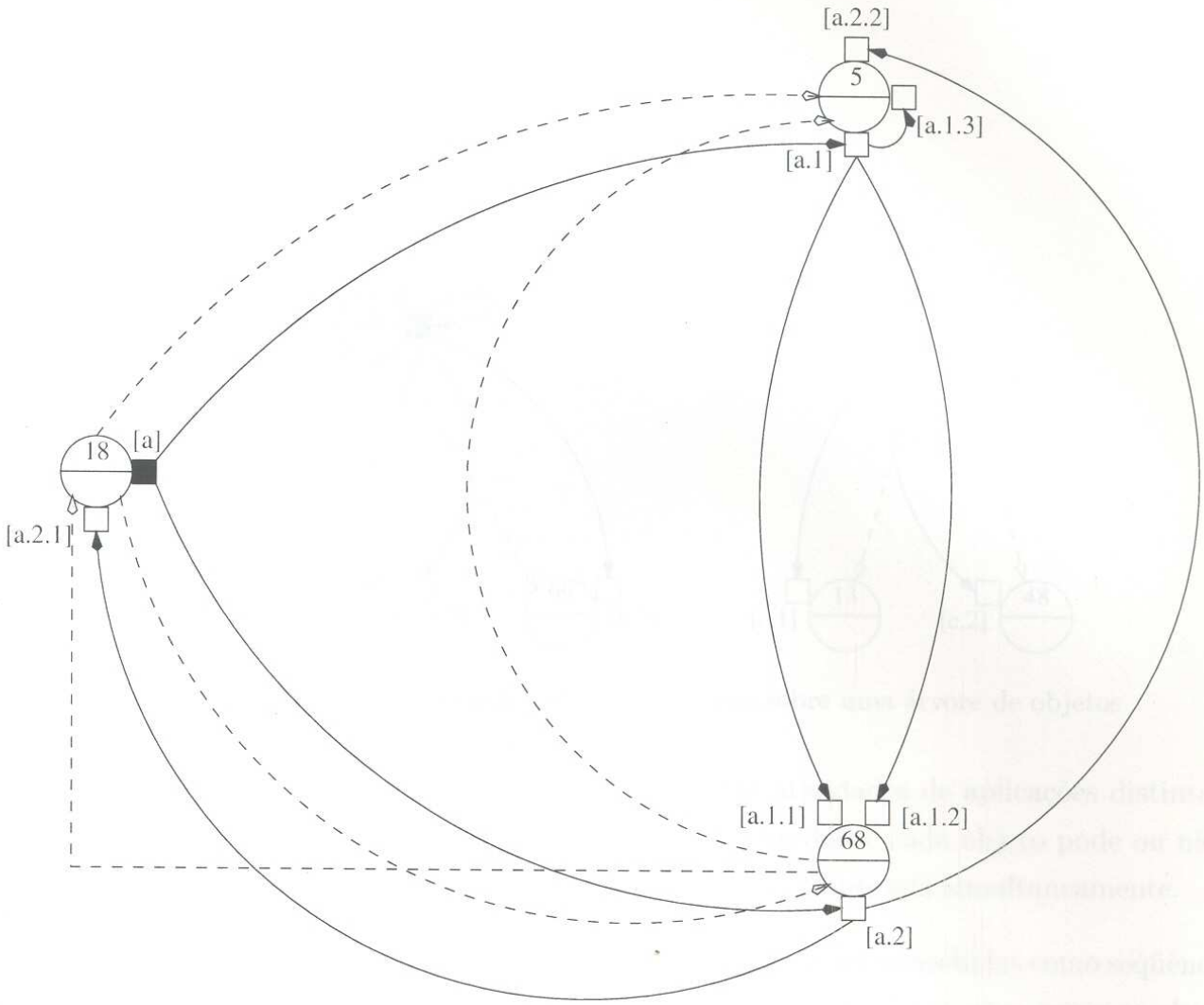


Figura 6.8 Árvore de execução síncrona sobre um grafo de objetos

no qual ocorrem atividades assíncronas sobre um grafo genérico de objetos. Neste exemplo, todos os objetos possuem atividades concorrentes.

6.10 Concepção de Aplicações

Toda aplicação inicia-se pela criação de um objeto, cuja atividade de construção dá início a uma ou mais atividades raizes (por construção). As atividades decorrentes destas, então, podem vir a invocar outras atividades raizes (por invocação), criando estruturas complexas de atividades, dependendo do modo de invocação e de como os objetos se referenciam. Poten-

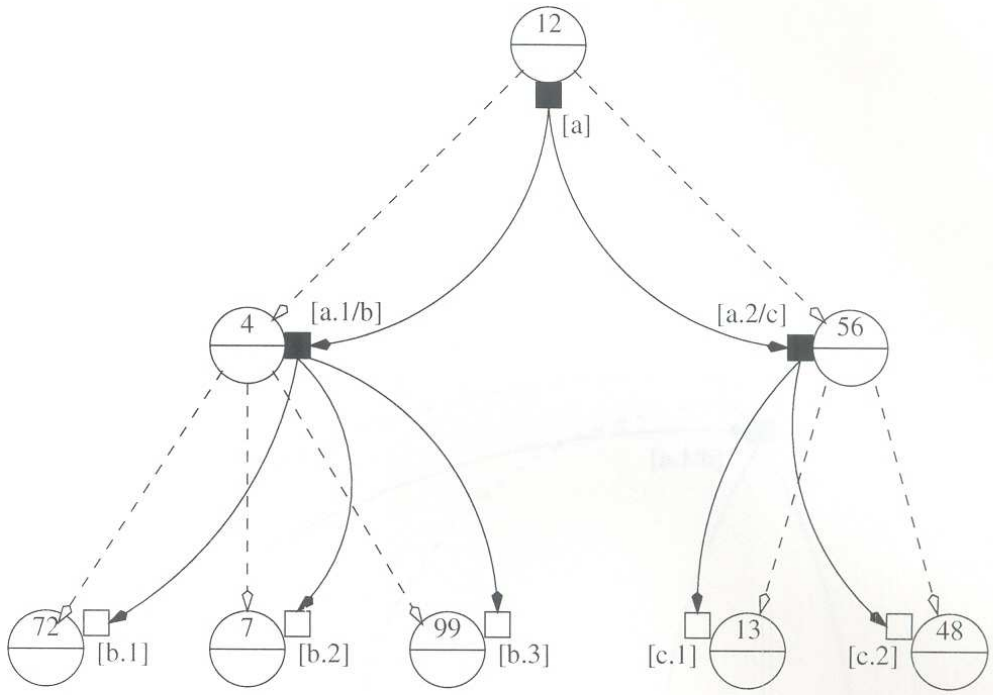


Figura 6.9 Árvores de execução síncrona distintas sobre uma árvore de objetos

cialmente, pode ocorrer concorrência (paralelismo) entre atividades de aplicações distintas e também entre atividades de uma mesma aplicação. Além disso, cada objeto pode ou não ter concorrência interna, isto é, mais de uma atividade não bloqueada simultaneamente.

As Figuras 6.11 e 6.12 ilustram como duas aplicações podem ser concebidas como seqüências de atividades sobre um mesmo conjunto de objetos. A Figura 6.11.a mostra como os objetos referenciam-se uns aos outros, definindo um grafo de objetos. A Figura 6.11.b mostra a estrutura de atividades de uma das aplicações, tendo início na atividade de construção x, enquanto que a Figura 6.11.c mostra a estrutura de atividades da outra aplicação, tendo início na atividade de construção y. A Figura 6.12 mostra o mapeamento das duas estruturas de atividades sobre o conjunto de objetos. Por uma questão de legibilidade, as referências não estão representadas. Uma aplicação inicia-se construindo o objeto 77, enquanto a outra inicia-se contruindo o objeto 92. Alguns objetos têm atividades das duas aplicações concorrentemente. O objeto 23, por exemplo, tem as atividades $a.2$ e $a.2.1.1$ de uma aplicação e a atividade $e.2.2$ da outra. Além disso, as duas atividades da mesma aplicação ($a.2$ e $a.2.1.1$) também são concorrentes entre si e, portanto, o objeto 23 possui três atividades concorrentes. Portanto, o objeto 23 tem concorrência interna.

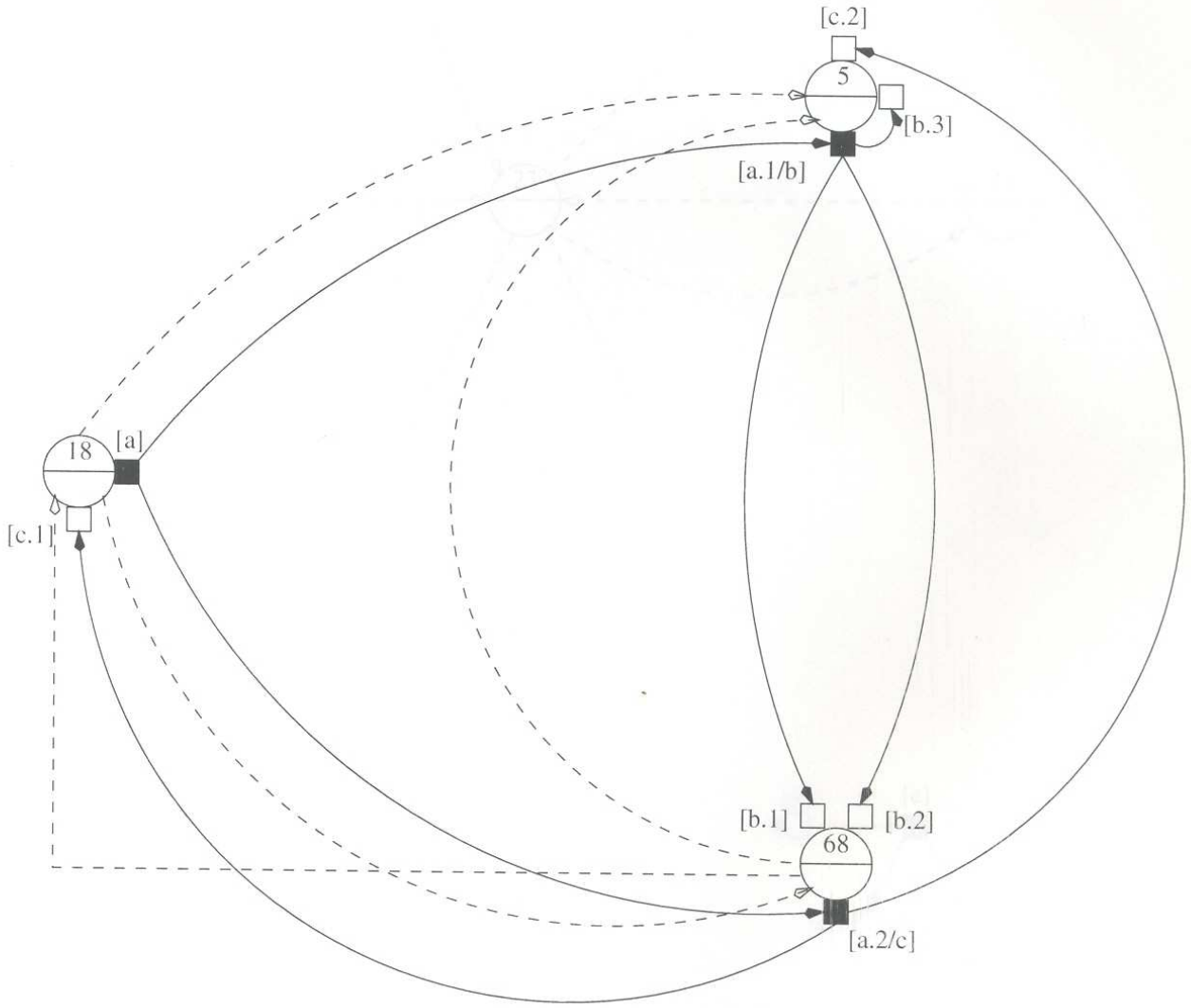
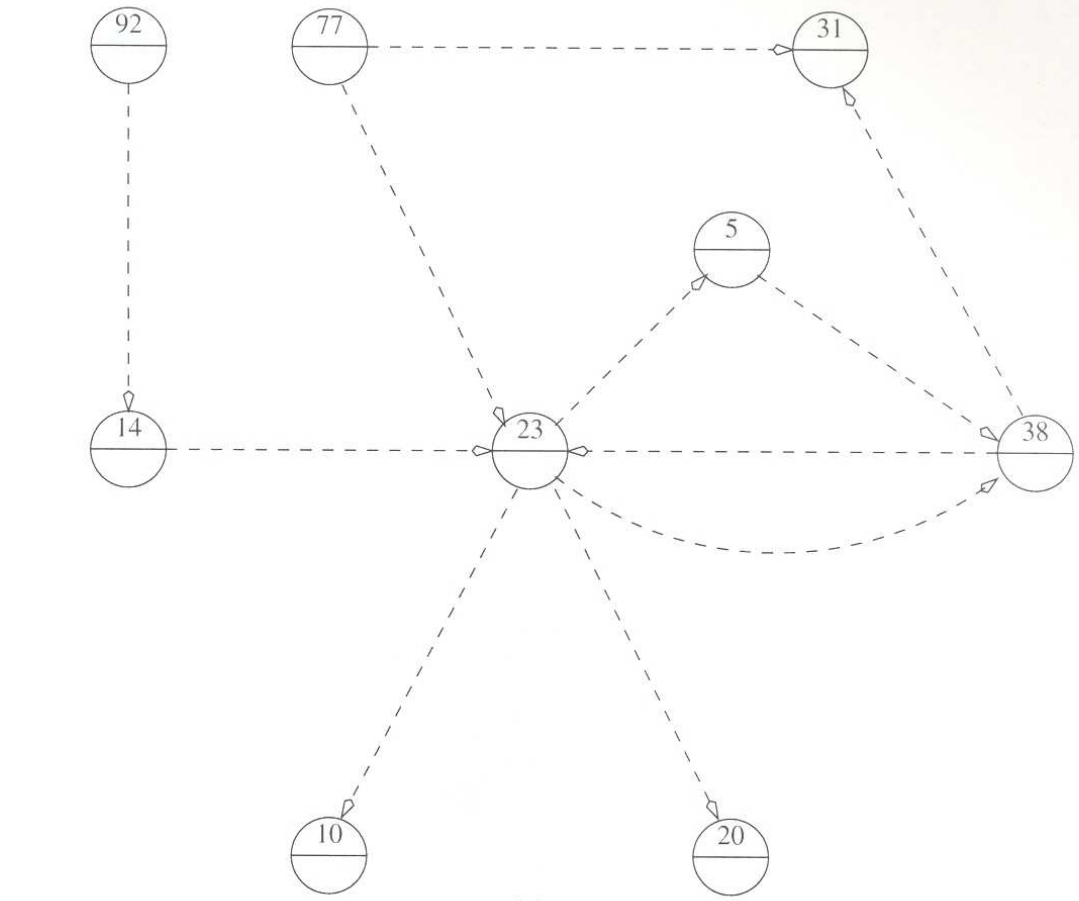
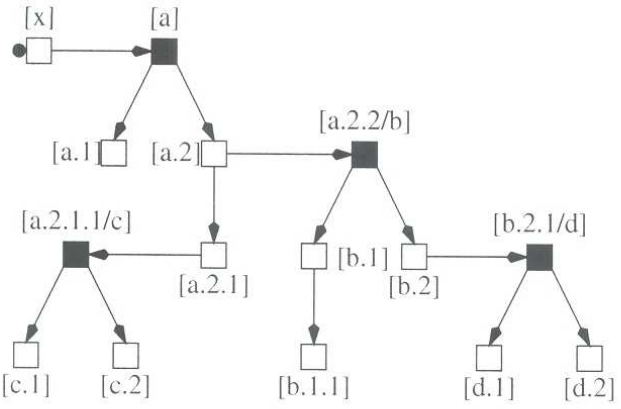


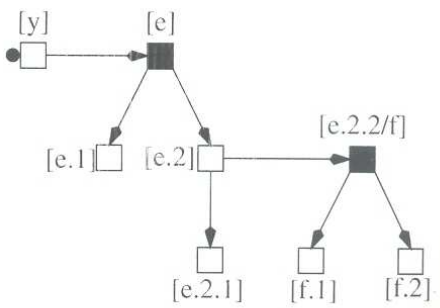
Figura 6.10 Árvores de execução síncrona distintas sobre um grafo de objetos



(a)



(b)



(c)

Figura 6.11 Conjunto de objetos e árvores de execução síncrona

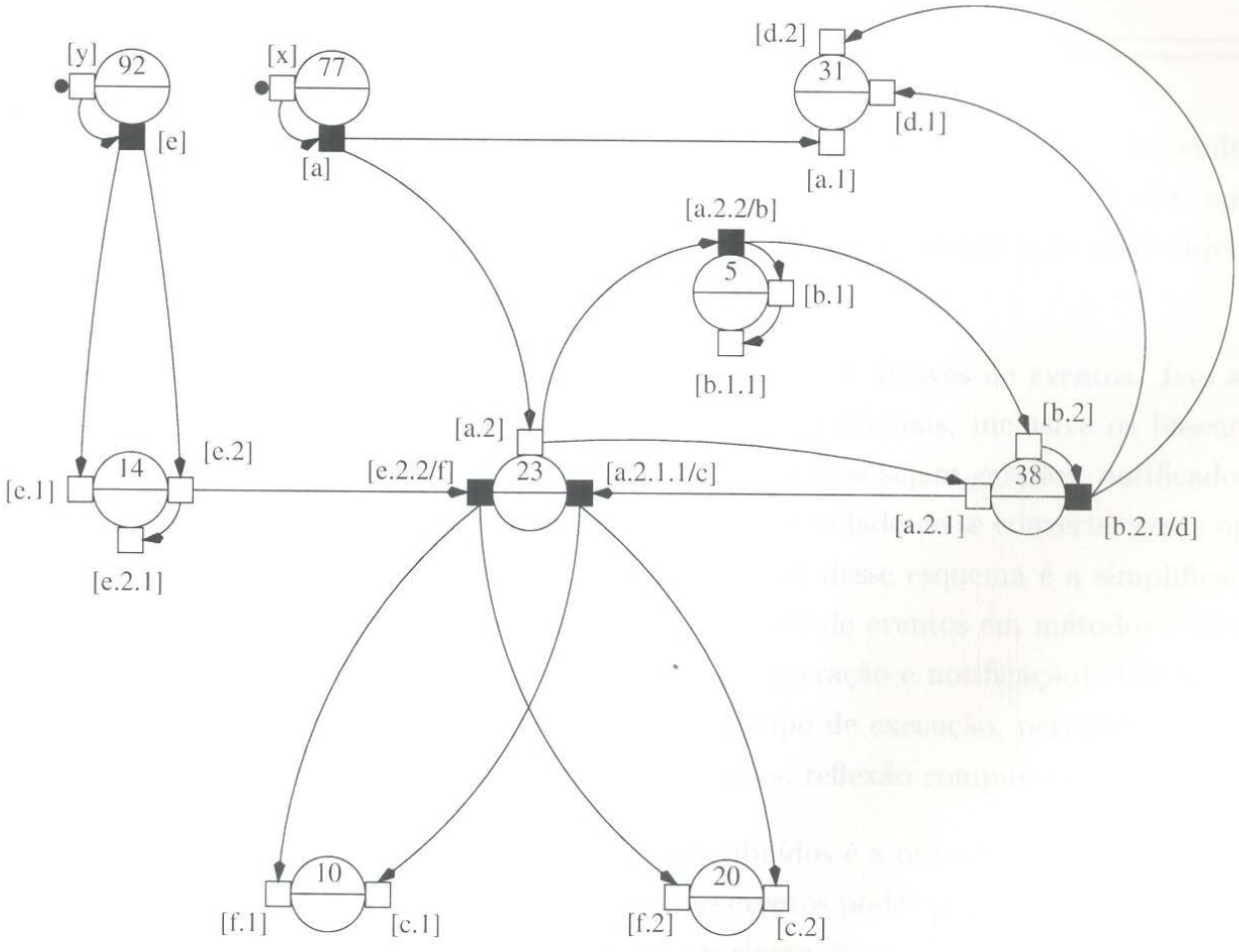


Figura 6.12 Atividades de aplicações distintas sobre um mesmo conjunto de objetos

Capítulo 7

Comunicação por Eventos

Este Capítulo especifica o esquema de comunicação por eventos da *Virtuosi*, definindo o papel de cada elemento envolvido na comunicação e introduzindo uma notação gráfica apropriada. Embora o esquema seja simples, é suficiente para prover a comunicação entre objetos. Obviamente, o esquema pode ser estendido no futuro.

A *Virtuosi* dá suporte direto à comunicação entre objetos através de eventos. Isto significa que, diferentemente dos ambiente de execução convencionais, inclusive os baseados em máquinas virtuais, a *Virtuosi* deve prover para que eventos sejam gerados, notificados e tratados de acordo com sua semântica original, sem a necessidade de se converter essas operações em simples invocações de métodos. Uma vantagem desse esquema é a simplificação dos compiladores, pois não há esforço para transformações de eventos em métodos e nem é necessário gerar código que faça o controle dos eventos (geração e notificação). Outra vantagem é a preservação da informação semântica em tempo de execução, permitindo que se insira mecanismos de segurança, estatísticas, otimizações, reflexão computacional e outros.

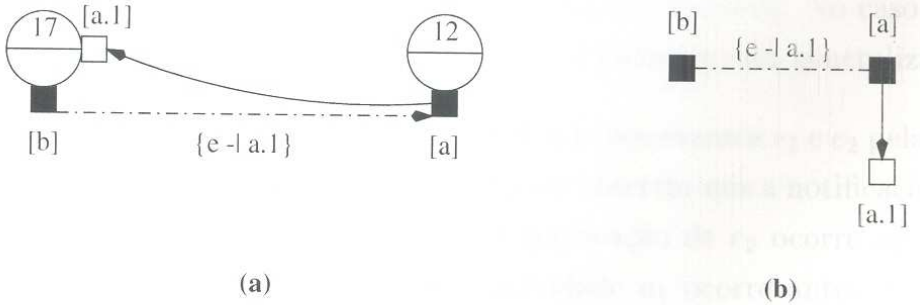
Um dos problemas mais críticos em sistemas distribuídos é a ordenação de eventos, pois uma simples inversão na ordem do tratamento de dois eventos pode comprometer a semântica da aplicação e a integridade da informação. Uma excelente discussão sobre esse problema e uma correspondente solução podem ser encontrados em [Lamport, 1978]. O esquema de comunicação por eventos definido neste Capítulo conta com disponibilidade de um mecanismo que cuide da ordenação de eventos.

Um evento na *Virtuosi* também é um objeto, isto é, também tem um estado e um conjunto de métodos. Dessa forma, um evento pode ser utilizado para levar informação de um objeto para outro e os seus métodos podem ser invocados, por exemplo, no seu tratamento.

7.1 Esquema Básico

Dois objetos comunicam-se através de um evento quando as suas atividades cuidam para demonstrar interesse pelo evento, notificar o evento e tratá-lo apropriadamente. Primeiramente, uma atividade que esteja interessada em um certo evento deve registrar esse interesse com o correspondente objeto gerador do evento; esse registro é feito através da invocação de uma atividade síncrona do objeto gerador. Após efetuado esse registro, toda vez que uma atividade qualquer do objeto gerador gerar o evento, a atividade que registrou seu interesse pelo evento recebe a notificação. A atividade que gera o evento é denominada *atividade geradora*, enquanto que a atividade que recebe o evento é denominada *atividade receptora*.

A Figura 7.1 ilustra o esquema básico de comunicação por eventos e apresenta a notação gráfica introduzida para esse fim. A Figura 7.1.a ilustra os objetos e as correspondentes atividades participantes, enquanto que a Figura 7.1.b ilustra a correspondente estruturação das atividades. Primeiramente, a atividade a do objeto 12 registra, fazendo invocação da atividade $a.1$, o seu interesse pelo evento e que pode ser gerado pelo objeto 17. Quando, mais tarde, a atividade b do objeto 17 gera o evento e, este é notificado à atividade a. Deve-se observar que a notificação de evento é representada por uma linha pontilhada, finalizada com uma seta indicando a atividade receptora. O evento sendo notificado é anotado entre chaves, indicando ainda a atividade do objeto gerador invocada para registrar o interesse pelo evento ($a.1$ no exemplo). Essa convenção permite identificar a ordem em que as atividades e a notificação de evento ocorrem.



(a) **(b)**

Figura 7.1 Esquema básico de comunicação por eventos

Esse esquema de comunicação por eventos requer a colaboração de todas as partes envolvidas, como segue:

- A atividade receptora deve ter uma identidade, isto é, uma identificação única que permita ser referenciada pela atividade geradora.
- O objeto gerador deve manter uma tabela de todos os eventos que suas atividades podem gerar e, para cada um destes, a lista de atividades receptoras que registraram interesse.
- A atividade receptora deve programar-se para receber a notificação de evento a qualquer momento após o correspondente registro de interesse, pois a notificação de evento ocorre de maneira assíncrona. Essa é razão pela qual a atividade de registro de interesse é síncrona. A atividade receptora tem ainda a opção de bloquear-se até a recepção do evento.
- A atividade geradora tem a opção de bloquear-se quando faz um notificação de evento, programando-se para receber um outro evento que permita retornar à execução.

7.2 Generalização do Esquema

A comunicação por evento não restringe-se a comunicação simples de um-para-um, ou uma atividade para uma atividade: é possível que uma atividade receptora programe-se para receber eventos de diversos objetos geradores e também é possível que um mesmo evento seja simultaneamente notificado a mais de uma atividade receptora. No caso geral, tem-se comunicação muitos-para-muitos. As Figuras 7.2 e 7.3 ilustram essa generalização.

Na Figura 7.2, a atividade a do objeto 12 é notificada dos eventos e_1 e e_2 pelas atividades b do objeto 17 e c do objeto 6, respectivamente. Pode-se observar que a notificação de e_1 ocorre após a atividade de registro a_1, enquanto que a notificação de e_2 ocorre após a atividade de registro a_2. Pode-se ainda observar que a atividade a_1 ocorre antes da atividade a_2. Entretanto, não é possível dizer em que ordem ocorrem as notificações. De qualquer forma, a atividade a tem que programar-se para receber os dois eventos de maneira assíncrona e em qualquer ordem.

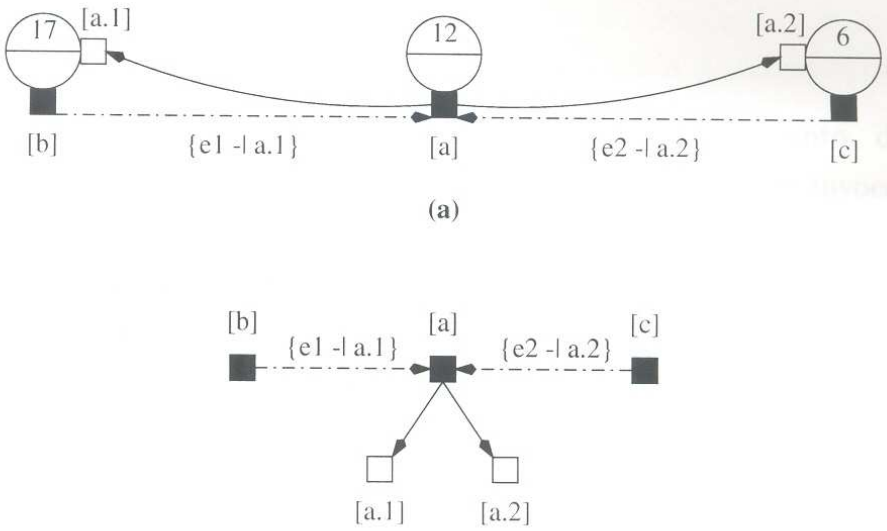


Figura 7.2 Notificação de eventos por mais de uma atividade

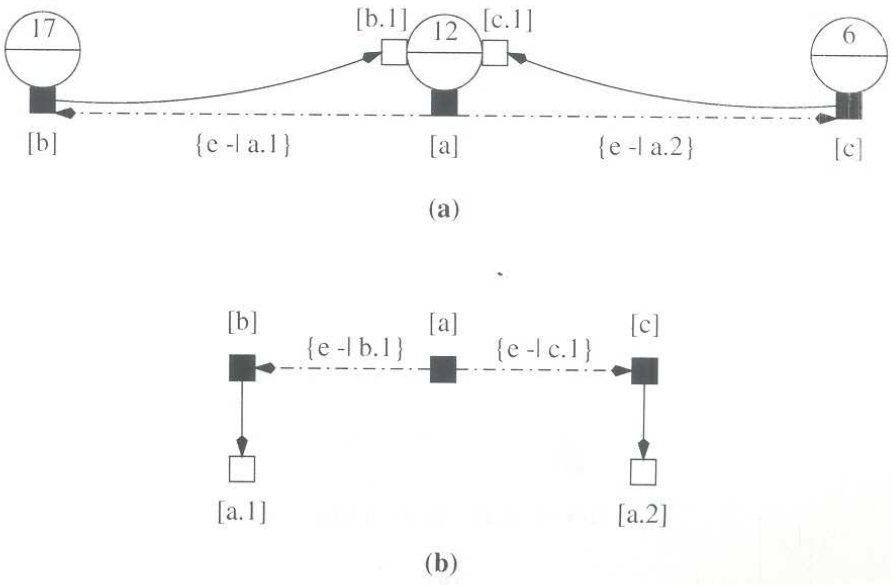


Figura 7.3 Notificação de um evento a mais de uma atividade

Na Figura 7.3, a atividade a notifica o evento e às atividades b do objeto 17 e c do objeto 6, simultaneamente. As atividades de registro b_1 e c_1 atualizam a tabela de notificação de eventos mantida pelo objeto 12, tal que quando o evento e é gerado pela atividade a, esta descobre que deve notificar as atividades b e c.

7.3 Tratamento de Eventos

Uma atividade receptora deve dar o devido tratamento para o evento, dependendo da semântica da aplicação. Opcionalmente, a atividade receptora pode invocar novas atividades, assim como pode criar novos objetos para fazer o tratamento. A Figura 7.4 ilustra uma situação em que novas atividades são invocadas no tratamento de eventos: a atividade a do objeto 12 invoca as atividades $a.3$ e $a.4$, no tratamento dos eventos e_1 e e_2, notificados pelas atividades b do objeto 17 e c do objeto 6. Não é possível determinar, entretanto, qual atividade é invocada no tratamento de cada evento recebido. Pode-se observar que a atividade $a.3$ é síncrona, enquanto que a atividade $a.4$, ou simplesmente d, é assíncrona, exemplificando a liberdade existente no arranjo de atividades.

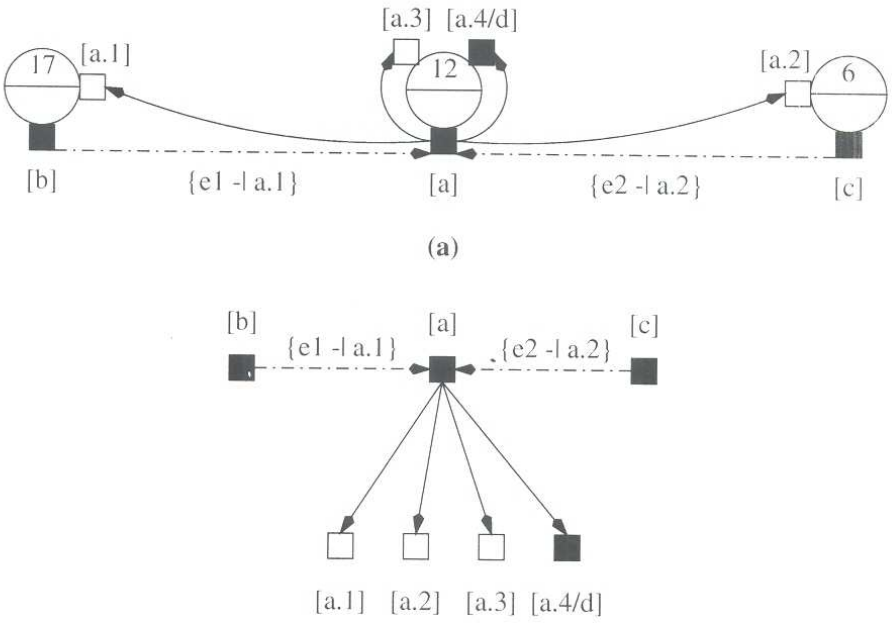
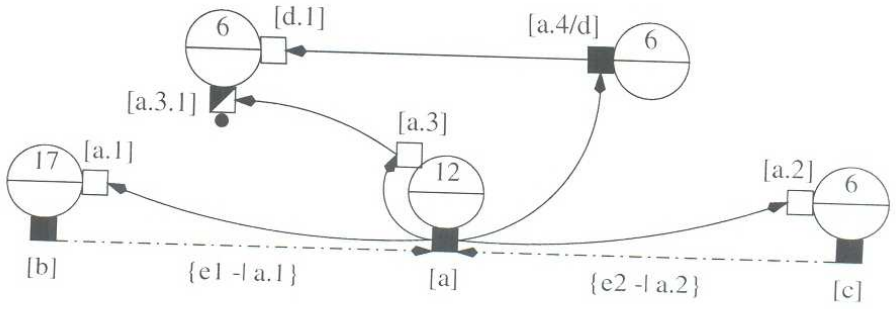


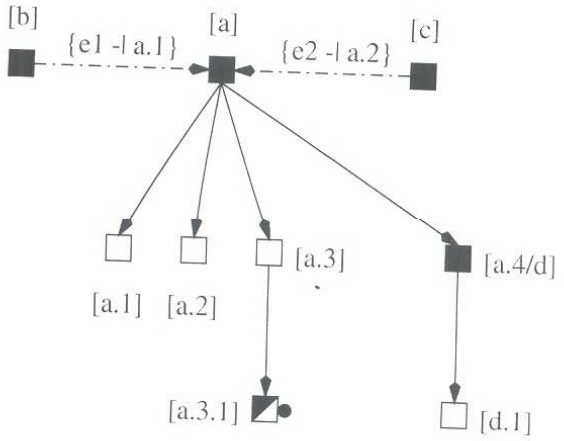
Figura 7.4 Iniciação de novas atividades no tratamento de eventos

A Figura 7.5 ilustra uma situação na qual um novo objeto é criado como conseqüência do tratamento de um evento: o objeto 6 é criado pela atividade $a.3$ do objeto 12 que foi invocada pela atividade receptora a deste mesmo objeto, como conseqüência do tratamento do evento e_1 ou do evento e_2. A Figura ainda mostra a invocação da atividade $a.4$ (ou

d) do objeto 6 como conseqüência do tratamento de um desses dois eventos, sendo que essa atividade, por sua vez, invoca a atividade *d*.1 do objeto 6, recém criado. Embora a notação não permita determinar a seqüência exata de atividades e eventos, certamente o encadeamento de atividades *d* e *d*.1 ocorre após o encadeamento *a*.3 e *a*.3.1.



(a)



(b)

Figura 7.5 Construção de objetos no tratamento de eventos

Capítulo 8

Interação com Sistemas Reais

Os objetos da *Virtuosi* realizam computações completas e, para tanto, precisam interagir com o mundo real, isto é, com os dispositivos do ambiente de execução que são externos às máquinas virtuais, tais como impressoras, discos, etc. Esses dispositivos não restrigem-se a elementos de hardware – um gerenciador de bancos de dados, por exemplo, também pode ser considerado com um dispositivo externo. Dessa forma, é possível construir aplicações na *Virtuosi* que interagem com qualquer elemento do sistema real de execução, incluindo componentes de software que executam sobre outros ambientes, concebidos ou não sob o paradigma de orientação a objetos. Este Capítulo define como essa interação ocorre, permitindo inclusive que objetos acessem de forma transparente dispositivos localizados em computadores remotos, e introduz uma notação gráfica apropriada.

8.1 Dispositivos Externos

Os dispositivos existentes em ambientes de execução que são externos à *Virtuosi* assumem as mais diversas formas, tanto em hardware quanto em software, incluindo porta paralela, porta serial, socket, sistema de mail, servidor Web, servidor de nomes, gerenciador de segurança, gerenciador de janelas, sistema legado, sistema gerenciador de banco de dados, sistema operacional, sistema gerenciador de arquivos, gerenciador de memória, teclado, mouse, câmera, relógio, sensor, disco, rede, impressora, etc. A designação *dispositivo externo* é utilizado em preferência *periférico*, uma vez que o conjunto de dispositivos externos à *Virtuosi* inclui o conjunto de periféricos dos computadores que a suportam e ainda inclui os processos e serviços disponíveis nesses computadores.

A Figura 8.1 ilustra uma situação na qual há um conjunto de objetos da *Virtuosi* e um conjunto de dispositivos externos. A *Virtuosi* está implementada por um conjunto de máquinas virtuais (não representadas na Figura) que executam nos computadores C_1, C_2, C_3, C_4 e C_5. Cada um desses computadores hospeda um ou mais dispositivos externos, representados por um retângulo com cantos arredondados. O computador C_1, por exemplo, hospeda o dispositivo *sensor*, enquanto que o computador C_2 hospeda os dispositivos *mouse*, *teclado* e *vídeo*. A questão a ser respondida é como os objetos da *Virtuosi*, que existem somente em um espaço virtual, podem interagir com os dispositivos externos.

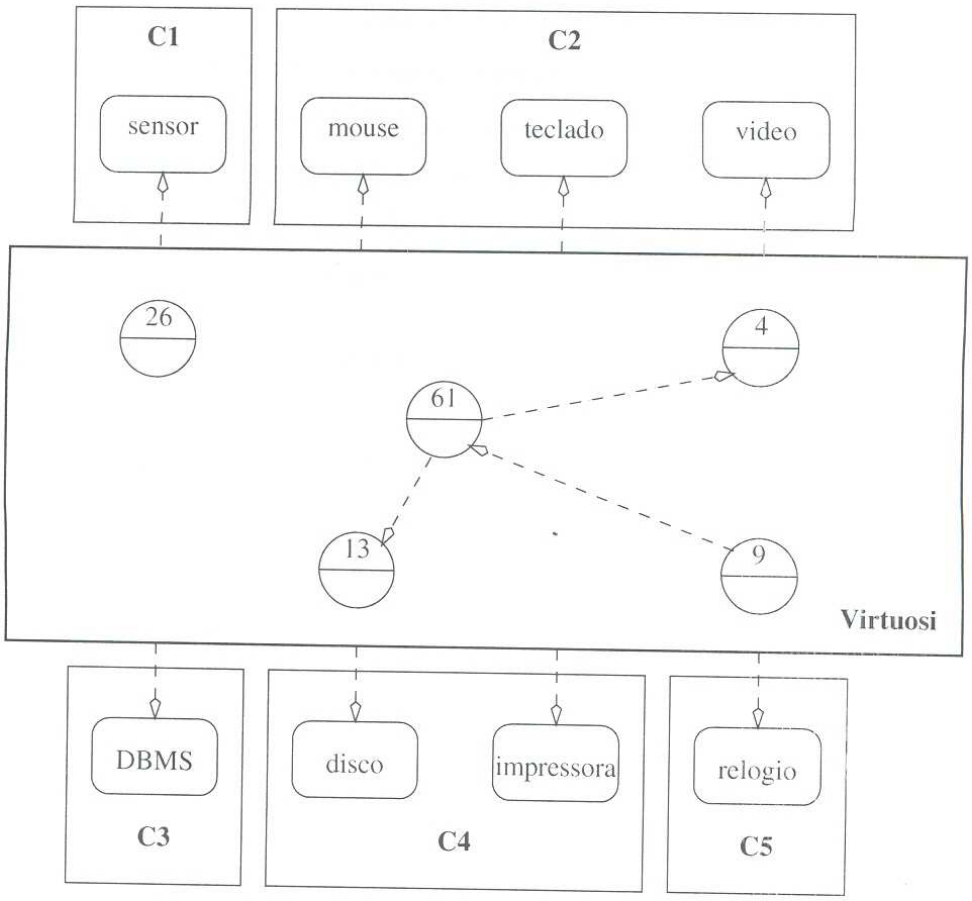


Figura 8.1 Dispositivos externos à *Virtuosi*

8.2 Objetos de Fronteira

A interação entre objetos da *Virtuosi* com dispositivos externos ocorre através de objetos especialmente projetados para esse fim. Esses objetos situam-se, logicamente, na fronteira entre a *Virtuosi* e o sistema real de execução, sendo, por essa razão, denominados *objetos de fronteira*. Os objetos da *Virtuosi* interagem com os objetos de fronteira que, por sua vez, interagem com os dispositivos externos. Essa interação pode ocorrer nos dois sentidos, ou seja, é possível que os objetos de fronteira sejam afetados por eventos gerados no contexto do dispositivo externo e, como conseqüência, invoque atividades dos objetos normais. Para cumprir esse papel, um objeto de fronteira deve ter uma interface de acordo com o modelo de objetos definido pela *Virtuosi* e, ao mesmo tempo, a implementação de seus métodos deve ser capaz de atuar nos dispositivos externos, o que somente é possível se for feita em código nativo. (Isso impede que tenham reflexão computacional.) Tipicamente, a interface de um objeto de fronteira disponibiliza métodos para operações de entrada e saída e métodos que fazem tratamento de eventos de origem externa.

A Figura 8.2 ilustra uma interação entre um objeto de fronteira e um dispositivo externo e introduz uma notação gráfica para indicar esse tipo de objeto. O objeto 51 é um objeto de fronteira que interage com o dispositivo externo *disco*. A atividade a do objeto é responsável pelo recebimento de eventos gerados pelo disco. No exemplo, o tratamento do evento e causa a invocação da atividade $a.1$ que, por sua vez, atua sobre o disco. A atividade b não está relacionada ao tratamento de eventos externos; pode ter sido invocada assincronamente por uma atividade de outro objeto ou pode ser uma atividade raiz por construção. Essa atividade também atua no disco.

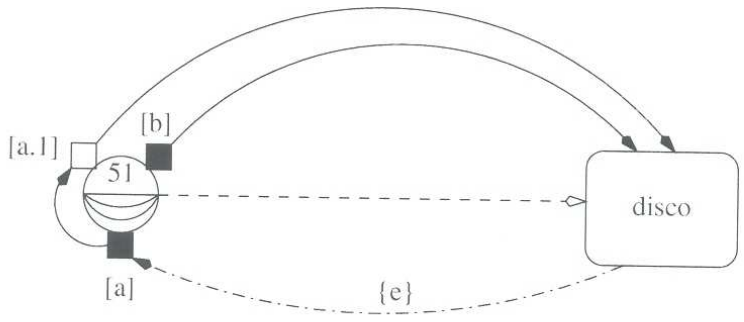


Figura 8.2 Interação entre um objeto de fronteira e um dispositivo externo

A Figura 8.3 ilustra o uso de objetos de fronteira para permitir a interação de objetos da *Virtuosi* com dispositivos externos. O exemplo é o mesmo mostrado na Figura 8.2, com acréscimo dos objetos de fronteira: foi acrescentado um objeto de fronteira para cada dispositivo externo. Pode-se notar que existem referências entre os objetos de fronteira e os demais, em ambos os sentidos, indicando que a interação é bi-direcional. A representação dos objetos de fronteira como parte da própria *Virtuosi* sugere que estes objetos têm estrutura e comportamento segundo o modelo de objetos da *Virtuosi* e, por isso, interagem normalmente com os demais. O objeto de fronteira 70, por exemplo, pode ter uma atividade invocada pelo objeto 9 a fim de programar o *relógio* para disparar um alarme num certo horário. Quando isso ocorre, o objeto 9 faz o tratamento do evento e comunica o objeto 70.

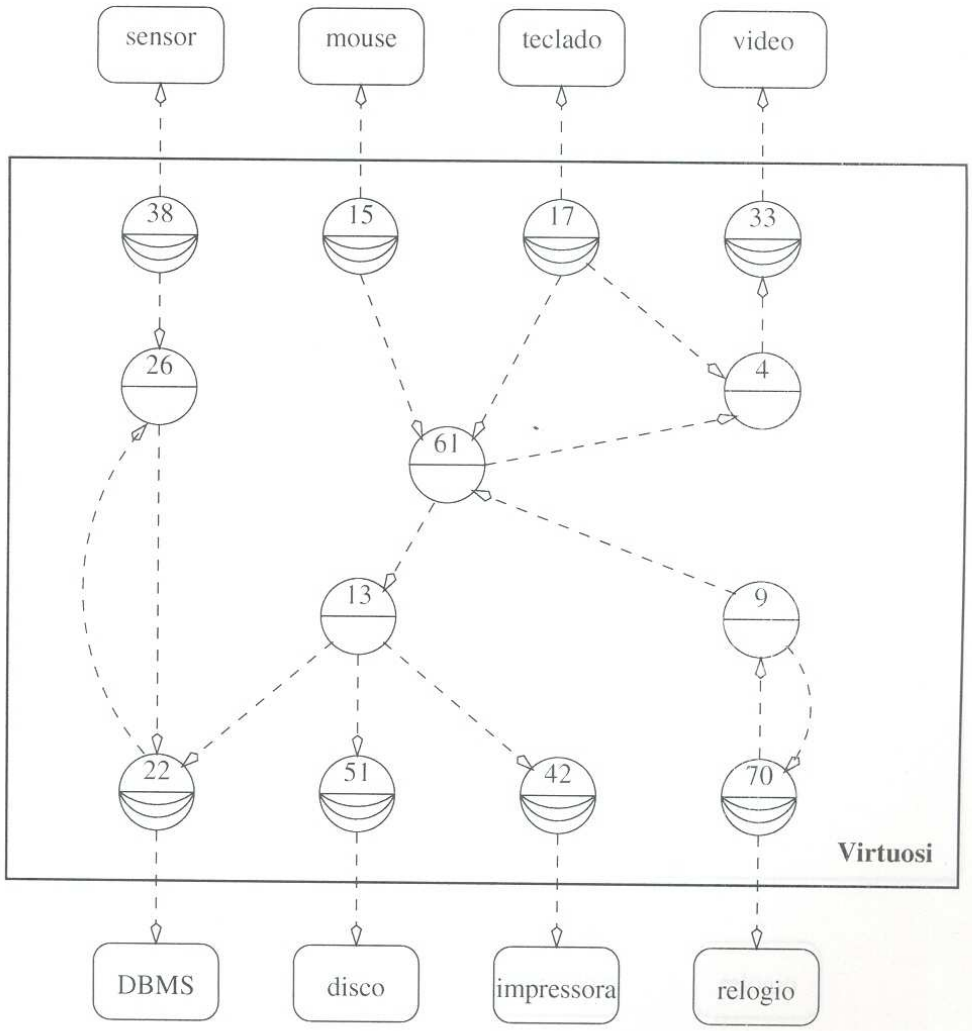


Figura 8.3 Objetos de fronteira na *Virtuosi*

8.3 Configuração de Objetos de Fronteira

A configuração dos objetos de fronteira, isto é, a sua alocação nas máquinas virtuais é livre e deve ser decidida de acordo com a aplicação, o tipo de dispositivo externo, as restrições de segurança do sistema real, etc.

A configuração dos objetos de fronteira correspondente ao exemplo da Figura 8.3 é ilustrada pelas Figuras 8.4 e 8.5, que mostram a distribuição dos objetos, respectivamente, pelas máquinas virtuais – identificadas por $V_1 \ldots V_7$ – e pelos computadores – identificados por $C_1 \ldots C_5$.

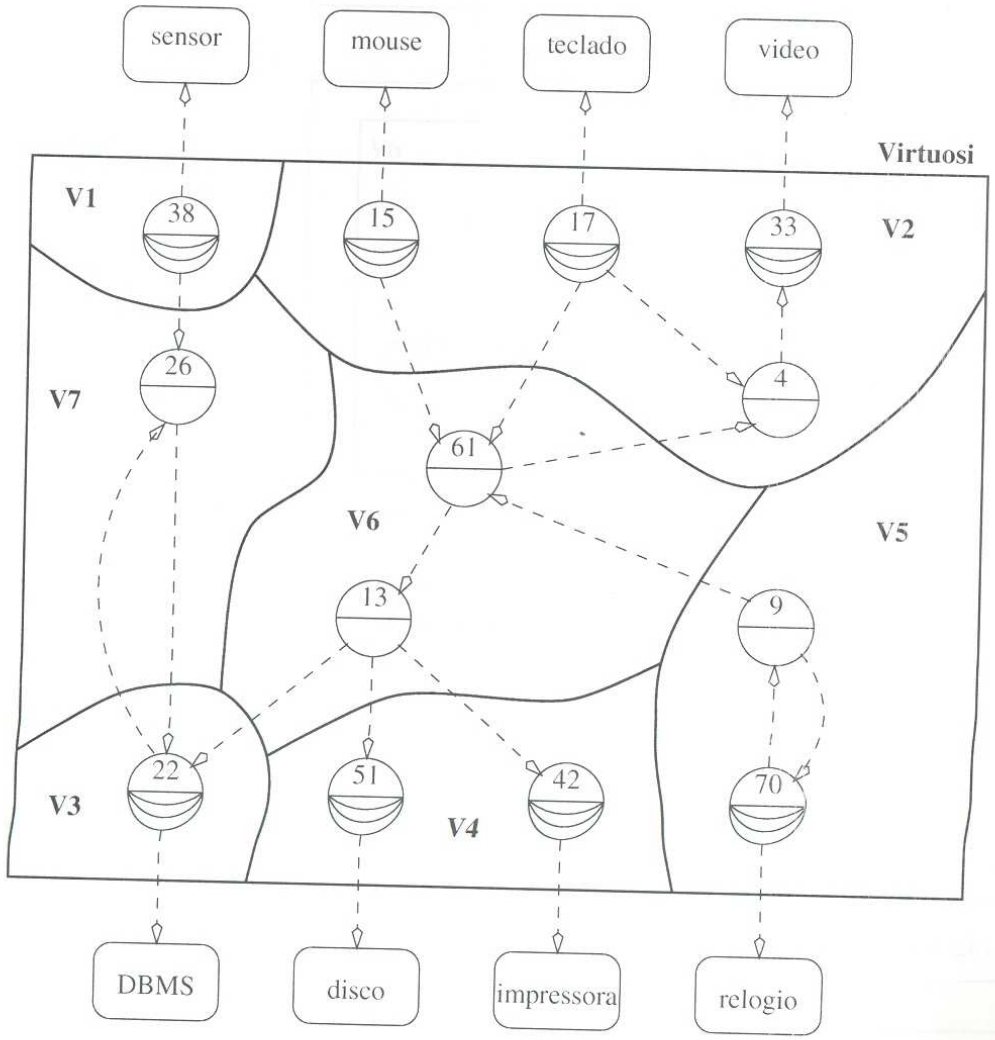


Figura 8.4 Particionamento lógico da *Virtuosi* na interação com dispositivos externos distribuídos

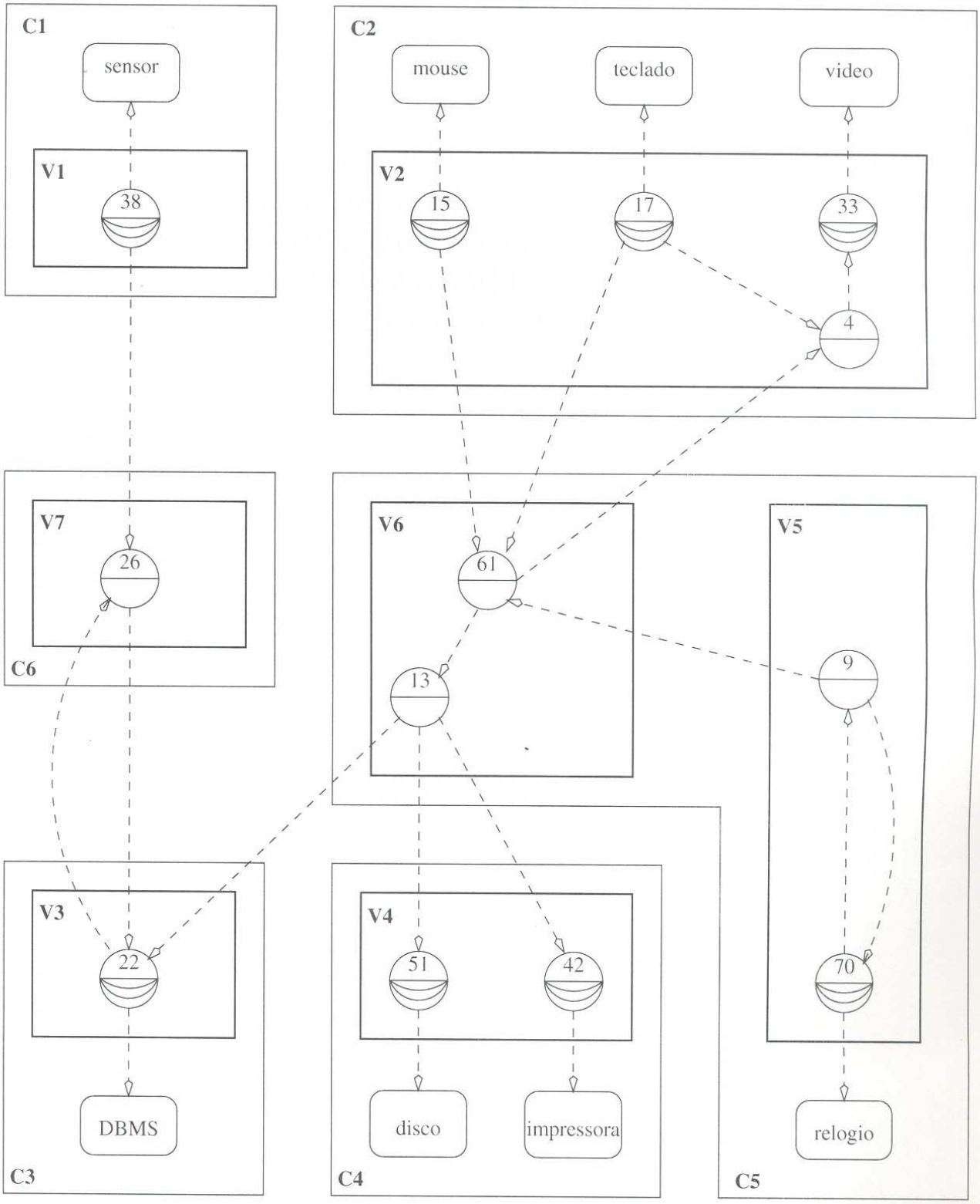


Figura 8.5 Particionamento físico da *Virtuosi* na interação com dispositivos externos distribuídos

Na Figura 8.4 observa-se que qualquer objeto da *Virtuosi* pode interagir com qualquer dispositivo externo através do correspodente objeto de fronteira, desde que haja uma referência entre eles. O objeto 13, por exemplo, interage com o *DBMS*, o *disco* e a *impressora*, através dos objetos de fronteira 22, 51 e 42, respectivamente.

A Figura 8.5, entretanto, mostra que essa interação é mais complexa do que aparenta, pois mostra que um objeto da *Virtuosi* hospedado em um certo computador pode interagir com um dispositivo externo hospedado em outro computador. O mesmo objeto 13 interage com dispositivos externos localizados em computadores distintos, pois enquanto este hospeda-se no computador C_5 – que hospeda a máquina virtual V_6 – os dispositivos externos hospedam-se nos computadores C_3 e C_4.

O exemplo ilustra a situação mais tipicamente esperada: há somente um objeto de fronteira para cada dispositivo externo, e vice-versa, e ambos hospedam-se no mesmo computador. Entretanto, no caso geral, podem haver vários objetos de fronteira, inclusive hospedados em máquinas virtuais distintas, associados a um único dispositivo externo. Por outro lado, um objeto de fronteira pode estar associado a vários dispositivos externos, hospedados em computadores distintos, inclusive.

Capítulo 9

Reflexão Computacional

Este Capítulo discute como o conceito de reflexão computacional aplica-se nos modelos de objeto e de comunicação da *Virtuosi*. A reflexão computacional é realizada sobre um objeto com a anexação a este de um ou mais *metacomponentes*. A reflexão computacional é discutida primeiramente sob a perspectiva das operações realizadas nos objetos em seu funcionamento normal, como objetos participantes de aplicações, listando as respectivas funções desempenhadas pelos metacomponentes. Em seguida, é discutida sob a perspectiva dos serviços de apoio que um ambiente de execução deve prover, detalhando como esses podem ser realizados através de atuações de metacomponentes sobre os objetos.

9.1 Operações Reflexivas

A operações básicas em objetos podem ser interceptadas por metacomponentes e reificadas de acordo com uma semântica própria. A lista a seguir mostra as operações básicas sobre objetos que podem sofrer reflexão computacional e que, por isso, são denominadas *operações reflexivas*. Ainda, para cada operação reflexiva consta uma relação específica de funções que podem ser desempenhadas pelos metacomponentes.

(1) Acessar o estado do objeto para leitura

   (a) controle de segurança

   (b) controle de concorrência

   (c) registro para fins estatísticos

   (d) registro para fins contábeis

(2) Acessar o estado do objeto para escrita

(a) controle de segurança

(b) controle de concorrência

(c) registro para fins estatísticos

(d) mecanismo de persistência

(e) atualização de índices de busca

(3) Receber uma invocação de método

(a) controle de segurança

(b) controle de concorrência

(c) registros para fins estatísticos

(d) verificação de licença

(e) redirecionamento da invocação para outro método do próprio objeto

(f) redirecionamento da invocação para um método de outro objeto, inclusive do próprio metacomponente

(g) cancelamento da invocação e envio de um objeto de retorno

(h) balanceamento de carga

(4) Enviar o objeto de retorno de uma atividade

(a) modificações no objeto de retorno

(b) registros para fins estatísticos

(5) Fazer uma invocação de método

(a) controle de segurança

(b) registros para fins estatísticos

(6) Criar um objeto

(a) controle de segurança

(b) registros para fins estatísticos

(c) alocação de objetos

(d) atualização de índices de busca

(7) Notificar um evento

(a) cancelamento da notificação

(b) notificações de outros eventos

(c) modificações no evento

(d) substituição do evento

(e) registros para fins estatísticos

(8) Receber um evento

(a) cancelamento do recebimento

(b) notificações de outros eventos

(c) modificações no evento

(d) substituição do evento

(e) registros para fins estatísticos

(9) Verificar uma asserção (pré-condição, pós-condição ou invariante)

(a) retorno de um resultado fixo (verdadeiro ou falso)

(b) substituição da asserção

(c) extensão da asserção

(d) registros para fins estatísticos

(10) Gerar uma exceção

(a) cancelamento da geração

(b) modificações na exceção

(c) substituição da exceção

(d) registros para fins estatísticos

(11) Receber uma exceção

(a) cancelamento do recebimento

(b) modificações na exceção

(c) substituição da exceção

(d) registros para fins estatísticos

9.2 Atuações de Metacomponentes

Podem haver metacomponentes que tomem a iniciativa de atuar sobre os correspondentes objetos, enquanto estes desempenham suas atividades funcionais. Essa atuação ocorre como parte das atividades normais do metacomponente. Para o objeto que sofre a atuação, entretanto, esta ocorre de forma assíncrona. A lista a seguir contém as principais formas de atuação de metacomponentes, descrevendo-as brevemente.

(1) Migrar o objeto: Tem por objetivo balancear a carga do sistema. Como conseqüência, todas as referências para o objeto devem ser atualizadas.

(2) Destruir o objeto (fazer coleta de lixo): Deve verificar se o objeto não possui qualquer atividade, não é referenciado por outro e não participa de uma transação atômica; se todas essas condições forem satisfeitas, o objeto pode ser destruído. Como conseqüência, os índices de busca devem ser atualizados.

(3) Persistir o estado do objeto: Salva o estado do objeto em memória estável de forma serializada.

(4) Fazer cópia do objeto por ocasião de sua entrada em uma transação atômica, armazenamento em cache ou início de computação desconectada: O estado serializado do objeto é copiado para uma área reserva alocada em memória estável nos casos de transação atômica e cache, enquanto que uma cópia temporária do objeto é criada no caso de computação desconectada.

(5) Restaurar o estado do objeto devido a falhas ou aborto de transação atômica: O estado do objeto retorna ao último estado copiado na área reserva.

(6) Terminar uma transação atômica com confirmação: A imagem persistente do objeto passa a corresponder ao novo estado do objeto e as cópias feitas na área reserva podem ser descartadas.

(7) Terminar uma transação atômica com aborto: O estado do objeto é restaurado.

(8) Serializar o estado do objeto: O estado do objeto é convertido para um formato que permita o seu armazenamento em memória estável e o seu transporte entre máquinas virtuais.

(9) Verificar se o estado do objeto foi modificado para fins de consistência de réplicas, cache e imagem persistente do objeto: Compara o estado atual do objeto com o último estado copiado na área reserva.

(10) Atualizar o estado do objeto para fins de consistência de réplicas: O estado da réplica é atualizado de acordo com a cópia temporária criada para computação desconectada.

(11) Atualizar o estado do objeto para fins de consistência de cache: O estado do objeto em cache é atualizado ou invalidado de acordo com a cópia temporária criada para computação desconectada.

(12) Atualizar o estado do objeto para fins de reunificação de computações desconectadas: O estado do objeto é atualizado de acordo com a cópia temporária criada para computação desconectada.

(13) Atualizar a versão do objeto devido a alterações na definição da correspondente classe: O formato do estado, assim como as assinaturas e implementações de métodos são atualizados.

(14) Verificar se o objeto satisfaz uma expressão de consulta (*query*): São feitas comparações do estado do objeto e resultados de invocações de métodos com elementos da expressão de consulta.

(15) Verificar se o objeto provê um méodo compatível com uma assinatura (*trading*): As assinaturas de métodos do objeto são comparadas com a assinatura fornecida.

(16) Verificar se o objeto contém uma referência para um outro objeto específico (relacionamento): Todas as referências mantidas pelo objeto são comparadas com a referência fornecida.

9.3 Organização de Metacomponentes

Metacomponentes são também objetos da *Virtuosi*, permitindo que comuniquem entre si e também com qualquer outro objeto, inclusive com objetos de fronteira, e permitindo que haja reflexão computacional sobre os próprios metacomponentes, caracterizando uma estrutura organizacional recursiva. Por exemplo, um metacomponente responsável pelo gerenciamento de uma transação atômica pode comunicar-se com os metacomponentes responsáveis por persistência, controle de concorrência e restauração; se a transação envolver vários objetos distribuídos, deve haver ainda cooperação entre os respectivos metacomponentes responsáveis por transação atômica. Além do mais, um metacomponente responsável por persistência pode interagir com um objeto de fronteira associado a um banco de dados ou a um sistema de arquivos. Ou seja, metacomponentes comportam-se e comunicam-se como qualquer outro objeto. O que difere os metacomponentes dos objetos básicos é a permissão que possuem para interpretar e alterar a estrutura e o comportamento dos objetos vinculados.

A Figura 9.1 ilustra um conjunto de objetos organizados tal que alguns atuam como metacomponentes e introduz a notação gráfica para indicar quando um objeto é metacomponente de outro objeto. Os objetos 1, 2, 3 e 4 são metacomponentes do objeto 6. A única indicação de que um objeto é um metacomponente é a referência que o metacomponente mantém para o(s) objeto(s) sobre o qual reflete; essa referência é anotada graficamente por uma linha cheia com uma seta indicando o objeto sobre o qual o metacomponente reflete. O exemplo ainda mostra que o metacomponente 4 possui referências para os demais para fins de invocação de métodos. O método componente 4 tem também uma referência para o objeto de fronteira 5, que interage com um *DBMS*. Neste exemplo, os objetos 1, 2, 3 e 4, podiam ter, respectivamente, as funções de gerenciamento de transação atômica, controle de concorrência, restauração e persistência.

A Figura 9.2 ilustra uma situação em que ocorre a reflexão computacional é aplicada recursivamente na organização dos objetos: o objeto 1 é um metacomponente do objeto 2 que, por sua vez, é um metacomponente do objeto 3. Além disso, o exemplo mostra a liberdade de interação dos metacomponentes entre si próprios e com os objetos básicos: o objeto 4, outro metacomponente do objeto 3, interage com o metacomponente 2, com o objeto básico 5 e com o objeto 6, que é metacomponente do objeto 7.

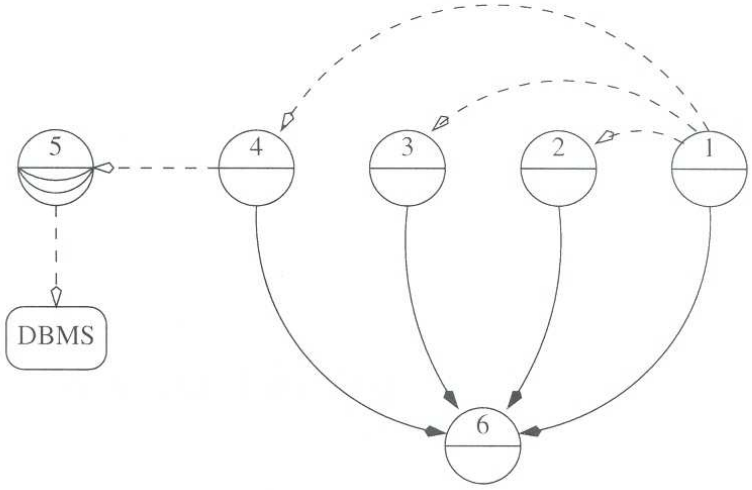


Figura 9.1 Reflexão computacional e metacomponentes

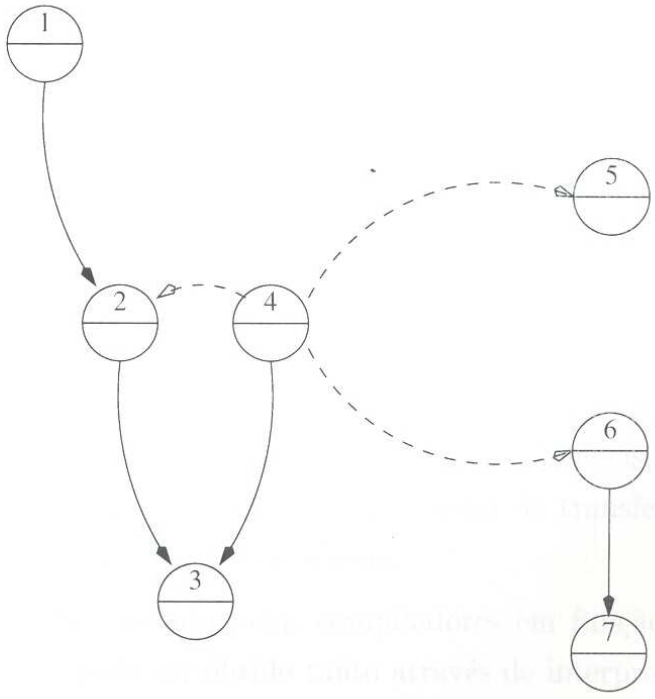


Figura 9.2 Recursividade na organização de metacomponentes

CAPÍTULO 10

Conclusão

10.1 Qualidades da *Virtuosi*

Este trabalho especifica a arquitetura de um ambiente de execução, denominada *Virtuosi*, que integra conceitos de orientação a objetos, sistemas distribuídos e reflexão computacional de forma consistente, o que viabiliza a sua implementação e o seu uso. Os principais benefícios decorrentes dessa integração são:

(1) Simplificação na compilação de programas escritos em linguagens orientadas a objetos e otimização na geração de código executável: o modelo de objetos e o modelo de eventos são diretamente suportados pela *Virtuosi*, eliminando as transformações encontradas em outras máquinas virtuais. Um exemplo é a máquina virtual de Java, que faz o tratamento de eventos através da invocação de métodos. Outro exemplo é o suporte direto para asserções, pré-condições, pós-condições e invariantes que, em outras máquinas, devem ser implementadas através de expressões e desvios gerados por compiladores.

(2) Simplificação na escrita de programas com relação à comunicação entre objetos distribuídos: a própria *Virtuosi* encarrega-se de resolver referências, localizar o objeto alvo em uma invocação ou notificação de evento e cuidar da transferência de parâmetros e retorno através de plataformas heterogêneas.

(3) Portabilidade do código gerado pelos compiladores em função de executar em uma máquina virtual: isso pode ser obtido tanto através de interpretadores de código (instruções de máquina ou árvore de programa) como através de geradores de código *just in time*.

(4) Possibilidade de se construir um mesmo sistema de software utilizando-se diferentes

linguagens: um conjunto de objetos que se comunicam na *Virtuosi* podem ter sido originalmente programados utilizando-se distintas linguagens, sem que isso afete a semântica da aplicação.

(5) Possibilidade de se executar um sistema de software sobre plataformas heterogêneas de hardware e sistema operacional: um conjunto de objetos que se comunicam na *Virtuosi* podem estar alocados em máquinas virtuais alocadas em computadores completamente distintos com relação a hardware e sistema operacional, sem que isso afete a semântica da aplicação.

(6) Escalabilidade do sistema computacional: na *Virtuosi* não existe qualquer entidade centralizadora, pois todo o controle do sistema é efetuado de maneira cooperativa entre as máquinas virtuais e entre os metacomponentes.

(7) Simplificação na construção de software pela possibilidade de separação entre os aspectos funcionais de uma aplicação e os aspectos de sistema envolvidos: isso é conseqüência direta do mecanismo de reflexão computacional.

(8) Possibilidade de se definir metacomponentes que especializem o comportamento padrão do sistema computacional e a conseqüente flexibilidade na configuração desse sistema computacional.

10.2 Trabalhos Futuros

Este trabalho é o primeiro passo na construção de um ambiente de execução distribuída para sistemas de software orientado a objetos. Os modelos definidos, acompanhados da notação gráfica e da terminologia introduzidas, fornecem uma base sólida sobre a qual uma série de trabalhos de detalhamento da especificação e de implementação do ambiente pode se seguir. Além disso, o ambiente de execução pode ser explorado para aplicações em diferentes áreas da Computação, inclusive para o aprendizado de construção de software. A seguir é apresentada de forma breve uma extensa lista de trabalhos que podem ser realizados a partir deste. A lista não reflete necessariamente a ordem em que os trabalhos devem ser realizados; quando há dependência entre os trabalhos, isso é explicitamente dito.

(1) Detalhamento e implementação do modelo básico de objetos, do modelo de comunicação por atividades, do modelo de comunicação por eventos e do modelo de reflexão

computacional. Essa tarefa por ser desenvolvida em etapas, a saber:

- Construção da célula mais elementar do ambiente: uma máquina virtual que suporte diretamente os princípios de orientação a objetos, mas que ainda opere isoladamente, permitindo somente uma aplicação ser executada por vez e ainda sem suporte para atividades assíncronas, isto é, sem suporte a concorrência. Essa implementação pode concretizar-se através de um interpretador de instruções de máquina, expressas tanto em termos de *bytecode* quanto em termos de árvore de programa. Opcionalmente, ainda, essa representação de instruções de máquina pode ser convertida para o código nativo do computador que hospeda a execução da máquina virtual. Inicialmente, a máquina virtual (interpretador ou programa executável) constrói um objeto que é uma instância de uma classe cujo nome é fornecido como parâmetro da execução. Como conseqüência, todas as atividades raizes por construção (deve haver ao menos uma) desse objeto são iniciadas, desencadeando a comunicação entre objetos através de iniciação de atividades. A máquina virtual permanece em execução enquanto houver pelo menos uma atividade em andamento.
- Inclusão do tratamento de referências remotas, provendo a interação de objetos entre máquinas virtuais distintas. Essa implementação pode fazer uso da tecnologia atualmente empregada em redes de computadores, isto é, basicamente o protocolo TCP/IP e a definição de processos servidores que utilizam um porta reservada para a *Virtuosi*; pode haver um único processo servidor por computador, atendendo todas as máquinas virtuais alocadas em um mesmo computador. Agora, cada máquina virtual permanece em execução enquanto houver pelo menos uma atividade em andamento ou enquanto houver algum objeto referenciado externamente, isto é, por um objeto de outra máquina virtual.
- Inclusão de suporte para chamadas assíncronas, com o devido controle de concorrência. Nesta etapa, deve ser eleita a semântica *default* do ambiente de execução para o controle de concorrência – uma candidata é a semântica de monitores.
- Inclusão de suporte para comunicação por eventos. O modelo definido neste trabalho é básico e, por isso, pode ser estendido com a inclusão de conceitos presentes em outros ambientes, como CORBA e Enterprise JavaBeans.
- Inclusão de suporte para reflexão computacional. Esse trabalho inclui uma ex-

tensão da representação de árvores de programas e de instruções de máquina para contemplar reflexão computacional.

(2) Construção de uma biblioteca de classes para os objetos atômicos de tipos mais comumente utilizados e pré-definidos em linguagens de programação, tais como inteiro, real, boolean, caracter e string.

(3) Construção de uma biblioteca de classes para suporte a vetores e matrizes.

(4) Construção de uma biblioteca de objetos de fronteira para dispositivos comumente encontrados em sistemas operacionais padrão, como impressoras, mouse, discos, etc, tanto para ambiente Windows como para Linux.

(5) Construção de uma biblioteca de classes de objetos de fronteira para componentes disponíveis em sistemas de desenvolvimento padrão, como componentes CORBA, ActiveX [Chappell, 1996] e JavaBeans [javabeans, 1996].

(6) Construção de uma biblioteca de classes para composição de interfaces gráficas, inclusive para interação homem-máquina.

(7) Construção de compiladores para as linguagens de programação orientada a objetos mais comuns que gerem código para a *Virtuosi*.

(8) Construção de uma biblioteca de metacomponentes para os requisitos básicos de sistemas distribuídos: persistência, controle de concorrência, recuperação, transação atômica, replicação, coleta de lixo, etc.

(9) Definição de mecanismos de escalonamento de atividades e de alocação de objetos a fim de otimizar a utilização de recursos do sistema, fazendo balanceamento de carga, reduzindo o tráfego em rede e explorando paralelismo na execução.

(10) Construção de compiladores *just in time*, isto é, que geram código nativo no instante de execução ou interpretação.

(11) Otimização do código gerado, empregando técnicas bem conhecidas que analisam a árvore de programa e fazendo uso da arquitetura alvo, como por exemplo o uso de registradores, no caso de geração de código nativo.

(12) Otimização na resolução de expressões aritméticas e booleanas, utilizando-se uma pilha de (objetos) operandos na qual cada elemento corresponde ao próprio estado de um objeto atômico.

(13) Construção de um depurador de aplicações distribuídas.

(14) Definição de um mecanismo para agrupamento de máquinas virtuais de forma hierárquica e, portanto, com alto grau de escalabilidade.

(15) Definição de um mecanismo de comunicação em grupo baseada na invocação de métodos.

(16) Extensão do modelo de comunicação por atividades para permitir que uma invocação assíncrona defina um evento a ser notificado quando do término da atividade invocada.

(17) Extensão do modelo de comunicação por atividades para fazer retornos parciais em invocação assíncrona de método, tanto para notificar sobre o andamento da atividade como para transferir possíveis resultados.

(18) Definição de mecanismos para suportar aplicações de tempo real.

(19) Extensão do modelo de objetos para suportar interfaces e herança múltipla.

(20) Extensão do modelo de objetos para suportar associações considerando todos os seus elementos: nome, multiplicidade e papel. Uma proposta para essa extensão pode ser encontrada em [Calsavara, 1996].

(21) Definição de mecanismos para suportar evolução de objetos.

(22) Definição de um mecanismo para localização de objetos baseado em expressões de consulta (*query*). Uma proposta para esse mecanismo pode ser encontrada em [Calsavara, 1996].

(23) Utilização do Domain Naming System (DNS) [Mockapetris, a, Mockapetris, b] para armazenamento e resolução de identidade dos objetos, das atividades e das máquinas virtuais.

(24) Definição de um mecanismo para detectar invocações polimórficas de métodos e prevenir invocações inválidas que resultariam em exceção.

(25) Definição de mecanismos para tratamento de falhas em comunicação e em processamento, isto é, na interpretação de código.

(26) Definição de mecanismos para prevenção e detecção de *deadlocks*.

(27) Definição de um mecanismo que permita migração de objetos.

(28) Definição de um mecanismo de cache de objetos. Uma proposta para esse mecanismo pode ser encontrada em [Calsavara, 1996].

(29) Definição de um mecanismo que permita objetos comportarem-se como agentes, isto é, migrarem para o local onde se encontra o objeto alvo em uma invocação de método no instante da invocação, ao invés de efetuar uma invocação remota [Lange, 1998].

(30) Extensão do modelo de objetos para contemplar os requisitos de segurança no acesso a métodos de acordo com cada aplicação. Essa extensão pode ter como base a dissertação de mestrado de [Cuche, 1999].

(31) Construção de um ambiente de desenvolvimento, incluindo um método de análise e projeto, uma linguagem de programação e uma ferramenta CASE. Um exemplo simples é definição de uma linguagem (ou extensão de uma linguagem existente) que explore o fato de em uma composição de objetos existir uma referência implícita do objeto contido para o seu contentor a fim de permitir fácil navegabilidade entre os objetos. Isso pode ser realizado através da disponibilização de uma palavra-chave que faça referência ao contentor de um objeto, quando se aplica; assim como em algumas linguagens existe a palavra *this* ou *self* para referenciar o próprio objeto, pode existir a palavra *container* para referenciar o objeto contentor.

(32) Averiguação das linguagens de programação e dos métodos de análise e projeto mais utilizados atualmente com relação ao suporte às formas canônicas de redefinição de métodos.

(33) Aplicação da *Virtuosi* na construção de aplicações Web. Essa tarefa requer a adaptação de um *browser* para implementar uma máquina virtual.

(34) Aplicação da *Virtuosi* no aprendizado de orientação objetos e no aprendizado de sistemas distribuídos.

(35) Aplicação da *Virtuosi* na construção de protótipos de aplicações.

(36) Aplicação da *Virtuosi* na simulação de sistemas.

(37) Aplicação da *Virtuosi* na construção de sistemas reais.

(38) Aplicação da *Virtuosi* no suporte a outros modelos de computação, como por exemplo os sistemas multi-agentes.

Apêndice A

Resumo da Notação Gráfica

a synchronous activity

an asynchronous activity without notification of end

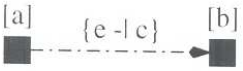
an asynchronous activity with notification of end

the construction activity of an object

the activity "a" invokes the activity "a.1" synchronously

the activity "a" invokes the activity "a.1" (also "b") asynchronously without notification of end

the activity "a" invokes the activity "a.1" (also "b") asynchronously with notification of end

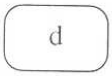


the activity "a" notifies the event "e" to activity "b" after the activity "c"

the object whose identity is 12

an object whose identity is not shown

a border object



the external device "d"

the object 12 has an association reference to the object 15

the object 12 has a composition reference to the object 15 (the object 12 logically contains the object 15)

the object 12 is a metacomponent of the object 15

Referências Bibliográficas

[Accetta et al., 1986] Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., and Young, M. (1986). Mach: a new kernel foundation for UNIX development. In *Proc. Summer 1986 USENIX Conf.*, pages 93–112.

[Agha and Hewitt, 1987] Agha, G. and Hewitt, C. (1987). Actors: a conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, Cambridge, MA. MIT Press.

[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.

[Baillarguet and Piumarta, 1999] Baillarguet, C. and Piumarta, I. (1999). An highly-configurable, modular system for mobility, interoperability, specialization, and reuse. In *2nd ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS'99)*.

[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.

[Birrel and Nelson, 1984] Birrel, A. D. and Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Transactions and Computer Systems*, 2(1):39–59.

[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.

[Buzato and Calsavara, 1992] Buzato, L. E. and Calsavara, A. (1992). Stabilis: a case-study in writing fault-tolerant distributed applications using persistent objects. In *Proceedings of the 5th International Workshop on Persistent Object Systems*, San Miniato, Italy.

[Calsavara, 1996] Calsavara, A. (1996). *Constructing Highly-Available Distributed Metainformation Systems.* PhD thesis, Department of Computing Science, University of Newcastle upon Tyne, Inglaterra.

[Chappell, 1996] Chappell, D. (1996). *Understanding ActiveX and OLE - A Guide for Developers & Managers.* Microsoft Press, Redmond, WA.

[Cuche, 1999] Cuche, J.-S. (1999). Access rules in object-oriented models. Master's thesis, Emoose, Nantes, França.

[Dahl and Nygaard, 1970] Dahl, O.-J. and Nygaard, K. (1970). Simula-67 common base language. Technical Report S-22, Norwegian Computing Centre, Oslo.

[DCE, 1992] DCE (1992). *Introduction to OSF DCE.* Prentice Hall, Englewood Cliffs, NJ.

[Eriksson and Penker, 1998] Eriksson, H.-E. and Penker, M. (1998). *UML Toolkit.* John Wiley & Sons, Inc.

[Flanagan, 1997] Flanagan, D. (1997). *Java in a Nutshell.* O'Reilly.

[Folliot et al., 1997] Folliot, B., Piumarta, I., and Riccardi, F. (1997). Virtual virtual machines. In *Proceedings of the 4th Cabernet Radical Workshop.*

[Folliot et al., 1998] Folliot, B., Piumarta, I., and Riccardi, F. (1998). A dynamically configurable, multi-language execution platform. In *SIGOPS'98 Workshop.*

[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.

[Gamma et al., 1995] Gamma, E. et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley.

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. (1983). *Smaltalk-80: The Language.* Addison-Wesley, Reading, MA.

[javabeans, 1996] javabeans (1996). Javabeans, version 1.00. http://java.sun.com/beans.

[Jini, 1999] Jini (1999). Jini architectural overview. Technical White Paper, Sun Microsystems.

[Kiczales et al., 1991] Kiczales, G., de Riviere, J., and Bobrow, D. G. (1991). *The art of the Metaobject Protocol.* MIT Press, Cambridge, MA.

[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.

[Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

[Lange, 1998] Lange, D. B. (1998). Mobile objects and mobile agents: The future of distributed computing? In *ECOOP 98*, pages 1–12.

[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.

[Mockapetris, a] Mockapetris, P. Domain names - concepts and facilities. RFC-1034, USC-ISI.

[Mockapetris, b] Mockapetris, P. Domain names - implementation and specification. RFC-1035, USC-ISI.

[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.

[Nelson, 1981] Nelson, B. J. (1981). *Remote Procedure Call.* PhD thesis, Carnegie-Mellon University.

[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.

[Oliva and Buzato, 1998] Oliva, A. and Buzato, L. E. (1998). An overview of molds: A meta-object library for distributed systems. In *II Workshop em Sistemas Distribuídos*, Curitiba.

[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).

[Rumbaugh et al., 1994] Rumbaugh, J. et al. (1994). *Modelagem e Projetos Baseados em Objetos.* Editora Campus.

[Rumbaugh et al., 1997] Rumbaugh, J., Jacobson, I., and Booch, G. (1997). *Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, MA.

[Silberchatz and Galvin, 1998] Silberchatz, A. and Galvin, P. B. (1998). *Operating System Concepts.* Addison-Wesley, fifth edition.

[Soley and Kent, 1995] Soley, R. M. and Kent, W. (1995). The OMG object model. In Kim, W., editor, *Modern Database Systems*, chapter 2, pages 18–41. Addison-Wesley.

[Stroustrup, 1986] Stroustrup, B. (1986). *The C++ Programming Language.* Addison Wesley, Reading, Massachusetts.

[Szyperski, 1998] Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley.

[Wirth and Gutknecht, 1992] Wirth, N. and Gutknecht, J. (1992). *Project Oberon - The Design of an Operating System and Compiler.* Addison-Wesley, Reading, MA.

[Wu et al., 1997] Wu, D., Agrawal, D., Abbadi, A. E., and Singh, A. (1997). A java-based framework for processing distributed objects. *Lecture Notes in Computer Science*, (1331). 16th International Conference on Conceptual Modeling – ER'97.

[Yokote, 1992] Yokote, Y. (1992). The apertos reflective operating system: The concept and its implementation. In *OOPSLA.*