PUC
RIO

Renato Fontoura de Gusmão Cerqueira

Um Modelo de Composição Dinâmica entre Sistemas de Componentes de Software

TESE DE DOUTORADO

Departamento de Informática

Rio de Janeiro, 21 de agosto 2000

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Rua Marquês de São Vicente, 225 – Gávea
CEP 22453-900 Rio de Janeiro RJ Brasil
http://www.puc-rio.br

Renato Fontoura de Gusmão Cerqueira

Um Modelo de Composição Dinâmica entre Sistemas de Componentes de Software

Tese apresentada ao Departamento de Informática da PUC-Rio como parte dos requisitos para obtenção do título de Doutor em Informática: Ciência da Computação

Orientadores : Roberto Ierusalimschy
Noemi de La Rocque Rodriguez

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
Rio de Janeiro, 21 de agosto 2000

Ao meu avô Joaquim e à minha esposa Cristina.

Agradecimentos

Gostaria de agradecer todo o amor, carinho e apoio que minha esposa Cristina me deu durante a realização deste trabalho. Sua dedicação e compreensão foram fundamentais para que eu pudesse concluí-lo. Agradeço também aos meus pais por sempre terem me apoiado e motivado na realização do meu trabalho, mesmo que isso representasse várias horas a menos em nosso convívio. Apesar de não estar presente nos últimos anos, a lembrança do meu avô Quim foi fundamental para que eu conseguisse manter a disposição e a vontade de realizar o meu trabalho da melhor forma que eu pudesse. Agradeço a Deus pela sorte de ter o amor dessas pessoas.

Completando um ciclo de sete anos como orientado do Roberto, espero ter conseguido absorver um pouco de sua objetividade e clareza de raciocínio. Devo a ele qualquer sinal de um legítimo pesquisador que eu apresente. Gostaria de agradecer também a Noemi por desde o início ter acreditado e apostado neste trabalho. Seu incentivo, interesse e motivação foram fundamentais para que o LuaOrb chegasse a onde chegou. Mais do que orientadores e padrinhos, Roberto e Noemi são grandes amigos.

Não poderia deixar de agradecer aos meus queridos amigos Eduardo, Circuns, Cassino, Clinio, Costa, Claúdio e Christiano, que compreenderam minha constante ausência e, mesmo assim, sempre me apoiaram.

Gostaria de agradecer ao Rodacki, Marco Catunda, Ana Lúcia, Alexandre, Letícia, Thais e Maria Júlia por terem sido tão corajosos e terem usado o LuaOrb em seus trabalhos, quando este ainda estava em desenvolvimento. Graças a eles, pudemos realizar uma série de estudos de caso. Cassino, Almendra, Catunda e Letícia ainda participaram diretamente da implementação do LuaOrb. Esse trabalho também é deles!

Por fim, gostaria de agradecer aos Professores Marcelo Gattass e Lucena e aos laboratórios Tecgraf e LES do Departamento de Informática da PUC-Rio, por criarem um ambiente onde se acredita que é possível desenvolver tecnologia de software de boa qualidade em nosso país. Todos nós ganharíamos muito se outros seguissem os seus exemplos.

Resumo

Diferentes sistemas de componentes de software, tais como CORBA, COM e JavaBeans, apresentam diferentes modelos de objetos e sistemas de tipos. Essas diferenças dificultam a integração de componentes oriundos de sistemas distintos e, conseqüentemente, são uma barreira para o reuso desses componentes.

Neste trabalho, defendemos a tese de que uma linguagem interpretada com um determinado conjunto de mecanismos reflexivos, aliada à compatibilidade estrutural de tipos, oferece um mecanismo de composição adequado tanto para a conexão dinâmica de componentes, quanto para a interoperabilidade entre diferentes sistemas de componentes. Esse mecanismo de composição realiza em tempo de execução as tarefas de conexão, adaptação, implementação e verificação de tipos de componentes, e trata de uma maneira uniforme componentes de diferentes sistemas, permitindo que estes sejam conectados de uma forma transparente.

O mecanismo de composição que propomos se baseia em um modelo que privilegia a flexibilidade em tempo de execução. Esse modelo de composição é composto por dois elementos principais. O primeiro elemento é um modelo de objetos que definimos com a finalidade de poder representar componentes dos diferentes sistemas tratados neste trabalho. Assim, esse modelo de objetos faz o papel de um *modelo integrador*, isto é, um modelo sob o qual objetos de diferentes sistemas podem ser representados e interagir de forma transparente.

O segundo elemento de nosso modelo de composição é um padrão de projeto (*design pattern*) para a implementação de *bindings* entre linguagens interpretadas e sistemas de componentes. Esse padrão de projeto, chamado *Dynamic Language Binding*, não utiliza a técnica tradicional de *stubs*. Ao invés disso, ele utiliza mecanismos de reflexividade e tipagem dinâmica para implementar tanto *proxies* genéricos, que podem representar qualquer componente de um determinado sistema, quanto adaptadores genéricos, que permitem a implementação de componentes utilizando a própria linguagem de composição.

Como instrumento de validação da nossa proposta, descrevemos uma implementação do modelo de composição denominada LuaOrb. LuaOrb utiliza a linguagem interpretada Lua como linguagem de composição dinâmica, e integra os sistemas CORBA, COM e Java.

Abstract

Different component systems, such as CORBA, COM, and Java, have different object models and type systems. Such differences make the interoperability between components of distinct systems more difficult, and thus are an obstacle for component reuse.

In this dissertation, we argue that an interpreted language with a specific set of reflexive mechanisms, together with a type system with structural compatibility, offers a composition mechanism suitable for dynamic component connection and for interoperability between different component systems. This composition mechanism performs at runtime the tasks of verifying types, connecting, adapting and implementing components, and handles components of different systems in a uniform way, allowing them to be connected transparently.

The proposed composition mechanism is based on a model that favors flexibility at runtime. This composition model is composed of two major elements. The first one is an object model, defined in order to represent components of the different systems addressed in this dissertation. Thus, this object model performs the role of a *unifying model*, that is, a model in which objects from different systems can interact and be represented transparently.

The second element of our composition model is a design pattern to implement bindings between interpreted languages and component systems. This design pattern, named *Dynamic Language Binding*, does not use the traditional stubs technique. Instead of this, it uses reflection and dynamic typing to implement generic proxies, which can represent any component of a specific system, and generic adapters, which allow component implementations using the composition language itself.

In order to validate our proposal, we describe the LuaOrb system, which is an implementation of our composition model. LuaOrb uses the interpreted language Lua as its dynamic composition language, and integrates the systems CORBA, COM and Java.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Para a elaboração da tese apresentada nesta dissertação, fomos motivados pela necessidade de melhores mecanismos para o desenvolvimento de aplicações baseadas em componentes de software. Como poderá ser observado ao longo dos próximos capítulos, nosso foco está nos mecanismos para conexão entre componentes, e não tratamos de técnicas específicas para a especificação e implementação dos próprios componentes. Neste capítulo, detalhamos o contexto que motivou nosso trabalho e apresentamos brevemente nossa tese. Por fim, descrevemos a estrutura geral da dissertação.

1.1 Motivação

O conceito de componentes reusáveis, ou *componentware*, está adquirindo uma crescente importância no processo de desenvolvimento de software. Um ideal almejado pela Engenharia de Software, na área de reuso de software, é a construção de novas aplicações a partir de componentes de software já existentes e devidamente testados, como ocorre em outros ramos da engenharia [CN91, Bro94, NT95, Szy98].

Entretanto, os componentes da Engenharia de Software possuem uma natureza mais abstrata do que a de seus correspondentes de outras áreas. Algumas vezes, a própria especificação de um "artefato"de software já é considerada como sendo um componente reutilizável [Bro94, Bro96, Sam97, SLP98]. Apesar do significado do termo *componente de software* ser muito abrangente, neste texto vamos considerar um componente de software como sendo uma unidade binária desenvolvida isoladamente para uma finalidade específica, e que deve ter sido projetada para interagir com outros componentes a fim de formar aplicações finais. Essas unidades binárias devem interagir entre si através de interfaces (conjuntos de operações) bem definidas. De um ponto de vista mais formal, cada interface representa um tipo abstrato de dado[1].

Essa definição de componente de software o coloca muito próximo ao conceito de um objeto em programação orientada a objetos. Na realidade, tanto programação orientada a componentes quanto programação orientada a objetos apresentam entre seus conceitos fundamentais o encapsulamento através de interfaces (tipos abstratos de dados) e *late-binding*. Entretanto, programação OO tem uma ênfase muito grande em herança de implementação. Em programação OO, é permitido que o encapsulamento de um tipo abstrato de dado, representado através de uma *classe*, seja quebrado para que sub-classes dessa classe possam ser derivadas [CW85, Sny86, Szy98].

Já uma abordagem orientada a componentes enfatiza o uso de polimorfismo, pois um dos objetivos das técnicas de componentware é oferecer suporte à extensão de sistemas já existentes através da

[1] A definição para o termo *componente de software* que utilizamos neste texto foi fortemente baseada na definição apresentada em [Szy98].

simples substituição ou adição de novos componentes. Um componente pode ser substituído por outro em uma aplicação desde que ambos ofereçam a mesma interface (ou *tipo*) ou, pelo menos, ofereçam interfaces compatíveis, isto é, a interface do novo componente é um *sub-tipo* da interface do componente original [CW85, RIR93]. Tipicamente, um componente é tratado como uma caixa-preta, da qual só é conhecida a interface. O uso de herança de implementação é geralmente desaconselhado ou até mesmo proibido [WK95, Szy98].

Apesar das diferenças de abordagem entre programação orientada a objetos e programação orientada a componentes e das diferenças entre os conceitos de *objeto* e *componente* que alguns autores apontam [NT95, Szy98], vamos utilizar estes dois termos como sinônimos ao longo do texto.

1.1.1 Composição Dinâmica

O aspecto de ligação dinâmica (*late binding*) em componentware vai um pouco além do que é geralmente definido em orientação a objetos. Em linguagens orientadas a objetos, a ligação entre um objeto cliente e um objeto servidor só é estabelecida em tempo de execução; essa característica é a base para o polimorfismo em linguagens OO. Se as interfaces dos objetos forem conhecidas em tempo de compilação, é possível conciliar um alto grau de flexibilidade com mecanismos rígidos de verificação estática de tipos [RIR93, Ier93, DGLM95].

Já modelos orientados a componentes geralmente estendem os mecanismos de *late binding* para permitir que até mesmo as interfaces dos componentes só sejam conhecidas em tempo de execução. Assim, a verificação de tipos também é postergada para tempo de execução. Essa abordagem algumas vezes é classificada como *very late binding* [Szy98]. É importante observar que esse modelo é semelhante àquele adotado por Smalltalk [GR83], e, se não for feita a verificação de tipos em tempo de execução, é possível a ocorrência de erros de execução do tipo "*message not understood*".

O recurso de *very late binding* é de grande relevância para componentware, pois a possibilidade de permitir que mudanças sejam realizadas dinamicamente na composição de uma aplicação tem se mostrado de grande importância em diversas situações que exigem um alto grau de flexibilidade, tais como sistemas operacionais e de *desktop* [Sie96, KSC+98, KCC99, KRL+00, RKC99, BHK+99], ferramentas de gerenciamento de redes [Sie96, Ban97, RMIC97], aplicações cooperativas [MGG96, GFCI98, FRB+98], e ambientes de desenvolvimento baseados em prototipação [Sie96, CIR97].

Além dos exemplos enumerados acima, o recurso de composição dinâmica de aplicações é notadamente importante para sistemas que exijam um alto grau de *disponibilidade*[2], já que, em caso de falhas ou necessidade de atualização, sistemas críticos precisam poder ter componentes substituídos ou adicionados sem que para isto sua execução seja interrompida [Sta91, GK96, HW96, WS96, MUWZ96, KC99].

Para dar suporte a *very late binding*, é necessário oferecer algum mecanismo para a exploração dinâmica das interfaces dos componentes. Os componentes devem ter agregado a eles informação suficiente para sua auto-descrição. Os mecanismos para prover tais informações podem variar, mas o problema central é o mesmo: preservar informações de tipo disponíveis em tempo de compilação para serem inspecionadas em tempo de execução.

Além de um mecanismo de exploração dinâmica de tipos, *very late binding* também necessita de um mecanismo de *dispatching* que permita a construção dinâmica de chamadas de métodos. Podemos caracterizar *very late binding* como sendo um mecanismo de *dispatching* dinâmico aliado a um sistema, também dinâmico, de verificação de tipos.

[2] Um sistema é considerado *disponível* quando estiver em execução e apto a realizar seus serviços.

1.1.2 Mecanismos de Composição

Um outro aspecto de grande importância para o desenvolvimento de aplicações baseadas em componentes é o mecanismo utilizado para conectar componentes. Freqüentemente, as conexões entre componentes são definidas na mesma linguagem usada para implementá-los, fazendo com que os aspectos referentes à arquitetura do sistema[3] fiquem misturados com as implementações dos componentes. Entretanto, existe uma crescente preocupação com o desenvolvimento de novos mecanismos orientados para a conexão de componentes, de tal forma que a descrição da arquitetura de uma aplicação fique separada da implementação de seus componentes [KMN89, MDEK95, SG96a, BCK98]. A separação dos aspectos de composição e implementação de uma aplicação permite uma melhor compreensão de sua arquitetura, e também simplifica algumas tarefas de manutenção, tais como alterações na arquitetura e substituições de componentes. Os sistemas que necessitam do recurso de composição dinâmica ainda requerem um mecanismo de composição que permita que novas conexões sejam definidas em tempo de execução do sistema.

Alguns exemplos de mecanismos de composição são as linguagens de *script*, tais como Tcl [Ous90, Ous94], Python [Lut96], Lua [IFC96], JavaScript [Fla97, ECM97] e Visual Basic [Cla96], as linguagens de descrição de arquiteturas (*Architecture Description Languages* – ADL) [MDK94, MDEK95, SG96a, SDZ96, BF96, Tho96, IB96, BCK98] e os editores gráficos especializados em realizar conexões entre componentes [KMN89, dM95, NK95, NKM96, NKMD96, Jav97].

1.1.3 Sistemas de Componentes

Para servir como infra-estrutura de suporte ao desenvolvimento baseado em componentes, algumas especificações e implementações de *sistemas de componentes* têm surgido, tais como CORBA [OMG98, OMG99a], COM [Bro95, Box98] e JavaBeans [Jav97, Tho98]. Cada um desses sistemas define um conjunto próprio de mecanismos para viabilizar a construção e a comunicação de seus componentes.

Assim como linguagens orientadas a objetos, sistemas de componentes de software também definem *modelos de objetos*. Um modelo de objetos é o conjunto de conceitos usados para descrever objetos em uma determinada linguagem, especificação ou metodologia orientada a objetos [Man95, Cer96]. Neste texto, o termo *modelo de objetos* também será usado para referenciar o conjunto de conceitos usados por um determinado sistema de componentes de software.

Os sistemas de componentes existentes apresentam diversas diferenças entre seus modelos de objetos e seus sistemas de tipos. Essas diferenças representam uma barreira para a integração de componentes oriundos de sistemas diferentes, o que limita as possibilidades de reuso de tais componentes.

1.2 A Tese

Neste trabalho, defendemos a tese de que uma linguagem interpretada com um determinado conjunto de mecanismos reflexivos, aliada à compatibilidade estrutural de tipos [RIR93, Ier93, DGLM95], oferece um mecanismo de composição adequado tanto para a conexão dinâmica de componentes (*very late binding*), quanto para a interoperabilidade entre diferentes sistemas de componentes. Esse mecanismo de composição realiza em tempo de execução as tarefas de conexão, adaptação, implementação e verificação de tipos de componentes, e trata de uma maneira uniforme componentes de diferentes sistemas, permitindo que estes sejam conectados de uma forma transparente.

[3] De uma forma simplificada, podemos definir que a arquitetura de um sistema baseado em componentes é o conjunto de seus componentes e como estes se relacionam estática e dinamicamente.

1.2.1 Decisões de Projeto

Duas decisões foram fundamentais para a definição do mecanismo de composição. A primeira foi a escolha de uma linguagem interpretada como elemento de ligação entre componentes. Linguagens interpretadas tipicamente oferecem um alto grau de interatividade e expressividade, fazendo com que sejam ferramentas adequadas para programação por um administrador de sistema ou até mesmo pelo usuário final (*end user programming*) [CIS92, CIR97, Ous98]. Com o auxílio de uma linguagem interpretada, é possível se oferecer um *console* de comandos para que um usuário ou administrador de sistema possa ter, em tempo de execução, total acesso aos serviços oferecidos pelos componentes disponíveis. Através desse console, um usuário pode configurar e reconfigurar sua aplicação dinamicamente. Tipicamente, quando estão desempenhando o papel de uma linguagem de composição, linguagens interpretadas são classificadas como *linguagens de script* [Ous98, SN98, SN99].

A segunda decisão foi a escolha de um esquema de verificação de tipos baseado em compatibilidade estrutural. Esse esquema permite realizar uma verificação dinâmica de tipos entre interfaces de componentes de sistemas diferentes. Para realizar essa verificação, utilizamos as informações reificadas sobre as interfaces dos objetos que os sistemas de componentes oferecem.

O uso de linguagens interpretadas como mecanismo de composição não é uma idéia nova. Diversos trabalhos têm realizado ou proposto a integração de linguagens desse tipo com sistemas de componentes [NM94, SN99, MGG97, Ous98, ICR98, Gat98, JML98, Sta98, vR98, Chi99, JSLJ99, OMG99b, OMG00]. Entretanto, esses trabalhos só realizam a integração de uma linguagem de *script* com um sistema de componentes específico, não permitindo, ou pelo menos dificultando, a interoperabilidade entre componentes de sistemas diferentes. O uso neste trabalho de um esquema de verificação de tipos baseado em compatibilidade estrutural tem como objetivo contornar essa limitação.

1.2.2 O Modelo de Composição

O mecanismo de composição que propomos se baseia em um modelo que privilegia a flexibilidade em tempo de execução. Esse modelo de composição define uma série de mecanismos para permitir a composição e a adaptação dinâmica de componentes através de uma linguagem interpretada.

Nosso modelo de composição é composto por dois elementos principais. O primeiro elemento é um modelo de objetos, que definimos com a finalidade de representar os componentes dos diferentes sistemas tratados neste trabalho. Assim, esse modelo de objetos faz o papel de um *modelo integrador*, isto é, um modelo sob o qual objetos de diferentes sistemas podem ser representados e interagir de forma transparente.

De acordo com esse modelo de objetos integrador, não existe a definição de classes de objetos: objetos são entidades auto-suficientes que possuem o seu próprio estado e comportamento, e se comunicam através de um mecanismo genérico de envio de mensagens. Essa abordagem é similar àquela utilizada em linguagens baseadas em protótipos [Weg87, SLU88, DMC92].

O segundo elemento de nosso modelo de composição é um padrão de projeto (*design pattern*) para a implementação de *bindings* entre linguagens interpretadas e sistemas de componentes. Para que se tenha acesso a componentes de um determinado sistema através de uma linguagem de programação, é necessário que haja um *binding* entre a linguagem e o sistema de componentes. Geralmente, *bindings* entre linguagens e sistemas de componentes são baseados na geração estática de *stubs*, que são as entidades responsáveis pela comunicação da aplicação com o sistema dos componentes. Entretanto, o nosso padrão de projeto, chamado *Dynamic Language Binding*, não utiliza a técnica tradicional de *stubs*. Ao invés disso, ele utiliza mecanismos de reflexividade, tipagem dinâmica e construção de chamadas em tempo de execução, para implementar *proxies genéricos* que podem se comunicar com qualquer componente de um determinado sistema.

Esse padrão de *binding* oferece mecanismos para a instanciação de componentes e, em tempo de

execução, consulta os mecanismos de introspecção oferecidos pelos sistemas de componentes para descobrir as interfaces dos componentes instanciados. Para realizar a requisição de operações dos componentes, o *binding* utiliza um mecanismo de *proxy* genérico que faz dinamicamente a verificação e conversão de tipos dos parâmetros e resultados das operações. Cabe destacar que, para realizar a verificação e conversão de tipos, são utilizados dois tipos de descrições dinâmicas: a descrição formal das interfaces dos componentes, fornecida pelos mecanismos de introspecção dos sistemas de componentes, e as informações fornecidas pela linguagem de composição a respeito dos tipos de seus objetos.

Da mesma forma que são oferecidos *proxies* genéricos para acessar componentes, também é disponibilizado um mecanismo de *adaptadores genéricos*, que é utilizado para implementar componentes dinamicamente. Para definir um componente dinamicamente, é necessário somente a descrição da interface desejada para o componente e um *objeto* definido pela linguagem de composição que implemente as operações da interface.

A combinação dos mecanismos de *proxies* e adaptadores genéricos permite a construção de *pontes* automáticas e transparentes entre os sistemas de componentes que estão sendo integrados. Por exemplo, pode-se instanciar um adaptador genérico CORBA em que seja registrado como sua implementação um objeto que é, na realidade, um *proxy* para um componente COM. A verificação da compatibilidade entre as interfaces dos dois componentes pode ser feita através de um esquema baseado em compatibilidade estrutural de tipos.

De acordo com as características do nosso modelo de composição, definimos os requisitos para que um sistema de componentes possa ser utilizado, assim como os requisitos para que uma linguagem interpretada possa ser usada como nosso elemento de composição.

1.2.3 O Sistema LuaOrb

Como instrumento de validação da nossa proposta, também apresentamos o sistema LuaOrb, uma implementação do nosso modelo de composição que integra componentes CORBA, COM e Java. Para dar suporte à composição dinâmica, LuaOrb utiliza a linguagem interpretada Lua [FIC94, FIC96, IFC96] como linguagem de composição.

Para permitir que LuaOrb tenha acesso a componentes dos sistemas tratados, foram implementados *bindings* entre a linguagem Lua e os sistemas CORBA, COM e Java. Os *bindings* implementados representam um componente em Lua como se este fosse um típico objeto Lua. Através de um console de comandos, LuaOrb permite que novas interfaces de componentes sejam identificadas em tempo de execução e que suas instâncias sejam efetivamente usadas e combinadas.

1.3 Estrutura do Texto

No restante deste texto, descrevemos com maiores detalhes o nosso modelo de composição dinâmica entre sistemas de componentes de software. No próximo capítulo, discutimos as características dos sistemas de componentes de software mais relevantes para esta tese. Para isso, fazemos um resumo das características básicas de CORBA, COM e JavaBeans, e descrevemos seus mecanismos reflexivos que são necessários para a aplicação do nosso modelo de composição. Também apresentamos algumas soluções de integração entre os três sistemas.

No capítulo 3, discutimos os conceitos básicos referentes a conexões entre componentes, e apresentamos os mecanismos de composição que consideramos mais relevantes, comparando-os em função de sua expressividade e suporte à composição dinâmica.

No capítulo 4, descrevemos o nosso modelo de composição. Detalhamos seu modelo de objetos integrador e o padrão de projeto para implementação de *bindings* dinâmicos entre linguagens inter-

pretadas e sistemas de componentes. Por fim, definimos os requisitos que uma linguagem interpretada e um sistema de componentes devem atender para que possamos aplicar o nosso modelo.

Já no capítulo 5, descrevemos a implementação do sistema LuaOrb. Fazemos um breve resumo da linguagem Lua e descrevemos as implementações dos *bindings* entre Lua e CORBA, COM e Java. Também apresentamos um exemplo completo que ilustra como funciona o mecanismo de interoperabilidade do LuaOrb.

Por último, no capítulo 6 apresentamos um resumo dos resultados obtidos em alguns estudos de caso realizados, e destacamos as principais contribuições da tese. Além disso, identificamos alguns trabalhos futuros.

Capítulo 2

Sistemas de Componentes

Para servir de infra-estrutura para o desenvolvimento baseado em componentes, têm surgido na indústria de software especificações e implementações de *sistemas de componentes*. Assim como ocorre com os componentes utilizados em outros ramos da Engenharia, um sistema de componentes de software define padrões de comunicação e ligação entre seus componentes. A adoção de padrões é a base para oferecer um alto grau de *interoperabilidade*[1] entre componentes.

Um sistema de componentes adota um modelo de objetos para definir o conjunto de conceitos e recursos que vão caracterizar seus componentes. Alguns exemplos de possíveis características definidas por um modelo de objetos são os mecanismos de polimorfismo, herança, envio de mensagens, identidade de objetos, sistema de tipos e reflexividade.

Além do modelo de objetos, um sistema de componentes define alguns serviços de apoio ao desenvolvimento tanto dos componentes propriamente ditos quanto das aplicações, tais como serviços de nomes, transações, segurança e persistência.

Entre as tecnologias disponíveis, três sistemas de componentes merecem destaque devido à sua disseminação nas comunidades da indústria e da academia:

- *Common Object Request Broker Architecture* (CORBA): padrão definido pela OMG (*Object Management Group*[2]) que especifica uma arquitetura orientada a objetos para dar suporte a aplicações distribuídas [OMG98, OMG92]. Apesar da OMG só definir o padrão, existem várias implementações disponíveis, tais como *TAO* [Sch99a], *ORBacus* [OOC98], *Mico* [Mic00] e *Visibroker* [Vis96].

- *Component Object Model* (COM): tecnologia desenvolvida pela Microsoft para a construção de sistemas baseados em componentes. Essa tecnologia serve de base para diversas outras tecnologias desenvolvidas por essa empresa, tais como OLE e ActiveX [Bro95, Rog97, Kir97, Kir99].

- *JavaBeans*: sistema de componentes desenvolvido pela Sun Microsystems para a linguagem Java [GJS96, Jav97, AG98]. Seu modelo privilegia o suporte a ferramentas gráficas para a composição de aplicações. Mais recentemente foi desenvolvida uma extensão para *JavaBeans*, chamada *Enterprise JavaBeans* [Tho98], para oferecer uma infra-estrutura melhor para aplicações distribuídas que requeiram um maior grau de confiabilidade.

[1] Interoperabilidade é a capacidade de dois ou mais componentes de software se comunicarem ou trabalharem em conjunto, independentemente da linguagem em que foram escritos ou de seus domínios de execução.

[2] A OMG é uma organização internacional de padronização que trabalha com uma grande gama de padrões baseados em sua arquitetura de gerenciamento de objetos [OMG92]. Atualmente, a OMG conta com mais de 700 companhias como membros de seus comitês.

Mesmo adotando abordagens diferentes, esses três sistemas apresentam várias características semelhantes. Uma dessas características é a dissociação entre a interface e a implementação de um componente: para uma mesma interface podem existir diversas implementações (componentes) diferentes, da mesma forma que um componente pode implementar diversas interfaces. De acordo com essa separação, componentes só podem ser acessados através das operações definidas em suas interfaces, garantindo o encapsulamento de suas implementações. A separação entre interface e implementação é a base para que esses sistemas possam integrar componentes que estão executando em processos, estações ou plataformas diferentes, ou até mesmo que foram desenvolvidos em linguagens diferentes.

Entretanto, há uma característica que diferencia profundamente JavaBeans dos outros dois sistemas. Enquanto COM e CORBA procuram integrar componentes desenvolvidos em diferentes linguagens de programação, JavaBeans só contempla componentes desenvolvidos em Java. Dessa forma, JavaBeans só trata de problemas de interoperabilidade entre componentes Java, como, por exemplo, comunicação em ambiente distribuído. De uma certa forma, isto simplifica em muito a arquitetura de JavaBeans. Para oferecer interoperabilidade entre componentes desenvolvidos em linguagens diferentes, CORBA e COM definem modelos de objetos que podem ser mapeados para diversas linguagens de programação, o que acaba aumentando a complexidade desses sistemas.

COM e CORBA são geralmente classificados como sendo sistemas de *middleware*. Um middleware é um software que reside entre uma aplicação e o nível básico do sistema operacional e do sistema de rede de um ambiente computacional [CRS95]. Um middleware é baseado em interfaces padronizadas — incluindo interfaces de programação de aplicações (APIs) e protocolos de comunicação — que aumentam a capacidade de distribuição de aplicações, a interoperabilidade entre aplicações, e a portabilidade das aplicações entre diferentes plataformas. Padrões de middleware, como COM e CORBA, oferecem uma especificação pública de suas interfaces e permitem o desenvolvimento de implementações do middleware para diferentes plataformas computacionais.

Nas seções a seguir, descrevemos os modelos de objetos dos sistemas CORBA, COM e JavaBeans, apresentando suas características mais relevantes para esta tese. Na seção 2.1, apresentamos o conceito de *Linguagem de Definição de Interface*, que é utilizado por CORBA e COM para descrever as interfaces de seus componentes de uma forma independente de linguagem de programação. Também apresentamos as principais características das linguagens de descrição de interface desses dois sistemas. As seções 2.2, 2.3 e 2.4 apresentam algumas características básicas dos três sistemas de componentes. Na seção 2.5, descrevemos os mecanismos reflexivos que esses sistemas oferecem e que são requisitos básicos para a aplicação do modelo proposto no capítulo 4. Já na seção 2.6, apresentamos as principais soluções utilizadas para oferecer interoperabilidade entre componentes de sistemas diferentes.

2.1 Linguagens de Definição de Interface

Tanto CORBA quanto COM oferecem mecanismos para definir as interfaces de seus componentes, de tal forma que essas definições sejam independentes de linguagem de programação, permitindo a interoperabilidade entre componentes desenvolvidos em linguagens diferentes. Para isto, são utilizadas *Linguagens de Definição de Interface* (*Interface Definition Languages* — IDL) para descrever as interfaces dos componentes [Win94, Bro95, JSLJ99, OMG98].

Linguagens de definição de interface já são tradicionalmente utilizadas em ferramentas de programação distribuída para a geração automática de *stubs* [Sun88, RKF92, Win94]. Esses *stubs* são responsáveis pelo empacotamento e desempacotamento das mensagens (requisições de serviços) recebidas por um servidor de determinada interface. Para uma mesma interface, geralmente são gerados dois *stubs*: um para ser usado pelos clientes do serviço, que é responsável pelo empacotamento dos parâmetros de uma operação e pela sua requisição; e um outro *stub* que é utilizado pelo servidor pro-

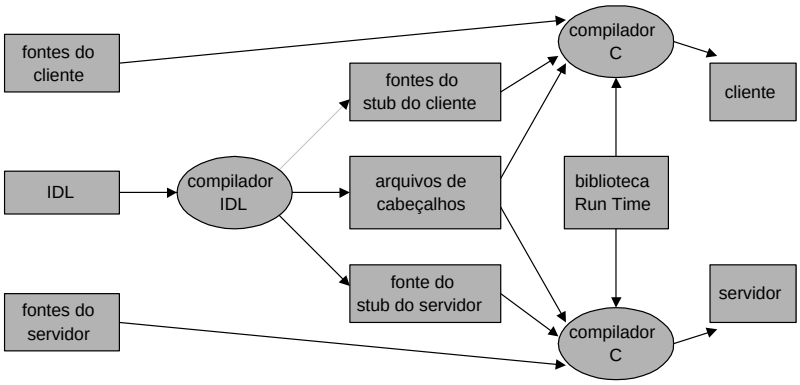

Figura 2.1: Geração de *stubs* a partir de uma IDL.

priamente dito para receber as requisições de serviços, desempacotar os parâmetros e empacotar os resultados que devem ser retornados para os seus clientes. Esses *stubs* também são úteis para permitir uma verificação estática de tipos por linguagens compiladas fortemente tipadas. A figura 2.1 ilustra um processo típico de geração de *stubs* a partir de declarações em uma IDL, utilizado pelo sistema DCE RPC [RKF92]. O mesmo desacoplamento entre cliente e servidor proporcionado por uma IDL para programação distribuída pode ser utilizado para permitir que cliente e servidor sejam implementados em linguagens diferentes.

Tipicamente, IDLs descrevem os aspectos lógicos e físicos da interface de um componente. O aspecto lógico da interface de um componente é composto pelo conjunto de operações e atributos que sua interface oferece, definindo tanto o encapsulamento da implementação do componente quanto suas relações com outros tipos abstratos de dados.

Entretanto, a partir de uma definição em IDL, também são obtidas informações a respeito do aspecto físico da interface de um componente. Essas informações podem ser utilizadas para a geração automática dos *stubs* responsáveis pelo empacotamento e desempacotamento das mensagens trocadas entre componentes, assim como para a geração de representações binárias padronizadas para interfaces de componentes, de tal forma que essas representações possam ser compartilhadas entre diferentes ambientes (ver seção 2.3).

As linguagens de definição de interface usadas por CORBA e COM — OMG IDL [OMG98] e COM IDL [Bro95], respectivamente — definem interfaces de acordo com seus modelos de objetos. Os mesmos conceitos básicos estão presentes em ambas as linguagens, apesar de apresentarem algumas diferenças em como são concretizados. As seções a seguir descrevem como alguns desses conceitos básicos estão presentes nas duas IDLs.

2.1.1 Interfaces

Tanto em OMG IDL quanto em COM IDL, uma interface define basicamente o conjunto de operações e atributos oferecidos por um componente. OMG IDL é baseada em um modelo de objetos tradicional, onde um objeto tem uma interface única que pode ser composta a partir de outras através de herança múltipla. No caso de herança, uma interface derivada é considerada um subtipo da interface da qual deriva. Assim, se uma interface `B` deriva de uma interface `A`, um componente que implemente a interface `A` pode ser substituído por um componente que implemente a interface `B`. Todas as interfaces em OMG IDL já derivam implicitamente da interface `CORBA::Object`.

Em OMG IDL, uma interface também pode conter a definição de constantes e tipos de dados, tais

```
exception OutOfRange {
  float value;
};
interface Calc {
  float add(in float x, in float y);
  float sqrt(in float x) raises (OutOfRange);
};
```

Figura 2.2: Exemplo de interface em OMG IDL.

como estruturas e uniões. Nesse caso, a interface funciona como um espaço de nomes para agrupar as definições de novos tipos de dados. Entretanto, uma interface não pode ser definida dentro de outra interface, isto é, não pode haver interfaces aninhadas.

COM IDL é baseada na linguagem de definição de interface utilizada na arquitetura DCE da Open Software Foundation (OSF) [RKF92, RT93]. A linguagem DCE IDL oferece suporte apenas a chamadas de procedimentos remotos, isto é, ela só permite a descrição da interface de procedimentos. Assim, COM IDL estende a linguagem de DCE para permitir a descrição de interfaces de objetos (tipos abstratos de dados).

De acordo com o modelo de objetos adotado por COM, um objeto pode implementar qualquer número de interfaces. Entretanto, COM IDL apenas oferece um mecanismo de herança simples de interface, que segue o mesmo esquema de OMG IDL para tratar a compatibilidade entre um tipo e seus subtipos. Todas as interfaces em COM derivam implícita ou explicitamente da interface `IUnknown`, que oferece uma operação que permite o acesso a todas as interfaces implementadas por um objeto (ver seção 2.3). Por convenção, os nomes de interfaces COM sempre começam com a letra `I`.

Cabe destacar que as duas IDLs só oferecem herança de interfaces. Em nenhum dos dois casos é possível definir uma relação de herança de implementação.

2.1.2 Operações

Tanto em OMG IDL quanto em COM IDL, a descrição de uma interface é composta pelas assinaturas completas das operações (métodos) oferecidas. Já a assinatura de uma operação é composta pelo seu nome, o tipo de seu valor de retorno, os tipos de todos os seus parâmetros, e as exceções que podem ser sinalizadas.

A descrição de um parâmetro também envolve a definição do sentido em que é feita a troca de informação entre o cliente, que requisita a operação, e o componente, que recebe a requisição (modificadores `in`, `out` e `inout`).

Uma chamada de método é tipicamente síncrona. Isso significa que um cliente fica bloqueado até receber o resultado de uma chamada de método. Entretanto, esse comportamento pode ser alterado em OMG IDL através do modificador `oneway`. Quando esse modificador é aplicado a um método, as requisições a esse método seguem uma regra de *melhor esforço*, que não garante a entrega da requisição ao objeto servidor. De acordo com o padrão CORBA, um ORB pode descartar requisições `oneway` caso haja problemas de comunicação entre cliente e servidor, sem precisar notificar as partes envolvidas. Uma operação `oneway` não pode ter nenhum valor de retorno e, tipicamente, não bloqueia o cliente que a requisita. COM IDL não permite a declaração de operações assíncronas, mas há uma previsão de incluir um modificador semelhante ao modificador `oneway`, chamado `async_uuid` [Box98].

Apesar de operações declaradas em ambas IDLs poderem sinalizar exceções, as duas linguagens diferem bastante na forma como representam isso em uma interface. Em OMG IDL, a definição de

```
interface Calc : IUnknown {
  HRESULT add([in] float x, [in] float y,
              [out, retval] float *pResult);
  HRESULT sqrt([in] float x, [out, retval] float *pResult);
};
```

Figura 2.3: Exemplo de interface em COM IDL.

um tipo de exceção é feita através da definição de uma estrutura composta por campos de dados. A assinatura de uma operação define as exceções que podem ser sinalizadas por ela através da cláusula `raises`. As exceções pré-definidas pela arquitetura CORBA, tipicamente relacionadas com falhas de comunicação, não precisam ser especificadas nas assinaturas das operações.

Já em COM IDL, uma exceção é representada por um identificador numérico, que tipicamente tem uma mensagem explicativa associada. Todas as operações retornam um código numérico do tipo `HRESULT`, que indica se a operação foi realizada com sucesso ou não. Caso ocorra uma falha, vem codificado nesse valor o identificador do tipo de falha ocorrida. Um código `HRESULT` pode representar tanto uma exceção pré-definida por COM, quanto uma exceção definida pelo desenvolvedor do componente.

As figuras 2.2 e 2.3 apresentam exemplos de uso de OMG IDL e COM IDL, respectivamente, através da representação de um mesmo tipo abstrato de dados nas duas linguagens.

2.1.3 Atributos

Freqüentemente, é útil indicar que um componente tem um certo atributo visível para seus clientes, que pode ser acessado através de sua interface (em COM IDL, esses atributos são chamados de *propriedades*). Esses atributos poderiam ser vistos como uma quebra do encapsulamento do componente. Entretanto, eles só representam uma visão lógica do atributo, e não uma variável de instância do componente. Esses atributos são tipicamente mapeados em pares de métodos de acesso, que são responsáveis pela leitura e escrita do atributo. Assim, o componente pode representar internamente esses atributos de qualquer forma.

O comportamento normal de um atributo de uma interface é permitir sua consulta e alteração pelos clientes da interface. Entretanto, esse comportamento pode ser alterado através de um mecanismo da IDL para tornar o atributo disponível apenas para leitura. No caso de COM IDL, ainda é possível definir um atributo que só possa ser alterado e não consultado. As figuras 2.4 e 2.5 mostram as representações de um mesmo tipo abstrato de dados nas duas IDLs. Esse tipo, chamado `Exemplo-Atributo`, possui um atributo que permite a leitura e escrita de seu valor (`valor`), e um atributo que permite somente a sua leitura (`valor_maximo`).

Em COM IDL, é necessário definir explicitamente os dois métodos de acesso a um atributo. Para isto, são usados os modificadores `propget` e `propput` para identificar os métodos de leitura e escrita de um atributo. Note que, além do uso desses modificadores, é necessário definir o restante das assinaturas dos métodos de forma coerente, isto é, os dois métodos têm que ter o mesmo nome e os parâmetros têm que obedecer o sentido de transferência de informação:

```
propget -> out
propput -> in
```

Uma outra diferença na representação de atributos nas duas linguagens é que, em OMG IDL, o acesso a um atributo pode gerar as exceções de comunicação pré-definidas em CORBA, mas não pode

```
interface ExemploAtributo {
  attribute short valor;
  readonly attribute short valor_maximo;
};
```

Figura 2.4: Exemplo de definição de atributos em OMG IDL.

```
interface ExemploAtributo : IUnknown {
  [propget] HRESULT valor([out, retval] short *pVal);
  [propput] HRESULT valor([in] short Val);

  [propget] HRESULT valor_maximo([out, retval] short *pVal);
};
```

Figura 2.5: Exemplo de definição de atributos em COM IDL.

gerar exceções definidas pelo desenvolvedor. Já em COM IDL, o acesso a um atributo pode retornar qualquer código de erro através do valor de retorno `HRESULT`.

2.1.4 Polimorfismo

Em ambas IDLs, polimorfismo é oferecido através da dissociação entre interfaces e implementações. Em OMG IDL, um polimorfismo adicional é oferecido via herança múltipla de interfaces, onde é adotado um mecanismo tradicional de compatibilidade entre um tipo e seus sub-tipos explicitamente declarados. Já em COM IDL, um polimorfismo adicional pode ser obtido através do conjunto de interfaces oferecidas por um objeto, que pode ser inspecionado dinamicamente através de operações oferecidas pela interface básica `IUnknown` (ver seção 2.3).

2.1.5 Tipos Não-Objeto

Tanto OMG IDL quanto COM IDL oferecem um conjunto de tipos não-objeto, tais como tipos primitivos (inteiros, números de ponto flutuante, *strings*, entre outros) e alguns tipos estruturados (estruturas, uniões, seqüências e arrays).

As duas linguagens também oferecem um tipo de dado especial que é muito similar a uma *união*, pois pode armazenar arbitrariamente outros tipos de dados, e permite a identificação em tempo de execução de qual tipo de dado está sendo correntemente armazenado. Entretanto, enquanto uma união define previamente um conjunto fixo de possíveis valores que pode assumir, esse tipo especial pode armazenar qualquer outro tipo de valor, sem precisar que isso seja previamente declarado.

Em OMG IDL, esse tipo especial é o tipo `any`, que pode armazenar qualquer outro tipo de valor que possa ser representado pela linguagem. Já em COM IDL, esse tipo é o tipo `VARIANT`, que apresenta uma restrição de só poder armazenar tipos de dados que possam ser representados na linguagem Visual Basic, isto é, ele só pode armazenar os tipos básicos de COM IDL, o que inclui referências para objetos dos tipos `IUnknown` e `IDispatch`. Essa restrição se deve ao fato de que o tipo `VARIANT` foi originalmente projetado para trabalhar somente com a linguagem Visual Basic [Bro95].

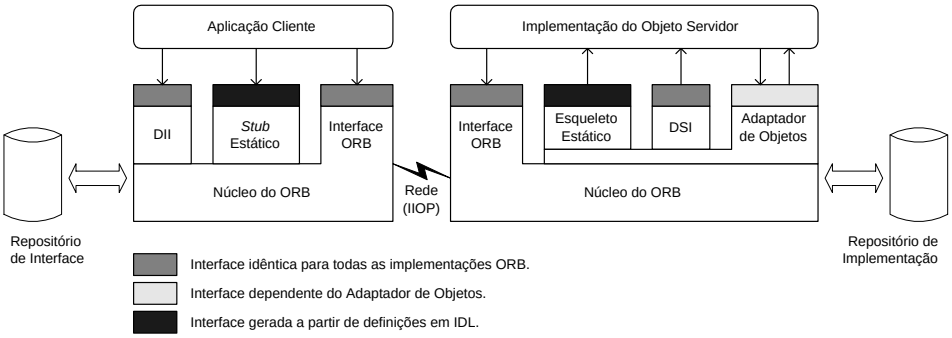

Figura 2.6: A arquitetura CORBA.

2.2 CORBA

O padrão CORBA é o principal elemento da arquitetura definida pela OMG. Entre as especificações de maior relevância do padrão CORBA estão os mapeamentos de OMG IDL para diversas linguagens de programação, tais como C++, Java, C e Smalltalk, e as interfaces do seu *Object Request Broker* (ORB). A figura 2.6 mostra a estrutura geral da arquitetura CORBA.

O ORB é o elemento central dessa arquitetura. Ele é o responsável por estabelecer as conexões entre clientes e servidores, e por repassar as requisições de operações dos clientes para os objetos servidores. Vários aspectos relacionados à heterogeneidade entre plataformas e linguagens de programação são tratados internamente pelo ORB.

Seguindo rigorosamente a nomenclatura adotada pela OMG, a arquitetura CORBA é baseada no conceito de objetos distribuídos, mas não apresenta o conceito de componentes de software. Somente recentemente a OMG definiu o conceito de componentes para a arquitetura CORBA [OMG99a]. Entretanto, para o escopo deste texto, os objetos servidores da arquitetura básica de CORBA apresentam todas as características necessárias para serem considerados componentes de software.

2.2.1 Protocolo de Comunicação

CORBA especifica um padrão para o formato das mensagens enviadas entre objetos através de uma rede de comunicação. Esse padrão é denominado *General Inter-ORB Protocol* (GIOP). O GIOP é aplicado quando um cliente requisita uma operação de um objeto servidor através do ORB. O *Internet Inter-ORB Protocol* (IIOP) é um mapeamento específico do GIOP para o protocolo de transporte TCP/IP. ORBs podem implementar o GIOP para outros protocolos de transporte, mas todas as implementações de ORB devem oferecer pelo menos o IIOP.

Já que o IIOP é um requisito básico para que uma implementação de ORB seja considerada em conformidade com o padrão CORBA, é possível viabilizar a comunicação entre diferentes ORBs. Assim, o IIOP é um elemento essencial para a interoperabilidade entre diferentes implementações de CORBA. Essa interoperabilidade permite que um cliente desenvolvido usando um determinado ORB possa utilizar os serviços oferecidos por um objeto servidor desenvolvido para outro ORB.

2.2.2 Interface entre Cliente e Servidor

Além do mecanismo tradicional de *stubs*, CORBA também oferece recursos para a construção dinâmica de requisições de operações, através de sua *Dynamic Invocation Interface* (DII). De forma seme-

lhante, CORBA provê uma interface para a implementação dinâmica de objetos servidores: a *Dynamic Skeleton Interface* (DSI). Voltaremos a discutir essas interfaces na seção 2.5.

A requisição de uma operação pode obedecer a diferentes regras de execução. Tipicamente, uma requisição segue o modelo síncrono de chamada de procedimentos. Entretanto, devido a seu foco em sistemas distribuídos, CORBA também permite que operações sejam executadas de uma forma assíncrona. Como foi visto na seção 2.1.2, o modificador `oneway` de IDL define uma operação que não pode ter nenhum tipo de valor de retorno, podendo ser implementada de uma forma assíncrona pelo ORB. Essas operações tipicamente não têm nenhuma garantia de entrega.

A DII também oferece um outro modo para realizar requisições, chamado *síncrono postergado* (*deferred synchronous*). Nesse modo de operação, um cliente pode requisitar um método de um objeto servidor sem esperar pelo seu término, continuar o seu processamento, e posteriormente consultar o resultado do método.

2.2.3 Referências de Objetos

Uma *referência de objeto* é o mecanismo utilizado para identificar e localizar um determinado objeto servidor. Para um cliente, referências de objetos são entidades opacas, ou seja, clientes usam essas referências para requisitar as operações dos objetos, mas não podem consultar ou modificar o conteúdo de uma referência. Uma referência de objeto só pode identificar um único objeto CORBA, mas um mesmo objeto pode ter várias referências para ele. Sob vários aspectos, essas referências são análogas aos ponteiros para instâncias de classes C++ [HV99].

Toda referência de objeto contém uma indicação de qual interface seu objeto oferece. Isso permite que um ORB ofereça alguma segurança de tipos em tempo de execução, verificando se a interface oferecida por uma referência tem a operação sendo requisitada. Em linguagens estaticamente tipadas, como C++ e Java, segurança de tipos também é garantida em tempo de compilação. O mapeamento da linguagem não permite que seja requisitada uma operação, a menos que o objeto destino tenha garantidamente a operação em sua interface. Essa garantia em tempo de compilação só existe se o objeto servidor estiver sendo acessado através dos *stubs* gerados automaticamente a partir de uma definição em IDL. O uso de DII faz com que essa garantia seja perdida em tempo de compilação.

As regras de compatibilidade de tipos tradicionais entre uma interface e suas sub-interfaces também valem para as referências de objetos, isto é, uma referência para um objeto com uma determinada interface derivada é considerada compatível com referências para objetos de suas super-interfaces.

Uma referência de objeto pode ser linearizada na forma de uma cadeia de caracteres (*string*), e essa cadeia de caracteres pode ser convertida de volta em uma referência que denota o mesmo objeto original. Essa cadeia de caracteres pode ser armazenada para um uso posterior. Por exemplo, um objeto servidor pode armazenar em um arquivo sua referência linearizada, e posteriormente um cliente pode ler esse arquivo para criar uma referência para esse objeto.

CORBA especifica um formato padrão para a representação de referências de objetos. Esse formato padrão é denominado *Interoperable Object Reference* (IOR). Uma IOR contém todas as informações necessárias para estabelecer a conexão entre um cliente e um objeto servidor. Uma IOR identifica os protocolos disponíveis para conexão com um determinado servidor. No caso do protocolo IIOP, uma IOR armazena o nome da máquina servidora, o número da porta TCP/IP associada ao processo servidor, e um *identificador de objeto* que identifica unicamente o objeto destino naquele processo servidor. Isso significa que um ORB pode usar referências criadas por outras implementações CORBA, tanto através de referências passadas como parâmetros de operações, quanto através de referências importadas em sua forma linearizada.

2.2.4 Repositórios de Interfaces e Implementações

Um ORB deve oferecer um *Repositório de Interfaces*, onde são armazenadas as descrições das interfaces dos objetos servidores disponíveis. O Repositório de Interfaces é o mecanismo básico de introspecção de CORBA, permitindo que um sistema consulte as interfaces dos objetos servidores disponíveis. Esse repositório é um objeto servidor CORBA, o que permite que ele seja utilizado por uma aplicação da mesma forma que outros objetos servidores. A sua interface define uma série de operações que permitem tanto a consulta de definições de interfaces quanto a alteração e criação de novas interfaces.

A arquitetura CORBA também define a existência de um *Repositório de Implementações*. Esse repositório deve auxiliar um cliente na obtenção de uma referência para um determinado objeto servidor. Além disso, um repositório de implementações pode oferecer serviços adicionais para auxiliar em tarefas como migração de objetos servidores, ativação automática de servidores e balanceamento de carga.

2.3 COM

De acordo com a nomenclatura de COM, Um componente é uma coleção de classes, onde cada uma dessas classes é unicamente identificada através de um *identificador de classe* (CLSID). Por sua vez, cada classe COM pode implementar várias *interfaces* COM. Uma interface COM oferece um conjunto de operações, e é unicamente identificada através de um *identificador de interface* (IID). COM diferencia os conceitos de componente e objeto: enquanto um componente é uma coleção de classes, um objeto COM é uma instância de uma classe COM.

2.3.1 Identificadores Globais Únicos

Os identificadores CLSID e IID são identificadores binários únicos, que são designados genericamente como *Identificadores Globalmente Únicos* (*Globally Unique Identifiers* – GUIDs). GUIDs são números de 128 bits que são garantidamente únicos tanto no tempo quanto no espaço. Para garantir que esses identificadores são únicos, eles são gerados a partir de um algoritmo que utiliza como dados de entrada o endereço da interface de rede da máquina onde ele está sendo gerado, a hora do relógio local, e mais dois contadores persistentes, que são usados para compensar imprecisões na leitura do relógio e mudanças anormais no relógio da máquina (horário de verão ou ajustes da hora feitos pelo usuário da máquina). Quando a máquina local não possui uma interface de rede, é gerado um identificador estatisticamente único, que só tem sua exclusividade garantida na máquina local. Os GUIDs de COM são baseados nos UUIDs (*Universally Unique Identifiers*) utilizados no sistema DCE RPC [RKF92].

A finalidade desses identificadores é evitar a colisão de nomes de interfaces e classes de objetos COM. Eles representam uma espécie de nome *físico* das entidades a eles associadas, que são definidos em tempo de projeto em conjunto com o nome lógico da entidade. Por exemplo, a interface mostrada na figura 2.7 tem o nome lógico `IClassFactory` e o IID `00000001-0000-0000-C000-000000000046`. A interface de programação de COM oferece funções e ferramentas para gerar automaticamente esses identificadores.

Uma vez “publicada”, uma interface COM é considerada imutável. Novas versões da interface requerem a geração de novos IIDs, forçando efetivamente a introdução de novas interfaces. Um objeto pode oferecer simultaneamente múltiplas versões de uma interface.

```
[object, uuid(00000001-0000-0000-C000-000000000046)]
interface IClassFactory : IUnknown {
   HRESULT CreateInstance([in] IUnknown *pUnkOuter,
                          [in] REFIID riid,
                          [out, iid_is(riid)] void** ppv);
   HRESULT LockServer([in] BOOL bLock);
}
```

Figura 2.7: A definição da interface `IClassFactory` em COM IDL.

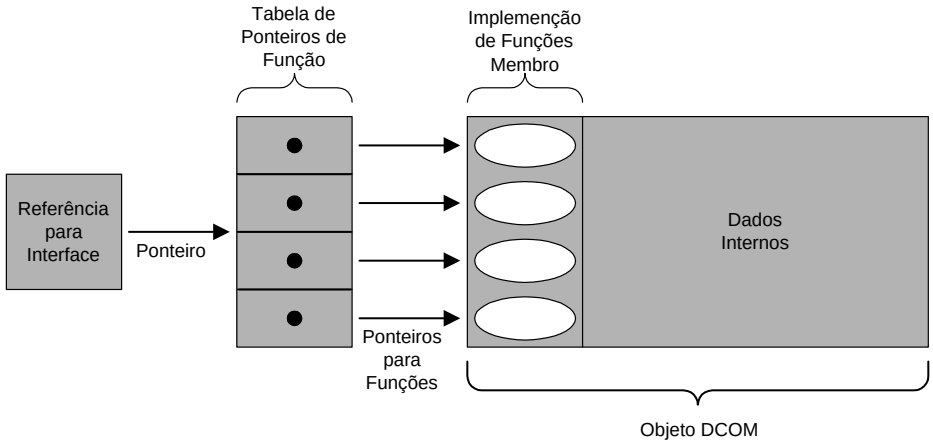

Figura 2.8: Uma interface COM.

2.3.2 Criação de Objetos

Para instanciar um objeto de uma determinada classe, a aplicação cliente chama uma função da interface de programação de COM fornecendo os identificadores da classe e da interface desejada (CLSID e IID, respectivamente). COM procura no registro do sistema pelo CLSID fornecido, para localizar a implementação da classe e poder instanciá-la. Algumas classes podem ter suas instâncias compartilhadas por vários clientes.

Um componente COM pode oferecer uma interface especial, chamada `IClassFactory` (figura 2.7). Essa interface é usada para criar novas instâncias de objetos de uma determinada classe.

2.3.3 Formato Binário de Interfaces

A comunicação entre componentes COM é feita exclusivamente através de suas interfaces. A arquitetura COM define um padrão para o formato binário da representação em memória de uma interface. Esse padrão estipula que uma interface tem que ser igual a uma tabela de métodos virtuais de C++ [ES90]. Assim, interfaces são representadas em memória por ponteiros para tabelas de ponteiros de função (figura 2.8). Esse formato permite uma comunicação eficiente entre objetos.

A princípio, esse padrão binário torna uma interface de componente independente de linguagem de programação. Entretanto, essa independência é um pouco limitada, pois exige que uma linguagem para ser integrada a COM represente objetos exatamente como C++, o que é uma restrição muito forte, ou

```
[object, uuid(00000000-0000-0000-C000-000000000046)]
interface IUnknown {
   HRESULT QueryInterface([in] REFIID riid, [out] void** ppv);
   ULONG AddRef(void);
   ULONG Release(void);
}
```

Figura 2.9: A definição da interface `IUnknown` em COM IDL.

disponibilize um suporte em tempo de execução para fazer a ponte necessária entre as representações nativas da linguagem e o padrão de COM. Visual Basic e Visual J++ (ferramenta de programação em Java da Microsoft) utilizam essa última técnica, onde as suas máquinas virtuais são responsáveis por esse suporte.

2.3.4 A Interface IUnknown

As interfaces oferecidas por um objeto para que seus clientes possam acessá-lo são denominadas *interfaces de entrada* (*incoming interfaces*). A interface `IUnknown` (figura 2.9) é a interface básica que todo objeto COM deve oferecer (implementar), e da qual todas as demais interfaces COM devem derivar. Isto significa que a representação binária de todas as interfaces COM são ponteiros para tabelas que começam com os três ítens `QueryInterface`, `AddRef` e `Release`. Qualquer método específico de uma interface irá aparecer depois desses três ítens comuns.

As operações `AddRef` e `Release` são utilizadas no controle do número de referências para um objeto. Tanto o desenvolvedor de uma classe de objeto quanto o desenvolvedor da aplicação cliente são responsáveis por manter o controle do número de referências para os objetos criados. Quando um objeto passa para um cliente uma de suas interfaces, ele deve incrementar o seu contador de referências. Quando um cliente faz uma cópia de um ponteiro de interface, ele deve incrementar o contador do objeto através da operação `AddRef`.

Quando um cliente não vai mais utilizar um ponteiro de interface, ele deve decrementar o contador de referências do objeto, chamando a operação `Release`. Quando o contador de referências do objeto voltar a zero, o objeto deve se destruir.

A operação `QueryInterface` é utilizada para fazer a coerção de tipos de um objeto. Essa operação permite requisitar, através de IIDs, qualquer outra interface implementada por um determinado objeto COM. Seu primeiro parâmetro é o nome físico (IID) da interface sendo requisitada. O seu segundo parâmetro deve ser um ponteiro para uma variável, que por sua vez deve ser um ponteiro de interface. Se a operação `QueryInterface` for concluída com sucesso, o parâmetro `ppv` vai conter o ponteiro para a interface desejada e o valor de retorno da operação será `S_OK`. Se o objeto não oferecer a interface sendo requisitada, a operação retornará o valor `E_NOINTERFACE` e a área de memória apontada por `ppv` vai conter o valor `NULL`.

É interessante observar que, apesar da operação `QueryInterface` ser a base para o sistema de tipos de COM, ela não oferece praticamente nenhuma segurança de tipo em C++. Como o tipo do parâmetro `ppv` é `void**`, não é possível para um compilador realizar qualquer verificação de tipos sobre o tipo real do ponteiro passado como parâmetro. Assim, fica a cargo dos clientes da interface `IUnknown` garantir a correção do tipo de `ppv`.

A especificação de COM define um conjunto de regras bem precisas para que uma implementação da interface `IUnknown` esteja de acordo com o seu modelo. Várias dessas regras estão diretamente relacionadas com propriedades que devem ser observadas em todas as implementações da operação `QueryInterface`:

Simetria — Se uma requisição de `QueryInterface` para obter uma interface `B` é satisfeita através de um ponteiro do tipo `A`, então uma requisição de `QueryInterface` para obter uma interface `A` a partir do ponteiro resultante do tipo `B` do mesmo objeto nunca pode falhar. Isto significa que

```
pA.QI(B)->pB
```

implica em

```
pB.QI(A)->pA
```

Transitividade — Se uma requisição de `QueryInterface` para obter uma interface `B` é satisfeita através de um ponteiro do tipo `A` e se uma segunda requisição de `QueryInterface` para obter uma interface `C` é satisfeita através do ponteiro do tipo `B`, então uma requisição de `QueryInterface` para obter uma interface `C` a partir do ponteiro original do tipo `A` também tem que ser satisfeita. Isto significa que

```
pA.QI(B)->pB e pB.QI(C)->pC
```

implica em

```
pA.QI(C)->pC
```

Reflexividade — Uma requisição de `QueryInterface` tem que ser sempre satisfeita se o tipo da interface requisitada for o mesmo do ponteiro de interface utilizado para realizar a requisição. Isto significa que

```
pA.QI(A)->pA
```

tem que ocorrer sempre.

O fato de `QueryInterface` ser simétrica, transitiva e reflexiva implica que qualquer ponteiro de interface para um determinado objeto deve sempre oferecer o mesmo tipo de resposta *sim/não* para uma determinada requisição de `QueryInterface`. Cabe observar que a operação `QueryInterface` define uma relação de equivalência.

Um corolário que pode ser inferido a partir dessas três propriedades de `QueryInterface` é que o conjunto de interfaces oferecidas por um objeto não pode mudar ao longo do tempo. A especificação de COM requer que esse corolário seja verdade para todos os objetos. Esse requisito implica que a hierarquia de tipos de um objeto é estática, independentemente do fato de que clientes devem interrogar dinamicamente um objeto para obter o seu conjunto de interfaces oferecidas. Entretanto, não há nenhuma restrição na especificação de COM que proíba duas instâncias (objetos) de uma mesma classe oferecerem diferentes conjuntos de interfaces.

O conjunto de interfaces que um objeto oferece, e que podem ser acessadas através do método `QueryInterface`, determina o *tipo* de um objeto COM. Um subtipo é um super-conjunto de interfaces. Por exemplo, assuma que um cliente requer um objeto com o conjunto de interfaces $\{A, B, C\}$. Um objeto que ofereça o conjunto de interfaces $\{A, B, C, D, E\}$ satisfaz os requisitos do cliente e poderia ser utilizado diretamente pelo cliente. Assim, é possível testar dinamicamente se um objeto é de um tipo compatível com o tipo requisitado pelo cliente, através de consultas com o método `QueryInterface`.

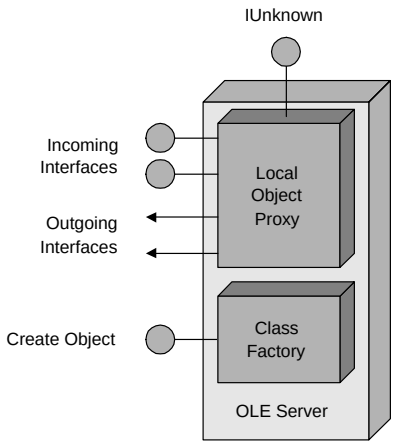

Figura 2.10: A estrutura de um componente COM.

COM também define a *identidade* de um objeto através de uma regra aplicada à operação `QueryInterface`. De acordo com essa regra, toda vez que for requisitado, através da operação `QueryInterface`, um ponteiro para a interface `IUnknown` de um determinado objeto, deve-se obter exatamente o mesmo ponteiro. Dessa forma, o ponteiro para a interface `IUnknown` funciona como a identidade de um objeto.

A infra-estrutura de COM não oferece nenhuma implementação *default* para a interface `IUnknown`, nem nenhuma facilidade adicional para auxiliar o desenvolvedor nessa tarefa. Assim, o desenvolvedor do componente é totalmente responsável por atender aos ítens de conformidade com o modelo de COM.

2.3.5 Interfaces de Saída e Pontos de Conexão

A figura 2.10 mostra a estrutura típica de um componente COM. Além das interfaces de entrada, o modelo de componentes COM também possui o conceito de *interfaces de saída* (*outgoing interfaces*). As interfaces de saída de um componente têm um fluxo de chamadas no sentido contrário das interfaces de entrada, isto é, ao invés do componente implementar as interfaces de saída, ele vai requerer outros componentes que as implementem.

COM utiliza o mecanismo de *pontos de conexão* para representar as interfaces de saída de um componente explicitamente [Box98]. Pontos de conexão permitem o registro e a chamada de métodos de objetos de *callback*. Pontos de conexão não têm o objetivo de dar suporte a redes de objetos altamente conectadas. Eles também não oferecem comunicação bi-direcional. Ao invés disso, um ponto de conexão expressa, através de um pequeno conjunto de interfaces padronizadas, o conceito mais geral de registro de objetos de *callback*, que podem atender a diferentes interfaces de saída. Esse conceito é muito similar ao de *eventos* gerados por um componente JavaBeans (um *event source*) que são interceptados por um outro *bean* (um *event listener*) [Jav97].

Um ponto de conexão é representado pela interface `IConnectionPoint` (figura 2.11). A operação `GetConnectionInterface` retorna o identificador da interface de saída que um determinado ponto de conexão está controlando (um ponto de conexão só pode ter uma interface associada a ele). A operação `GetConnectionPointContainer` retorna o objeto ao qual o ponto de conexão pertence. Já as operações `Advise` e `Unadvise` são responsáveis, respectivamente, por fazer e cancelar o registro de objetos de `callback` em um determinado ponto de conexão. Os objetos de

```
interface IConnectionPoint : IUnknown {
  HRESULT GetConnectionInterface([out] IID* pIID);
  HRESULT GetConnectionPointContainer(
             [out] IConnectionPointContainer** ppCPC);
  HRESULT Advise([in] IUnknown* pUnkSink; [out] DWORD* pdwCookie);
  HRESULT Unadvise([in] DWORD dwCookie);
  HRESULT EnumConnections([out] IEnumConnections** ppEnum);
};

interface IConnectionPointContainer : IUnknown {
  HRESULT EnumConnectionPoints([out] IEnumConnectionPoints** ppEnum);
  HRESULT FindConnectionPoint([in] REFIID riid,
                              [out] IConnectionPoint** ppCP);
};
```

Figura 2.11: As interfaces `IConnectionPoint` e `IConnectionPointContainer`.

callback registrados devem implementar a interface de saída representada pelo ponto de conexão (a mesma retornada por `GetConnectionInterface`). Por fim, a operação `EnumConnections` retorna uma coleção com todos os objetos de *callback* atualmente registrados em um determinado ponto de conexão.

A princípio, um componente deve oferecer uma implementação diferente da interface `IConnectionPoint` para cada uma de suas interfaces de saída. Entretanto, a interface `IConnectionPoint` não é exportada no tipo do componente, isto é, ela não pode ser obtida através do método `QueryInterface` do componente que possui os pontos de conexão. Assim, para permitir a consulta dos pontos de conexão oferecidos por um componente e, conseqüentemente, de suas interfaces de saída, COM define a interface `IConnectionPointContainer` (figura 2.11). Todo componente que quiser oferecer pontos de conexão deve implementar essa interface (de entrada). Essa interface faz parte do tipo desses componentes, podendo ser obtida através do método `QueryInterface`.

2.3.6 Categorias de Componentes

COM define o conceito de *Categoria de Componentes* para facilitar a identificação de componentes que têm determinados conjuntos de interfaces de entrada e de saída. Categorias são identificadas pelos seus próprios GUIDs (CATID), e são definidas como conjuntos de identificadores de interfaces (IIDs).

Um objeto COM pode ser membro de diversas categorias, e não existe nenhuma definição de relacionamento entre categorias. Uma classe COM pode especificar de quais categorias suas instâncias fazem parte (categorias implementadas), assim como pode especificar quais categorias de componentes seus objetos requerem (categorias requisitadas). Dessa forma, um cliente pode verificar mais facilmente quais classes de componentes oferecem as interfaces que ele necessita, como também pode verificar se ele é capaz de prover os serviços que um componente requer. Uma categoria pode definir uma determinada classe como sua implementação *default*.

2.4 Java e JavaBeans

Apesar de ser sintaticamente similar a C++, Java é uma linguagem orientada a objetos pura, que apresenta uma semântica bem distinta de C++ [GJS96, AG98]. Algumas das características básicas de Java que a diferenciam de C++ são a adoção de uma semântica referencial, que define que um programa

manipula somente referências para objetos, e o gerenciamento automático de memória através de um mecanismo de coleta de lixo.

Uma outra característica fundamental de Java é a separação dos conceitos de tipos e classes, assim como de suas respectivas hierarquias. Da mesma forma que Duo-Talk [Lun89], Pool [AvdL90] e School [RIR93], Java considera um tipo como sendo uma especificação e uma classe como sendo a definição de uma implementação. O tipo de um objeto é sua aparência externa, isto é, sua interface para o mundo externo. Uma interface em Java é muito similar às interfaces de CORBA e COM. Já a classe de um objeto define sua estrutura interna, isto é, seu estado e o código que o manipula. Por exemplo, considere a seguinte interface Java que especifica um tipo abstrato de dados `Point`:

```
public interface Point {
   double getX();
   double getY();
   void moveBy(Point pt);
}
```

Essa interface poderia ter uma implementação dada pela classe `PointCart`, apresentada a seguir:

```
class PointCart implements Point {
   protected double x;
   protected double y;
   public double getX() { return x; }
   public double getY() { return y; }
   public void moveBy(Point pt) {
      x = x + pt.getX();
      y = y + pt.getY();
   }
}
```

Java oferece mecanismos de herança tanto de interfaces quanto de classes. Entretanto, uma classe pode implementar e uma interface pode herdar de tantas interfaces quanto forem necessárias (herança múltipla de interfaces), mas uma classe só pode herdar de uma única outra classe (herança simples de classes).

As classes que não herdam explicitamente de uma outra classe são implicitamente subclasses da classe primitiva `Object`. Todos os objetos são polimorficamente da classe `Object`, e assim `Object` é o tipo genérico usado para referências que devem poder designar objetos de qualquer classe. A classe `Object` define métodos de suporte a *threads* e mais alguns métodos utilitários, tais como métodos para teste de igualdade entre objetos, clonagem e introspecção (ver seção 2.5).

2.4.1 A Máquina Virtual de Java

Java foi projetada para maximizar sua portabilidade. Para isso, ela adotou para suas implementações uma arquitetura baseada em uma máquina virtual (*Java Virtual Machine* — JVM). Essa máquina virtual define uma linguagem de máquina para a qual um programa Java deve ser compilado. O formato de um programa Java já compilado é tipicamente chamado de *bytecode*, e uma máquina virtual Java é responsável por executar esses *bytecodes* em diferentes plataformas. Para podermos executar um programa Java em uma determinada plataforma, precisamos apenas de uma implementação da máquina virtual de Java para a plataforma desejada. Assim, a princípio, um programa Java compilado em *bytecodes* pode ser executado em qualquer plataforma com uma implementação disponível da JVM.

A JVM provê um ambiente de execução que permite que um programa Java tenha acesso à própria máquina virtual e ao ambiente externo. A JVM verifica operações com possíveis riscos de segurança através de um *gerente de segurança*. O gerente de segurança pode, por exemplo, proibir que uma aplicação tenha acesso de leitura ou escrita ao disco local, ou pode restringir conexões remotas a apenas algumas máquinas específicas.

Quando classes são carregadas em uma máquina virtual, esta primeiro verifica se os *bytecodes* estão com o formato correto e se as exigências de segurança são atendidas. Por exemplo, a máquina virtual verifica se os *bytecodes* de uma classe não tentam usar um inteiro como uma referência para ter acesso direto a áreas de memória.

Esses recursos de segurança combinados com a independência de plataforma do *bytecode* de um programa Java oferecem um modelo seguro e apropriado para executar códigos transferidos através de uma rede de computadores, com diferentes níveis de confiabilidade.

A padronização entre diferentes plataformas da representação binária dos tipos de dados de Java facilita o oferecimento de um mecanismo para a serialização de objetos. Através desse mecanismo de serialização, Java permite que o estado de um objeto seja convertido em uma cadeia de *bytes*, que pode ser armazenada em um arquivo ou transferida através de uma rede de computadores.

2.4.2 RMI

Java oferece um mecanismo de suporte a chamadas remotas de métodos, denominado RMI (*Remote Method Invocation*). Esse mecanismo é disponibilizado para o desenvolvedor através de um pacote (biblioteca de classes e interfaces) chamado `java.rmi`. O RMI oferece um modo de criar objetos cujos métodos podem ser chamados a partir de outras máquinas virtuais, incluindo aquelas que estão executando em outras estações de uma rede.

A infra-estrutura oferecida pelo RMI para programação distribuída é muito similar à oferecida por CORBA. Entretanto, uma diferença básica entre essas duas infra-estruturas é que RMI só trata de objetos Java, o que simplifica diversos de seus aspectos. Por exemplo, a passagem de objetos por valor em uma chamada remota de método em RMI é bem mais simples e direta do que em CORBA, pois esses objetos podem ser serializados e copiados para uma outra máquina virtual utilizando os recursos básicos de serialização oferecidos por Java. Isto é, ao ser passado como parâmetro em uma chamada de método de um servidor RMI, um objeto que não seja um servidor RMI é serializado e transferido para o processo no qual o objeto servidor reside. Cabe observar que objetos passados por valor em chamadas RMI são cópias dos objetos originais, e não mantêm nenhum vínculo com estes. Assim, alterações feitas de um lado não afetam o outro, pois cada lado possui sua própria cópia do objeto.

2.4.3 JavaBeans

O sistema JavaBeans define um sistema de componentes de software para Java, preservando o modelo de objetos dessa linguagem. De acordo com [Jav97], um *bean* é um componente reusável que pode ser manipulado visualmente através de uma ferramenta de composição. Uma ferramenta de composição pode ser um construtor de páginas WWW, um construtor visual de aplicações, um construtor de interfaces com o usuário, e até um editor de documentos compostos.

Um *bean* pode ser um simples elemento de interface com o usuário, tal como um botão ou uma lista de opções, ou um sofisticado componente visual, tal como um visualizador de banco de dados ou um formulário de entrada de dados. Mesmo que um *bean* não ofereça uma interface gráfica com o usuário, ele ainda deve poder ser composto visualmente com outros *beans* através de uma ferramenta de construção de aplicações.

Diferentes *beans* irão variar com relação à funcionalidade que eles oferecem, mas alguns recursos são comuns a todos os *beans*:

- *introspecção*: uma ferramenta de composição pode analisar as características de um *bean*, tais como seus métodos, eventos e propriedades;
- *configuração*: um usuário pode configurar a aparência e o comportamento de um *bean* através de uma ferramenta de composição;
- *eventos*: *beans* podem ser conectados através de uma metáfora simples de comunicação baseada em eventos;
- *propriedades*: um *bean* tipicamente oferece um conjunto de propriedades (ou atributos) que podem ser utilizadas para fins tanto de configuração quanto de programação;
- *persistência*: um *bean* pode ser configurado através de uma ferramenta de composição, e então ter sua configuração armazenada externamente e recarregada posteriormente.

A princípio, um *bean* não precisa herdar de nenhuma classe básica ou interface. Entretanto, um *bean* visual tem que herdar de `java.awt.Component`, para que ele possa ser adicionado a um recipiente (*container*) visual de componentes. Outras classes e interfaces úteis para a implementação de *beans* são oferecidas por Java através do pacote `java.beans`.

As três características mais importantes de um *bean* são o conjunto de *propriedades* que ele expõe, o conjunto de *métodos* que ele permite que outros objetos chamem, e o conjunto de *eventos* que ele gera[3]. Propriedades são basicamente atributos associados a um *bean*, que podem ser consultados e alterados através de métodos apropriados do *bean*. Esses métodos para manipulação de propriedades seguem certas regras de formação de seus nomes. Por exemplo, se um *bean* tem uma propriedade chamada `Color`, sua interface deve oferecer as operações `getColor` e `setColor` para consultar e alterar o valor dessa propriedade.

Os métodos exportados por um *bean* são apenas métodos regulares de Java, que podem ser chamados por outros componentes. Por *default*, todos os métodos públicos de um *bean* são exportados, mas um *bean* pode escolher exportar apenas um subconjunto de seus métodos públicos.

O mecanismo de eventos oferece um modo para que um componente notifique outros componentes que ocorreu algum evento que lhes interesse. De acordo com esse mecanismo, um objeto observador de eventos (*listener*) pode ser registrado em um objeto gerador de eventos. Quando um objeto gerador de eventos detecta que ocorreu alguma alteração em seu estado, ele chama os métodos apropriados de seus objetos *listeners*. Cabe observar que um objeto *listener* pode ser um gerador de eventos para outros objetos. Uma ferramenta de construção de aplicações baseada em JavaBeans pode fazer a conexão entre componentes através de ligações entre objetos geradores de eventos e objetos *listeners*.

Uma ferramenta de construção de aplicações pode obter as informações necessárias sobre as características de um *bean* através de mecanismos de introspecção. JavaBeans oferece dois mecanismos para se obter essas informações. O primeiro mecanismo é baseado na interface `BeanInfo`. Se para um determinado *bean* existir uma classe associada a ele que implemente a interface `BeanInfo`, um ambiente de desenvolvimento para JavaBeans pode utilizar essa classe para obter informações sobre as características do *bean*, tais como eventos, métodos e propriedades oferecidos e o ícone que deve ser utilizado para sua representação gráfica no ambiente de desenvolvimento. A associação entre um *bean* e uma implementação da interface `BeanInfo` é feita através de uma convenção de nomes. Por exemplo, se a classe que implementa um *bean* se chama `Foo`, a classe associada a esse *bean* que implementa a interface `BeanInfo` deve se chamar `FooBeanInfo`.

O outro mecanismo que pode ser utilizado para se obter as características de um *bean*, no caso de não haver uma implementação da interface `BeanInfo` para o componente, é o próprio mecanismo de

[3]Como veremos na seção 3.1, essas três características estão presentes de alguma forma em todos os sistemas de componentes.

introspecção oferecido por Java (ver seção 2.5). Nesse caso, JavaBeans define uma série de regras para a formação de nomes de métodos, classes e interfaces Java. Essas regras de definição de nomes em conjunto com o mecanismo de introspecção de Java permitem que um ambiente de desenvolvimento infira quais são os métodos, eventos e propriedades oferecidos por um *bean*.

Um dos principais objetivos de JavaBeans é oferecer uma arquitetura de componentes independente de plataforma. A princípio, um *bean* deve poder ser usado em outros sistemas de componentes. Mas para que isso seja possível, é necessário existir uma ponte entre JavaBeans e o outro sistema de componentes (ver seção 2.6).

Cabe destacar que JavaBeans, apesar de ser um sistema de componentes, não pode ser classificado como um sistema de middleware, ao contrário de CORBA e COM. Isto se deve ao fato de que o elemento realmente responsável pela comunicação entre os componentes de Java é a sua máquina virtual (incluindo o suporte a RMI). Ela é o elemento que fica entre a aplicação e o nível básico do sistema operacional e do sistema de rede, e que permite a distribuição, a interoperabilidade e a portabilidade de aplicações entre diferentes plataformas. Dessa forma, quem faz o papel de middleware é a máquina virtual de Java. JavaBeans fica responsável pela definição de uma arquitetura para orientar o desenvolvimento de componentes reutilizáveis, através de um modelo para o desacoplamento e a composição de componentes.

2.4.4 Enterprise JavaBeans

A tecnologia Enterprise JavaBeans (EJB) define um modelo para o desenvolvimento e a disponibilização de componentes servidores reutilizáveis para Java [Tho98, EJB99]. EJB é uma extensão do sistema de componentes de JavaBeans para oferecer um suporte mais adequado para componentes servidores.

De acordo como a terminologia de EJB, *componentes servidores* são partes de uma aplicação que executam em um servidor de aplicações. A tecnologia EJB faz parte da plataforma da Sun Microsystems chamada *Enterprise Java*, que é um ambiente Java robusto para o suporte a aplicações com rigorosos requisitos de escalabilidade, distribuição e disponibilidade. A tecnologia EJB oferece suporte ao desenvolvimento de aplicações baseadas em uma arquitetura de objetos distribuídos em várias camadas (*multitier*), na qual grande parte da lógica da aplicação é movida do cliente para o servidor. A lógica da aplicação é particionada em um ou mais objetos que são disponibilizados em um servidor de aplicações.

Um servidor de aplicações Java oferece um ambiente de execução otimizado para componentes Java. Combinando tecnologias tradicionais de processamento de transações *online* com tecnologias de objetos distribuídos, EJB espera que um servidor de aplicações Java ofereça um ambiente de execução robusto com elevados graus de escalabilidade e desempenho, especialmente adequado para aplicações para a Internet.

A tecnologia EJB também espera elevar o grau de portabilidade oferecido por Java. Um componente EJB deve não apenas poder executar em qualquer plataforma, como também ser completamente portável entre diferentes implementações de servidores de aplicações compatíveis com EJB. Para isso, o ambiente EJB deve mapear automaticamente um componente para uma determinada infra-estrutura de serviços.

A infra-estrutura de EJB define uma série de interfaces padronizadas para que uma aplicação possa ter acesso a serviços de chamadas remotas de métodos (RMI), de nomes e diretórios (JNDI), de integração com CORBA (Java IDL), de criação dinâmica de páginas HTML (*Servlets* e JSP), de mensagens (JMS), de transações (JTS), e de acesso a bancos de dados (JDBC).

Uma descrição mais detalhada da arquitetura EJB está além do escopo deste trabalho. Entretanto, para o modelo de composição que apresentamos no capítulo 4, o dado mais importante é que tanto JavaBeans quanto EJB seguem o modelo de objetos de Java, fazendo com que um componente

JavaBeans ou EJB seja, em última análise, um objeto Java.

2.5 Reflexividade e Metaprogramação

Os sistemas de componentes discutidos anteriormente têm progressivamente incluído mecanismos de reflexividade entre seus recursos [OMG99a, WL98]. Tipicamente, esses mecanismos têm o objetivo de atender a sistemas que necessitam se adaptar em tempo de execução a novos requisitos de software e hardware. Originalmente, eram oferecidos mecanismos que permitiam a adaptação das aplicações, mas existe uma forte tendência de também oferecer esses mecanismos para a adaptação do próprio sistema de componentes [RKC99, KRL$^+$00, WSL00, KS00].

Sistemas reflexivos são estruturados em níveis: um nível base que contém entidades que representam abstrações do domínio da aplicação, e um ou mais meta-níveis que contêm entidades que representam abstrações do próprio sistema computacional [KdRB91, Ste94, Lis97]. Um meta-nível permite obter informações e realizar transformações em tempo de execução sobre suas abstrações, podendo influir no comportamento do nível base da aplicação ou de um meta-nível subjacente. Essa estruturação em níveis possibilita a implementação de novas propriedades em aplicações de forma não intrusiva.

Um exemplo de sistema reflexivo seria um modelo de objetos que disponibilizasse para a aplicação algumas de suas abstrações — tais como classes, tipos e requisições de métodos — na forma de objetos que possam ser manipulados durante a execução da aplicação. O ato de tornar tais abstrações do meta-nível disponíveis para manipulação pela aplicação durante a sua execução é chamado de *reificação*. As informações reificadas podem estar disponíveis apenas para consulta ou podem permitir a sua própria alteração. A programação que manipula as informações reificadas é chamada de *metaprogramação*.

Quais informações devem ser reificadas em um sistema reflexivo são definidas de acordo com o objetivo que se espera alcançar com a reflexão. No caso de sistemas de componentes, essas informações são tipicamente as descrições de interfaces, as requisições de métodos, as informações sobre interfaces oferecidas e utilizadas por um componente, as relações de dependências estáticas e dinâmicas entre os componentes (informações sobre a arquitetura do sistema), entre outras.

Uma vez escolhido o conjunto de informações a serem reificadas, um sistema reflexivo deve definir uma interface para manipular essas meta-informações. Em sistemas reflexivos orientados a objetos, essa interface é denominada *Protocolo de Meta-objetos* (MOP — *Metaobject Protocol*) [KdRB91, Lis97].

O modelo de composição proposto nesta tese se baseia fortemente em recursos de reflexão computacional. Do ponto de vista dos sistemas de componentes, estamos particularmente interessados nos mecanismos reflexivos que esses sistemas oferecem para *very late binding*:

- Introspecção: mais especificamente, a habilidade de inspecionar em tempo de execução as descrições completas das interfaces dos componentes.
- Chamadas dinâmicas: a habilidade de construir novas chamadas de métodos em tempo de execução.
- Implementações dinâmicas: a habilidade de construir novas implementações de componentes em tempo de execução.

Como descrevemos nas próximas seções, esses recursos estão presentes de diferentes formas nos três sistemas de componentes apresentados anteriormente.

```
interface Request {
   Status add_arg (in Identifier name,   //nome do argumento
                   in TypeCode arg_type, //tipo do argumento
                   in void *value,       //valor do argumento
                   in long len,          //tamanho em bytes do valor
                   in Flags arg_flags);  //in, out, ou inout
   Status invoke (in Flags invoke_flags);
   Status delete();
   Status send (in Flags invoke_flags);
   Status get_response (in Flags response_flags)
      raises (WrongTransaction);
};
```

Figura 2.12: A interface `Request` em OMG IDL.

2.5.1 Reflexividade em CORBA

Em CORBA, as descrições das interfaces de objetos podem ser obtidas dinamicamente através de consultas ao Repositório de Interfaces. Esse repositório é um objeto servidor CORBA que mantém as descrições das interfaces disponíveis, e através dele podemos consultar e alterar essas descrições.

Para oferecer chamadas dinâmicas, CORBA possui uma interface específica chamada *Interface de Invocação Dinâmica* (DII – *Dynamic Invocation Interface*). Através da DII, um programa pode dinamicamente construir chamadas de métodos e invocá-las. Uma chamada de método pode ser manipulada através da interface `Request` (figura 2.12), que permite a construção da lista de parâmetros, a invocação da operação no objeto servidor, e a obtenção dos valores de retorno. Além disto, essa interface permite a escolha entre os modos síncrono ou assíncrono para realizar uma chamada de método. É completamente transparente para um objeto servidor se uma operação foi requisitada através da DII ou através de um *stub* estaticamente definido. Dessa forma, chamadas dinâmicas podem ser aplicadas a qualquer objeto CORBA.

CORBA também oferece uma interface pré-definida, chamada *Interface de Esqueleto Dinâmico* (DSI – *Dynamic Skeleton Interface*), que permite implementar objetos que não conhecem, em tempo de compilação, o tipo que estão implementando [OMG98]. A idéia básica da DSI é implementar todas as chamadas a um objeto particular através de um único ponto de entrada, a *Rotina de Implementação Dinâmica* (DIR – *Dynamic Implementation Routine*). Essa rotina é responsável pelo desempacotamento dos argumentos e pela invocação da rotina apropriada para a chamada sendo executada. O uso mais freqüente da DSI tem sido a implementação de pontes entre ORBs diferentes [Sie96, ZM97]. Nesse contexto, o servidor dinâmico atua como um representante (*proxy*) de um objeto que reside em outro ORB. Entretanto, a DSI também pode ser usada para implementar objetos servidores em tempo de execução, através, por exemplo, de uma linguagem interpretada [Sie96, MRI99].

Para acionar um servidor dinâmico, o ORB invoca a DIR correspondente passando um único argumento, um objeto do tipo `ServerRequest`. A figura 2.13 mostra a interface `ServerRequest` em IDL. O atributo `operation` identifica o método que está sendo chamado. A lista de parâmetros é obtida através do método `arguments`. Os demais métodos são relacionados com o retorno de resultados, sinalização de exceções e recuperação da informação de contexto especificada em IDL para a operação sendo requisitada.

Também é completamente transparente para um cliente se um objeto CORBA é implementado usando DSI ou não. Assim, clientes CORBA, dinâmicos ou estáticos, podem utilizar simultaneamente diferentes implementações estáticas e dinâmicas de uma mesma interface, de uma forma totalmente transparente.

```
interface ServerRequest {
  readonly attribute Identifier operation;
  void arguments (inout NVList nv);
  Context ctx();
  void set_result (in Any value);
  void set_exception (in Any value);
};
```

Figura 2.13: A interface `ServerRequest` em OMG IDL.

```
interface IDispatch : IUnknown {
   HRESULT Invoke ([in] DISPID id, [in] REFIID riid, [in] LCID lcid,
                   [in] WORD wFlags, [in,out] DISPPARAMS *pDispParams,
                   [out] VARIANT *pVarResult,
                   [out] EXCEPINFO *pExcepInfo, [out] UINT *puArgErr);
   HRESULT GetTypeInfoCount ([out] UINT *pctinfo);
   HRESULT GetTypeInfo ([in] UINT iTInfo, [in] LCID lcid,
                        [out] ITypeInfo **ppTInfo);
   HRESULT GetIDsOfNames ([in] REFIID riid,
                          [in, size_is(cNames)] LPOLESTR *rgszNames,
                          [in] UINT cNames, [in] LCID lcid,
                          [out, size_is(cNames)] DISPID *rgid);
};
```

Figura 2.14: A interface `IDispatch` em COM IDL.

2.5.2 Reflexividade em COM

A abordagem adotada por COM é bem similar a de CORBA. Apesar de não existir explicitamente um repositório de interfaces, um componente COM pode prover uma *biblioteca de tipos* (*type library*), que é uma biblioteca dinâmica (DLL) que pode ser inspecionada em tempo de execução através de uma interface COM específica para esse fim (`ITypeInfo`). A princípio, ao contrário de CORBA, as informações nessas bibliotecas não podem ser alteradas, a menos que toda a biblioteca seja substituída.

Para permitir chamadas dinâmicas, um componente COM tem que oferecer uma interface especial, chamada `IDispatch` (figura 2.14). Essa interface tem uma operação principal chamada `Invoke`, que recebe um seletor da operação desejada (parâmetro `id`) e uma estrutura com os argumentos da operação (parâmetro `pDispParams`). Em contraste com CORBA e Java, a estrutura usada para a passagem de parâmetros através da operação `Invoke` não pode representar todos os tipos de dados do modelo de COM. Sendo assim, nem todas as operações COM podem ser acessadas através da interface `IDispatch`, mas somente aquelas compatíveis com a tecnologia de Automação OLE[4][Bro95]. Uma outra restrição de COM é que só podem ser aplicadas chamadas dinâmicas a componentes que implementem explicitamente a interface `IDispatch`. Apesar de ser comum a implementação dessa interface pelos componentes, ela não é obrigatória de acordo com a arquitetura COM.

A interface `IDispatch` também pode ser utilizada para implementar novos componentes em tempo de execução. Para isto, é necessário prover um componente que ofereça a interface `IDispatch`, e possa receber uma requisição da operação `Invoke` e re-enviar a operação selecionada para a implementação real. Essa implementação real pode ser definida em tempo de execução através,

[4] A tecnologia de Automação OLE define um subconjunto de recursos de COM que estão disponíveis para ferramentas de script como o Visual Basic.

```
public final class Class extends Object implements Serializable {
   public static native Class forName(String className)
      throws ClassNotFoundException;
   public Field[] getFields() throws SecurityException;
   public Method[] getMethods() throws SecurityException;
   public Constructor[] getConstructors() throws SecurityException;
   public Field getField(String name)
      throws NoSuchFieldException, SecurityException;
   public Method getMethod(String name, Class parameterTypes[])
      throws NoSuchMethodException, SecurityException;
   public Constructor getConstructor(Class parameterTypes[])
      throws NoSuchMethodException, SecurityException;
   public native Object newInstance()
      throws InstantiationException, IllegalAccessException;
   public native boolean isInstance(Object obj);
   public native boolean isInterface();
   public native boolean isPrimitive();
   ...
}
```

Figura 2.15: A classe `Class` de Java.

por exemplo, de uma linguagem interpretada. Novamente, COM apresenta uma restrição ao uso de implementações dinâmicas, pois uma implementação dinâmica só pode ser acessada através da interface `IDispatch`, fazendo com que seus clientes tenham que ter conhecimento disto.

2.5.3 Reflexividade em Java

Java oferece suporte a meta-programação através de sua interface de programação denominada *Core Reflection API*. O ponto de partida da arquitetura reflexiva de Java é a classe `Class`, que descreve outras classes Java (figura 2.15). Essa classe permite a inspeção dos métodos, campos e construtores que pertencem a uma determinada classe ou interface (métodos `getMethods`, `getMethod`, `getFields`, `getField`, `getConstructors` e `getConstructor`). Dependendo da política de segurança que estiver sendo usada, é possível se ter acesso tanto aos membros públicos quanto aos privados da classe. Ao contrário de CORBA e COM, objetos Java sempre carregam suas próprias informações de tipo. Para se obter a classe de uma determinada instância de objeto, a classe básica `Object` implementa o método `getClass`:

```
public class Object {
   public final native Class getClass();
   ...
}
```

As informações sobre o tipo de um objeto só estão disponíveis para consulta, não podendo ser alteradas em tempo de execução. A partir de um objeto `Class`, também é possível se criar novas instâncias da classe que ele representa, através do método `newInstance` ou dos construtores obtidos a partir dos métodos `getConstructors` e `getConstructor`.

Para a construção de chamadas dinâmicas, o pacote reflexivo de Java oferece as classes `Method` e `Field`, cujas instâncias são obtidas exclusivamente através de um objeto `Class`. A classe `Method` (figura 2.16) permite a consulta à assinatura completa de um método, e a construção e invocação de

```
public final class Method extends Object implements Member {
   public Class getDeclaringClass();
   public String getName();
   public native int getModifiers();
   public Class getReturnType();
   public Class[] getParameterTypes();
   public Class[] getExceptionTypes();

   public native Object invoke(Object obj, Object args[])
      throws IllegalAccessException, IllegalArgumentException,
             InvocationTargetException;
   ...
}

public final class Field extends Object implements Member {
   public Class getDeclaringClass();
   public String getName();
   public native int getModifiers();
   public Class getType();
   public native Object get(Object obj)
      throws IllegalArgumentException, IllegalAccessException;
   public native void set(Object obj, Object value)
      throws IllegalArgumentException, IllegalAccessException;
   ...
}
```

Figura 2.16: As classes `Method` e `Field` de Java.

uma chamada dinâmica através do método `invoke`. A classe `Field` (figura 2.16) permite a consulta e atribuição aos campos de um objeto, através de suas operações `get` e `set`. Da mesma forma que em CORBA, é completamente transparente para um objeto se um de seus métodos ou atributos está sendo requisitado por uma chamada dinâmica ou estática.

Java não oferece propriamente uma interface para implementar novos objetos dinamicamente. Entretanto, Java oferece a classe abstrata `ClassLoader` para permitir que novas classes sejam carregadas em tempo de execução. Assim, é necessário que o código objeto da nova classe seja obtido de alguma forma em tempo de execução, e que ele seja carregado na máquina virtual de Java através de um novo objeto `ClassLoader`. Como será visto no capítulo 5, esse recurso permitiu que contornássemos a limitação de Java não prover uma interface específica para implementações dinâmicas.

2.6 Interoperabilidade entre Sistemas

Quando optamos por um dos sistemas de componentes vistos anteriormente para desenvolver um componente, de uma certa forma estamos restringindo as possibilidades de uso do componente a aplicações baseadas no mesmo sistema de componentes. Entretanto, existem várias soluções que permitem a interoperabilidade entre sistemas de componentes diferentes. Nas seções a seguir, apresentamos as principais soluções de interoperabilidade entre sistemas CORBA, COM e Java.

2.6.1 Integração de Java com CORBA e COM

O mecanismo mais básico de integração de Java com os sistemas CORBA e COM é através dos mapeamentos desses sistemas para essa linguagem, da mesma forma como é feito com outras linguagens. O padrão CORBA [OMG98] define um mapeamento de CORBA para Java que especifica como objetos CORBA devem ser representados em Java, permitindo que aplicações clientes de servidores CORBA sejam implementadas em Java, e como um servidor CORBA deve ser implementado em Java.

O mapeamento de COM para Java funciona de forma similar [OH97], apesar de não seguir um padrão definido pela indústria. O que ocorre é que a implementação de Java feita pela Microsoft, denominada Visual J++, oferece uma ponte de integração entre os dois sistemas. De uma maneira simplificada, o Visual J++ é composto por três elementos principais: uma implementação da máquina virtual de Java, um ambiente de desenvolvimento integrado (IDE), e uma ponte Java-COM.

Programas desenvolvidos com o Visual J++ podem acessar objetos COM como se estes fossem objetos nativos de Java. A máquina virtual lê a *type library* do objeto COM em tempo de execução, e disponibiliza internamente o objeto COM através de um *stub* Java gerado dinamicamente. A ponte nessa direção é transparente para o desenvolvedor, mas requer que o componente COM forneça uma *type library*.

Para um objeto Java ser disponibilizado como um objeto COM, uma *type library* deve ser criada. O Visual J++ é capaz de gerar essa *type library* automaticamente. Quando um objeto COM é implementado em Java, a JVM da Microsoft trata de forma transparente diversos detalhes de baixo nível da implementação de um objeto COM, tais como o controle do número de referências para o objeto. Essa solução de integração entre Java e COM é proprietária da Microsoft e não está disponível para outros ambientes de programação Java.

Além dos mapeamentos desses sistemas de componentes para Java, outras formas de interoperabilidade têm sido desenvolvidas. Como já comentamos anteriormente, o sistema JavaBeans tem como um de seus objetivos principais oferecer uma arquitetura de componentes independente de plataforma. Assim, a especificação do sistema JavaBeans procurou definir um conjunto de interfaces de programação que fossem facilmente mapeadas para outros sistemas de componentes de software, permitindo que fossem implementadas pontes entre JavaBeans e esses sistemas. Um exemplo de ponte de JavaBeans é a que permite o uso de um *bean* como um componente ActiveX (componente COM de interface com o usuário) [Jav00].

A integração de Java com CORBA também tem evoluido continuamente [OMG99c]. O primeiro passo dessa evolução foi a especificação de um mapeamento entre o protocolo do RMI e o IIOP. Esse mapeamento identificou o subconjunto dos dois protocolos que pode ser diretamente mapeado de um para o outro. O próximo passo foi estender o IIOP para oferecer suporte completo ao protocolo do RMI.

Como RMI é um mecanismo específico de Java, ele é fácil de ser usado para implementar clientes e servidores Java e permite a passagem de objetos por valor em chamadas remotas de métodos. Por outro lado, a independência de linguagem oferecida por CORBA restringia a passagem de objetos como parâmetros a ser feita somente por referência. A solução para esse problema envolveu a adição de um novo tipo de dado à OMG IDL, chamado `valuetype`, que oferece algum suporte à transferência de objetos por valor. A adição desse novo tipo de dado ao IIOP atendeu os requisitos do protocolo do RMI.

O último passo foi a definição de um mapeamento de Java para IDL, chamado *Java IDL*. Assim, um objeto definido e implementado originalmente em Java pode ser facilmente representado por uma interface equivalente em IDL, permitindo que tanto clientes CORBA quanto clientes Java possam acessar o objeto.

A especificação da arquitetura EJB [EJB99] ainda define que o IIOP é seu protocolo padrão para permitir uma maior interoperabilidade em ambientes distribuídos e heterogêneos. A recente

especificação do modelo de componentes de CORBA (*CORBA Component Model* – CCM) também define uma série de mecanismos de compatibilização com a arquitetura EJB [OMG99a].

2.6.2 Integração CORBA-COM

A principal solução de interoperabilidade entre CORBA e COM é a construção de *pontes* entre esses dois sistemas. Em [OMG98], a OMG especifica um padrão de arquitetura de integração entre esses dois sistemas, baseado no uso de pontes.

A arquitetura de integração proposta pela OMG define mapeamentos distintos entre CORBA e as interfaces regulares de COM e as de automação OLE. Esses mapeamentos identificam uma série de similaridades entre COM e CORBA, de forma a facilitar a criação da arquitetura de integração. A principal similaridade identificada é que os dois sistemas são baseados na noção de interfaces como sendo a única parte visível de um componente.

Além de identificar as similaridades entre os modelos de objetos dos dois sistemas, que podem ser mapeadas diretamente de um sistema para o outro, os mapeamentos definidos pela OMG dão soluções para os diversos aspectos incompatíveis entre os dois modelos, tais como o modelo de múltiplas interfaces de componentes COM, o mecanismo de herança múltipla de CORBA, as representações de exceções nos dois sistemas, e a representação dos tipos básicos não oferecidos por interfaces de automação OLE.

A arquitetura de integração define os conceitos de *views* e *targets*. De acordo com essa especificação, uma *view* é associada com um cliente (CORBA ou COM), enquanto o *target* é associado com o servidor (COM ou CORBA). Como a arquitetura de integração trata tanto das interfaces regulares de COM quanto das interfaces de automação OLE, os mapeamentos entre as seguintes *views* e *targets* são definidos:

- uma *view* com uma interface COM de um *target* CORBA;
- uma *view* CORBA de um *target* com uma interface COM;
- uma *view* com uma interface de automação OLE de um *target* CORBA;
- uma *view* CORBA de um *target* com uma interface de automação OLE.

Um par *view–target* define uma *ponte* entre os dois sistemas de componentes. Uma ponte faz com que um objeto de um modelo de objetos apareça para o desenvolvedor como sendo do outro modelo de objetos. Por exemplo, através de uma ponte, um desenvolvedor em Visual Basic pode usar um objeto servidor CORBA como se fosse um componente nativo de COM, com todos os recursos e características de um objeto COM.

O mapeamento usado para criar pontes entre CORBA e COM pode ser genérico ou específico para uma interface. O mapeamento genérico é feito através de um mapeamento realizado em tempo de execução, onde todas as adaptações necessárias são feitas dinamicamente. Já o mapeamento específico para uma interface é feito através da criação de uma ponte específica em tempo de compilação da aplicação, que é incluída na aplicação cliente. Aspectos como flexibilidade e desempenho devem ser considerados ao se escolher entre a abordagem de mapeamento genérico (dinâmico) e a abordagem de mapeamento específico para uma interface (estático). Enquanto mapeamentos dinâmicos oferecem uma maior flexibilidade, mapeamentos estáticos oferecem melhores resultados de desempenho.

Um detalhamento completo da arquitetura de integração entre CORBA e COM especificada pela OMG está além dos escopo deste trabalho. Mais informações sobre essa especificação e outras soluções de interoperabilidade entre CORBA e COM podem ser obtidas em [OMG98, RC98, Moo98, Pri99]. Entretanto, alguns dos conceitos apresentados nesta seção são de grande importância para o modelo de composição apresentado no capítulo 4, tais como a noção de um modelo de objetos com

conceitos comuns a diferentes sistemas, o mecanismo de pontes construídas a partir de objetos *view* e objetos *target*, e as noções de mapeamentos dinâmicos e estáticos.

2.7 Considerações Finais

Neste capítulo, não apresentamos todos os aspectos de CORBA, COM e JavaBeans. Descrições mais completas desses sistemas podem ser obtidas em [HV99, OMG98, Sie96, Bro95, Box98, Rog97, Kir97, AG98, GJS96, Jav97, Tho98].

Também apresentamos apenas os mecanismos reflexivos relevantes para este trabalho. Entretanto, outras características reflexivas estão presentes nesses sistemas de componentes. Um exemplo de outra característica reflexiva é o suporte à interceptação de chamadas de métodos. CORBA oferece esse recurso através do mecanismo de *Interceptadores* [OMG98], e uma nova proposta para substituir esse mecanismo está sendo avaliada pela OMG [OMG99e]. COM somente incluiu esse recurso à sua arquitetura recentemente, através de sua extensão COM+ [Kir97]. Além dessa extensão oficial, existem outras propostas de mecanismos de interceptação para COM [HS97, WL98, HS99]. Java e JavaBeans ainda não oferecem um mecanismo para interceptação de chamadas de métodos, mas existem algumas propostas de mecanismos desse tipo [LK98, OB99].

Como será visto no capítulo 4, apesar de todas as diferenças entre os modelos de objetos e os mecanismos reflexivos oferecidos pelos três sistemas de componentes analisados, foi possível definir um padrão de projeto (*design pattern*), baseado nesses modelos e mecanismos, para a implementação de um *binding* dinâmico entre linguagens interpretadas e esses sistemas de componentes.

As soluções de interoperabilidade entre sistemas de componentes apresentadas neste capítulo tratam desse problema caso a caso. Nenhuma das soluções vistas anteriormente define um modelo mais geral, que possa ser aplicado a um conjunto mais amplo de sistemas de componentes. O modelo de composição que propomos no capítulo 4 procura ser abstrato e flexível o suficiente para que possa ser aplicado a qualquer sistema de componentes com um determinado conjunto de características.

Capítulo 3

Composição Dinâmica

No capítulo 2, descrevemos as características dos sistemas de componentes e os recursos oferecidos por estes. Neste capítulo discutimos algumas das principais técnicas utilizadas para realizar a *composição* de componentes. Como estamos particularmente interessados no suporte à composição dinâmica, analisamos os principais aspectos e dificuldades que os mecanismos de composição devem tratar para oferecer esse tipo de suporte.

Os mecanismos de composição são responsáveis por fazer a ligação efetiva entre componentes de software. Freqüentemente, a definição dessas ligações está embutida na implementação do próprio componente. Esse alto grau de acoplamento entre um componente e suas conexões dificulta em muito a reconfiguração de uma aplicação, tanto estática quanto dinamicamente. Entretanto, existem vários mecanismos que promovem o desacoplamento entre componentes e suas conexões. Além de facilitar o entendimento e a alteração da composição de uma aplicação, esse desacoplamento também aumenta as possibilidades de reuso dos componentes, permitindo que estes possam participar de diferentes configurações de interação.

Para permitir adaptações dinâmicas na composição de aplicações, os mecanismos de composição precisam tratar algumas dificuldades adicionais. As seções a seguir tratam exatamente dessas dificuldades e dos principais mecanismos de composição que permitem o desacoplamento da implementação dos componentes. Inicialmente discutimos alguns conceitos básicos referentes a conexões entre componentes. Depois analisamos uma série de aspectos relacionados com a adaptação dinâmica de sistemas baseados em componentes de software. Uma vez definido o conjunto de aspectos relacionados à composição de aplicações que estamos interessados, apresentamos um dos principais mecanismos de composição adotados atualmente [OMG96, OMG97a, OMG97b, Ous98, Gat98]. Esse mecanismo, que é baseado na utilização de linguagens de *script*, oferece diversos recursos para o apoio à adaptação dinâmica de aplicações. Depois apresentamos resumidamente mais alguns mecanismos de composição que consideramos relevantes. Finalmente, traçamos algumas comparações entre as diferentes abordagens, dando destaque às suas características referentes à expressividade e à composição em tempo de execução.

3.1 Conceitos Básicos

Relembrando a definição apresentada no capítulo 1, consideramos um componente de software como sendo uma unidade binária desenvolvida isoladamente para uma finalidade específica, e que deve ter sido projetada para colaborar com outros componentes a fim de formar aplicações finais. Essas unidades binárias encapsulam seu estado e comportamento, e interagem entre si através apenas de interfaces (conjuntos de operações) bem definidas.

```
interface ServicoA {
   double soma (in double x, in double y);
};

interface ServicoB {
   attribute ServicoA a;
   double produto (in double x, in double y);
};

interface ServicoC {
   void setA(in ServicoA a);
   ServicoA getA();
   double produto (in double x, in double y);
};
```

Figura 3.1: Exemplos de conexões entre componentes através de atributos da interface.

As interfaces de um componente definem suas *fronteiras*. Essas fronteiras podem ser classificadas em quatro categorias [WUK99]:

Serviços oferecidos — São as interfaces pelas quais um componente oferece seus serviços para seus clientes.

Serviços requisitados — São as interfaces pelas quais um componente tem acesso a outros serviços necessários para seu processamento interno.

Eventos gerados — São as interfaces pelas quais um componente sinaliza para outros componentes mudanças em seu estado interno ou a ocorrência de algum outro evento específico.

Eventos observados — São as interfaces pelas quais um componente é notificado da ocorrência de eventos de seu interesse.

Essa classificação pode ser observada nos sistemas COM e JavaBeans. A recente especificação para o modelo de componentes de CORBA [OMG99a] também adotou essa diferenciação para as interfaces de um componente CORBA.

Como foi visto no capítulo 2, os sistemas de componentes oferecem algum mecanismo para descrever suas interfaces. Em CORBA e COM, essas interfaces são descritas através de IDLs; enquanto que em JavaBeans, elas são descritas por interfaces e classes da própria linguagem Java.

As conexões entre componentes podem ser representadas por diferentes tipos de abstrações. A forma mais básica para representar conexões é através de atributos do componente que referenciam diretamente os componentes a ele conectados. A figura 3.1 mostra um exemplo com dois estilos típicos para representar esse tipo de conexão em OMG IDL. Nesse exemplo, as interfaces `ServicoB` e `ServicoC` são oferecidas por serviços que vão utilizar internamente um serviço do tipo `ServicoA`. `ServicoB` representa essa conexão através de um atributo explícito (atributo `a`), que pode ter seu valor alterado em tempo de execução. Já `ServicoC` oferece um par de métodos de acesso a esse atributo interno (par *get/set*). Caso não se queira permitir que o valor de um atributo seja alterado por uma manipulação externa ao componente, basta definir o atributo como sendo somente de leitura (`readonly`), no caso da interface `ServicoB`, ou não oferecer o método *set* correspondente ao atributo, no caso da interface `ServicoC`. JavaBeans chega a adotar um padrão de formação de nomes para métodos e classes, de forma que um ambiente de desenvolvimento possa extrair automaticamente as relações de dependência entre componentes, a partir da simples análise desses nomes.

Um outro modelo também comum para a representação de conexões adota os conceitos de *portas* e *conectores* [KMN89, MDK94, MDEK95, SG96a, BCK98, WUK99]. Uma porta representa um ponto de conexão de um componente, e pode ser de *entrada* (serviços oferecidos e eventos observados) ou de *saída* (serviços requisitados e eventos sinalizados). As ligações entre portas são realizadas por uma entidade específica para esse fim: os *conectores*. Um *conector* é tipicamente representado por um componente com interfaces específicas para a ligação entre portas. Conectores podem apresentar padrões de comportamento bem sofisticados, tais como *multicasting*, armazenamento e persistência de mensagens. Esse modelo é claramente inspirado na arquitetura de circuitos eletrônicos.

Os conceitos de portas e conectores estão parcialmente presentes nas arquiteturas de CORBA, COM e JavaBeans. Em COM, uma porta de saída é especificamente representada pela interface `IConnectionPoint` [Bro95, Box98]. De forma similar, portas de entrada e saída podem ser representadas através do mecanismo de *listeners* de JavaBeans. Os serviços de Eventos e de Notificação de CORBA [OMG97c, OMG99d] e o de filas de mensagens de COM [Kir99] podem ser vistos como conectores de componentes.

Essa forma mais estruturada para representar conexões entre componentes tem servido de base para o estudo de *arquiteturas de software* [SG96a, PMS96, BCK98, PCS98]. Há uma forte tendência para descrever arquiteturas de software como configurações de componentes e conectores, proporcionando uma visão mais abstrata da estrutura de uma aplicação. Essas descrições podem ser aplicadas de forma a criar vários níveis de abstração, isto é, uma configuração de componentes e conectores pode ser usada como um componente de um outro sistema.

Entretanto, para compor novas aplicações, não basta apenas conectar as diferentes interfaces dos componentes. Um mecanismo de composição de aplicações ainda precisa tratar dos seguintes aspectos:

Configuração — Tipicamente, um componente apresenta uma série de atributos que podem ser configurados para ajustar o seu comportamento a um determinado contexto ou aplicação. Essa tarefa também é conhecida como *customização* do componente.

Adaptação — Muitas vezes é necessário adaptar uma interface de um componente para que este possa ser usado em conjunto com outros componentes que esperam por uma outra interface. Essa tarefa também é conhecida como *gluing* entre componentes.

Dessa forma, um mecanismo de composição deve oferecer algum meio que permita a realização das tarefas de conexão, configuração e adaptação de componentes. Geralmente, um mecanismo de composição realiza essas tarefas utilizando uma linguagem de programação como elemento de composição. Essa linguagem de programação pode ser a mesma usada na implementação dos componentes, ou alguma outra mais específica para esse fim.

Para a implementação de um componente, o desenvolvedor tipicamente tem sua atenção voltada para a implementação de soluções eficientes e de estruturas de dados complexas. Já para a tarefa de composição, o foco está na configuração de componentes e na implementação de diferentes e complexos esquemas de interação entre objetos. Assim, o uso de linguagens com enfoques diferentes para implementar e compor componentes pode ser uma solução eficaz para atender às diferenças de requisitos entre esses dois tipos de tarefa. Além disso, sistemas de componentes como CORBA e COM, que permitem que componentes sejam implementados por diferentes linguagens de programação, naturalmente induzem ao uso de linguagens diferentes para composição e implementação.

Uma linguagem de composição basicamente deve oferecer facilidades para

- a customização de componentes;
- a implementação de adaptadores entre interfaces incompatíveis (*gluing*);

- a implementação de roteiros (*scripts*) de execução de componentes, de acordo com os seus diferentes protocolos;
- a coordenação (sincronização) da execução concorrente de componentes, tipicamente em um ambiente distribuído;
- a criação de novos componentes a partir da combinação de outros (componentes compostos);
- e a conexão entre as interfaces dos componentes, de modo a definir a arquitetura de uma aplicação.

Essas facilidades podem ser usadas para orientar a definição do poder computacional de uma linguagem de composição, assim como para facilitar uma análise comparativa entre diferentes mecanismos de composição [NM94, SN99].

Para atender a aplicações que devam permitir que sua composição seja alterada dinamicamente, uma ferramenta de composição ainda deve oferecer algum mecanismo interativo através do qual um administrador do sistema, ou até mesmo um usuário final, possa realizar alterações na composição da aplicação. Também é importante que a ferramenta ofereça ao usuário uma maneira simples de expressar suas intenções. Para viabilizar esse suporte dinâmico, a linguagem de composição deve ter acesso em tempo de execução às descrições das interfaces dos componentes e suas inter-dependências.

3.2 Adaptação Dinâmica

Quando falamos de adaptação dinâmica de um sistema baseado em componentes de software, podemos estar tratando de dois tipos de adaptação. O primeiro tipo consiste da substituição de componentes, sem alterar a arquitetura do sistema, que é composta pelas interfaces dos componentes e seus relacionamentos. Tipicamente, esse tipo de adaptação está relacionado com requisitos não-funcionais do sistema. Um componente pode não estar mais atendendo a um determinado requisito porque seu comportamento degradou de alguma forma, ou porque o requisito mudou em tempo de execução. Alguns exemplos de requisitos não-funcionais que podem exigir a adaptação dinâmica de um sistema são desempenho, segurança, e mudanças no ambiente operacional. Uma eventual necessidade de corrigir um erro na implementação de um determinado componente (um *bug*) também pode exigir uma adaptação desse tipo. Tipicamente, esse tipo de adaptação só afeta aspectos dinâmicos da estrutura de um sistema.

O segundo tipo de adaptação envolve mudanças na arquitetura do sistema, seja por alterações nas interfaces que já estão sendo utilizadas na arquitetura, ou seja pela adição de novas interfaces de componentes à arquitetura do sistema. Adaptações desse tipo geralmente são motivadas pela necessidade de adicionar novas funcionalidades a um sistema, o que pode acarretar a alteração de alguns aspectos "estáticos" da estrutura do sistema.

Os mecanismos de suporte para adaptação dinâmica mais comumente encontrados em sistemas baseados em componentes atendem ao primeiro tipo de adaptação. Os sistemas estabelecem um conjunto fixo de interfaces para os componentes que podem ser utilizados, e as possíveis adaptações dinâmicas envolvem apenas a substituição ou o acréscimo de componentes com interfaces conhecidas previamente.

À primeira vista, uma adaptação desse tipo pode parecer bem simples. Entretanto, ela envolve alguns problemas complexos. Primeiro, é necessário manter todos os relacionamentos dinâmicos entre os componentes para que, ao se substituir um determinado componente, seja possível redirecionar para o novo componente todos os componentes que eram clientes do antigo. Da mesma forma, é necessário remover os componentes que só são utilizados pelo componente que está sendo substituído, e carregar

os componentes requisitados pelo novo componente. Essa reconfiguração precisa ser feita de uma forma atômica para não deixar o sistema em um estado inconsistente durante a substituição.

Um outro problema na substituição de componentes é garantir que ela será feita em um momento apropriado, não comprometendo nenhum processamento em andamento no sistema. De um modo geral, para oferecer essa garantia, o componente sendo substituido não deve estar no meio de um processamento. Em muitos casos, uma substituição também envolve uma transferência de estado entre o componente original e seu substituto.

Geralmente, esses problemas são tratados pela própria aplicação suscetível à adaptação, através de mecanismos explicitamente definidos em sua arquitetura. Alguns trabalhos mais recentes procuram incluir o suporte para resolver esses problemas na infra-estrutura oferecida pelo sistema de componentes [KRL+00, KC00].

O suporte à adaptação dinâmica que envolva a inclusão de novas interfaces à arquitetura de um sistema já é mais difícil de ser encontrado. Isso se deve à necessidade de alterar a estrutura do sistema de tal forma que se possa usar efetivamente a nova interface. Uma nova interface provavelmente representa uma nova funcionalidade, que não foi prevista na concepção original do sistema.

Os meta-mecanismos disponíveis nos sistemas de componentes permitem que novos tipos de componentes sejam descobertos em tempo de execução, e que chamadas de métodos sejam criadas dinamicamente. Entretanto, eles não oferecem nenhum mecanismo que facilite a integração efetiva de novos tipos de componentes às aplicações já existentes.

Apesar desse recurso de adaptação dificilmente estar disponível para um sistema em operação, ele é o princípio básico de ambientes de suporte à prototipação de sistemas [CIS92, DMC92, SU95]. De uma maneira simplificada, esses ambientes oferecem mecanismos para que o desenvolvedor possa alterar dinamicamente as funcionalidades que o protótipo de um sistema possui. Facilitando a realização de alterações desse tipo, esses ambientes permitem que um desenvolvedor possa experimentar e testar diversas opções de projeto em um curto espaço de tempo.

Dependendo da flexibilidade desejada, uma linguagem de composição pode ser usada de diferentes formas para permitir que novas interfaces sejam incluídas em um sistema. A primeira forma é através de um console de comandos interativos, que funciona como uma interface genérica de acesso a componentes. Esse tipo de solução pode ser apropriado para situações em que o uso do novo componente só será feito esporadicamente, ou para ferramentas de administração de sistemas.

Uma outra possibilidade é utilizar o mesmo console de comandos para reconfigurar a interface com o usuário de uma aplicação. Para isso, a interface com o usuário também precisa ser reconfigurável em tempo de execução. Dessa forma, é possível adicionar à interface da aplicação os pontos de acesso à nova funcionalidade.

Uma possibilidade mais restrita seria usar uma linguagem de composição para implementar adaptadores entre a nova interface e uma interface já esperada pelo sistema. Somente esses adaptadores poderiam ter acesso à alguma funcionalidade extra oferecida pelo novo componente. Esse tipo de solução é apropriado para fazer a adaptação entre interfaces funcionalmente compatíveis, mas que apresentam diferenças na sintaxe de suas operações ou no protocolo (seqüência de chamadas) esperado pelos componentes envolvidos. Um adaptador ainda poderia suprir alguma funcionalidade que o novo componente não ofereça. Esse tipo particular de solução é denominado *adaptação dinâmica de componentes* [Kni99].

3.3 Adaptação Antecipada e Não-Antecipada

De acordo com a seção anterior, quanto mais cedo for possível prever uma adaptação de um sistema, mais facilmente essa adaptação pode ser realizada. Entretanto, permitir que um sistema possa evoluir

em direções não previstas originalmente oferece uma grande flexibilidade ao sistema.

A discussão sobre evoluções *antecipadas* e *não-antecipadas* já estava presente em trabalhos que procuraram identificar as raízes das diferenças entre linguagens orientadas a objetos baseadas em classes e herança, e linguagens OO baseadas em protótipos e delegação [SLU88]. Segundo esses trabalhos, a grande diferença entre a abordagem baseada em classes e a baseada em protótipos está em quando é definido o compartilhamento (*sharing*) de comportamento e estado entre objetos ou classes de objetos.

A forma de compartilhamento mais tradicional em linguagens orientadas a objetos é o compartilhamento *antecipado*. Durante a fase de projeto de um sistema, o projetista pode freqüentemente prever aspectos comuns a diferentes partes do sistema, levando ao compartilhamento de procedimentos e dados entre as partes similares. Esse tipo de compartilhamento é melhor capturado por mecanismos da linguagem de programação que ofereçam meios ao desenvolvedor para definir, antecipadamente, a estrutura a ser compartilhada por diferentes componentes do sistema. Em linguagens OO tradicionais — como Simula, C++ e Java — classes e herança são os mecanismos para representar o compartilhamento antecipado de comportamento entre objetos. Essas classes podem ser utilizadas por um número não antecipado de instâncias de objetos.

Por outro lado, o compartilhamento *não-antecipado* não é bem representado através dos mecanismos tradicionais de herança de classes. A necessidade de compartilhamento não-antecipado pode surgir quando um desenvolvedor precisa introduzir um novo recurso em um sistema, que talvez não pudesse ser previsto quando o sistema original foi desenvolvido. Suponha que esse desenvolvedor venha a observar que o novo recurso pode ser atendido parcialmente através de componentes já disponíveis no sistema, e o restante da funcionalidade necessária para oferecer o novo recurso poderia ser obtido através da introdução no sistema de mais alguns procedimentos e dados. Dessa forma, surge uma relação de compartilhamento com os componentes que são utilizados tanto para os seus propósitos originais (antecipados), quanto para os novos propósitos (não-antecipados). Obrigar que os relacionamentos de compartilhamento sejam definidos a priori impõe restrições aos tipos de extensões que podem ser introduzidas sem modificar o sistema original.

Uma linguagem de programação oferece um melhor suporte ao compartilhamento não-antecipado se o novo comportamento puder ser introduzido através da simples definição de quais são as diferenças entre o novo comportamento desejado e o já existente no sistema. Delegação é apontada como um mecanismo adequado para esse tipo de suporte, pois permite que novos objetos reusem o comportamento de objetos já existentes, sem requerer a especificação estática desse relacionamento [SLU88, DMC92, Kni99].

A análise das alternativas de mecanismos de compartilhamento deve ser feita considerando a necessidade de adaptações futuras no sistema para oferecer extensões em seu comportamento. Sob esse ponto de vista, toda a análise sobre compartilhamento antecipado e não-antecipado pode ser utilizada para avaliar as diferentes abordagens para adaptação de sistemas e componentes.

Generalizando a noção de antecipação e não-antecipação, podemos adotá-la em relação a outros aspectos de uma linguagem de programação. Por exemplo, essa noção pode ser usada para analisar o sistema de tipos de uma linguagem OO. Como veremos na seção 4.1, o modelo de objetos que propomos neste trabalho é fortemente baseado na não antecipação de decisões a respeito de uma série de aspectos dos objetos que vão compor um sistema.

3.4 Linguagens de *Script*

As linguagens de *script* são projetadas para compor e configurar aplicações, e são apontadas como sendo um dos principais tipos de ferramentas utilizadas para realizar essas duas tarefas [OMG96, OMG97a, OMG97b, Ous98, Gat98, SN98, SN99, Sch99b]. Tipicamente, elas trabalham embutidas

em alguma aplicação hospedeira, que disponibiliza um conjunto de funcionalidades para o ambiente da linguagem. Através de uma linguagem de *script* é possível configurar o comportamento de uma aplicação, definindo *roteiros* de execução. Esses roteiros definem como os componentes da aplicação devem colaborar para atingir um determinado objetivo. Alguns dos usos mais comuns para linguagens de *script* são [SN98]:

- automação de tarefas administrativas (ex.: automação de *backup*);
- automação de testes de componentes e aplicações;
- processamento de texto (extração e formatação de dados);
- desenvolvimento de aplicações para Web, tanto do lado servidor (programação CGI) quanto do lado cliente (JavaScript e VBScript);
- prototipação rápida de aplicações;
- desenvolvimento de aplicações baseadas em componentes.

Os dois últimos usos são os de maior interesse para esta tese. Na próxima seção, descrevemos as principais características das linguagens de *script*, para depois analisarmos sua integração com os sistemas de componentes.

3.4.1 Principais Características das Linguagens de *Script*

Linguagens de *script* oferecem um modelo de programação de mais alto nível do que linguagens como C, C++ e Java. Isto é, elas oferecem mecanismos de programação que tratam de diversos detalhes de implementação automaticamente, permitindo que programadores possam escrever menos código para obter a mesma funcionalidade. Essas linguagens privilegiam a velocidade de desenvolvimento em detrimento da eficiência em tempo de execução [Ous98].

Geralmente, linguagens de *script* são classificadas como sendo *interpretadas*. A definição do que é uma *linguagem interpretada* não é muito precisa. Isto ocorre pelo menos por duas razões: uma mesma linguagem pode ser tanto interpretada quanto compilada, e muitas implementações de linguagens usam uma combinação das duas técnicas. Entretanto, a expressão *linguagem interpretada* muitas vezes é útil para descrever uma classe de linguagens com características distintas das chamadas *linguagens convencionais*, tais como C ou FORTRAN. Assim, para sermos mais precisos, adotaremos a seguinte definição:

> Uma linguagem será classificada como *interpretada* se oferecer algum mecanismo para a execução de trechos de código criados dinamicamente, no mesmo ambiente do programa sendo executado; em outras palavras, se o interpretador de código fonte for diretamente acessível pela linguagem.

De acordo com essa definição, linguagens como Lisp, Lua, Mumps e TCL são interpretadas, enquanto Pascal, C, C++ e Java não são.

Algumas características importantes de linguagens interpretadas, tais como interatividade, programação pelo usuário final e reflexividade, derivam diretamente da definição acima. Outras propriedades, como tipagem dinâmica, ausência de declarações e gerência automática de memória, não são conseqüências diretas dessa definição, mas facilitam o projeto de uma linguagem interpretada.

Outra característica importante de linguagens interpretadas é que não existe a necessidade de recompilação e ligação após cada alteração em um programa. Essa característica é útil tanto no desenvolvimento de pequenos programas (principalmente para programação pelo usuário final) quanto na

prototipação rápida de aplicações, pois permite que diversas decisões de projeto sejam implementadas e testadas em pequenos períodos de tempo. Essa característica é fundamental para a interatividade oferecida por uma linguagem interpretada, pois possibilita que um operador tenha acesso direto a todos os recursos disponíveis em um sistema através de um *console interativo*, sem a necessidade de ferramentas específicas para o acesso a cada recurso. Durante a prototipação e desenvolvimento de um sistema, um console interativo permite que novos componentes sejam adicionados e testados sem que seja necessário o desenvolvimento de módulos específicos para teste.

Além das características herdadas de linguagens interpretadas, as linguagens de *script* apresentam outras características típicas de sua categoria. Por exemplo, muitas vezes elas trabalham embutidas em algum sistema, e oferecem uma série de mecanismos de extensibilidade. Através desses mecanismos, é possível aumentar os conjuntos de funcionalidades e tipos de dados disponibilizados através da linguagem, de forma a adaptar a linguagem a novos domínios de aplicação.

Ferramentas para integração com software legado são bem comuns para linguagens de *script*. Essas ferramentas facilitam o processo de extensão da linguagem, gerando automaticamente adaptadores entre a linguagem e o software legado. Exemplos de ferramentas desse tipo podem ser vistos em [Bea98] e [Cel97].

Freqüentemente, linguagens de *script* não são consideradas seguras, devido ao uso de tipagem dinâmica. Entretanto, elas podem ser tão seguras quanto linguagens com tipagem estática. A grande diferença é que a verificação de tipos é postergada até o último momento. Caso ocorra algum erro de tipos em tempo de execução, ele pode ser tratado pelos mecanismos de tratamento de exceções da linguagem. Linguagens com tipagem estática fazem essa verificação em tempo de compilação, evitando os custos de realizar essa verificação em tempo de execução. Enquanto a tipagem estática gera programas mais eficientes e menos flexíveis, a tipagem dinâmica exige um maior esforço do desenvolvedor para prover os tratamentos de exceções necessários.

Como linguagens de *script* são tipicamente usadas para pequenas tarefas de processamento, o tempo total de execução da aplicação tende a ser dominado pelo tempo na execução dos componentes, que geralmente são implementados em uma linguagem compilada.

Resumidamente, podemos descrever as linguagens de *script* como tendo as seguintes características:

- são interpretadas (oferecem uma primitiva similar à função *eval* de Lisp);
- dão apoio à programação pelo usuário final (devem ser fáceis de se aprender);
- dão apoio à prototipação;
- são completas computacionalmente;
- oferecem abstrações de dados de alto nível;
- usam tipagem dinâmica;
- não necessitam de declarações de tipos e variáveis;
- oferecem mecanismos de reflexividade;
- oferecem gerência automática de memória (coleta de lixo);
- são extensíveis;
- trabalham embutidas em algum sistema;

- podem ser especializadas em algum domínio de aplicação, tais como Web, processamento de texto e administração de sistemas (*linguagens específicas de domínio*).

Entretanto, nem todas as linguagens de *script* apresentam todas essas características. Dentre as principais linguagens de *script*, podemos citar Bourne Shell [KP84], Tcl [Ous94], Visual Basic [Cla96], Lua [IFC96], Python [Lut96], Perl [WS91], e JavaScript [Fla97, ECM97].

3.4.2 Integração com Sistemas de Componentes

As linguagens de *script* têm desempenhado um papel muito importante na consolidação dos sistemas de componentes de software. Por exemplo, Visual Basic foi um dos principais responsáveis pela popularização do uso de COM [Ude94].

Através de um ambiente de programação híbrido — uma parte da programação é visual e outra é em Basic — Visual Basic permitiu que a programação para o ambiente Windows chegasse a programadores não especializados. Nessa ferramenta, sofisticados controles de interface com o usuário são disponibilizados através de componentes COM [TT95]. O aspecto visual da interface da aplicação com o usuário é desenvolvido graficamente, por manipulação direta dos componentes visuais. Já o comportamento da aplicação é descrito através de *scripts* em Basic.

No ambiente de programação do Visual Basic, o conjunto de componentes disponíveis para o desenvolvedor pode ser estendido. Através de seu ambiente, pode-se ter acesso a todos os componentes COM disponíveis tanto no sistema Windows local quanto em uma rede local.

Visual Basic e suas extensões têm um papel fundamental na estratégia de desenvolvimento de aplicações proposta pela Microsoft [Gat98]. Através de sua extensão conhecida como *Visual Basic for Applications* (VBA), Visual Basic pode ser embutido em outras aplicações para desempenhar o papel de uma linguagem de extensão e configuração de aplicações. Atualmente, essa linguagem também já pode ser usada como linguagem de *script* tanto para navegadores WWW (VBScript), quanto para a construção dinâmica de páginas HTML (*Active Server Pages* – ASP).

Depois de Visual Basic, várias outras ferramentas de desenvolvimento para ambiente Windows seguiram o seu modelo, como foi o caso de Delphi [Bor95]. Existem também algumas ferramentas experimentais de *script* para COM que procuram explorar aspectos diferenciais oferecidos pelas linguagens adotadas, como são os casos dos *bindings* para COM das linguagens Haskell [JML98], BETA [Mad99] e Lua (ver seção 5.2).

Para os outros sistemas de componentes, o uso de linguagens de *script* ainda é bem modesto. Entretanto, existem vários trabalhos tentando reverter essa situação. Basicamente, o que se tem feito é a implementação de *bindings* entre os sistemas de componentes com linguagens de *script* já estabelecidas. Este é o caso de trabalhos de integração entre

- Tcl e CORBA [ASG+94, Pil00],
- Tcl e Java [Joh98, Sta98],
- Python e CORBA [Chi99, JSLJ99, OMG00],
- Python e Java [CSW97, vR98, Ang99],
- Lua e CORBA [CRI97, CIR97, ICR98],
- Lua e Java [CI99, CIR99a].

Existem também algumas poucas linguagens que foram desenvolvidas especialmente para determinados sistemas, tais como CorbaScript para sistemas CORBA [MGG97], e Pnuts para Java [McC00].

Para a implementação de ferramentas de *script* para Java, duas abordagens têm sido adotadas. A primeira é baseada na interface de programação com o sistema nativo de Java (JNI – *Java Native Interface*), onde é implementado um módulo de extensão para a linguagem de *script* que permite que esta se comunique com o ambiente de Java, como ocorre em TclBlend [Sta98], JPI [CSW97] e LuaJava [CI99, CIR99a]. A outra solução envolve a implementação do interpretador da linguagem de *script* em Java. Essa solução foi adotada em JPython [vR98, Ang99], Jacl [Joh98] e Pnuts [McC00].

Para os sistemas CORBA, a OMG, que é a entidade responsável pela especificação de CORBA, definiu um padrão para *script* de componentes [OMG99b]. Esse padrão não propõe uma linguagem específica, mas define um modelo geral para os mapeamentos entre linguagens de *script* e CORBA. Isto é, a partir desse modelo, podem ser definidos mapeamentos específicos de diversas linguagens de *script* para CORBA.

Esse padrão foi proposto como uma complementação para a especificação do Modelo de Componentes CORBA [OMG97a, OMG97b, OMG99a, OMG99b]. Algumas ferramentas de *script* para CORBA já seguem esse padrão, como é o caso de CorbaScript [MGG97], LuaOrb [CIR97, CRI97, ICR98] e Combat [Pil00]. Essas ferramentas são implementações de mapeamentos de suas respectivas linguagens de programação para CORBA.

A grande diferença entre os mapeamentos de linguagens de *script* para CORBA e os de linguagens como C++, Java e C, é que estes são baseados em declarações estáticas de classes, que estão disponíveis antes da compilação da aplicação, enquanto aqueles são tipicamente baseados em tipagem dinâmica e classes ou protótipos de objetos gerados em tempo de execução.

3.5 Outros Mecanismos de Composição

Há uma forte tendência de se definir novos mecanismos de composição que ofereçam um controle mais rígido sobre a realização de conexões entre componentes. Esta seção trata exatamente de alguns desses mecanismos.

3.5.1 Linguagens de Descrição de Arquiteturas

Com a finalidade de descrever precisamente arquiteturas de software e estilos arquitetônicos[1], diversas linguagens têm sido definidas. Essas linguagens são denominadas *Linguagens de Descrição de Arquiteturas* (ADLs – *Architecture Description Languages*). Schneider [Sch99b] dá a seguinte definição para ADLs:

> Uma ADL é uma notação que permite a descrição precisa e a análise das propriedades externamente visíveis de uma arquitetura de software, oferecendo diferentes estilos arquitetônicos em diferentes níveis de abstração.

Uma ADL deve oferecer um vocabulário que possa ser facilmente compreendido por arquitetos de software. Tipicamente, as abstrações de componentes e conectores são diretamente oferecidas por uma ADL. Essas linguagens definem uma semântica precisa para componentes e conectores, resolvendo ambigüidades e auxiliando na detecção de inconsistências em uma arquitetura. Uma ADL também deve oferecer um conjunto de técnicas que facilite a análise e a verificação de estilos arquitetônicos e de propriedades específicas de um sistema [Sch99b].

Analisando os aspectos lingüísticos de ADLs, Shaw e Garlan [SG96a] elaboraram os seguintes conceitos que uma ADL deve oferecer:

[1] Um estilo arquitetônico é uma abstração sobre um conjunto de arquiteturas de software relacionadas. Um estilo define um vocabulário comum de tipos de componentes e conectores, e um conjunto de regras de como componentes e conectores podem ser combinados.

abstração — os componentes de um sistema e suas interações são descritos através de abstrações explícitas e claras.

composição — um sistema é descrito como uma composição de componentes e conectores independentes;

reutilização — componentes, conectores e padrões de arquitetura podem ser reutilizados em diferentes descrições de arquiteturas;

configuração — descrições de sistemas podem ser ajustadas para diferentes situações e sistemas podem ser reconfigurados dinamicamente;

heterogeneidade — múltiplas e heterogêneas descrições de arquiteturas podem ser combinadas;

análise — descrições de arquiteturas são passíveis de análises e verificações.

Schneider [Sch99b] identifica mais algumas características desejáveis para uma ADL, tais como o suporte direto aos estilos arquitetônicos mais comuns, e a possibilidade de definir novos estilos arquitetônicos a partir da adição de novos tipos de componentes, conectores e regras de composição.

Existe uma grande variedade de ADLs surgindo de grupos de pesquisa tanto da indústria quanto da academia. Uma comparação detalhada dessas linguagens está além do escopo deste trabalho. Mais detalhes sobre esse tópico e exemplos de ADLs podem ser obtidos em [KMN89, PS93, MDK94, MDEK95, SG96a, SDZ96, BF96, Tho96, IB96, WS96, MT97, BCK98, PCS98, PA98].

Dada a natureza das abstrações tipicamente presentes em uma ADL, diversas ferramentas gráficas têm sido desenvolvidas para dar apoio à descrição de arquiteturas. Essas ferramentas basicamente substituem as descrições textuais de uma ADL por representações gráficas equivalentes. Assim, arquiteturas são definidas através de diagramas que especificam as conexões entre componentes.

Apesar dessas ferramentas poderem desempenhar o papel de uma ADL na definição das ligações entre componentes e conectores, elas tipicamente são dependentes de uma linguagem textual para descrever os componentes e conectores existentes. Assim, essas ferramentas gráficas funcionam como *front-ends* para ADLs. Exemplos dessas ferramentas são os ambientes gráficos para as linguagens Conic e Darwin [KMN89, KMNS93, NKMD95, NK95, NKMD96, FS96, NKM96], e o ambiente de composição visual Vista [dM95].

O sistema JavaBeans também sugere o uso de ferramentas de composição visual [Jav97]. Apesar dessas ferramentas para JavaBeans não estarem associadas a uma ADL propriamente dita, seu paradigma de composição é muito similar ao utilizado pelas ferramentas gráficas baseadas em ADLs. Neste caso, as informações sobre os pontos de conexão dos componentes são obtidas através das interfaces definidas em Java.

3.5.2 Composição Semântica

Assim como diversas ADLs, outros mecanismos de composição adotam abordagens que advogam que a compatibilidade entre componentes definida exclusivamente com base no casamento sintático das assinaturas de interfaces não é suficiente. Assim, esses mecanismos procuram adicionar informações da semântica dos componentes em suas interfaces. Esse é o caso dos mecanismos baseados em interfaces declarativas, que permitem que as interfaces dos componentes sejam estendidas com atributos que representam, pelo menos parcialmente, a semântica dos componentes. Singh e Gisi [SG96b] usam essa abordagem em seu sistema baseado nas linguagens ACL, KIF e KQML.

Uma outra abordagem é baseada na definição de *protocolos*. De uma maneira simplificada, um protocolo especifica a seqüência na qual as operações de um conjunto componentes devem ser chamadas para que estes funcionem da maneira esperada. De uma certa forma, esses protocolos podem ser vistos como *scripts*. Exemplos dessa abordagem são os trabalhos [YS94, YS97, Cho98].

A técnica de projeto por *contratos* também tem sido proposta como um mecanismo para orientar a conexão de componentes [Mey88, Mey92, BJPW99, CR99]. Basicamente, essa técnica acrescenta um conjunto de pré e pós-condições para as operações oferecidas pelas interfaces dos componentes. Essas condições são usadas como um contrato estabelecido entre componentes cliente e servidor.

Oscar Nierstrasz também tem desenvolvido um trabalho importante na área de linguagens de composição. Ele tem procurado definir, através de modelos formais, as características necessárias para uma linguagem de composição. A partir desses modelos formais, é possível analisar e verificar uma série de propriedades de uma configuração de componentes. Ao longo de seu trabalho, Nierstrasz definiu algumas linguagens de composição, tais como OC [Nie92, Nie93], Pict [LSN96, NSL96] e Piccola [SN99, Sch99b, ALSN00, AN00, LAN00]. Piccola modela componentes e abstrações de composição através de um modelo formal unificador para a comunicação entre agentes concorrentes. De acordo com Nierstrasz, uma linguagem de composição é uma generalização do conceito de linguagens de *script*, pois deve incluir um suporte de mais alto nível para a definição de estilos arquitetônicos, abstrações de composição, abstrações de coordenação e abstrações de adaptação. Dessa forma, Piccola agrega idéias de linguagens de *script*, linguagens e modelos de coordenação, técnicas de adaptação e ADLs.

3.6 Considerações Finais

Neste capítulo, vimos uma série de conceitos relacionados com a composição de aplicações, e demos uma atenção especial àqueles relacionados com a composição e a adaptação dinâmica de aplicações. Analisamos vários aspectos das linguagens de *script*, que são a principal técnica utilizada para compor aplicações baseadas em componentes. Também fizemos uma breve descrição de outros mecanismos de composição, que procuram trazer um maior rigor para a tarefa de composição.

Os mecanismos de composição vistos neste capítulo diferem bastante com relação à capacidade de expressão e ao suporte à composição dinâmica. Por serem computacionalmente completas, linguagens de *script* geralmente podem ser utilizadas para implementar componentes, e também podem facilitar a implementação de adaptadores entre interfaces incompatíveis.

Já as ADLs permitem no máximo a criação de novos componentes a partir da composição de outros já existentes. A adaptação entre interfaces incompatíveis muitas vezes é feita através da definição de novos conectores, que ficam responsáveis por ajustar as interfaces incompatíveis. Entretanto, muitas ADLs não permitem a definição de novos tipos de conectores, o que dificulta bastante a adaptação entre interfaces incompatíveis.

Por outro lado, as ADLs permitem uma especificação mais abstrata da estrutura de uma aplicação. As linguagens de *script* geralmente não oferecem suporte explícito para estilos arquitetônicos, ou oferecem apenas um único estilo, como é o caso de Bourne Shell, que adota o estilo de filtros e *pipes*.

A integração com sistemas de componentes, como CORBA, COM e JavaBeans, é mais comumente encontrada em linguagens de *script*. Mesmo assim, nenhuma das abordagens ou ferramentas descritas neste capítulo oferece alguma solução de interoperabilidade entre sistemas de componentes, de tal forma que fosse possível construir aplicações a partir de componentes de diferentes sistemas de uma maneira transparente. Apesar de Piccola poder representar múltiplos modelos de objetos, ela ainda não trata dessa questão de interoperabilidade [ALSN00].

Entre os mecanismos vistos na seção 3.5, o suporte à composição dinâmica é bem limitado. Quando uma ADL oferece algum suporte à composição dinâmica, tipicamente só são permitidas adaptações que preservem a arquitetura original da aplicação, isto é, não é permitido a inclusão de novos tipos de componentes e conectores. Mesmo linguagens de composição mais completas como Piccola ainda não permitem adaptações dinâmicas na configuração de uma aplicação.

Devido a própria origem das linguagens de *script*, um suporte mais flexível para a composição

dinâmica é bem mais natural para essas linguagens. As linguagens de *script* foram fortemente influenciadas pelos paradigmas de programação exploratória e de prototipação. Esses paradigmas são intrinsecamente dinâmicos, e suas ferramentas de programação geralmente usam uma linguagem interpretada ou algum outro mecanismo interativo, para permitir que um operador interaja diretamente com a ferramenta para estender ou adaptar uma determinada aplicação. Também é importante para essas ferramentas oferecer ao usuário uma maneira simples de expressar suas intenções.

Uma ferramenta de composição baseada em uma linguagem de *script* pode ser de grande valia tanto para um processo de desenvolvimento baseado em prototipação rápida, quanto para a obtenção de uma aplicação final com um alto grau de flexibilidade.

Capítulo 4

Uma Proposta de Modelo de Composição

Neste capítulo, apresentamos nossa proposta de modelo para composição de aplicações. Esse modelo define um mecanismo de composição dinâmica baseado em uma linguagem de *script*, que permite tanto o uso e adaptação de componentes já existentes quanto a interoperabilidade entre diferentes sistemas de componentes. O nosso principal objetivo ao definir um modelo de composição é identificar um conjunto de mecanismos que ofereça uma melhor estruturação para a integração entre uma linguagem de *script* e um sistema de componentes. O modelo não é amarrado a nenhuma linguagem de *script* específica, nem é restrito a um único sistema de componentes.

Esse modelo de composição procura oferecer um mecanismo que possa ser usado para comandar os dois tipos de adaptação dinâmica de aplicações vistos na seção 3.2. Para isso, o modelo adota diversas características presentes em ambientes de prototipação, com a finalidade de oferecer para a adaptação de um sistema já em operação a mesma flexibilidade disponível naqueles ambientes. Cabe destacar que o nosso modelo oferece apenas um mecanismo interativo, através de uma linguagem de *script*, para comandar as reconfigurações das conexões entre os componentes. Além desse mecanismo, a aplicação ou o sistema de componentes precisa oferecer mecanismos que permitam a reconfiguração dinâmica das conexões entre seus componentes (*re-wiring*).

Nosso modelo de composição é formado basicamente por um modelo de objetos e um padrão de projeto (*design pattern*) para a implementação de *bindings* dinâmicos entre linguagens de *script* e sistemas de componentes. A motivação para definirmos um modelo de objetos foi a necessidade de termos um conjunto de conceitos que pudessem descrever os aspectos semânticos de objetos de diferentes sistemas de componentes, de uma maneira uniforme. Assim, classificamos nosso modelo de objetos como um modelo de objetos integrador, ou apenas *modelo integrador*. Esse modelo integrador é a base para permitir a interoperabilidade entre sistemas de componentes diferentes.

Procuramos definir um modelo de objetos que fosse simples e expressivo o suficiente para representar componentes de diferentes sistemas de uma maneira uniforme. Para fazer a definição desse modelo, utilizamos alguns conceitos básicos que estão presentes em todos os sistemas de componentes tratados no capítulo 2. Nosso modelo de objetos integrador trata apenas de aspectos que estão relacionados com a composição e interação entre objetos, sem abordar aspectos relacionados com suas implementações. Para que uma linguagem de *script* possa ser utilizada como elemento de ligação de nosso mecanismo de composição, ela também deve oferecer um modelo de objetos compatível com esse modelo integrador.

É importante observar que tratamos de dois tipos diferentes de modelo neste capítulo. O primeiro é o modelo de composição dinâmica, que define um conjunto de mecanismos para permitir a utilização de uma linguagem de *script* para compor, adaptar e implementar componentes. O outro é o modelo de objetos, que serve de base conceitual para o mecanismo de composição, permitindo que componentes de diferentes sistemas sejam unificados através de uma representação comum. Quando nos referirmos

ao modelo integrador, estaremos fazendo referência ao modelo de objetos que faz parte de nosso modelo de composição.

A partir do nosso modelo de objetos integrador, definimos o padrão de projeto para a implementação de *bindings* dinâmicos entre linguagens de *script* e sistemas de componentes. Esse padrão de projeto tem como objetivo definir um conjunto de mecanismos que faça a ligação entre uma linguagem de *script* e um sistema de componentes, sem utilizar o mecanismo tradicional de geração prévia de *stubs*. Um *binding* que siga esse padrão permite que uma série de tarefas sejam realizadas em tempo de execução:

- a incorporação e o uso efetivo em uma aplicação de novos tipos de componentes, identificados apenas em tempo de execução;
- a implementação de componentes através da própria linguagem de composição;
- a extensão e adaptação de componentes também através da linguagem de composição.

O nosso padrão de *binding* pode ser aplicado a qualquer sistema de componentes e linguagem interpretada que atendam a um determinado conjunto de requisitos.

Como veremos na seção 4.3, o modelo de objetos integrador e o padrão de *bindings* dinâmicos permitem que pontes entre componentes de diferentes sistemas sejam criadas em tempo de execução. Através dessas pontes dinâmicas, componentes de sistemas diferentes podem se comunicar de uma forma transparente.

Cabe destacar que o aspecto de reuso de componentes é abordado em nosso modelo de composição de duas formas. Primeiro, o mecanismo de composição oferece recursos para a adaptação de componentes já existentes a novas situações de uso. Por último, o mecanismo de pontes dinâmicas aumenta as possibilidades de reuso de um componente, pois não restringe a sua utilização a um único sistema de componentes.

4.1 O Modelo de Objetos Integrador

Para facilitar o mapeamento dos modelos de objetos dos sistemas de componentes analisados neste trabalho, e para não introduzir restrições desnecessárias que dificultassem o mapeamento de outros sistemas, procuramos definir um modelo de objetos bem simples, que só tivesse os conceitos necessários para este trabalho. Se por um lado esse modelo integrador deve unificar os conceitos de objeto de diferentes sistemas de componentes, por outro ele deve poder ser representado em uma linguagem de composição, de tal forma que componentes possam ser manipulados através dessa linguagem. Assim, nosso modelo de objetos deve procurar conciliar características tanto dos sistemas de componentes quanto de linguagens de *script*.

Definimos um modelo de objetos integrador que privilegia a flexibilidade em tempo de execução, seguindo uma abordagem similar àquela utilizada em linguagens baseadas em protótipos [Weg87, SLU88, DMC92]. Além de definir aspectos semânticos dos objetos, o nosso modelo integrador também define um sistema de tipos que oferece o máximo de flexibilidade em tempo de execução sem comprometer a correção de um programa. Assumindo o uso de uma linguagem de composição sem declarações e com tipagem dinâmica, que são características comuns a linguagens de *script*, um sistema de tipos baseado em *compatibilidade estrutural* [RIR93, DGLM95] para fazer a verificação de tipos é uma alternativa interessante.

Para apresentar esse modelo de objetos integrador, vamos caracterizar inicialmente o seu conceito mais básico de objeto. Depois estendemos o modelo com um sistema de tipos para permitir a verificação, em tempo de execução, da compatibilidade de tipos entre componentes. A descrição da arquitetura do modelo de objetos é feita com o auxílio da linguagem OMG IDL, que é utilizada para

```
typedef sequence<Dispatcher> ListOfObjects;
typedef Dispatcher ExceptionDescr;

interface Dispatcher {
   boolean invoke (in string method_name, in ListOfObjects params_in,
                   out ListOfObjects params_out,
                   out ExceptionDescr ex);
};
```

Figura 4.1: A interface `Dispatcher`.

representar as operações oferecidas pelas entidades do modelo. O Apêndice A apresenta a definição completa do nosso modelo de objetos em IDL.

4.1.1 O Modelo Básico

A unidade básica de informação em nosso modelo é o *objeto*. Um objeto é uma entidade auto-suficiente com seu próprio estado e comportamento. Um objeto é capaz de responder a mensagens, e seu comportamento é determinado pela forma como reage às mensagens recebidas. Já o estado de um objeto é o reflexo dos efeitos do tratamento de mensagens sobre si mesmo. Apesar do estado de um objeto ser tipicamente representado por variáveis de instância, em nosso modelo esse estado só pode ser acessado através das operações oferecidas explicitamente pelo próprio objeto, isto é, as variáveis de instância não são visíveis diretamente por entidades externas.

Ao contrário de modelos de objetos mais tradicionais — como o de Simula, Smalltalk, C++ e Java — nosso modelo não possui a noção de classes de objetos. Como a princípio só estamos interessados em acessar os serviços oferecidos por objetos e realizar conexões entre eles, a noção de classe, vista como um molde para a implementação de objetos, não se mostra relevante, podendo ser encarada como um mero detalhe de implementação do componente.

Sem a noção de classe, um objeto unifica em si mesmo o seu próprio estado e comportamento, seguindo uma abordagem muito similar à adotada por linguagens baseadas em protótipos. Esse tipo de abordagem é freqüentemente descrito como sendo mais adequado para o início do desenvolvimento de um sistema, quando ainda se está modelando o domínio da aplicação, ou para evoluir objetos e sistemas de uma forma independente e dinâmica [Weg87, SLU88, DMC92, SU95, Kni99].

Como esse modelo básico não define nenhum mecanismo para compartilhamento de comportamento entre objetos, pois um objeto é definido como uma entidade totalmente auto-suficiente, ele pode ser classificado mais especificamente como sendo *baseado em encapsulamento* (*embedding-based*) [Car95].

Para oferecer uma maneira uniforme de representar objetos em nosso modelo, e também permitir que se tenha acesso às diferentes interfaces funcionais oferecidas pelos objetos, modelamos objetos como sendo implementações independentes de uma interface genérica para recepção e tratamento de mensagens (*dispatching*). A figura 4.1 apresenta essa interface genérica de *dispatching*, utilizando a notação da linguagem OMG IDL.

A interface `Dispatcher` facilita a conexão dinâmica entre objetos que oferecem diferentes interfaces funcionais. Apesar das interfaces funcionais dos objetos não serem explicitamente representadas no modelo, a operação `invoke` permite a construção dinâmica de chamadas para as operações oferecidas pela interface funcional de um objeto.

O parâmetro `method_name` faz o papel do seletor da operação que deve ser executada. O parâme-

tro `params_in` contém uma lista com todos os parâmetros que devem ser passados para a operação selecionada. Ao término da execução da operação, o parâmetro `params_out` contém uma lista com todos os valores retornados pela operação. Como `params_out` pode conter mais de um valor, esse parâmetro pode ser usado para retornar tanto o valor de retorno de uma operação propriamente dito, que seria o primeiro elemento da lista, quanto os seus parâmetros `out`.

Uma operação também pode sinalizar uma exceção. Nesse caso, o parâmetro `ex` irá conter a descrição da exceção sinalizada e o parâmetro `params_out` será uma lista vazia. Como exceções podem ter diferentes formas de representação em diferentes sistemas de componentes, desde simples identificadores numéricos até objetos com operações e estado, elas também são representadas como objetos em nosso modelo, para oferecer uma forma mais genérica de representação.

O valor de retorno da operação `invoke` é um valor booleano que indica se o objeto oferece tratamento para a mensagem enviada, que é composta pelo nome do método requisitado e pelos parâmetros de entrada (`params_in`). Se o valor de retorno de `invoke` for verdadeiro, significa que o objeto tratou a mensagem enviada, e o resultado do tratamento da mensagem está codificado nos parâmetros `params_out` e `ex`. Já se o valor de retorno for falso, indica que o objeto não é capaz de tratar a mensagem recebida e os parâmetros de saída `params_out` e `ex` não possuem valores válidos.

Apesar dos sistemas de componentes analisados no capítulo 2 poderem representar na interface de um componente a abstração de um atributo, não incluímos métodos específicos para o acesso a atributos na interface `Dispatcher`. Entretanto, a manipulação de atributos pode ser feita através de mensagens do tipo `get`/`set`, enviadas através do método `invoke`. Dessa forma, mantemos o modelo simples, sem comprometer o seu poder de representação.

4.1.2 O Sistema de Tipos

Na seção anterior, definimos o conceito básico de objeto de nosso modelo, através do qual componentes de diversos sistemas podem ser representados. Entretanto, não tratamos diretamente da questão dos *tipos* dos objetos. Essa questão é de grande relevância quando pensamos em integração com os sistemas de componentes existentes, pois estes consideram o tipo de um componente, também conhecido como sua *interface*, como um contrato de serviços estabelecido entre cliente e servidor. O procedimento de verificação de tipos pode ser visto como uma ferramenta independente para verificar uma importante propriedade de um programa: a ausência de erros de tipo em tempo de execução.

Seguindo a diretriz de oferecer o máximo de flexibilidade em tempo de execução, definimos para o nosso modelo de objetos um sistema de tipos flexível que permite a verificação dinâmica de tipos sem alterar a semântica básica dos objetos. Para possibilitar a incorporação de componentes com interfaces somente conhecidas em tempo de execução, um sistema de tipos baseado em verificação dinâmica se torna fundamental.

Procuramos tratar o sistema de tipos como um mecanismo ortogonal ao restante do modelo de objetos. De uma certa forma, os sistemas de componentes já induzem esse tipo de independência entre o sistema de tipos e os aspectos relacionados com o comportamento dos objetos, ao proporem uma rígida separação entre a interface de um componente e sua implementação.

A Definição de Tipo

Em modelos orientados a objetos puros, objetos carregam suas próprias operações, e estas são a única parte visível de um objeto. De acordo com esses modelos, a única forma de se comunicar com um objeto é através da requisição de suas operações. Assim, se caracterizarmos um *tipo* como sendo uma coleção de valores compartilhando uma estrutura ou forma semelhante, devemos considerar somente as operações como a estrutura visível de um objeto. Dessa forma, uma boa definição para tipos em

```
interface InterfaceDescr {
   NameSeq getAllOperationNames();
   MethodSignature getSignature(in string name);
};

typedef sequence<InterfaceDescr> InterfaceDescrSeq;

interface MethodSignature {
   string getName();
   InterfaceDescrSeq getParamsIn();
   InterfaceDescrSeq getParamsOut();
   InterfaceDescrSeq getExceptions();
};
```

Figura 4.2: As interfaces do nosso sistema de tipos.

linguagens orientadas a objetos é que o tipo de um objeto é sua interface, isto é, sua coleção de operações [RIR93].

Seguindo essa definição para tipo, podemos dizer que um erro de tipo em um sistema OO é a requisição de uma operação para um objeto que não tem um método para tratá-la. Modelos estaticamente tipados são aqueles que podem garantir a ausência de tais erros por um procedimento de verificação de tipos efetuado em tempo de compilação do sistema. Já um modelo dinamicamente tipado efetua o mesmo procedimento de verificação em tempo de execução do sistema. Nesse caso, fica por conta do ambiente de execução da linguagem o correto tratamento de um eventual erro de tipo.

De acordo com a definição de tipo vista acima, podemos representar um tipo em nosso modelo de objetos através da interface `InterfaceDescr` (figura 4.2). Com essa interface, um tipo oferece a descrição completa das assinaturas de suas operações. A operação `getAllOperationNames` fornece uma seqüência com os nomes de todas as operações definidas em seu tipo. Já a operação `getSignature` recebe o nome de uma operação e retorna sua assinatura completa.

A assinatura de uma operação em nosso modelo, representada pela interface `MethodSignature`, é composta pelo nome da operação e pelos tipos dos parâmetros de entrada, dos parâmetros de saída e das exceções que podem ser sinalizadas pela operação (figura 4.2).

A Associação de Tipos em Nosso Modelo de Objetos

Uma vez definido o que é um tipo em nosso modelo de objetos, precisamos estabelecer uma maneira de associar um tipo a um objeto. Para isso, podemos estender a interface `Dispatcher` com a operação `getInterface` (figura 4.3). Essa operação retorna um objeto do tipo `IntefaceDescr`, descrevendo completamente a interface funcional oferecida por um objeto. Assim, podemos dizer que o tipo de um determinado objeto é o conjunto de assinaturas que a sua operação `getInterface` retorna.

Essa versão final da interface `Dispatcher`, em conjunto com as interfaces definidas na figura 4.2, oferecem um mecanismo de introspecção de tipos para o nosso modelo de objetos; e assim podemos definir um esquema para verificar a compatibilidade entre tipos de objetos em tempo de execução, baseado na definição de subtipo apresentada a seguir.

```
interface Dispatcher {
   boolean invoke(in string method_name,
                  in ListOfObjects params_in,
                  out ListOfObjects params_out,
                  out ExceptionDescr ex);
   InterfaceDescr getInterface();
};
```

Figura 4.3: A versão final da interface `Dispatcher`.

Compatibilidade Estrutural

Usualmente, em linguagens orientadas a objetos, um tipo é considerado um subtipo de outro somente quando ele é declarado dessa forma, diretamente ou por transitividade. Entretanto, podemos flexibilizar essa definição, sem comprometer a correção de um programa, estabelecendo que um tipo é um subtipo de um outro se eles forem compatíveis de alguma forma apropriada.

Para o nosso modelo de objetos, procuramos definir um sistema de tipos tão flexível quanto fosse possível, sem que com isso a correção do programa fosse comprometida. Mais especificamente, definimos uma regra para o nosso sistema de tipos que estipula que um objeto só será considerado incompatível com um tipo quando houver uma boa razão para isso. De acordo com essa regra, consideramos que um tipo A é subtipo de B caso não haja possibilidade de ocorrer um erro de tipo quando um objeto do tipo A for usado no lugar de B.

Para verificar a possibilidade de ocorrer um erro de tipo, escolhemos um esquema de verificação baseado em *compatibilidade estrutural* [RIR93, Ier93, DGLM95]. De acordo com esse esquema de verificação, um tipo A é subtipo de B se para cada operação de B existe uma operação compatível em A, e a compatibilidade entre duas operações é determinada em função de suas assinaturas completas. Assim, podemos dar uma definição mais precisa para a relação de subtipo em nosso modelo de objetos:

> Um tipo A é um subtipo de B ($A \prec B$) se e somente se, para cada método X em B, com n parâmetros de entrada ($P_{B_{1\ldots n}}$) e m parâmetros de saída ($R_{B_{1\ldots m}}$), existe um método X em A, também com n parâmetros de entrada ($P_{A_{1\ldots n}}$) e m parâmetros de saída ($R_{A_{1\ldots m}}$), onde para todo $i \leq m$, $R_{A_i} \prec R_{B_i}$, e, para todo $i \leq n$, $P_{B_i} \prec P_{A_i}$.

A aparente inversão da última condição enunciada ($P_{B_i} \prec P_{A_i}$) é conhecida como a *regra da contravariância*, e é necessária para garantir a correção do sistema de tipos [CW85]. Essa definição é não só suficiente para garantir a ausência de erros de tipo em tempo de execução, como também é necessária. Em qualquer modelo que aceite A no lugar de B com $A \not\prec B$, é possível criar uma situação em que ocorra um erro em tempo de execução.

Observe que a definição acima não é formal, já que ela pode resultar em uma recursão infinita quando aplicada a tipos recursivos, isto é, tipos que fazem referência a eles mesmos. Uma definição formal de subtipo e uma prova formal de que a definição acima evita erros de tipo podem ser encontradas em [Ier93, IR95].

Para exemplificar as regras de compatibilidade estrutural, considere as interfaces definidas na figura 4.4. De acordo com essas interfaces, podemos observar os seguites relacionamentos: $Point2 \prec Point$, $Point \not\prec Point1$ (falta o método `copy` em `Point`), e $Point1 \not\prec Point$ (devido ao parâmetro incompatível do método `moveBy`).

Para verificar a compatibilidade estrutural entre dois tipos, deve-se observar exclusivamente as assinaturas de suas operações, independentemente de qualquer declaração explícita de tipo e da ordem em que as operações são declaradas na definição do tipo. Como demonstrado em [Ier93], um sistema

```
interface Point {
   double getX();
   double getY();
   void moveBy(in Point pt);
};
interface Point1 {
   double getX();
   double getY();
   void moveBy(in Point1 pt);
   Point1 copy();
};
interface Point2 {
   double getX();
   double getY();
   void moveBy(in Point pt);
   Point2 copy();
};
```

Figura 4.4: Um exemplo para a aplicação das regras de compatibilidade estrutural.

de tipos com compatibilidade estrutural oferece o mecanismo mais flexível possível para a verificação de compatibilidade entre tipos, sem comprometer a correção de um programa.

Um Algoritmo para a Verificação de Compatibilidade Estrutural

Para podermos verificar a compatibilidade entre os tipos dos objetos de um sistema, de tal forma que possamos identificar e tratar possíveis erros de tipo, é necessário que o sistema como um todo esteja corretamente tipado, isto é, todos os objetos implementam de forma consistente as operações definidas em suas interfaces. Em um sistema corretamente tipado, ao requisitarmos uma operação através do método `Dispatcher::invoke`, se as seguintes condições forem observadas:

- o objeto receptor possui uma operação em sua interface (obtida através da operação `getInterface`) com o mesmo nome da operação sendo requisitada (parâmetro `method_name`);
- os tipos dos argumentos passados através do parâmetro `params_in` são subtipos dos parâmetros formais definidos na assinatura da operação requisitada (observando a ordem apropriada);

então sempre ocorrerá que:

- o método `Dispatcher::invoke` retornará verdadeiro;
- caso a execução da operação seja concluída com sucesso, os tipos dos resultados retornados pelo método `invoke`, através do parâmetro `params_out`, serão subtipos dos tipos dos parâmetros de saída correspondentes definidos na assinatura da operação requisitada;
- caso a operação requisitada sinalize uma exceção, o tipo desta deve ser subtipo de pelo menos um dos tipos de exceções definidos na assinatura da operação.

Caso contrário, o método `invoke` retornará falso, indicando um erro de tipo em tempo de execução.

```
function TypedInvoke (obj, method_name, params_in)
   local interface = obj:getInterface()
   local op_sig = interface:getSignature(method_name)
   if not op_sig then
      TypeError()
   end
   if not checkParams(params_in, op_sig:getParamsIn()) then
      TypeError()
   end
   local ok, params_out, ex = obj:invoke(method_name,params_in)
   assert(ok)
   if ex then
      assert(checkException(ex, op_sig:getExceptions()))
   else
      assert(checkParams(params_out, op_sig:getParamsOut()))
   end
   return params_out, exception
end
```

Figura 4.5: Um esboço do algoritmo de verificação de tipos executado em uma chamada de método.

Assim, assumindo um sistema corretamente tipado, podemos definir um esboço para o algoritmo de verificação de tipos executado durante a chamada de um método de um objeto `obj`[1] (figura 4.5). De acordo com esse algoritmo, primeiro verificamos se o objeto sobre o qual o método vai ser chamado possui uma operação com o mesmo nome em sua interface. Depois, verificamos, através da função `checkParams`, se os tipos dos argumentos contidos na lista `params_in` são compatíveis com os tipos dos parâmetros definidos na assinatura da operação. Se todos os objetos envolvidos na chamada estiverem corretamente tipados, após a operação ter sido efetivamente requisitada, os valores retornados pelo método serão compatíveis com os tipos definidos na assinatura da operação. Se um erro de tipo for detectado durante esse processo, ele será sinalizado para o ambiente de execução através da função `TypeError`. Já se for verificado que o sistema não está corretamente tipado, isto é, se um objeto não implementar realmente a interface que ele diz implementar, a execução do programa será interrompida.

As funções `checkParams` e `checkException`, que utilizamos para verificar os tipos dos argumentos e exceções envolvidos no processo de chamada de um método, são apresentadas na figura 4.6. A função `checkParams` inicialmente verifica se uma lista de argumentos tem o mesmo número de elementos que a lista de parâmetros formais definida na assinatura da operação. Depois, ela verifica para cada argumento se seu tipo é um subtipo do parâmetro formal correspondente. Observe que usamos a função `checkParams` para verificar tanto os argumentos de entrada quanto os de saída, obedecendo a regra de contra-variância: os valores passados de um ponto para outro devem ser sempre de um subtipo dos tipos esperados no ponto de destino.

Já a função `checkException` verifica se uma uma exceção sinalizada pelo método requisitado é de um subtipo de alguma das exceções definidas na assinatura da operação. Para fazer essa verificação, essa função utiliza a função `belongs`, que recebe como parâmetros um tipo e uma lista de tipos, e testa se o primeiro parâmetro é subtipo de algum dos tipos contidos no segundo parâmetro.

[1] Descrevemos os algoritmos desta seção com o auxílio de uma versão simplificada da linguagem Lua. Um resumo dessa linguagem pode ser visto na seção 5.1, enquanto uma descrição mais completa da linguagem pode ser obtida em [IFC96, IFC99, FIC96].

```
function checkParams(args, interfaces)
   if length(args) == length(interfaces) then
      for i = 1, length(args) do
         if not isSubtype(args[i]:getInterface(), interfaces[i]) then
            return false
         end
      end
      return true
   else
      return false
   end
end

function checkException(exception, interfaces)
   return belongs(exception:getInterface(), interfaces)
end
```

Figura 4.6: As funções `checkParams` e `checkException`.

Tanto a função `checkParams` quanto a função `belongs` utilizam o predicado `isSubtype` para verificar se um tipo é subtipo de um outro. Apresentamos um esboço desse predicado na figura 4.7. Esse predicado verifica a relação $A \prec B$ de acordo com a definição de subtipo vista na seção anterior, incluindo a verificação dos tipos das exceções sinalizadas em cada operação.

Para resolver o problema de recursão infinita que ocorre quando aplicamos a nossa definição de subtipo a tipos recursivos, o predicado `isSubtype` utiliza as funções auxiliares `addToHierarchy` e `removeFromHierarchy` e o predicado `inHierarchy`. Apesar de não descrevermos completamente essas funções, o funcionamento delas é basicamente o seguinte:

`addToHierarchy` — recebe dois tipos A e B como parâmetros e registra a existência de um relacionamento de subtipo entre eles ($A \prec B$).

`removeFromHierarchy` — recebe dois tipos A e B como parâmetros e desfaz o registro de relacionamento de subtipo entre eles ($A \not\prec B$).

`inHierarchy` — recebe dois tipos A e B como parâmetros e retorna verdadeiro caso haja um registro do relacionamento de subtipo entre A e B ($A \prec B$), e falso no caso contrário ($A \not\prec B$).

Tendo definido essas funções auxiliares, podemos analisar o funcionamento do predicado `isSubtype`. Inicialmente, ele testa se anteriormente já foi verificado que A é subtipo de B (linhas 2–4). Caso o relacionamento ainda não tenha sido verificado, este é registrado através da função `addToHierarchy`. Assim, assumimos que o relacionamento $A \prec B$ é verdadeiro, e só desfazemos o registro do relacionamento caso encontremos algum motivo que prove o contrário durante a execução do algoritmo de verificação de compatibilidade estrutural. Como pode ser visto em [Ier93, IR95], esse artifício resolve problemas de recursão do tipo "A é subtipo de B se e somente se A for subtipo de B". No laço das linhas 7–30, percorremos cada uma das operações de B e verificamos se A possui uma operação com uma assinatura compatível. Além de possuirem o mesmo nome, para que duas operações sejam compatíveis, as seguintes condições devem ser observadas:

- `length(PA) == length(PB)` e `length(RA) == length(RB)`;
- `isSubtype(PB[i], PA[i]) == true`, para `i` de 1 a `length(PA)`;

```
 1 function isSubtype(A, B)
 2    if inHierarchy(A,B) then
 3       return true
 4    end
 5    addToHierarchy(A,B)
 6
 7    for op_name in B:getAllOperationNames() do
 8       local opsig_A = A:getSignature(op_name)
 9       local opsig_B = B:getSignature(op_name)
10       if not opsig_A then
11          removeFromHierarchy(A,B)
12          return false
13       end
14       local PA = opsig_A:getParamsIn()
15       local RA = opsig_A:getParamsOut()
16       local EA = opsig_A:getExceptions()
17       local PB = opsig_B:getParamsIn()
18       local RB = opsig_B:getParamsOut()
19       local EB = opsig_B:getExceptions()
20       if not (checkTypeLists(PB, PA) and checkTypeLists(RA, RB)) then
21          removeFromHierarchy(A,B)
22          return false
23       end
24       for ex_type in EA do
25          if not belongs(ex_type,EB) then
26             removeFromHierarchy(A,B)
27             return false
28          end
29       end
30    end
31    return true
32 end
```

Figura 4.7: A função `isSubtype`.

- `isSubtype(RA[i], RB[i]) == true`, para `i` de 1 a `length(RA)`;
- `belongs(ex_type,EB) == true`, para `ex_type` em `EA`.

Caso qualquer uma dessas condições seja falsa, o relacionamento $A \prec B$ não é verdadeiro. Para verificar se as listas de parâmetros de entrada e de saída de duas assinaturas são compatíveis, usamos a função `checkTypeLists`:

```
function checkTypeLists (list1, list2)
   if length(list1) ~= length(list2) then
      return false
   end
   for i = 1, length(list1) do
      if not isSubtype(list1[i], list2[i]) then
         return false
      end
   end
   return true
end
```

Conseqüências do Uso de Compatibilidade Estrutural

Uma conseqüência da abordagem estrutural é que diversas alterações na interface de um componente podem não afetar o seu uso, como ocorre com a adição ou a reordenação de operações na interface. Essa idéia, quando aplicada ao contexto de um sistema de componentes, pode oferecer uma flexibilidade ainda maior. Por exemplo, se em CORBA as estruturas forem consideradas como sendo tipos abstratos de dados, com pares de operações *get* e *set* para cada um de seus campos, a adição e a reordenação de campos podem ser toleradas em diversas situações. Da mesma forma, os tipos `array` e `sequence` de CORBA podem ser considerados tipos compatíveis, dado que possuem as mesmas operações abstratas *set_index* e *get_index*.

Um outro aspecto importante da compatibilidade estrutural é a amarração de um tipo com sua implementação. O mecanismo adotado por linguagens como C++, onde um tipo define o formato de uma tabela de ponteiros para métodos (*tabela de métodos virtuais*) e as classes preenchem essas tabelas com os endereços de seus métodos, não se aplica a um esquema de compatibilidade estrutural. O modelo de compatibilidade estrutural requer algum mecanismo de *dispatching*, como aqueles usados por Smalltalk [GR83, NH94], pelas interfaces dinâmicas de CORBA e COM, e pela interface `Dispatcher`.

Compatibilidade estrutural também pode ser usada para resolver um problema recorrente em sistemas de componentes, que está relacionado com versões de interfaces. Tipicamente, o problema ocorre quando é feita uma alteração na definição da interface de um serviço, mas nem todos os clientes do serviço podem ser atualizados. Assim, podem existir em uma mesma aplicação componentes trabalhando com diferentes versões de uma mesma interface, ou melhor, com diferentes interfaces com o mesmo nome. Um esquema de compatibilidade estrutural poderia ser usado nesse caso para verificar a compatibilidade entre as interfaces, independentemente de seus nomes.

Como nosso modelo de objetos tem entre seus objetivos principais permitir a interoperabilidade entre diferentes sistemas de componentes, e conseqüentemente entre diferentes sistemas de tipos, o uso de um esquema baseado em compatibilidade estrutural aparece novamente como uma solução interessante. Um esquema desse tipo permite que façamos a verificação da compatibilidade entre tipos de diferentes sistemas de componentes, pois não exige que os relacionamentos de subtipo sejam declarados explicitamente, o que não poderia ser feito diretamente entre sistemas de tipos diferentes.

A grande vantagem da compatibilidade estrutural é sua flexibilidade. Entretanto, essa mesma flexibilidade algumas vezes é considerada uma desvantagem. O argumento é que um tipo pode ser considerado subtipo de um outro erroneamente. Isto é, apesar dos tipos serem sintaticamente compatíveis, não haveria nenhuma garantia de uma compatibilidade semântica. Por outro lado, cabe lembrar que essa mesma garantia também não é oferecida pela regra tradicional de compatibilidade de tipos. A única diferença nesse sentido é que se espera que no modelo tradicional os relacionamentos de subtipo sejam definidos com mais cuidado.

Novamente podemos aplicar a análise de decisões antecipadas e não-antecipadas. O modelo tradicional de compatibilidade entre tipos, baseado na definição explícita de subtipos, promove a antecipação da definição dos relacionamentos de compatibilidade, garantindo uma maior confiabilidade. Já o modelo baseado em compatibilidade estrutural promove a não antecipação desses relacionamentos, permitindo uma maior flexibilidade para evoluções não previstas em um sistema.

4.2 O Padrão para o *Binding* Dinâmico de Linguagens

Para permitir que uma linguagem de *script* se comunique com um sistema de componentes de software, é necessário que seja implementado um *binding* entre a linguagem e o sistema de componentes. Esse *binding* pode ser baseado na geração prévia de *stubs* e esqueletos, como tipicamente ocorre com linguagens compiladas.

Como em nosso modelo de composição esperamos que seja possível a incorporação dinâmica de novos componentes, cuja interface só será conhecida em tempo de execução (*very late binding*), a geração dos *stubs* para esses componentes faz-se necessária também em tempo de execução. Para permitir essa geração dinâmica, é fundamental que o sistema de componentes ofereça algum metamecanismo que permita a consulta dinâmica das definições das interfaces dos componentes. De posse dessas definições, é possível a geração dinâmica dos *stubs* de acesso aos componentes.

Entretanto, tendo como base o modelo de objetos descrito na seção anterior, é possível fazer o *binding* de uma linguagem de *script* com um sistema de componentes sem utilizar *stubs* específicos para cada interface. O *binding* em questão consiste em fazer a ligação entre a linguagem de *script* e a interface `Dispatcher`.

Como os tipos dos componentes possivelmente só serão conhecidos em tempo de execução, é necessário poder realizar a verificação de tipos também em tempo de execução. Assim, além do sistema de componentes disponibilizar as definições das interfaces, a linguagem também deve oferecer um mecanismo para consultar em tempo de execução os tipos de seus valores. De posse dessas informações, a implementação do *binding* é capaz de realizar a verificação de tipos entre os valores manipulados em tempo de execução pela linguagem e os tipos formais definidos nas interfaces dos componentes. Nesse esquema dinâmico, o *binding* também pode adotar um conjunto de coerções válidas entre os tipos da linguagem e os tipos esperados pelo sistema de componentes.

Nesta seção, definimos um padrão de projeto (*design pattern*) para a implementação de *bindings* entre linguagens de *script* e sistemas de componentes, de acordo com o modelo de objetos integrador descrito na seção anterior. Todo *binding* entre uma linguagem de programação e um sistema de componentes envolve a definição de um mapeamento do modelo de objetos do sistema de componentes para a linguagem de programação. Isto é, define-se uma forma de representar através dos mecanismos da linguagem de programação os conceitos presentes no modelo de objetos do sistema de componentes. Assim como outros mapeamentos entre linguagens de programação e sistemas de componentes, mapeamentos que sigam o nosso padrão devem representar componentes externos da mesma forma que objetos nativos da linguagem.

Os objetos nativos da linguagem também devem ser, em um determinado nível de abstração, compatíveis com a interface `Dispatcher`. Além disso, eles devem oferecer facilidades para a adaptação

```
interface  ExtendedDispatcher : Dispatcher {
   void setDelegatee(in Dispatcher obj);
   Dispatcher getDelegatee();

   void setMethod(in string name, in Method meth);
   void removeMethod(in string name);
   Method getMethod(in string name);
};
```

Figura 4.8: A interface `ExtendedDispatcher`.

dinâmica de objetos, com a finalidade de dar apoio às estratégias de adaptação vistas nas seções 3.2 e 3.3.

O nosso padrão de projeto para *bindings* dinâmicos é composto por dois mecanismos básicos, denominados *proxy genérico* e *adaptador genérico*. Esses mecanismos são responsáveis por fazer a ligação entre os objetos do sistema de componentes e os da linguagem de composição.

4.2.1 Os Recursos de Adaptação Dinâmica da Linguagem de Composição

Da forma como definimos nosso modelo de objetos na seção 4.1.1, um objeto não oferece recursos explícitos para adaptação dinâmica. Para que a linguagem utilizada por nosso mecanismo de composição possa realizar adaptações não-antecipadas sobre componentes, vamos acrescentar ao nosso modelo de composição recursos para o compartilhamento de comportamento através de delegação, e para a adição, remoção e substituição de tratadores de mensagens (métodos).

O mecanismo de delegação permite a adaptação seletiva de componentes, através de objetos adaptadores customizados para um determinado tipo de cliente. Já o mecanismo para manipulação de métodos permite a adaptação global de um componente, isto é, a adaptação pode afetar todos os clientes do objeto alterado.

Para representar esses mecanismos de adaptação e mantê-los isolados do nosso objeto básico, podemos definir uma versão estendida da interface `Dispatcher`, denominada `ExtendedDispatcher` (figura 4.8), que oferece todos os recursos de adaptação. A interface `ExtendedDispatcher` estende a interface `Dispatcher` com mecanismos reflexivos que permitem a manipulação da implementação de um objeto. A qualquer momento, tanto a aplicação quanto o próprio objeto podem interromper seu processamento e modificar o comportamento futuro do objeto através dessa interface. Cabe destacar que esses recursos de adaptação só se aplicam à linguagem de composição, já que, tipicamente, os sistemas de componentes tradicionais não oferecem diretamente mecanismos de suporte a esse tipo de adaptação dinâmica.

Delegação

Para dar suporte a um mecanismo de delegação, essa interface oferece as operações `setDelegatee` e `getDelegatee` para definir e consultar o objeto que deve ter seu comportamento compartilhado pelo mecanismo de delegação.

A semântica implícita do mecanismo de delegação é que toda mensagem que não puder ser tratada pelo próprio objeto que a recebeu (objeto *delegador*) deve ser repassada para o objeto *delegado*, e assim sucessivamente. Se um objeto não souber tratar uma determinada mensagem recebida e nem tiver um objeto delegado associado a ele, a operação `invoke` deve retornar falso para indicar a falha.

```
interface Method {
   void call(in Dispatcher self,
             in ListOfObjects params_in,
             out ListOfObjects params_out,
             out ExceptionDescr ex);
};
```

Figura 4.9: A interface `Method`.

A princípio, em qualquer instante da execução da aplicação, pode-se alterar o comportamento do objeto delegador, substituindo o objeto delegado. Da mesma forma, é possível criar dinamicamente um novo objeto que estende a funcionalidade de um outro. O novo objeto é responsável por implementar a nova funcionalidade, e delega para o objeto original as demais operações.

Manipulação de Métodos

Para possibilitar a extensão do comportamento de um objeto de uma forma global, a interface `ExtendedDispatcher` oferece operações que permitem alterar e consultar a implementação de um objeto. Para permitir a manipulação da implementação de um objeto, reificamos parcialmente a estrutura de sua implementação através da interface `Method` (figura 4.9). Essa interface representa um tratador de mensagem (método) que é usado para compor a implementação de um objeto. De acordo com essas interfaces, a implementação de um objeto é composta por um conjunto de objetos do tipo `Method`.

A operação `call` realiza efetivamente o tratamento de uma mensagem. Além de alguns parâmetros em comum com o método `Dispatcher::invoke`, essa operação também recebe como parâmetro a referência do objeto sobre o qual ela deve ser aplicada (parâmetro `self`).

Essa forma de reificação de um método faz com que nossa linguagem de composição trate métodos como objetos de primeira-classe. A amarração explícita do parâmetro `self` permite uma outra forma de compartilhamento em nosso modelo de composição: um mesmo método pode ser usado na implementação de mais de um objeto. Além disso, a passagem explícita do parâmetro `self` para a operação `call` possibilita a implementação de um mecanismo de delegação *real*; isto é, ao se delegar o tratamento de uma mensagem para um outro objeto, é possível especificar que o método correspondente seja executado sobre o objeto que originalmente recebeu a mensagem, e não sobre o objeto que possui o método.

Uma vez definida a interface `Method`, podemos incluir na interface `ExtendedDispatcher` operações para manipular a implementação de um objeto. A operação `setMethod` permite adicionar ou substituir um método em um determinado objeto, dependendo da existência ou não de um método com o mesmo nome registrado no objeto. A operação `removeMethod` remove da implementação de um determinado objeto o método com o nome passado como parâmetro. Já a operação `getMethod` retorna o método, se existir algum, com nome igual ao passado como parâmetro. É importante observar que, por motivos de simplificação, esse mecanismo de manipulação de métodos não permite a sobrecarga de métodos, isto é, não é possível um objeto ter mais de um método com o mesmo nome.

É importante observar que os mecanismos oferecidos pela interface `ExtendedDispatcher` podem ter impacto direto sobre o tipo de um objeto. Deixamos como responsabilidade do desenvolvedor garantir que um sistema continuará corretamente tipado após realizar alterações em um objeto através das operações da interface `ExtendedDispatcher`.

4.2.2 *Proxy* Genérico

O primeiro problema a ser tratado por nosso padrão de projeto, para fazer a integração entre uma linguagem de composição e um sistema de componentes, é como oferecer um mecanismo que permita que componentes externos sejam manipulados através da linguagem de composição. Para isso, definimos um mecanismo que denominamos de *proxy genérico*.

Um *proxy genérico* é um objeto nativo da linguagem, criado para representar um componente externo. Sua funcionalidade básica é delegar para o componente externo que ele representa qualquer operação aplicada sobre ele, fazendo as conversões necessárias entre os tipos de dados da linguagem de composição e os do sistema de componentes.

Um *proxy* genérico segue o padrão de projeto *proxy* apresentado em [GHJV95]. Entretanto, ao contrário dos mecanismos de *proxy* tradicionais, onde para cada tipo de interface é necessário prover uma implementação de *proxy* específica, um *proxy* genérico tira proveito de mecanismos de reflexividade para oferecer uma implementação única de *proxy*, capaz de atender a qualquer tipo de interface de objeto.

Basicamente, um *proxy* genérico é uma implementação da interface `ExtendedDispatcher`. Como essa interface herda as operações da interface `Dispatcher`, um *proxy* permite que qualquer operação da interface funcional do componente que ele representa seja requisitada através da operação `invoke`. Através dos serviços específicos da interface `ExtendedDispatcher`, o *proxy* genérico também permite redefinir o componente externo por ele representado, redefinir a implementação de uma operação do componente externo, e estender a sua própria interface com novos métodos. Assim, um *proxy* pode fazer o papel de um adaptador do componente para algum uso específico.

Tendo como base o nosso modelo de objetos integrador e os mecanismos de meta-programação dos sistemas de componentes, vistos nas seções 4.1 e 2.5 respectivamente, podemos definir um algoritmo de *dispatching* padrão para ser usado por um *proxy* genérico, sempre que o seu método `invoke` for acionado. Esse algoritmo, que é fortemente baseado em mecanismos de reflexividade, tipagem dinâmica e compatibilidade estrutural, pode ser sintetizado pelos passos a seguir:

1. Procurar na interface do componente externo associado ao *proxy* um método com o mesmo nome do seletor fornecido na mensagem recebida. Para isso, deve-se utilizar o mecanismo de introspecção oferecido pelo sistema de componentes.

2. Utilizando o mesmo mecanismo de introspecção, obter a descrição completa da assinatura do método requisitado.

3. Para cada parâmetro formal definido na assinatura do método, verificar se o argumento correspondente tem um tipo estruturalmente compatível. Para realizar essa verificação, são utilizados os mecanismos de introspecção do sistema de componentes e de tipagem dinâmica da linguagem de composição.

4. Utilizando o mecanismo de construção de chamadas dinâmicas do sistema de componentes, construir a requisição do método, fazendo as conversões dos tipos de dados da linguagem para os do sistema de componentes.

5. Fazer a requisição do método efetivamente através do mecanismo de construção de chamadas dinâmicas.

6. Por fim, devolver os resultados do método, fazendo as conversões apropriadas dos tipos de dados do sistema de componentes para os da linguagem de composição.

Caso ocorra uma falha em qualquer um dos passos do algoritmo, a operação `invoke` deve retornar falso para indicar o erro.

Esse algoritmo é um mecanismo fundamental de nosso modelo de composição, pois permite que sejam chamadas operações de componentes cujos tipos só vão ser conhecidos em tempo de execução. Entretanto, ele pode precisar de pequenos ajustes para se adaptar aos sistemas de componentes. Por exemplo, como será visto no capítulo 5, Java e COM exigem que o algoritmo trate o caso de sobrecarga de métodos. Essas adaptações ficam por conta das instanciações do algoritmo para os seus diferentes cenários.

Cabe observar que, toda vez que um *proxy* delega a execução de uma operação para o seu componente externo, o parâmetro *self* é amarrado a uma referência para o próprio componente externo, e não a uma referência para o *proxy*. Isto é, o método é executado sobre o componente externo. Assim, o mecanismo de *proxy* genérico, ao invés de implementar um esquema de delegação real, onde o parâmetro `self` seria sempre amarrado ao *proxy*, implementa apenas um esquema de reenvio automático de mensagens [Kni99].

Para não recair no mesmo problema das interfaces dinâmicas dos sistemas de componentes (alto grau de complexidade em seu uso), a linguagem de composição deve oferecer um mecanismo com o qual operações aplicadas a um *proxy*, através da sintaxe regular da linguagem, possam ser detectadas e redirecionadas para a operação `invoke`, que é oferecida pelo *proxy* através da interface `Dispatcher`. Tipicamente, isso pode ser feito em linguagens dinamicamente tipadas através de mecanismos de tratamento de exceções. Dessa forma, a linguagem de composição oferece um *açúcar sintático* para se acessar a interface `Dispatcher`.

4.2.3 Adaptador Genérico

Em uma primeira análise, pode parecer que o nosso *binding* entre uma linguagem de composição e um sistema de componentes pode trabalhar em uma única direção: a linguagem tem que ser capaz de agir como cliente dos componentes externos, mas não precisa implementar novos componentes para serem integrados ao sistema de componentes. Entretanto, para permitir a implementação de objetos de *callback*, tais como os *listeners* de JavaBeans, o *binding* tem que trabalhar nas duas direções, já que esses objetos devem ser chamados a partir de componentes externos.

Como objetos de *callback* são uma técnica amplamente utilizada em programação orientada a objetos (por exemplo, a maioria dos padrões de projeto apresentados em [GHJV95] usa essa técnica), uma linguagem sem esse tipo de suporte teria um uso extremamente limitado.

Para permitir a implementação dinâmica de objetos com a linguagem de composição, nosso padrão prescreve o uso de *adaptadores genéricos*. Um adaptador genérico é um objeto do próprio sistema de componentes que oferece uma determinada interface, e delega a implementação real das operações de sua interface para um objeto definido na linguagem de composição. Seu funcionamento é muito similar ao de um *proxy* genérico: qualquer mensagem enviada para o adaptador é reenviada para a implementação real. O adaptador também é responsável por fazer as conversões de tipos de dados necessárias entre o sistema de componentes e a linguagem de composição.

O adaptador genérico pode ser visto como uma implementação da interface `Dispatcher`, o que permite que uma única implementação de adaptador seja utilizada para instanciar adaptadores para diferentes interfaces. A ligação entre um adaptador e um objeto da linguagem de composição também é feita através de um algoritmo de *dispatching*, e usa os mesmos mecanismos reflexivos utilizados pelo *proxy* genérico, só que dessa vez no sentido inverso: um objeto do sistema de componentes delega a execução das operações para um objeto definido na linguagem de composição. Assim, a linguagem também deve oferecer mecanismos de introspecção e construção dinâmica de chamadas de métodos, isto é, os objetos nativos da linguagem de composição também devem implementar a interface `Dispatcher`.

A definição das implementações dos componentes através de uma linguagem interpretada trás duas conseqüências importantes para o nosso modelo de composição. A primeira é que as imple-

mentações dos componentes podem ser definidas e associadas dinamicamente a um adaptador. Assim, um desenvolvedor pode postergar até o instante da criação efetiva do componente a definição de sua implementação. A segunda conseqüência é que, assumindo que os objetos da linguagem de composição são compatíveis de alguma forma com a interface `ExtendedDispatcher`, as implementações de componentes definidas através dos adaptadores genéricos podem ser manipuladas em tempo de execução, permitindo a adaptação dinâmica desses componentes.

Como pode ser observado, os adaptadores e *proxies* genéricos são muito semelhantes. A diferença básica entre eles é que, enquanto um *proxy* oferece a interface `Dispatcher` para a linguagem de composição e delega as requisições de operações para o sistema de componentes, um adaptador oferece a mesma interface para o sistema de componentes e delega as requisições para um objeto implementado na linguagem de composição. Dessa forma, esses dois mecanismos fazem o papel de *pontes* entre modelos de objetos e domínios de execução diferentes.

4.2.4 Instanciação do Padrão

Ao se instanciar esse padrão de projeto a um determinado par de linguagem de composição e sistema de componentes, é necessário definir um mapeamento entre o nosso modelo de objetos, com o qual a linguagem de composição deve ser compatível, e o sistema de componentes em questão. Assim, versões específicas do *proxy* e do adaptador genéricos devem ser fornecidas para implementar esse mapeamento. Essa implementação do mapeamento representa uma instância do padrão de *binding* dinâmico, que vamos chamar simplesmente de *binding*.

Tipicamente, um *binding* define um conjunto de coerções entre os tipos de dados da linguagem de composição e do sistema de componentes, e ainda pode exigir algumas adaptações no nosso modelo de objetos. Um exemplo de adaptação que pode ser necessária é a necessidade de tratar a sobrecarga de métodos em uma mesma interface. O capítulo 5 apresenta três exemplos de *binding*.

Outros tipos de problemas podem aparecer ao instanciarmos o nosso padrão em um determinado *binding*. Por exemplo, quando o uso do mecanismo de introspecção de interfaces oferecido pelo sistema de componentes compromete o desempenho do *binding*, podemos utilizar uma técnica de armazenamento local dos dados necessários a respeito das interfaces dos componentes. Esse tipo de técnica é conhecida pelo nome de *cache*.

É interessante observar como a interface `Dispatcher` está presente em nosso padrão de *binding* com diferentes implementações e níveis de abstração:

- O mecanismo de construção de chamadas dinâmicas oferecido pelo sistema de componentes pode ser visto como uma implementação da interface `Dispatcher`.
- A linguagem de composição também deve oferecer tal interface, para que o adaptador genérico possa construir chamadas dinâmicas a métodos definidos na linguagem de composição.
- Os *proxies* genéricos implementam a interface `Dispatcher`, e a disponibilizam para ser acessada através da linguagem de composição.
- Os adaptadores genéricos também podem ser considerados como sendo implementações da interface `Dispatcher`. Só que dessa vez essa interface é acessada pelo próprio sistema de componentes.

4.3 Pontes Dinâmicas

A utilização do modelo de objetos apresentado na seção 4.1 e do padrão para *bindings* dinâmicos permite que componentes de sistemas diferentes sejam tratados de uma forma uniforme através de uma

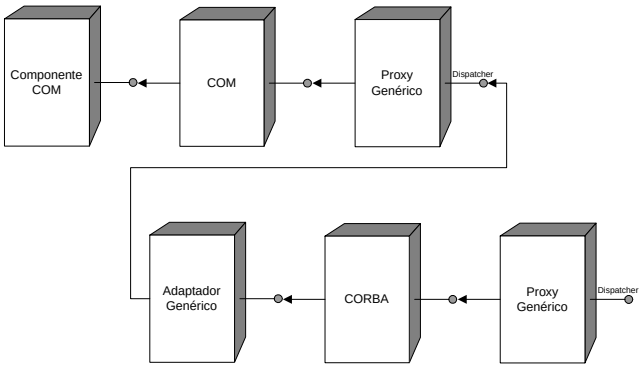

Figura 4.10: Um exemplo de ponte dinâmica.

linguagem de composição. Para isso, os *proxies* e adaptadores genéricos encapsulam vários aspectos dos sistemas de componentes.

Essa representação uniforme de componentes de sistemas diferentes possibilita a criação de pontes dinâmicas entre esses sistemas. Como um adaptador genérico pode delegar a execução das requisições recebidas para qualquer objeto do domínio da linguagem de composição, para criar pontes dinâmicas podemos *conectar* um adaptador de um determinado *binding* com um *proxy* de outro. Isto é, usando a linguagem de composição, instanciamos um *proxy* para um componente externo, e então usamos esse *proxy* como implementação para um adaptador de outro *binding*. A figura 4.10 ilustra um exemplo dessa arquitetura de integração. Nesse exemplo, um componente COM é exportado para um ambiente CORBA através de uma cadeia de objetos que implementam a interface `Dispatcher` de alguma forma. Todas as vezes que for realizada uma requisição de método para o serviço disponível no ambiente CORBA, será disparada uma seqüência de mapeamentos:

$$\text{linguagem} \Rightarrow \text{CORBA} \Rightarrow \text{linguagem} \Rightarrow \text{COM}$$

Essa seqüência de mapeamentos naturalmente acarretará em uma perda de desempenho, se for comparada com uma solução estática. Entretanto, essas pontes são criadas com um custo mínimo para o desenvolvedor da aplicação, e podem ser aplicadas em situações que não puderam ser antecipadas durante o desenvolvimento inicial de um sistema. Além disso, uma ponte dinâmica desse tipo pode ser substituída por uma solução mais eficiente quando o desempenho se mostrar crítico. Por exemplo, no caso descrito acima, uma ponte estática direta entre CORBA e COM poderia ser utilizada para substituir nossa ponte dinâmica. A seção 5.3 apresenta um exemplo completo desse tipo de integração dinâmica entre sistemas de componentes.

É importante observar que a adoção de um mecanismo de compatibilidade estrutural de tipos também desempenha um papel fundamental na criação de pontes dinâmicas, pois possibilita a verificação em tempo de execução da compatibilidade entre interfaces de objetos de sistemas diferentes.

4.4 Requisitos para a Aplicação do Modelo

A partir do modelo de composição descrito neste capítulo, podemos identificar os requisitos que um sistema de componentes e uma linguagem de *script* devem atender para que possam ser integrados ao

nosso modelo. Naturalmente, o quanto mais os seus modelos de objetos forem similares ao nosso, mais facilmente eles poderão ser integrados. Entretanto, três recursos são essencias para essa integração, por parte tanto do sistema de componentes quanto da linguagem de composição:

- capacidade de obter a descrição dos tipos dos objetos em tempo de execução;
- construção dinâmica de chamadas de métodos;
- e definição em tempo de execução de novas implementações de objetos.

A linguagem de composição ainda deve oferecer algum mecanismo que permita que ela seja usada como um açúcar sintático para o acesso à interface `Dispatcher`, isto é, as operações devem ser requisitadas através da sintaxe regular da linguagem de composição, ao invés do acesso direto ao método `invoke`.

O capítulo 5 mostra como três sistemas de componentes e uma linguagem de *script* foram integrados ao nosso modelo. Entretanto, qualquer outro sistema ou linguagem que apresente as características descritas acima pode ser utilizado.

4.5 Considerações Finais

De uma maneira resumida, podemos dizer que o modelo de composição proposto nesta tese é composto por um modelo de objetos integrador e um padrão para a implementação de *bindings* dinâmicos entre linguagens de *script* e sistemas de componentes. A definição desses dois elementos é a principal diferença entre o nosso mecanismo de composição e os outros mecanismos apresentados no capítulo 3.

Nenhum dos mecanismos de composição que vimos anteriormente define um modelo de objetos que possa ser mapeado em tempo de execução e de forma transparente a diferentes sistemas de componentes. Os demais trabalhos também não definem um padrão para a implementação de *bindings* entre linguagens de *script* e diferentes sistemas de componentes.

Apesar de várias outras linguagens de *script* trabalharem integradas a sistemas de componentes, tipicamente essa integração é feita de uma forma *ad hoc*. Neste trabalho, identificamos mecanismos que permitiram uma melhor estruturação dessa integração. A possibilidade de interoperabilidade entre diferentes sistemas de componentes foi praticamente uma conseqüência natural dessa estruturação.

Optamos por um modelo simples, porém genérico e flexível. Como nosso enfoque é o reuso de componentes, procuramos definir um modelo de objetos para o qual componentes já existentes pudessem ser mapeados mais facilmente. Também procuramos oferecer ao desenvolvedor de aplicações um mecanismo que lhe desse bastante flexibilidade para tirar o máximo proveito dos componentes disponíveis.

Assim como classificamos o nosso modelo de objetos como integrador, CORBA e COM também são considerados modelos integradores, pois possibilitam a integração de diferentes linguagens. Entretanto, esses sistemas procuram definir um modelo que oriente o desenvolvimento de componentes. Assim, a integração desses sistemas com as linguagens de programação é feita através da representação de seus modelos de objetos nas linguagens de programação, de forma a guiar o uso dessas linguagens.

Já em nosso modelo integrador, não entramos no mérito de como os componentes devem ser implementados, nem definimos um modelo de objetos para orientar essa implementação. Procuramos apenas representar os componentes dos sistemas integrados de uma maneira homogênea.

Por outro lado, procuramos oferecer um ferramental para auxiliar na construção de aplicações a partir de componentes já existentes. Mesmo assim, ainda deixamos o desenvolvedor com bastante liberdade para definir o seu estilo de composição, pois não definimos esquemas de composição tão estruturados quanto aqueles tipicamente oferecidos por linguagens de descrição de arquiteturas.

Assim como qualquer outro mecanismo que ofereça muita flexibilidade ao desenvolvedor, um mecanismo de composição baseado em nosso modelo pode ser usado de forma desordenada, levando a um sistema com uma arquitetura no mínimo confusa. Assim, é importante que um desenvolvedor tenha alguma disciplina na escolha de estilos arquitetônicos ao adotar nosso modelo de composição.

Naturalmente, espera-se que haja uma considerável perda de desempenho em *bindings* de linguagens baseados em nosso modelo, quando comparados com soluções estáticas. Entretanto, como veremos no capítulo 6, nosso modelo de composição tem se mostrado apropriado em diversos domínios de aplicação.

Capítulo 5

LuaOrb: Uma Implementação do Modelo de Composição

Neste capítulo, apresentamos o sistema LuaOrb. Esse sistema é uma implementação do modelo de composição descrito no capítulo anterior. LuaOrb utiliza a linguagem de *script* Lua [IFC96, FIC96] como seu elemento de composição, e é composto por três *bindings* diferentes, entre essa linguagem e os sistemas de componentes CORBA, COM e Java, chamados respectivamente de LuaCorba, LuaCom e LuaJava.

Esses três *bindings* são implementações do padrão de projeto para *bindings* dinâmicos apresentado no capítulo anterior e, a princípio, podem trabalhar de forma completamente independente. Entretanto, quando integrados em um mesmo ambiente, esses *bindings* permitem que um desenvolvedor tenha acesso a todos os componentes disponíveis dos três sistemas de uma maneira uniforme e integrada, podendo inclusive criar pontes dinâmicas entre os sistemas.

Para descrever o sistema LuaOrb, inicialmente faremos um resumo da linguagem Lua (seção 5.1), procurando dar destaque somente aos aspectos diretamente relacionados com este trabalho. Maiores detalhes da linguagem podem ser obtidos em [IFC96, IFC99, FIC96].

Em seguida, apresentaremos os três *bindings* implementados (seção 5.2). Apesar desses *bindings* poderem ser tratados como três ferramentas independentes, vamos apresentá-los em paralelo devido às suas semelhanças. Também apresentaremos um exemplo para ilustrar como LuaOrb possibilita a interoperabilidade entre diferentes sistemas de componentes (seção 5.3).

Por último, faremos uma breve comparação dos *bindings* que compõem o sistema LuaOrb com outras ferramentas de *script* para sistemas de componentes de software.

5.1 A Linguagem Lua

Lua é uma linguagem de *script* de uso geral, que alia uma sintaxe simples a recursos de reflexividade e de descrição de dados. Lua foi projetada para trabalhar embutida em aplicações, funcionando como um mecanismo de extensão e configuração destas. Assim, Lua não tem a noção de programa principal, e sempre trabalha acoplada a uma aplicação *hospedeira*.

Uma das principais diretrizes de projeto de Lua foi manter a linguagem simples. Essa diretriz foi motivada por duas razões. A primeira era permitir um aprendizado mais fácil da linguagem, para que desenvolvedores principiantes ou usuários finais um pouco mais especializados pudessem utilizá-la com mais facilidade. A outra razão era possibilitar a implementação de um interpretador pequeno para a linguagem, para que as aplicações hospedeiras não fossem sobrecarregadas ao embutir o interpretador da linguagem.

Lua é disponibilizada através de uma biblioteca de funções C a ser ligada com as aplicações hospedeiras. Essa biblioteca oferece uma interface de programação (API) através da qual uma aplicação hospedeira pode interagir com o interpretador Lua.

5.1.1 Características Básicas

Pelo seu lado "tradicional", Lua é uma linguagem procedural com as estruturas de controle usuais (*while*,*if*, *for*, etc.), e com definições de funções com parâmetros e variáveis locais, entre outros recursos típicos de linguagens procedurais.

Como uma linguagem interpretada, Lua disponibiliza diretamente o seu interpretador através da própria linguagem. Para isso, ela oferece as funções `dostring` e `dofile`, que executam trechos de código Lua diretamente sobre o ambiente global da linguagem:

```
> a = 3
> print(a)
3
> dostring("a = a + 2")
> print(a)
5
```

Lua é dinamicamente tipada e, assim, variáveis não têm tipos, mas seus valores sim. O tipo do valor associado a uma variável pode ser consultado a qualquer instante através da função `type`:

```
> a = 5
> print(type(a))
number
> a = "teste"
> print(type(a))
string
```

Funções são tratadas como valores de primeira classe, isto é, elas podem ser manipuladas como qualquer outro tipo de valor da linguagem:

```
> f = print
> f("Oi")
Oi
> print = 3 + 4
> f(print)
7
```

Assim, o nome de uma função nada mais é do que uma variável cujo valor é uma função. A associação de nomes a funções pode ser feito basicamente de duas formas:

```
add = function (x,y) return  x + y end
```

e

```
function add(x,y)
   return x + y
end
```

A primeira é a forma mais básica de se fazer essa associação; já a segunda forma é apenas um açúcar sintático para a primeira.

O corpo de uma função pode fazer referência às suas próprias variáveis locais, que inclui seus parâmetros, e às variáveis globais, desde que estas não sejam redefinidas por variáveis locais de funções envolventes. Uma função não pode acessar uma variável local de uma função envolvente, pois tais variáveis podem não existir mais quando a função for chamada. Entretanto, uma função pode acessar o *valor* de uma variável local de uma função envolvente, usando um mecanismo denominado *upvalue*.

Um *upvalue* é similar a uma variável, mas seu valor é congelado quando a função na qual ele aparece é instanciada. O nome usado em um *upvalue* pode ser o nome de qualquer variável visível no ponto onde a função foi definida, precedido pelo caráter especial `%`.

Através do mecanismo de *upvalue*, Lua permite a criação de *closures*, de forma bem semelhante a linguagens funcionais. Por exemplo, considere a seguinte função:

```
function criaIntervalo(a,b)
   return function (x)
             return (x >= %a and x <= %b)
          end
end
```

Essa função gera outras funções que podem ser usadas para testar se um valor está dentro de um determinado intervalo. O intervalo da função criada é determinado pelos parâmetros `a` e `b`. A função `criaIntervalo` pode ser usada da seguinte forma:

```
> testaIntervalo = criaIntervalo(3,6)
> print(testaIntervalo(4))
1
> print(testaIntervalo(7))
nil
```

O tipo *nil* é um tipo de dado especial em Lua que só pode assumir um único valor, também denominado `nil`. Esse tipo de dado serve para indicar variáveis não inicializadas e também é utilizado para representar o valor *falso* de expressões booleanas (qualquer outro valor é considerado *verdadeiro*).

O tipo *userdata* é oferecido para permitir que ponteiros C sejam armazenados em variáveis Lua. Ele corresponde ao tipo `void*` de C e não tem nenhuma operação pré-definida sobre ele em Lua, exceto atribuição e teste de igualdade.

Lua oferece um mecanismo unificador de estruturas de dados denominado *tabelas*. O tipo tabela implementa um array associativo, isto é, arrays que podem ser indexados não apenas por números, mas por qualquer outro valor (exceto `nil`). Assim, esse tipo pode ser usado para representar tanto arrays ordinários, quanto tabelas de símbolos, conjuntos, registros, entre outros. O tipo tabela é o principal mecanismo de estruturação de dados em Lua. Por exemplo, para representar registros, Lua usa o nome do campo como um índice. A linguagem facilita essa representação oferecendo `a.campo` como um açúcar sintático para `a["campo"]`.

Tabelas precisam ser explicitamente criadas antes de serem usadas. Para isso, existem os *construtores* de tabelas, que podem trabalhar de duas formas básicas. A primeira cria uma tabela vazia:

```
> t = {}
```

A segunda forma permite que seja criada uma tabela com alguns campos já inicializados:

```
> t1 = {1, 1, 2, 3, 5, 8}
> ponto = {x = 0.3, y = 0.4}
```

Durante o ciclo de vida de uma tabela, campos podem ser adicionados ou removidos livremente. Por exemplo, o construtor definido acima para criar a tabela associada à variável `ponto` é equivalente a:

```
> ponto = {}
> ponto.x = 0.3
> ponto.y = 0.4
```

Tabelas também podem ser usadas para organizar o espaço de nomes de Lua. Como funções são valores de primeira classe, tabelas podem ter campos que contenham funções:

```
> console = {}
> console.out = print
> console.out("Teste")
Teste
```

Apesar de Lua não ser originalmente uma linguagem orientada a objetos, objetos também podem ser representados através de tabelas. A forma `t:f(x)` é um açúcar sintático para `t.f(t,x)`, que chama o "método" `f` da tabela `t` passando ela mesma como primeiro parâmetro. Por exemplo, a tabela `ponto` pode ser estendida com uma operação para imprimir o seu conteúdo:

```
> ponto.dump = function (self) print(self.x, self.y) end
> ponto:dump()
0.3 0.4
```

Observe que tabelas são objetos, e não valores. Variáveis não podem conter tabelas, somente referências para elas. Atribuições, passagem de parâmetros e retornos de funções sempre manipulam referências para tabelas, e não implicam em nenhum tipo de cópia.

Lua oferece a função `call` que permite a construção dinâmica de uma chamada de função. Por exemplo, as duas chamadas a seguir são equivalentes:

```
> a = add(3,4)
> a = call(add,{3,4})
```

O segundo parâmetro que a função `call` recebe é uma tabela com os argumentos que devem ser passados para a função sendo chamada. Em seu modo normal, `call` retorna diretamente todos os resultados da função chamada. Mas, como uma função em Lua pode ter mais de um valor de retorno, a função `call` pode ser chamada com o argumento opcional de modo `"p"` que especifica que todos os valores de retorno devem ser agrupados em uma tabela. Por exemplo, considere uma função `extremos` que recebe como parâmetro um grupo de números e retorna o menor e o maior valor desse grupo; essa função poderia ser chamada de três maneiras diferentes:

```
> a, b = extremos(16,7,98,23)           --> a = 7, b = 98
> a, b = call(extremos,{16,7,98,23})    --> a = 7, b = 98
> t = call(extremos,{16,7,98,23}, "p")  --> t = {7, 98, n = 2}
```

A última forma é interessante quando não sabemos a priori quantos valores vão ser retornados por uma função.

Lua também permite a definição de funções com número variável de parâmetros. Para isto, basta incluir o símbolo `...` ao final da lista de parâmetros da função, que assim os demais argumentos serão agrupados em um parâmetro implícito chamado `arg`. Como um exemplo de definição de uma função com número variável de parâmetros, podemos usar a função `extremos` do exemplo anterior:

```
function extremos(...)
   return call(min,arg), call(max,arg)
end
```

Observe que as funções `min` e `max` também recebem um número variável de parâmetros, e por isso é necessário usar a função `call` para chamá-las nesse caso.

5.1.2 Mecanismos de Extensão

Lua também oferece um conjunto de mecanismos que permitem a extensão de sua funcionalidade para adaptar a linguagem a diferentes domínios de aplicação. O seu mecanismo mais básico de extensão é através de sua API C. Essa API permite que a aplicação hospedeira registre novas funções no ambiente Lua. Tipicamente, essas funções são implementadas em C e permitem o acesso aos recursos exportados pela aplicação. Apesar desse mecanismo de extensão ser muito útil, ele não estende propriamente a linguagem, mas somente o conjunto de funções disponíveis para ela.

Lua oferece um outro mecanismo de extensão bem mais poderoso chamado *métodos de tag*, que permite alterar a própria semântica da linguagem. Um método de *tag* é uma função definida pelo programador que é chamada em pontos específicos durante a execução de um programa Lua, permitindo que o programador altere o comportamento padrão do interpretador nesses pontos. Cada um desses pontos é chamado de *evento*.

Um método de *tag* chamado em qualquer evento específico é selecionado de acordo com o *tag* dos valores envolvidos no evento. Todos os valores em Lua têm um *tag* associado. Entretanto, somente valores dos tipos `userdata` e tabela podem ter essa associação redefinida.

Tags podem ser criados com a função `newtag`, e a função `tag` retorna o *tag* de um dado valor. Para mudar o *tag* de uma tabela, existe a função `settag`. O *tag* de um *userdata* só pode ser alterado através da API C de Lua, isto é, ele só pode ser alterado pela aplicação hospedeira. A função `settagmethod` substitui o método de *tag* associado com um determinado par *(tag, evento)*. Já a função `gettagmethod` recebe um *tag* e o nome de um evento, e retorna o método associado a esse par.

Através do mecanismo de métodos de *tag*, é possível estender Lua com novos tipos de dados, ou até mesmo alterar o seu modelo de objetos. A partir dos eventos que podem ser redefinidos para um determinado *tag*, é possível representar diferentes modelos de objetos em Lua simultaneamente (diferentes *tags* representando diferentes modelos).

Neste trabalho, estamos particularmente interessados em estender o modelo de objetos de Lua de tal forma que seja possível representar o modelo proposto no capítulo 4. Para facilitar a comparação entre o modelo de objetos de Lua e o nosso modelo, vamos representar alguns aspectos do modelo de Lua através de uma notação em OMG IDL (o Apêndice A apresenta a definição completa em IDL da parte do modelo de objetos de Lua que é relevante para esse trabalho). Essa representação proporciona uma interpretação um pouco mais orientada a objetos da arquitetura de Lua.

Inicialmente, podemos representar o tipo tabela através da interface `LuaObject` (figura 5.1). Essa interface oferece as operações `get` e `set` que permitem consultar e atribuir valores aos campos de uma tabela. A interface `LuaFunction` representa os valores do tipo função, e objetos com essa interface, assim como qualquer outro, podem ser associados a campos de uma tabela. Assim, uma expressão da forma `t:f(x)` poderia ser vista como:

```
> metodo = t:get("f")
> params_out = metodo:call({t,x})
```

Observe que uma função em Lua pode ter mais de um valor de retorno.

Para representar o mecanismo de métodos de *tag* associados a tabelas, definimos a interface `LuaTag`, que funciona como um meta-objeto associado a um objeto Lua (figura 5.2). Essa interface

```
interface LuaObject {
   any get (in any index);
   void set (in any index, in any value);

   LuaTag getTag();
   void setTag (in LuaTag tag);
};

interface LuaFunction {
   ListOfAny call (in ListOfAny params_in);
};
```

Figura 5.1: As interfaces `LuaObject` e `LuaFunction`.

oferece operações para manipular os métodos de *tag* associados aos eventos que podem ser gerados por acessos a um objeto em Lua. Estamos particularmente interessados em três desses eventos:

SetTable — esse evento é gerado todas as vezes que a operação `set` da interface `LuaObject` é acionada, isto é, sempre que se tenta atribuir um valor a um campo de uma tabela Lua. O tratador desse evento recebe o objeto (tabela), o índice do campo e o valor a ser atribuido ao campo (interface `SetTableMethod`).

GetTable — esse evento é gerado todas as vezes que a operação `get` da interface `LuaObject` é acionada, isto é, sempre que se tenta consultar o valor de um campo de uma tabela. O tratador desse evento recebe o objeto (tabela) e o índice do campo a ser consultado, e retorna o valor associado ao campo, se houver algum (interface `GetTableMethod`).

Index — esse evento é muito parecido com o evento *GetTable*. Entretanto, ele somente é gerado quando a operação `get` é acionada e não existe nenhum valor associado ao campo consultado, isto é, sempre que se tenta consultar um campo cujo valor é `nil`. O tratador desse evento também recebe o objeto e o índice do campo a ser consultado, e pode retornar um valor, como se este estivesse associado ao campo. Como a interface do tratador desse evento é igual ao do tratador de *GetTable*, podemos definir a interface `IndexMethod` como um sinônimo para a interface `GetTableMethod`:

```
typedef GetTableMethod IndexMethod;
```

Redefinindo o tratador de um desses eventos para um determinado *tag*, podemos associar qualquer semântica desejada à atribuição e consulta de campos de uma tabela. A associação de um meta-objeto a um objeto Lua é feita através das operações `setTag` e `getTag` da interface `LuaObject`. Observe que essa associação pode ser refeita a qualquer momento da execução de um programa Lua.

Através da redefinição dos tratadores dos eventos de *SetTable*, *GetTable* e *Index*, é possível implementar diferentes esquemas de compartilhamento de comportamento em Lua. Como a associação desses métodos de *tag* é feita por objeto, podemos ter diferentes esquemas de compartilhamento coexistindo simultaneamente em Lua.

5.2 Uma Visão Geral do Sistema LuaOrb

Os três requisitos básicos que uma linguagem deve atender para pode ser usada com nosso modelo de composição já são diretamente satisfeitos por Lua: os tipos de todos os valores podem ser consulta-

```
interface LuaTag {
   void setSetTableMethod (in SetTableMethod method);
   SetTableMethod getSetTableMethod();
   void setGetTableMethod (in GetTableMethod method);
   GetTableMethod getGetTableMethod();
   void setIndexMethod (in IndexMethod method);
   IndexMethod getIndexMethod();
};

interface SetTableMethod {
   void call (in LuaObject self, in any index, in any value);
};

interface GetTableMethod {
   any call (in LuaObject self, in any index);
};
```

Figura 5.2: As meta-interfaces do modelo de objetos de Lua.

dos em tempo de execução; a função `call` permite a construção dinâmica de chamadas de funções e, conseqüentemente, de métodos; os objetos em Lua podem ser estendidos com novos métodos em qualquer instante de seu ciclo de vida, permitindo que um objeto possa ter seu tipo e sua implementação construídos em tempo de execução. Além disso, devido a flexibilidade que o modelo de objetos de Lua oferece através de seus mecanismos reflexivos, é possível estender a linguagem para se adequar de uma forma mais completa ao nosso padrão de *binding* dinâmico. Como os métodos de *tag* vistos anteriormente podem ser usados para interceptar e controlar todos os acessos feitos a um objeto Lua, podemos utilizá-los para implementar o nosso algoritmo de *dispatching*, assim como o esquema de delegação de nosso modelo. Todo esse mapeamento pode ser feito preservando a sintaxe regular da linguagem para chamadas de métodos.

De acordo com os mecanismos de metaprogramação descritos na seção 2.5, os sistemas CORBA, COM e Java também atendem aos requisitos do nosso modelo de composição. Essa compatibilidade entre nosso modelo e esses três sistemas não é acidental, pois nosso modelo foi definido, em parte, pela observação das características comuns a esses sistemas. Uma vez verificado que os sistemas e a linguagem de composição escolhidos são compatíveis com nosso modelo, precisamos descrever como instanciamos o nosso padrão de projeto de *bindings* dinâmicos para Lua e esses três sistemas de componentes. Essa descrição consiste essencialmente da representação dos mecanismos de *proxy* genérico e adaptador genérico em Lua, e do mapeamento da interface `Dispatcher` para os sistemas de componentes.

Inicialmente, vamos descrever como os mecanismos de *proxy* e adaptador genéricos podem ser implementados através de Lua. Assumimos que as interfaces dinâmicas dos sistemas de componentes podem ser representadas através da interface `Dispatcher`, e assim descrevemos a base da implementação dos três *bindings* de uma maneira uniforme. Essa descrição uniforme é apropriada, pois os *proxies* e adaptadores de cada *binding* são tratados praticamente da mesma forma.

Entretanto, sempre que for necessário, vamos analisar aspectos específicos de cada *binding*. Alguns aspectos da implementação dos *bindings* são descritos com o auxílio de Lua para facilitar a sua compreensão, apesar dos *bindings* reais terem sido implementados diretamente em C/C++ por motivos de desempenho.

Em seguida, vamos apresentar de forma simplificada como a interface `Dispatcher` pode ser

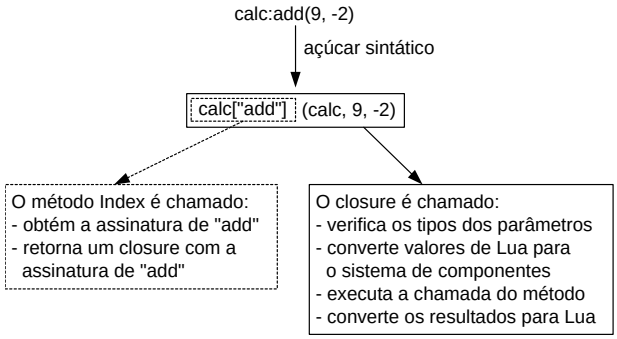

Figura 5.3: Uma chamada de método através de um *proxy*.

mapeada para os diferentes sistemas de componentes, sem abordar todos os detalhes de implementação. Apesar dos três *bindings* que compõem o sistema LuaOrb seguirem o mesmo padrão de projeto, eles não chegam a compartilhar código entre suas implementações. As implementações reais dos *bindings* foram feitas em C/C++, e acessam diretamente as interfaces de programação dos sistemas de componentes. Detalhes mais específicos sobre as implementações de cada *binding* podem ser vistos em [CIR99b, CIR97, CRI97, ICR98, RIC98, CI99, CIR99a].

5.2.1 *Proxy* Genérico

Para ilustrar como desejamos acessar um componente externo através de Lua, considere a seguinte classe Java:

```
public class Calc {
   public float add (float a, float b) {
      return a + b;
   }
}
```

Agora suponha que uma variável Lua, chamada `calc`, referencia um *proxy* para uma instância dessa classe. Chamadas ao método `add` devem ser feitas como se `calc` fosse um objeto Lua regular:

```
> print(calc:add(9,-2))
7
```

Além disso, campos devem poder ser acrescentados e substituídos em `calc` da mesma forma como isso pode ser feito com um objeto Lua regular. Entretanto, essas alterações não podem afetar o componente externo representado pelo *proxy*, devido a restrições impostas por sistemas de componentes como CORBA, COM e Java.

Para implementar esse esquema em Lua, precisamos apenas redefinir o método de *tag* para o evento *Index* do objeto *proxy*. Se não alterarmos os tratadores dos eventos *SetTable* e *GetTable*, o comportamento original da linguagem já se encarregará de oferecer a possibilidade de acrescentar e redefinir métodos em um *proxy*. O evento de *Index* vai ser acionado apenas quando se tentar acessar um campo que o próprio *proxy* não ofereça. Nesse caso, o componente externo deve ser consultado para ver se ele é capaz de responder à requisição.

A figura 5.3 ilustra como uma chamada de método é realizada através de um *proxy* em nossos *bindings*. A chamada de método `calc:add`, que está no topo da figura, pode ser reescrita sem o açúcar

sintático de Lua. O primeiro passo da chamada, representado pela caixa tracejada, ocorre quando a expressão `calc["add"]` é avaliada. Como um *proxy* tem um *tag* especial, essa expressão gera um evento de *Index*, que chama o método de *tag* correspondente. Esse método é responsável pelos dois primeiros passos do algoritmo de *dispatching* de nosso *binding*: ele procura na interface do componente externo um método com nome igual ao índice requisitado e obtém sua assinatura. Caso o componente possua uma operação com o nome fornecido, o resultado devolvido pelo método do evento de *Index* é um *closure* que encapsula as informações necessárias sobre a operação requisitada.

Uma versão simplificada da implementação desse método de *tag* poderia ser feita da seguinte forma:

```
function index_method (table, index)
   local interface = table.delegatee:getInterface()
   local signature = interface:getSignature(index)
   if signature then
      return makeClosure(signature)
   else
      return nil
   end
end
```

O campo `delegatee` da tabela referencia o componente externo representado pelo *proxy*. Se o componente possuir uma operação com nome igual ao índice fornecido, a função retorna um *closure*, que é efetivamente criado pela função `makeClosure`. Esse *closure* será responsável por concluir o algoritmo de *dispatching*. A figura 5.4 mostra um esboço da implementação do *closure* criado.

Quando for chamado, o *closure* irá verificar se os argumentos recebidos têm tipos compatíveis com os parâmetros formais da operação. Se todos os argumentos foram compatíveis, a requisição é efetivamente realizada, através da operação `invoke` da interface `Dispatcher` (seção 4.1). Para simplificar o esboço do algoritmo, assumimos que todas as conversões entre os valores de Lua e do sistema de componentes são realizadas internamente pelo método `invoke`, tanto para os parâmetros quanto para os resultados da operação.

O esboço de nosso algoritmo também ilustra como tratamos as exceções sinalizadas pela operação requisitada. Para tratar suas exceções, o próprio *proxy* deve implementar um método chamado `exception_handler`. Esse método recebe como parâmetros o nome da operação que falhou e um objeto que descreve a exceção sinalizada. Nesse método, o *proxy* pode dar o tratamento desejado à exceção, e até mesmo definir um valor de retorno apropriado para a chamada que falhou. Se o *proxy* não definir uma implementação própria para o método `exception_handler`, o *binding* fornece um tratamento pré-definido, que gera um erro Lua. Esse esquema para o tratamento das exceções sinalizadas pelos componentes externos foi baseado no esquema já utilizado em Lua.

Apesar de toda a explicação anterior ter sido baseada em um exemplo com um objeto Java, a mesma se aplicaria a um objeto CORBA que tivesse a interface:

```
interface Calc {
   float add (in float a, in float b);
};
```

Ou a um componente COM que oferecesse a interface:

```
interface Calc : IUnknown {
   HRESULT add ([in] float a, [in] float b,
                [out, retval] float *pResult);
};
```

```
function makeClosure(signature)
  return function (self, ...)
    if checkParams(arg, %signature:getParamsIn()) then
       local name = %signature:getName()
       local ok,params_out,exception = self.delegatee:invoke(name,arg)
       assert(ok)
       if exception then
          return self:exception_handler(name,exception)
       else
          return params_out
       end
    else
       error("Tipo de Parâmetro Inválido")
    end
  end
end
```

Figura 5.4: A função `makeClosure`.

Isso porque, de acordo com a abordagem de compatibilidade estrutural, essas três interfaces são compatíveis.

Apesar do uso de *proxies* de objetos com essas três interfaces ser o mesmo em Lua, a forma de criá-los é diferente. Cada um dos *bindings* oferece uma função específica para criar *proxies* explicitamente. Essa diferença é necessária porque de alguma forma é preciso indicar em qual sistema de componentes deve-se procurar ou instanciar um novo objeto. Por exemplo, poderíamos criar um *proxy* para cada versão da interface `Calc` da seguinte forma:

```
calc1 = LuaOrb.JavaProxy("Calc")
calc2 = LuaOrb.ComProxy("CalcTst.CalcTst.1")
calc3 = LuaOrb.CorbaProxy({interface = "Calc",
                           ior_file = "calc.ref"})
```

No caso de Java, cria-se uma nova instância da classe `Calc` e, a partir dessa instância, cria-se o *proxy* Lua (`calc1`). Para o sistema COM, o procedimento é praticamente o mesmo: é fornecido o identificador da classe COM desejada (`CalcTst.CalcTst.1`), e o *proxy* é criado a partir de uma nova instância da classe (na verdade, a implementação da classe COM é que vai decidir se vai ser criada uma nova instância ou se uma única instância é compartilhada por todos os clientes). Já para o sistema CORBA, precisamos fornecer dois tipos de informação para criar um *proxy*: a interface do objeto e o seu IOR (seção 2.2.3). Nesse exemplo, o IOR do servidor é obtido a partir de um arquivo que deve conter a representação do IOR em forma de *string*. Tipicamente, o servidor CORBA já vai estar executando quando criarmos um *proxy* para ele. Uma vez criados, esses três *proxies* são utilizados da mesma forma.

Além da criação explícita, *proxies* podem ser criados implicitamente. Isso ocorre basicamente quando alguma operação de um componente retorna uma referência para um outro componente externo, ou quando um componente implementado em Lua recebe um parâmetro que é uma referência para um outro componente. Nesses casos, os *bindings* criam automaticamente *proxies* para representar esses objetos.

Apesar de não ter incluído na descrição da implementação do *proxy* genérico o tratamento de atributos definidos nas interfaces dos componentes externos, podemos facilmente introduzir esse tra-

tamento através de métodos de *tag* para os eventos de *SetTable* e *GetTable* dos *proxies* genéricos. Dessa forma, atributos são tratados como se fossem campos regulares de uma tabela Lua.

5.2.2 Adaptador Genérico

O mecanismo de adaptador genérico é utilizado para permitir que um objeto Lua sirva como implementação de um componente que será exportado para o ambiente de um determinado sistema de componentes. Por exemplo, gostaríamos que o objeto Lua:

```
> calc = { add = function(self, x, y) return x + y end }
```

pudesse ser usado para implementar as três versões da interface `Calc`, vistas na seção anterior. Na realidade, o objeto `calc` pode ser usado simultaneamente como uma implementação da interface `Calc` nos três sistemas de componentes.

O primeiro passo para disponibilizar um objeto Lua como um componente de um sistema é passar esse objeto para o *binding*, para que este possa criar um adaptador que será efetivamente exportado para o sistema de componentes. O adaptador criado irá receber as requisições de operações através do sistema de componentes e irá repassá-las para o objeto Lua.

A princípio, a criação de um adaptador pode ser requisitada para um *binding* de uma forma explícita ou implícita. A criação explícita é feita através de uma função com essa finalidade específica, que é exportada para Lua pelo próprio *binding*. Por exemplo, o objeto `calc` pode ser registrado no *binding* CORBA da seguinte forma:

```
> calc_adapter = LuaOrb.CorbaAdapter("Calc", calc)
```

A função `LuaOrb.CorbaAdapter` recebe o nome da interface com a qual o adaptador deve ser exportado para o sistema de componentes e o objeto que deve ser usado como implementação. Como resultado, essa função retorna uma referência para o adaptador criado, para que ele também possa ser manipulado diretamente através de Lua. Já a criação implícita é feita todas as vezes que um objeto Lua deve ser convertido pelo *binding* para um objeto do sistema de componentes, seja na passagem de parâmetros ou no retorno de resultados.

Podemos considerar que o adaptador genérico oferece a interface `Dispatcher` para ser chamada pelo sistema de componentes no qual ele se registra. Mais uma vez, vale lembrar que o adaptador genérico funciona no sentido contrário do *proxy*: ele recebe uma requisição de operação através do sistema de componentes e a repassa para um objeto Lua.

A figura 5.5 mostra um esboço de uma possível implementação em Lua do método `Dispatcher::invoke` para o adaptador genérico. Primeiro, o adaptador deve verificar se o objeto Lua, para o qual ele deve delegar suas requisições, tem um método com o mesmo nome da operação requisitada. Em seguida, ele monta uma tabela com os argumentos que devem ser passados para o método, começando pela referência para o próprio objeto Lua. Depois, ele efetua a chamada do método através da função `call`. Os resultados da operação são agrupados em uma tabela (`params_out`). Se a operação retornar uma exceção, esta é repassada para o sistema de componentes através do parâmetro de saída correspondente da operação `invoke`. Para indicar que um valor retornado pela função `call` deve ser tratado pelo *binding* como sendo uma exceção, usamos um *tag* especial para marcar esses valores (`BindingException`).

Se a operação executou com sucesso, o adaptador verifica a compatibilidade de tipos entre os resultados do método e os tipos formais do resultado e dos parâmetros de saída definidos na assinatura da operação na interface do componente. Por fim, os resultados são retornados para o cliente que fez a requisição da operação. Novamente, para simplificar o esboço do algoritmo de *dispatching*, não incluímos as conversões entre os tipos de dados dos parâmetros e resultados da operação.

```
function Adapter:invoke(method_name, params_in)
   local method = self.delegatee[method_name]
   if type(method) == "function" then
      local args = { self.delegatee }
      for i = 1, params_in.n do
         args[i+1] = params_in[i]
      end
      local params_out = call(method, args, "p")
      local interface = self:getInterface()
      local sig = interface:getSignature(method_name)
      if tag(params_out[1]) == BindingException then
         assert(checkException(params_out[1], sig:getExceptions()))
         return true, {}, params_out[1]
      else
         assert(checkParams(params_out, sig:getParamsOut()))
         return true, params_out, nil
      end
   else
      return nil   -- operação não implementada
   end
end
```

Figura 5.5: Uma implementação para o método `invoke` em Lua.

É importante observar que, enquanto o *proxy* genérico verifica a compatibilidade de tipos dos argumentos recebidos de Lua e retorna os resultados da operação diretamente para Lua sem verificar seus tipos, o adaptador genérico faz o inverso. Isso ocorre porque os métodos definidos em Lua não possuem uma assinatura, o que não permite que façamos uma verificação dos tipos dos valores que devem ser passados para Lua. Basicamente, podemos definir que um *binding* de Lua verifica o tipo de um valor todas as vezes que o fluxo de informação é no sentido

$$\text{Lua} \Rightarrow \text{sistema de componentes}$$

e ele não faz essa verificação de tipos no sentido contrário. Nesse último caso, fica por conta da aplicação verificar em Lua se está recebendo um dado do tipo certo. Essa característica não faz parte do padrão de *binding* dinâmico, mas é típica dos *bindings* de Lua.

Como a implementação de um componente através de um adaptador genérico é dada por um objeto Lua, e tipicamente esses objetos Lua podem ter sua implementação alterada em tempo de execução, LuaOrb possibilita a extensão dinâmica da implementação de componentes servidores. Esse recurso pode ser bem útil durante a fase de prototipação de um sistema, ou em outros sistemas em que se deseje fazer a atualização das implementações de componentes, sem interromper suas execuções. Já exploramos esse tipo de recurso no *binding* CORBA [MRI99]. Com a possibilidade de manipular as descrições de interfaces contidas no Repositório de Interfaces de CORBA, também é possível evoluir em conjunto e dinamicamente a implementação e a interface de um componente. Entretanto, ainda fica faltando um mecanismo que permita a atualização da visão que todos os clientes têm do componente. Assim, esse mecanismo só funciona adequadamente para determinadas evoluções ou em sistemas bem controlados.

5.2.3 O Mapeamento da Interface Dispatcher

Nas seções anteriores, mostramos como implementar os mecanismos de *proxy* genérico e adaptador genérico, assumindo que o sistema de componentes oferece interfaces de programação compatíveis com as interfaces definidas em nosso modelo de composição. Apesar de nenhum dos três sistemas de componentes integrados pelo LuaOrb ser diretamente compatível com as interfaces do nosso modelo, podemos representar essas interfaces através dos mecanismos de meta-programação oferecidos pelos sistemas de componentes.

Os mecanismos de introspecção oferecidos por CORBA, COM e Java são capazes de fornecer todas as informações sobre as interfaces dos componentes que são necessárias para o nosso modelo. O mesmo ocorre com as interfaces dos sistemas de componentes para construção de chamadas dinâmicas, que já são conceitualmente compatíveis com a interface `Dispatcher`, pelo menos sob o ponto de vista do cliente de um componente. Essas interfaces, nos casos de CORBA e COM, também são compatíveis com a interface `Dispatcher` para a implementação de um componente. Entretanto, como já foi visto na seção 2.5, Java não oferece diretamente um mecanismo compatível com a interface `Dispatcher` para a implementação de objetos.

Para contornar esse problema na implementação de LuaJava, adotamos uma técnica baseada na classe `ClassLoader` oferecida por Java. Essa classe permite que novas classes sejam carregadas pela máquina virtual de Java em tempo de execução. Com o auxílio dessa classe, todas as vezes que o *binding* precisar criar uma instância de um novo tipo que será implementado pela linguagem de composição, o código binário da classe correspondente será gerado em memória e carregado na máquina virtual de Java (tudo isso é feito em tempo de execução). Apesar do *binding* criar classes diferentes para cada tipo de objeto, todas essas classes seguem um mesmo modelo de implementação: para cada método da interface do objeto, é gerado um método na nova classe que delega para um objeto Lua a sua execução. Assim, ao contrário dos *bindings* LuaCorba e LuaCom, LuaJava não trabalha com uma implementação única para os adaptadores genéricos.

Apesar de utilizar uma técnica bem diferente dos outros dois *bindings*, a solução de LuaJava para a implementação dos adaptadores genéricos apresenta praticamente o mesmo resultado final. Ao criar uma nova instância de adaptador genérico, LuaJava apresenta uma maior perda de desempenho do que os outros dois *bindings*. Entretanto, dependendo de como o objeto for utilizado, essa perda pode ser compensada, pois as chamadas de seus métodos não precisam fazer tantas consultas à interface sendo implementada. A única limitação da solução de LuaJava, quando comparada com LuaCorba e LuaCom, é que ela não permite uma evolução conjunta da interface e implementação de um componente em tempo de execução.

Uma outra diferença de LuaJava para os demais *bindings* é que LuaJava pode usar como base para a implementação de um adaptador genérico tanto uma interface quanto uma classe Java já existente, bastando para isso passar como parâmetro para a função `LuaOrb.JavaAdapter` o nome de uma interface ou de uma classe. Nesse último caso, a classe gerada para implementar o adaptador genérico é definida como uma subclasse da classe já existente. Cada método dessa subclasse verifica se pode delegar a sua execução para o objeto Lua associado; caso este não possua um método para tratar a chamada, a execução da operação é delegada para a super-classe.

Ao contrário dos *bindings* LuaCom e LuaJava, as consultas realizadas por LuaCorba para obter as descrições das interfaces de seus componentes tipicamente envolvem requisições remotas ao Repositório de Interfaces. Para minimizar o impacto que essas consultas podem ter no seu desempenho, LuaCorba utiliza uma técnica de *cache* para armazenar localmente as interfaces já consultadas.

Como mencionamos anteriormente, precisamos fazer algumas adaptações em nosso modelo de objetos e em nosso padrão de *binding* para podermos mapeá-los para um determinado sistema de componentes. Nos *bindings* que implementamos, uma das adaptações mais significativas foi a que fizemos no algoritmo de *dispatching* do *proxy* genérico para resolver a sobrecarga de métodos em

```
[object, uuid(69E133E2-E67E-11D2-9210-444553540000), dual]
interface IPasswordTst : IDispatch {
   HRESULT validate([in] BSTR user, [in] BSTR passwd,
                    [out, retval] VARIANT_BOOL* result);
};
```

Figura 5.6: A interface `IPasswordTst`.

COM e Java. Nos *bindings* com esses dois sistemas, o método de *tag* para o evento *Index*, responsável pelos dois primeiros passos de nosso algoritmo, seleciona uma lista com as assinaturas dos métodos candidatos; e o *closure* resolve qual método chamar em função dos tipos dos argumentos. O primeiro método cujos parâmetros formais casarem com os argumentos recebidos é o escolhido. No lado do adaptador genérico, o tratamento da sobrecarga de operações fica por conta do objeto Lua. Como assumimos que esse objeto só oferece um único método associado a um determinado nome, o adaptador genérico repassa os parâmetros e fica por conta do objeto Lua tratá-los de forma adequada.

5.3 Um Exemplo de Interoperabilidade

Para ilustrar como podemos criar pontes dinâmicas entre os *bindings* que compõem o LuaOrb, vamos utilizar um exemplo que integra objetos dos três sistemas de componentes. Nesse exemplo, usamos o pacote AWT de Java para criar um console através do qual um usuário informa o seu nome e sua senha para ter acesso a uma determinada aplicação Java. Usamos um componente COM, que está executando em outra estação de trabalho, para fazer a validação da senha, e um servidor CORBA para fazer a conexão entre o console Java e o componente COM em um ambiente distribuído.

A figura 5.6 apresenta a interface em COM IDL do componente validador de senhas. O primeiro passo da construção do nosso exemplo é disponibilizar esse componente COM em um ambiente CORBA, através do LuaOrb. Para isso, precisamos definir uma interface CORBA compatível estruturalmente com a interface `IPasswordTst`, que vamos chamar de `Login`:

```
interface Login {
   boolean validate(in string user, in string passwd);
};
```

Uma vez essas interfaces definidas, podemos implementar o primeiro *script* de nosso exemplo (figura 5.7). Na linha 1, criamos um *proxy* Lua para o componente COM; na linha 2, criamos um servidor CORBA associado a esse *proxy*. Para disponibilizar esse servidor em um ambiente CORBA, utilizamos o serviço de nomes de CORBA [OMG97c]. Nas linhas 4-5, criamos um *proxy* para o servidor de nomes (nesse exemplo, o serviço de nomes está executando na máquina `Host1` e conectado na porta 7777). Finalmente, na linha 6, registramos o servidor `login_server` no serviço de nomes, com o nome `LoginNT`. Esse *script* é executado na mesma estação em que o componente COM está disponível.

Para criar o console apresentado na figura 5.8, através do qual um usuário pode fornecer seu nome e sua senha, utilizamos o pacote AWT de Java para construção de interfaces gráficas. O *script* a seguir cria e configura todos os elementos de interface utilizados nesse console:

```
frame = LuaOrb.JavaProxy("java.awt.Frame","System Login")
label_user = LuaOrb.JavaProxy("java.awt.Label","User:")
```

```
1  login_proxy = LuaOrb.ComProxy("PasswordTst.PasswordTst.1")
2  login_server = LuaOrb.CorbaAdapter("Login", login_proxy)
3
4  name_service = LuaOrb.CorbaProxy{interface="CosNaming::NamingContext",
5                    ior = "corbaloc::Host1:7777/NameService"}
6  name_service:bind({{id="LoginNT", kind=""}}, login_server)
```

Figura 5.7: Um *script* Lua para criar uma ponte entre COM e CORBA.

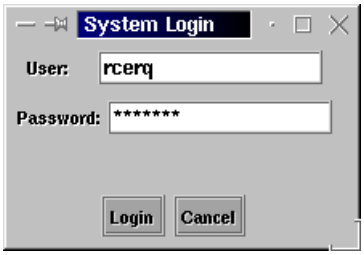

Figura 5.8: Um console AWT para validação de senhas.

```
label_pass = LuaOrb.JavaProxy("java.awt.Label","Password:")
text_user = LuaOrb.JavaProxy("java.awt.TextField","")
text_pass = LuaOrb.JavaProxy("java.awt.TextField","")
button_login = LuaOrb.JavaProxy("java.awt.Button","Login")
button_cancel = LuaOrb.JavaProxy("java.awt.Button","Cancel")
panel_user = LuaOrb.JavaProxy("java.awt.Panel")
panel_pass = LuaOrb.JavaProxy("java.awt.Panel")
panel_buttons = LuaOrb.JavaProxy("java.awt.Panel")
BorderLayout = LuaOrb.JavaProxy("java.awt.BorderLayout")

text_pass:setEchoChar("*")
text_user:setColumns(20)
text_pass:setColumns(20)
panel_user:add(label_user)
panel_user:add(text_user)
panel_pass:add(label_pass)
panel_pass:add(text_pass)
panel_buttons:add(button_login)
panel_buttons:add(button_cancel)
frame:add(panel_user, BorderLayout.NORTH)
frame:add(panel_pass, BorderLayout.CENTER)
frame:add(panel_buttons, BorderLayout.SOUTH)
frame:pack()
frame:setVisible(1)
```

Esse *script* é apenas a primeira parte do programa que deve ser executado em todas as estações clientes que precisam acessar o componente de validação de senhas. Com os elementos de interface com o usuário já criados, devemos agora associar os tratadores de eventos necessários para que o diálogo

```
 1  name_service=LuaOrb.CorbaProxy{interface="CosNaming::NamingContext",
 2                   ior="corbaloc::Host1:7777/NameService"}
 3  object = name_service:resolve({{id="LoginNT", kind=""}})
 4  login_service = CORBA_narrow(object)
 5
 6  login_cb = {
 7     actionPerformed = function (self, ev)
 8        local user = text_user.getText()
 9        local pass = text_pass.getText()
10        local key = login_service:validate(user,pass)
11        if key ~= nil then
12          App = LuaOrb.JavaProxy("Application")
13          frame:dispose()
14        end
15     end
16  }
17
18  button_login:addActionListener(login_cb)
```

Figura 5.9: Um *script* Lua para acessar o serviço de validação de senhas.

de validação de senhas tenha o comportamento esperado. Para isso, definimos um objeto *listener* que será acionado quando o botão de confirmação do diálogo for pressionado pelo usuário.

A figura 5.9 apresenta o restante do *script* cliente do nosso exemplo. Nas linhas 1-2, criamos um *proxy* para o serviço de nomes de CORBA e, na linha 3, obtemos um *proxy* para o serviço de validação de senhas registrado anteriormente no serviço de nomes. Como o objeto retornado pela operação `resolve` do serviço de nomes é do tipo `CORBA::Object`, não podemos acessar as operações específicas da interface `Login` através desse *proxy*. Para contornar esse tipo de problema, o *binding* CORBA oferece a função `CORBA_narrow`, que recebe uma referência para um objeto, e retorna uma outra referência para o mesmo objeto, só que esta última é associada ao tipo mais especializado do objeto (linha 4).

Nas linhas 6-16, é criado o objeto Lua que vai ser usado para tratar o evento gerado quando o botão `button_login` for pressionado. Esse objeto, `login_cb`, possui apenas a operação `actionPerformed`, que consulta o conteúdo dos campos de entrada de dados do diálogo (linhas 8-9) e acessa o serviço de validação de senhas através do *proxy* `login_service` (linha 10). Se o nome e a senha do usuário estiverem corretos, a aplicação protegida pelo diálogo de validação de senhas é ativada (linha 12). Finalmente, o objeto `login_cb` é registrado como o tratador de eventos do botão `button_login` (linha 18). Observe que um adaptador genérico é criado implicitamente para o objeto `login_cb` poder ser usado pelo objeto Java (um botão AWT).

Uma vez que os *scripts* do lado servidor e do lado cliente tiverem executado, a configuração final obtida em nossa aplicação exemplo será equivalente àquela representada na figura 4.10 (página 63). O *script* cliente pode ser executado em mais de uma estação cliente simultaneamente. Nesse caso, todos os consoles clientes vão estar compartilhando o mesmo serviço de validação de senhas.

Muitas vezes, precisamos adaptar as interfaces entre os componentes cliente e servidor. Por exemplo, suponha que o console da interface com o usuário espere que o serviço de validação de senhas tenha a seguinte interface:

```
struct LoginInfo {
   string user;
```

```
    string passwd;
};

interface Login {
    boolean validate (in LoginInfo info);
};
```

Nesse caso, podemos adaptar o *script* do lado servidor para compatibilizar as interfaces `Login` e `IPasswordTst`:

```
login_proxy.old_validate = login_proxy.validate
login_proxy.validate = function (self, info)
    return self:old_validate(info.user, info.passwd)
end
```

Nessa solução, definimos um novo método `validate` para o *proxy* Lua, que chama a sua versão original em COM para efetivamente validar os dados do usuário.

5.4 Comparação com Outros Sistemas

Como apresentamos na seção 3.4, existem várias ferramentas de *script*, e muitas delas com um alto grau de integração com sistemas de componentes de software. Entretanto, nenhuma das ferramentas analisadas oferece uma solução de interoperabilidade entre sistemas como a apresentada pelo LuaOrb. O fato de seguirem um padrão de projeto comum dá aos *bindings* do LuaOrb uma grande uniformidade de uso. Apesar de não conhecermos uma outra ferramenta de *script* para compararmos com o sistema LuaOrb completo, podemos fazer uma análise comparativa de seus *bindings* separadamente.

É importante observar que a linguagem Lua não precisou ser modificada em nada para a implementação do LuaOrb. Tanto Lua quanto os sistemas de componentes foram utilizados como pacotes fechados para a implementação do LuaOrb. Outras ferramentas de *script* para sistemas de componentes tipicamente exigem alterações na implementação do interpretador da linguagem, quando não utilizam linguagens desenvolvidas especificamente para trabalhar com um determinado sistema de componentes.

5.4.1 LuaCorba e Outras Ferramentas de *Script* para CORBA

O *binding* LuaCorba foi o primeiro componente do sistema LuaOrb a ser desenvolvido [CRI97]. Esse *binding* foi desenvolvido em C++ e, atualmente, trabalha com três implementações CORBA: ORBacus C++ [OOC98], Visibroker C++ [Vis96] e Mico [Mic00]. LuaCorba e a linguagem CorbaScript [MGG96, MGG97] foram das primeiras ferramentas de *script* para CORBA. Os recursos oferecidos por essas duas ferramentas e as técnicas utilizadas em suas implementações são muito similares. A diferença básica entre esses dois sistemas é que CorbaScript, como o próprio nome já sugere, é uma linguagem desenvolvida exclusivamente para ser integrada a CORBA. Esse forte acoplamento com CORBA trouxe algumas conseqüências práticas para CorbaScript, tais como a ausência de mecanismos mais flexíveis para estender a linguagem e para embutí-la em aplicações. Até recentemente, CorbaScript só podia ser utilizada de forma independente de uma aplicação específica, isto é, ela não podia ser embutida diretamente em uma aplicação hospedeira.

Por outro lado, LuaCorba integrou uma linguagem de *script* já existente a um sistema de componentes e, dessa forma, procuramos identificar os mecanismos e as técnicas básicas para implementar esse tipo de *binding* entre linguagens de *script* e sistemas de componentes. Em LuaCorba, também

procuramos trazer para o domínio de aplicações CORBA todas as facilidades que uma linguagem de *script* tipicamente traz para outros domínios de aplicações. Por exemplo, embutindo LuaCorba em uma aplicação, podemos combinar em um mesmo processo objetos clientes e servidores implementados tanto em Lua quanto em C++ (usando *stubs* estaticamente gerados). Dessa forma, podemos dividir uma aplicação em seu núcleo, gerado estaticamente em C++, e seus pontos de extensão, que são configurados com o auxílio de Lua.

Com a definição do padrão da OMG para ferramentas de *script* de objetos CORBA [OMG99b], alguns outros *bindings* foram definidos e implementados com base nesse novo padrão [Pil00, Chi99, OMG00]. As técnicas de implementação desses *bindings* podem ser divididas em dois grupos: as que utilizam as interfaces dinâmicas de CORBA (DII e DSI) e as que geram em tempo de execução os *stubs* e esqueletos típicos dos *bindings* estáticos. Enquanto LuaCorba, CorbaScript e Combat [Pil00] estão no primeiro grupo, os *bindings* de Python tipicamente estão no segundo grupo [Chi99, JSLJ99, OMG00]. Apesar de não termos realizado efetivamente uma comparação quantitativa entre as duas técnicas, acreditamos que a segunda técnica deve apresentar um melhor desempenho nas chamadas de operações dos componentes (se não considerarmos o tempo necessário para gerar os *stubs* e esqueletos em tempo de execução), enquanto a primeira deve utilizar menos memória em tempo de execução.

É importante destacar que LuaCorba já atende ao padrão para ferramentas de *script* para CORBA. Apesar desse padrão ter sido definido posteriormente ao desenvolvimento de LuaCorba, o modelo adotado pelo padrão é compatível com o usado na implementação de LuaCorba.

Uma vez que todas as ferramentas de *script* para CORBA seguem de alguma forma o padrão definido em [OMG99b], a escolha de uma determinada ferramenta deve ser feita em função das características oferecidas pela linguagem de *script*, tais como facilidade de uso, desempenho, consumo de memória em tempo de execução e bibliotecas disponíveis. Dessas características, as mais fáceis de serem medidas são desempenho e consumo de memória. Apesar de não termos medidas do impacto provocado no consumo de memória em tempo de execução pelo LuaCorba, sabemos que essa ferramenta adiciona ao tamanho final de uma aplicação aproximadamente 200 KB em plataformas Intel com sistemas operacionais Linux e MS Windows NT. Essas medidas incluem o próprio *binding* e a linguagem Lua com suas bibliotecas básicas, mas não incluem o ORB utilizado. Só para uma simples comparação, o console básico de comandos de CorbaScript 1.2 tem aproximadamente 1 MB em plataforma Linux/Intel.

Com relação ao desempenho de LuaCorba, podemos esperar uma perda se o compararmos com *binding* estáticos, mesmo utilizando o recurso de armazenar localmente as descrições das interfaces obtidas através do Repositório de Interfaces. Essa perda de desempenho deve ser proporcional à complexidade dos tipos dos parâmetros passados através de uma chamada de operação. Em um primeiro estudo [Mou99], observamos perdas de 23 a 52% em chamadas remotas e de 240 a 430% em chamadas locais. As perdas em chamadas remotas, onde predomina o tempo de transmissão de dados através da rede, não se mostraram críticas; ainda mais se considerarmos que implementações CORBA feitas em Java chegam a ser de 3 a 4 vezes mais lentas do que implementações CORBA em C++ [BWW99].

5.4.2 LuaCom e Outras Ferramentas de *Script* para COM

Para a plataforma Windows/COM, Visual Basic ainda é a principal ferramenta de *script*. A tecnologia de Automação OLE, que é a base para a implementação de ferramentas de *script* para COM, foi originalmente projetada para servir de infra-estrutura para a implementação do Visual Basic, e posteriormente sofreu algumas alterações para poder ser utilizada por outras ferramentas de *script* [Bro95]. Mesmo assim, ainda podemos observar uma grande influência de Visual Basic sobre a atual tecnologia de Automação OLE. Um exemplo dessa influência é a restrição dos tipos de dados que podem ser usados na interface de um componente passível de automação, isto é, só podem ser usados os tipos de dados válidos em Visual Basic, que são um subconjunto do tipos válidos em COM IDL.

Em função das restrições impostas pela tecnologia de Automação OLE, todas as ferramentas de *script* para COM são bem similares com relação ao modo como se integram ao sistema de componentes. Dessa forma, o que diferencia essas ferramentas são as características das linguagens utilizadas. Uma das poucas exceções dessa uniformidade nas técnicas de integração é o próprio Visual Basic. Como a linguagem de Visual Basic é extremamente acoplada a COM e ao sistema Windows, ela também permite o acesso a componentes COM através de suas interfaces na forma de tabelas de métodos virtuais, o que representa um considerável ganho de desempenho.

LuaCom se restringe exclusivamente aos recursos oferecidos através da tecnologia de Automação OLE. Em comparação com os outros dois *bindings* de LuaOrb, este é o que apresenta o maior número de restrições:

- nem todos os tipos de dados disponíveis no sistema de componentes podem ser disponibilizados para Lua;
- somente os componentes que oferecem a interface `IDispatch` podem ser acessados através de Lua;
- componentes implementados em Lua só podem ser acessados através da interface `IDispatch`, isto é, não é transparente para os clientes como o componente foi implementado;
- como as referências para objetos sempre aparecem com os tipos `IUnknown` ou `IDispatch` nas assinaturas de operações de interfaces compatíveis com Automação OLE, LuaCom só permite a criação explícita de adaptadores genéricos, para que o *binding* possa saber qual interface o componente realmente está implementando.

Como LuaCom foi o último *binding* do LuaOrb a ser implementado, ainda não temos medidas de desempenho para compará-lo com outras ferramentas de *script* para COM.

5.4.3 LuaJava e Outras Ferramentas de *Script* para Java

Como vimos na seção 3.4, as implementações de *bindings* entre linguagens de *script* e Java têm seguido basicamente duas abordagens. A primeira utiliza a interface de programação com o sistema nativo oferecida por Java (JNI – *Java Native Interface*), para implementar um módulo de extensão para a linguagem de *script* que permite que esta se comunique com o ambiente de Java. A outra abordagem envolve a implementação do interpretador da linguagem de *script* em Java. LuaJava adotou a primeira abordagem, da mesma forma que TclBlend [Sta98] e JPI [CSW97]. JPython [vR98, Ang99], Jacl [Joh98] e Pnuts [McC00] são exemplos de ferramentas que seguiram a segunda abordagem.

As ferramentas de *script* implementadas diretamente em Java apresentam a vantagem de poderem ser usadas mesmo quando não é possível utilizar código nativo, como, por exemplo, em *applets*. No caso de linguagens de *script* que já possuem interpretadores para diversas plataformas, a reimplementação desses interpretadores em Java pode trazer algumas desvantagens, além do próprio trabalho envolvido nessa tarefa. Uma das desvantagens é a dificuldade de manter as implementações totalmente compatíveis. Por exemplo, Jacl já possui uma lista descrevendo suas diferenças em relação à linguagem Tcl original.

Uma outra desvantagem é que uma implementação em Java pode piorar consideravelmente o desempenho da linguagem de *script*. De acordo com [CIR99a], o desempenho dessas linguagens pode piorar de 10 a 100 vezes, quando comparado com o das implementações originais das linguagens.

O desempenho de LuaJava é compatível com o de ferramentas similares. De acordo com [CIR99a], quando medimos os tempos de chamada de métodos de objetos Java a partir da linguagem de *script*, LuaJava apresenta praticamente os mesmos resultados de JPython, enquanto Jacl é dez vezes mais lento que as outras duas ferramentas. Enquanto o tempo para chamar um método de um objeto Java

através de um *proxy* LuaJava é de 49μs, o tempo de chamada de um método de um objeto Lua é de aproximadamente 3μs. Esse aumento da ordem de 10 vezes é devido ao uso da API reflexiva de Java e da JNI, mais as conversões dos argumentos (incluindo o *self*) e resultados de uma linguagem para a outra.

Uma outra diferença importante nas técnicas de implementação de *bindings* de linguagens de *script* para Java está relacionada ao uso de objetos da linguagem de *script* para serem chamados a partir de Java (objetos *callback*). As ferramentas de *script* têm adotado duas técnicas para permitir esse uso:

- a geração de novas classes Java em tempo de execução, com o auxílio do mecanismo de Java para a carga dinâmica de novas classes (*Class Loader*);
- o uso de objetos *callbacks* restritos a apenas algumas classes pré-definidas (tipicamente classes que seguem o padrão dos *listeners* de JavaBeans).

A segunda técnica representa uma perda de generalidade para a ferramenta que a utilize, pois a ferramenta não pode implementar tipos quaisquer de objetos. LuaJava e JPython adotam a primeira técnica, enquanto TclBlend, Jacl e JPI adotam a segunda.

Capítulo 6

Conclusões

O nosso modelo de composição e o sistema LuaOrb foram alguns dos resultados obtidos durante um trabalho de pesquisa desenvolvido ao longo de cinco anos. Em [Cer96], fizemos o nosso primeiro estudo sobre os modelos de objetos de sistemas de componentes, e desenvolvemos um estudo de caso onde definimos o mapeamento da linguagem School para CORBA. Desse trabalho resultou uma primeira comparação de tecnologias para interoperabilidade entre objetos [CI96].

Em seguida, começamos o desenvolvimento do *binding* de Lua para CORBA, onde tínhamos dois objetivos principais: verificar a capacidade anunciada por CORBA de integrar diferentes modelos de linguagens de programação, e desenvolver uma ferramenta com a qual pudéssemos explorar o uso de linguagens interpretadas em sistemas distribuídos orientados a objetos [CIR97, CRI97, RIC98].

Após o desenvolvimento da primeira versão do sistema LuaOrb, que só incluia o *binding* com CORBA, começamos a identificar os mecanismos básicos utilizados em sua implementação [ICR98]. Uma vez tendo identificado esses mecanismos básicos, iniciamos o desenvolvimento dos *bindings* LuaJava e LuaCom [CI99, CIR99a]. Com isso, verificamos que o mesmo modelo utilizado para implementar o *binding* CORBA poderia ser aplicado aos outros dois sistemas de componentes. A partir desse modelo comum, começamos a explorar a possibilidade de interoperabilidade entre os diferentes sistemas [CCI99]. Somente após termos completado o sistema LuaOrb, iniciamos a formalização do modelo de composição apresentado nesta dissertação.

Com o modelo de composição descrito no capítulo 4 e com o sistema LuaOrb, que é uma implementação desse modelo, pudemos alcançar o objetivo principal deste trabalho, que era comprovar a tese de que uma linguagem interpretada com um determinado conjunto de mecanismos reflexivos, aliada à compatibilidade estrutural de tipos, oferece um mecanismo de composição adequado tanto para a conexão dinâmica de componentes (*very late binding*), quanto para a interoperabilidade entre diferentes sistemas de componentes.

Apesar da complexidade intrínseca dos sistemas de componentes tratados, procuramos manter o modelo de composição o mais simples possível, representando e destacando apenas os aspectos que consideramos essenciais. Um dos indicadores da simplicidade do nosso modelo é o número de interfaces IDL utilizadas para representar seus conceitos. Como pode ser visto nas seções A.1 e A.2 do apêndice, todo o modelo de composição foi descrito com apenas 5 interfaces e 14 operações. Além de facilitar a compreensão do modelo, essa simplicidade também facilitou a implementação do padrão de projeto para *bindings* dinâmicos para um maior número de sistemas, o que permitiu uma melhor avaliação da aplicabilidade do modelo proposto. Manter essa simplicidade foi um dos principais desafios encontrados durante este trabalho.

No restante deste capítulo, vamos apresentar as principais contribuições deste trabalho. Também apresentamos resumidamente alguns estudos de casos que já foram realizados utilizando o sistema LuaOrb, e os principais resultados que foram observados. Finalmente, identificamos alguns trabalhos

futuros.

6.1 Principais Contribuições

Consideramos que este trabalho teve duas contribuições principais. Uma delas foi o próprio modelo de composição, que trouxe resultados tanto conceituais quanto práticos. A outra foi a ferramenta de desenvolvimento LuaOrb.

A contribuição dada pelo modelo de composição foi a unificação, em um único modelo, de mecanismos para a conexão dinâmica de componentes e para a interoperabilidade entre diferentes sistemas de componentes. Já o sistema LuaOrb representa uma valiosa ferramenta por permitir a aplicação e experimentação de novas técnicas no desenvolvimento de sistemas baseados em componentes. Na seção 6.2, apresentamos alguns estudos de caso que reforçam essa contribuição.

Além do resultado imediato proporcionado pelo nosso modelo de composição, o seu processo de definição produziu mais alguns resultados relevantes. Um desses resultados foi que, a partir da identificação dos mecanismos de extensão e reflexividade que permitiram a construção do modelo, podemos definir parâmetros mais específicos para se fazer uma análise comparativa da flexibilidade oferecida por diferentes sistemas de componentes. Por exemplo, CORBA mostrou ter a arquitetura mais flexível para a implementação de nosso modelo: Java não oferece um mecanismo para permitir implementações dinâmicas diretamente, e suas interfaces não podem evoluir dinamicamente; COM apresenta uma série de restrições para o uso de suas interfaces dinâmicas.

Os mecanismos de extensão e reflexividade de nosso modelo também podem ser usados de forma semelhante para a análise dos mecanismos oferecidos por linguagens de *script*. Análises desse tipo ficam como mais uma sugestão de trabalho futuro, além daquelas apresentadas na seção 6.3.

Para definir o nosso modelo de objetos integrador, tivemos que analisar os modelos de objetos adotados por CORBA, COM e Java. Assim, no capítulo 2, apresentamos esses sistemas a partir de seus modelos de objetos. Geralmente, os sistemas de componentes são analisados e comparados de acordo com aspectos puramente tecnológicos, tais como protocolos, mecanismos de segurança, suporte a transações e escalabilidade. Esses aspectos têm se mostrado muito efêmeros ao longo da evolução desses sistemas. Análises e comparações desses sistemas a partir das características de seus modelos de objetos são, de uma certa forma, mais eficazes, pois essas características têm se mostrado muito mais estáveis, e são elas que ditam como os sistemas de componentes podem ser usados e quais são suas limitações intrínsecas.

No processo de definição de nosso modelo de composição, também resgatamos alguns resultados obtidos em trabalhos desenvolvidos com linguagens OO baseadas em protótipos, e os aplicamos no contexto da integração de linguagens de *script* e sistemas de componentes.

6.2 Alguns Estudos de Caso

Desde o início do projeto, tivemos a preocupação de verificar e explorar a aplicação das ferramentas desenvolvidas e, conseqüentemente, do modelo de composição proposto. Assim, vários estudos de caso já foram desenvolvidos. Como o *binding* com CORBA foi o primeiro a ser implementado, esse *binding* foi a principal ferramenta utilizada nos experimentos realizados.

Os nossos estudos de caso podem ser divididos em dois grupos. O primeiro é composto por experimentos desenvolvidos em conjunto com outros trabalhos de pesquisa. O outro grupo consiste de aplicações finais que foram colocadas em produção.

O primeiro experimento realizado com LuaOrb foi na área de gerência de sistemas distribuídos [RMIC97]. Nesse experimento, integramos LuaOrb a uma ferramenta de gerência de rede, chamada LuaMan [ML97, Lim98], que utiliza o padrão SNMP de gerência de redes. O objetivo desse

trabalho é oferecer uma ferramenta de gerência integrada de serviços CORBA e agentes SNMP tradicionais. Essa ferramenta, além de permitir a gerência integrada de serviços, pode ser estendida dinamicamente para atender a mudanças nos serviços e na configuração da rede gerenciada. A partir desse experimento, foram desenvolvidos mecanismos para instalar e estender remotamente serviços CORBA, e uma ferramenta de apoio à evolução dinâmica das interfaces de serviços CORBA [Mar98, MRI99].

Em um outro experimento, exploramos o uso do LuaOrb no desenvolvimento de um sistema de CAD cooperativo [GFCI98, FRB+98, Gom99]. Nesse sistema, *scripts* Lua definem e modificam, em tempo de execução, os relacionamentos (*restrições*) entre os elementos que compõem um projeto de CAD desenvolvido concorrentemente por diversos usuários. Devido às características dinâmicas do LuaOrb, esse sistema de CAD oferece uma grande flexibilidade para tarefas como a adição de novas entidades no projeto e a configuração dinâmica dos relacionamentos entre as diversas entidades de um projeto. Quando um usuário faz uma alteração em alguma propriedade de uma entidade, é disparado um algoritmo para verificar e adaptar todas as propriedades de outras entidades que dependem da propriedade alterada.

Com o amadurecimento do LuaOrb, começamos a utilizá-lo no desenvolvimento de aplicações que foram posteriormente colocadas em produção. A primeira dessas aplicações foi um visualizador 3D distribuído, que foi desenvolvido para servir de ferramenta para a construção de ambientes imersivos de realidade virtual [FCCG99, Fer99]. Originalmente, esse visualizador foi desenvolvido para um sistema de treinamento da Marinha Brasileira, mas, a princípio, ele pode ser utilizado na composição de outros ambientes imersivos. Esse sistema de visualização é composto basicamente por dois componentes: um componente de visualização 3D, que tanto pode ser operado localmente em uma estação isolada como pode ser controlado remotamente; e um sistema de controle, que é responsável pela coordenação de diversas unidades de visualização. A infra-estrutura de comunicação adotada nesse sistema foi CORBA, e todo o sistema de controle e as interfaces de operação remota das unidades de visualização foram implementadas exclusivamente com o LuaOrb. Já o núcleo do componente de visualização, que é totalmente isolado de CORBA, foi desenvolvido em C++ e exporta uma interface de configuração e operação através de Lua.

Uma outra aplicação desenvolvida foi um simulador de console sonar para um outro sistema de treinamento da Marinha Brasileira. Esse simulador foi implementado para trabalhar integrado a um sistema de simulação distribuída, desenvolvido para o treinamento tático de equipes de navios de guerra. Basicamente, o simulador de console sonar monitora um conjunto de veículos simulados, aplica um algoritmo de detecção simplificado, e apresenta em um console gráfico os contatos correspondentes aos alvos detectados. Ao contrário do visualizador 3D, esse sistema foi desenvolvido de forma híbrida, isto é, ele é composto por objetos CORBA desenvolvidos em C++ e Lua.

A partir desse conjunto de estudos de caso, pudemos observar uma série de resultados do uso do sistema LuaOrb. O primeiro resultado significativo foi que, ao contrário do que poderíamos esperar, em nenhum dos estudos de caso o uso do LuaOrb comprometeu o desempenho da aplicação. Isso se deve a algumas características comuns a todas as aplicações analisadas: uso de componentes implementados em C++, distribuição e forte interação do usuário. Lua foi utilizada basicamente para fazer a composição e a coordenação de componentes, e todos os componentes que demandavam um maior esforço computacional foram implementados em C++. Em aplicações distribuídas, o tempo de comunicação entre os processos acaba minimizando o impacto do uso de uma linguagem interpretada. Da mesma forma, as intervenções do usuário e os tempos de resposta demandados por este se mostraram compatíveis com o desempenho oferecido pelo LuaOrb. Isso se verificou até mesmo no sistema de visualização 3D, que exige altas taxas de redesenho de tela para oferecer a sensação de imersão desejada.

Em todos as aplicações desenvolvidas, pudemos comprovar a adequação do LuaOrb para o suporte a mudanças rápidas nas implementações dos sistemas, realizadas tanto em tempo de execução quanto

com o sistema inativo. No simulador do console sonar, também foi possível verificar a adequação do LuaOrb como ferramenta de automação de testes para sistemas distribuídos.

Vários outros aspectos interessantes puderam ser observados. Por exemplo, no sistema de visualização 3D a implementação do componente de visualização ficou totalmente isolada do sistema de componentes, que nesse caso foi CORBA. Toda a integração com CORBA foi feita através de adaptadores implementados com o LuaOrb. Essa mesma arquitetura pode ser aplicada em várias outras situações. Já o simulador do console sonar mostrou que implementações híbridas também funcionam muito bem, mesmo quando componentes CORBA implementados em C++ residem no mesmo processo que componentes CORBA implementados em Lua.

O uso de LuaOrb também se mostrou muito adequado para a implementação de serviços como fábricas de objetos, grupos, repositórios e coleções de objetos distribuídos de um modo geral. Nesses casos, as implementações em LuaOrb funcionam como uma espécie de molde (*template*) para esses serviços, que são parametrizadas por alguma interface que define o tipo dos objetos que vão ser gerenciados pelo serviço. Acreditamos que essa adequação também seja verdade para qualquer outro serviço que envolva basicamente o gerenciamento de outros objetos de uma forma mais abstrata.

Todos os estudos de caso já desenvolvidos serviram para avaliar a eficácia do nosso modelo de composição dinâmica. Mas ainda não realizamos nenhum experimento que realmente explorasse os recursos de interoperabilidade entre os diferentes sistemas de componentes integrados pelo LuaOrb. Esses recursos só foram utilizados em exemplos pequenos, como aquele apresentado na seção 5.3. Precisamos de uma aplicação mais complexa para podermos fazer uma avaliação mais completa dos recursos de interoperabilidade do LuaOrb.

6.3 Trabalhos Futuros

A partir deste trabalho, identificamos várias evoluções possíveis. Com relação ao modelo de composição, acreditamos que seja de grande importância a inclusão de um modelo de concorrência, que permita a representação de diferentes esquemas de coordenação e sincronismo entre objetos. Com o auxílio de um modelo de concorrência, poderíamos oferecer um mecanismo para descrever de forma mais clara e precisa os esquemas de coordenação entre os componentes de uma aplicação. De acordo com as características vistas na seção 3.1 para uma linguagem de composição, um modelo de concorrência é o único aspecto ausente em nosso modelo de composição.

Seria interessante aplicar o nosso modelo de composição a outras linguagens de *script*, para que tivéssemos uma melhor avaliação de sua generalidade. Provavelmente, a utilização de outras linguagens de *script* levaria a um estudo dos diferentes mecanismos de reflexividade e extensão oferecidos por essas linguagens.

O sistema LuaOrb também pode evoluir de diversas formas. Seria extremamente interessante que fosse realizada uma análise mais detalhada do desempenho de LuaOrb. Esse análise é importante tanto para termos uma melhor avaliação do impacto que a abordagem dinâmica do LuaOrb causa, quanto para avaliarmos o desempenho dos mecanismos dinâmicos dos sistemas de componentes. A partir de uma análise de desempenho mais detalhada, poderíamos identificar gargalos nas implementações desses mecanismos dinâmicos, o que talvez permitisse a proposição de otimizações para esses mecanismos. Para fazer essa análise, poderíamos adotar uma abordagem similar à proposta de Jurič et al. [JWR+99], que define um modelo para a análise de desempenho de arquiteturas de objetos distribuídos.

Também com relação à análise de desempenho do LuaOrb, podemos investigar como o sistema se comporta em cenários com um elevado número de clientes e servidores por processo. Em cenários desse tipo, seria possível avaliar a influência no consumo de memória da ausência de *stubs* no LuaOrb.

Com relação aos recursos de interoperabilidade oferecidos pelo LuaOrb, precisamos desenvolver

estudos de caso mais elaborados, que permitam uma melhor avaliação desses recursos. Essa avaliação auxiliaria a identificação de possíveis extensões e limitações da abordagem que adotamos. Uma ferramenta útil para a interoperabilidade entre sistemas através do LuaOrb seria um tradutor entre as linguagens de definição de interfaces dos diferentes sistemas tratados pelo LuaOrb. Esse tradutor facilitaria a publicação da interface de um componente de um determinado sistema para um outro sistema de componentes. Por exemplo, esse tradutor poderia ser utilizado para gerar automaticamente a interface `Login` a partir da interface `IPasswordTst`, vistas na seção 5.3.

A partir do LuaOrb, podemos investigar a construção de ferramentas de mais alto nível que utilizem o LuaOrb como base. Por exemplo, podemos investigar extensões para o LuaOrb que permitam a sua utilização como uma linguagem de descrição de arquiteturas (ADL). Com o auxílio de extensões desse tipo, LuaOrb poderá oferecer mecanismos mais apropriados para a representação estruturada de arquiteturas de aplicações.

Uma outra linha de pesquisa que nos parece promissora é a integração do LuaOrb com middlewares reflexivos. Diversos trabalhos têm sido realizados na investigação de novos mecanismos para que tanto os middlewares quanto as aplicações construídas sobre estes possam se adaptar dinamicamente a novos requisitos [BP97, BCD+97, SSC97, CNM98, BCRP98, CBC98, FBC+98, KC99, RKC99, HKC+99, RSC+99, KC00, KRL+00, BBI+00, JTMJ00, KS00]. De um modo geral, esses trabalhos propõem a incorporação de técnicas de reflexividade aos middlewares. Middlewares reflexivos tipicamente oferecem mecanismos para o monitoramento de parâmetros de qualidade de serviço (QoS), o gerenciamento de recursos, a reificação das dependências dinâmicas entre componentes, e a reconexão entre componentes (*re-wiring*).

Acreditamos que a integração do LuaOrb com um middleware reflexivo seria vantajosa para essas duas tecnologias. Por um lado, essa integração ofereceria para um middleware reflexivo um mecanismo para a programação das possíveis reconfigurações de um sistema. Por outro lado, as informações sobre a configuração dinâmica de um sistema, que tipicamente um middleware reflexivo oferece, podem ser utilizadas para orientar um usuário do LuaOrb em suas tarefas de reconfiguração.

Da mesma forma como já exploramos a flexibilidade oferecida pelo LuaOrb em uma série de estudos de caso desenvolvidos, acreditamos que o mesmo ainda pode ser feito em vários outros domínios de aplicação. Novos estudos de caso ajudariam a explorar ainda mais as potencialidades e limitações de nossa abordagem dinâmica, de tal forma que pudéssemos identificar outras situações em que a flexibilidade de nosso modelo de composição pudesse ser aproveitada. Em última análise, esperamos que este trabalho, assim como suas evoluções, venham a contribuir de alguma forma para o aprimoramento das técnicas de desenvolvimento baseadas em componentes de software.

Apêndice A

Definições em IDL

A.1 O Modelo de Objetos Integrador

```
interface Dispatcher;
interface InterfaceDescr;
interface MethodSignature;

typedef Dispatcher ExceptionDescr;
typedef sequence<Dispatcher> ListOfObjects;
typedef sequence<InterfaceDescr> InterfaceDescrSeq;
typedef sequence<string> NameSeq;

interface Dispatcher {
   boolean invoke(in string method_name,
                  in ListOfObjects params_in,
                  out ListOfObjects params_out,
                  out ExceptionDescr ex);
   InterfaceDescr getInterface();
};

interface InterfaceDescr {
   NameSeq getAllOperationNames();
   MethodSignature getSignature(in string name);
};

interface MethodSignature {
   string getName();
   InterfaceDescrSeq getParamsIn();
   InterfaceDescrSeq getParamsOut();
   InterfaceDescrSeq getExceptions();
};
```

A.2 Os Mecanismos de Adaptação do Padrão de *Binding* Dinâmico

```
interface Method {
   void call(in Dispatcher self,
```

```
            in ListOfObjects params_in,
            out ListOfObjects params_out,
            out ExceptionDescr ex);
};

interface ExtendedDispatcher : Dispatcher {
   void setDelegatee(in Dispatcher obj);
   Dispatcher getDelegatee();

   void setMethod(in string name, in Method meth);
   void removeMethod(in string name);
   Method getMethod(in string name);
};
```

A.3 O Modelo de Objetos de Lua

```
interface LuaObject;
interface LuaTag;
interface SetTableMethod;
interface GetTableMethod;
interface LuaFunction;

typedef GetTableMethod IndexMethod;
typedef sequence<any> ListOfAny;

interface LuaObject {
   any get (in any index);
   void set (in any index, in any value);

   LuaTag getTag();
   void setTag (in LuaTag m_obj);
};

interface LuaTag {
   void setSetTableMethod (in SetTableMethod method);
   SetTableMethod getSetTableMethod();
   void setGetTableMethod (in GetTableMethod method);
   GetTableMethod getGetTableMethod();
   void setIndexMethod (in IndexMethod method);
   IndexMethod getIndexMethod();
};

interface SetTableMethod {
   void call (in LuaObject self, in any index, in any value);
};

interface GetTableMethod {
   any call (in LuaObject self, in any index);
```

```
};

interface LuaFunction {
   ListOfAny call (in ListOfAny params_in);
};
```

Referências Bibliográficas

[AG98] Ken Arnold e James Gosling. *The Java Programming Language*. Addison Wesley, segunda edição, 1998.

[ALSN00] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, e Oscar Nierstrasz. Piccola - a small composition language. Em Howard Bowman e John Derrick, editores, *Formal Methods for Distributed Processing, an Object Oriented Approach*. Cambridge University Press, 2000.

[AN00] Franz Achermann e Oscar Nierstrasz. Applications = components + scripts – a tour of Piccola. Em Mehmet Aksit, editor, *Software Architectures and Component Technology*. Kluwer, 2000.

[Ang99] Kirby Angell. Examining JPython. *Dr. Dobb's Journal*, 24(4):78–83, Abril 1999.

[ASG+94] G. Almasi, A. Suvaiala, C. Goina, C. Cascaval, e V. Jagannathan. *TclDii: A TCL Interface to the Orbix Dynamic Invocation Interface*, 1994. (`http://www.cerc.wvu.edu/dice/iss/TclDii/TclDii.html`).

[AvdL90] P. America e F. van der Linden. A parallel object-oriented language with inheritance and subtyping. *Sigplan Notices*, 25(10), 1990. (OOPSLA/ECOOP'90).

[Ban97] Bela Ban. *A Generic Management Model for CORBA, CMIP and SNMP*. Tese de Doutorado, University of Zurich and IBM Zurich Research Laboratory, Rueschlikon, Switzerland, 1997.

[BBI+00] Gordon Blair, Lynne Blair, Valérie Issarny, Petr Tuma, e Apostolos Zarras. The role of software architecture in constraining adaptation in component-based middleware platforms. Em J. Sventek e G. Coulson, editores, *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, páginas 164–184, New York, USA, Abril 2000. Springer-Verlag. (LNCS 1795).

[BCD+97] G. Blair, G. Coulson, N. Davies, P. Robin, e T. Fitzpatrick. Adaptive middleware for mobile multimedia applications. Em *Proceedings of the 8th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '97)*, St. Louis, Missouri, USA, Maio 1997.

[BCK98] Len Bass, Paul Clements, e Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[BCRP98] G. Blair, G. Coulson, P. Robin, e M. Papathomas. An architecture for next generation middleware. Em N. Davies, K. Raymond, e J. Seitz, editores, *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, Lake District, UK, 1998. Springer-Verlag.

[Bea98] David Beazley. SWIG and automated C/C++ scripting extensions. *Dr. Dobb's Journal*, 23(2):30–36, 1998.

[BF96] Judy Bishop e Roberto Faria. Connectors in configuration programming languages: are they necessary? Em *Proceedings of the Third International Conference on Configurable Distributed Systems*, páginas 11–18, Annapolis, Maryland, Maio 1996.

[BHK+99] Francisco J. Ballesteros, Christopher Hess, Fabio Kon, Sergio Arévalo, e Roy H. Campbell. Object Orientation in Off++ - A Distributed Adaptable μKernel. Em *ECOOP'99 Workshop on Object Orientation and Operating Systems*, páginas 49–53, Lisboa, Portugal, Junho 1999.

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noel Plouzeau, e Damien Watkins. Making components contract aware. *IEEE Computer*, 32(7):38–45, Julho 1999.

[Bor95] Borland Internacional. *Borland Delphi Users Manual*, 1995.

[Box98] Don Box. *Essential COM*. Addison-Wesley, 1998.

[BP97] G. Blair e M. Papathomas. The case for reflective middleware. Em *Proceedings of 3rd Cabernet Plenary Workshop*, Rennes, França, Abril 1997.

[Bro94] C. Brown. NATO standard for the development of reusable software components. Relatório técnico, Public Ada Library, 1994.
(`http://wuarchive.wustl.edu/languages/ada/docs/nato_ru`).

[Bro95] Kraig Brockschmidt. *Inside OLE*. Microsoft Press, segunda edição, 1995.

[Bro96] Alan Brown, editor. *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute*. IEEE Computer Society Press, 1996.

[BWW99] László Böszörményi, Andreas Wickner, e Harald Wolf. Performance evaluation of object oriented middleware. Em *Euro-Par'99 Parallel Processing*, páginas 258–261, Toulouse, França, Setembro 1999. Springer-Verlag. (LNCS 1685).

[Car95] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[CBC98] F. Costa, G. Blair, e G. Coulson. Experiments with reflective middleware. Em *ECOOP Workshop on Reflective Object-Oriented Programming and Systems (ROOPS'98)*, Bruxelas, Bélgica, 1998. Springer-Verlag.

[CCI99] Renato Cerqueira, Carlos Cassino, e Roberto Ierusalimschy. Dynamic component gluing across different componentware systems. Em *International Symposium on Distributed Objects and Applications (DOA'99)*, Edinburgh, 1999. OMG, IEEE Press.

[Cel97] Waldemar Celes. *toLua — accessing C/C++ code from Lua*. Tecgraf/PUC-Rio, 1997.
(`http://www.tecgraf.puc-rio.br/~celes/tolua/tolua.html`).

[Cer96] Renato Cerqueira. Um estudo sobre interoperabilidade entre linguagens orientadas a objetos. Dissertação de Mestrado, Dep. Informática, PUC-Rio, Rio de Janeiro, Brasil, 1996.

[Chi99] Martin Chilvers. *Fnorb – Version 1.0*. Distributed Systems Technology Centre, University of Queensland, Brisbane, Australia, Abril 1999.
(`http://www.dstc.edu.au/Fnorb`).

[Cho98] Il-Hyung Cho. Testing components using protocols. Em *OOPSLA'98 Doctoral Symposium*, 1998.
(`http://www.cs.clemson.edu/~ihcho`).

[CI96] Renato Cerqueira e Roberto Ierusalimschy. Uma avaliação das arquiteturas para interoperabilidade entre objetos. Em *X Simpósio Brasileiro de Engenharia de Software*, páginas 371–386, São Carlos, SP, Brasil, 1996.

[CI99] Carlos Cassino e Roberto Ierusalimschy. LuaJava – uma ferramenta de scripting para Java. Em *III Simpósio Brasileiro de Linguagens de Programação*, Porto Alegre, Brasil, 1999.

[CIR97] Renato Cerqueira, Roberto Ierusalimschy, e Noemi Rodriguez. A dynamic approach for composing CORBA objects. Monografias em Ciência da Computação 43/97, PUC-Rio, Rio de Janeiro, Brasil, 1997.

[CIR99a] Carlos Cassino, Roberto Ierusalimschy, e Noemi Rodriguez. LuaJava – a scripting tool for java. Monografias em Ciência da Computação 02/99, PUC-Rio, Rio de Janeiro, Brasil, Fevereiro 1999.

[CIR99b] Renato Cerqueira, Roberto Ierusalimschy, e Noemi Rodriguez. *The LuaOrb Manual*. Rio de Janeiro, Brasil, 1999.
(`http://www.tecgraf.puc-rio.br/luaorb/pub/luaorb.ps.gz`).

[CIS92] D. Cowan, R. Ierusalimschy, e T. Stepien. Programming environments for end-users. Em *12th World Computer Congress*, volume 3, páginas 54–60, Madri, Setembro 1992. IFIP.

[Cla96] David Clark, editor. *OLE Automation Programmer's Reference*. Microsoft Press, 1996.

[CN91] Brad Cox e Andrew Novobilski. *Object-oriented Programming: an evolutionary approach*. Addison Wesley, 1991.

[CNM98] Roy H. Campbell, Klara Nahrstedt, e M. Dennis Mickunas. 2K: A Component-Based Network-Centric Operating System. Project home page: `http://choices.cs.uiuc.edu/2K`, 1998.

[CR99] Cynthia Cicalese e Shmuel Rotenstreich. Behavioral specification of distributed software component interfaces. *IEEE Computer*, 32(7):46–53, Julho 1999.

[CRI97] Renato Cerqueira, Noemi Rodriguez, e Roberto Ierusalimschy. Binding an interpreted language to CORBA. Em *II Simpósio Brasileiro de Linguagens de Programação*, páginas 23–36, Campinas, Brasil, 1997.

[CRS95] John Colonna-Romano e Patricia Srite. *The Middleware Source Book*. Butterworth-Heinemann, 1995.

[CSW97] Douglas Cunningham, Eswaran Subrahmanian, e Arthur Westerberg. User-centered evolutionary software development using python and java. Em *Proceedings of the 6th International Python Conference*, 1997.
(`http://www.python.org/workshops/1997-10/proceedings/cunningham.htm`).

[CW85] Luca Cardelli e Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4), 1985.

[DGLM95] M. Day, R. Gruber, B. Liskov, e A. C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. Em *Proceedings of OOPSLA' 95*, 1995.

[dM95] Vicki de Mey. Visual composition of software applications. Em Oscar Nierstrasz e Dennis Tsichritzis, editores, *Object-oriented Software Composition*. Prentice-Hall, 1995.

[DMC92] Christophe Dony, Jacques Malenfant, e Pierre Cointe. Prototype-based languages: From a new taxonomy to constructive proposals and their validation. Em *OOPSLA'92 Proceedings*, páginas 201–217, 1992.

[ECM97] ECMA. *EMCAScript: A general-purpose, cross-platform programming language*, Junho 1997.
(`http://www.omg.org`).

[EJB99] Sun Microsystems. *Enterprise JavaBeans Specification, Version 1.1*, Dezembro 1999.
(`http://java.sun.com/products/ejb`).

[ES90] M. Ellis e B. Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, 1990.

[FBC+98] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, e P. Robin. Supporting adaptive multimedia applications through open bindings. Em *Proceedings of the 4th International Conference on Configurable Distributed Systems (ICCDS'98)*, Annapolis, Maryland, U.S.A., 1998.

[FCCG99] Alexandre Ferreira, Renato Cerqueira, Waldemar Celes, e Marcelo Gattass. Multiple display viewing architecture for virtual environments over heterogeneous networks. Em *SIBGRAPI'99*, páginas 83–92, Campinas, Brasil, 1999. SBC, IEEE Computer Society.

[Fer99] Alexandre Guimarães Ferreira. Uma arquitetura para visualização distribuída de ambientes virtuais. Dissertação de Mestrado, Dep. Informática, PUC-Rio, Rio de Janeiro, Brasil, Dezembro 1999.

[FIC94] Luiz Figueiredo, Roberto Ierusalimschy, e Waldemar Celes. The design and implementation of a language for extending applications. Em *XXI Semish*, páginas 273–283, 1994.

[FIC96] Luiz Figueiredo, Roberto Ierusalimschy, e Waldemar Celes. Lua: An extensible embedded language. *Dr. Dobb's Journal*, 21(12):26–33, Dezembro 1996.

[Fla97] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, segunda edição, 1997.

[FRB+98] Bruno Feijó, Paulo Rodacki, João Bento, Sérgio Scheer, e Renato Cerqueira. Distributed agents supporting event-driven design processes. Em *5th International Conference on Artificial Intelligence in Design'98*, Lisboa, Portugal, Julho 1998.

[FS96] Halldor Fosså e Morris Sloman. Implementing interactive configuration management for distributed systems. Em *Proceedings of the Third International Conference on Configurable Distributed Systems*, páginas 44–51, Annapolis, Maryland, Maio 1996.

[Gat98] Bill Gates. VBA and COM. *Byte*, 23(3):70–72, Março 1998.

[GFCI98] Paulo Gomes, Bruno Feijó, Renato Cerqueira, e Roberto Ierusalimschy. Reactivity and proactiveness in virtual prototyping. Em *2th International Symposium on Tools and Methods for Concurrent Engineering*, páginas 242–253, Manchester, U.K., Abril 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, e John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[GJS96] James Gosling, Bill Joy, e Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[GK96] Kaveh Moazammi Goudarzi e Jeff Kramer. Maintaining node consistency in the face of dynamic change. Em *Proceedings of the Third International Conference on Configurable Distributed Systems*, páginas 62–69, Annapolis, Maryland, Maio 1996.

[Gom99] Paulo César Rodacki Gomes. *Prototipação Virtual em Modelagem de Sólidos Distribuída*. Tese de Doutorado, Dep. Informática, PUC-Rio, Rio de Janeiro, Brasil, Maio 1999.

[GR83] A. Goldberg e D. Robson. *Smalltalk-80 : The Language and its Implementation*. Addison Wesley, 1983.

[HKC+99] Christopher K. Hess, Fabio Kon, Roy H. Campbell, Manuel Román, Dulcineia Carvalho, e Luiz Magalhães. Dynamic Resource Management for Smart Environments: The 2K Approach. Em *Interagency Workshop on Smart Environments*, Atlanta, Georgia, Julho 1999. Georgia Institute of Technology.

[HS97] Galen Hunt e Michael Scott. Coign: Efficient instrumentation for inter-component communication analysis. Relatório Técnico URCS 648, Computer Science Department, University of Rochester, Rochester, New York, Fevereiro 1997.

[HS99] Galen Hunt e Michael Scott. Intercepting and instrumenting COM applications. Em *Proc. 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, páginas 45–56, San Diego, CA, Maio 1999.

[HV99] Michi Henning e Steve Vinoski. *Advanced CORBA Programming with C++*. Addison Wesley, 1999.

[HW96] Steffen Hauptmann e Josef Wasel. On-line maintenance with on-the-fly software replacement. Em *Proceedings of the Third International Conference on Configurable Distributed Systems*, páginas 70–80, Annapolis, Maryland, Maio 1996.

[IB96] Valérie Issarny e Christophe Bidan. Aster: A CORBA-based software interconnection system supporting distributed system customization. Em *Proceedings of the Third International Conference on Configurable Distributed Systems*, páginas 194–201, Annapolis, Maryland, Maio 1996.

[ICR98] Roberto Ierusalimschy, Renato Cerqueira, e Noemi Rodriguez. Using reflexivity to interface with CORBA. Em *International Conference on Computer Languages 1998*, Chicago, 1998. IEEE.

[Ier93] Roberto Ierusalimschy. A denotational approach for type-checking in object-oriented programming languages. *Computer Language*, 19(1):19–40, 1993.

[IFC96] Roberto Ierusalimschy, Luiz Figueiredo, e Waldemar Celes. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[IFC99] Roberto Ierusalimschy, Luiz Figueiredo, e Waldemar Celes. *Reference Manual of the Programming Language Lua version 3.2*. Rio de Janeiro, Brasil, 1999.
(available by ftp at `ftp://ftp.tecgraf.puc-rio.br/pub/lua/refman.ps.gz`).

[IR95] Roberto Ierusalimschy e Noemi Rodriguez. Side effect free functions in object-oriented languages. *Computer Languages*, 21(3):129–146, 1995.

[Jav97] Sun Microsystems. *JavaBeans, Version 1.01*, Julho 1997. (`http://java.sun.com/beans`).

[Jav00] Sun Microsystems. *Using JavaBeans with Microsoft ActiveX Components*, 2000. (`http://java.sun.com/products/plugin/1.2/docs/script.html`).

[JML98] Simon Peyton Jones, Erik Meijer, e Daan Leijen. Scripting COM components in Haskell. Em *Proceedings of the Fifth International Conference on Software Reuse*, Victoria, British Columbia, Junho 1998. IEEE Computer Society Press.

[Joh98] Ray Johnson. *Tcl and Java Integration*, Janeiro 1998. (`http://sunscript.sun.com/java/tcljava.ps`).

[JSLJ99] Bill Janssen, Mike Spreitzer, Dan Larner, e Chris Jacobi. *ILU 2.0beta1 Reference Manual*. Xerox Corporation, Setembro 1999.

[JTMJ00] Bo Jørgensen, Eddy Truyen, Frank Matthijs, e Wouter Joosen. Customization of object request brokers by application specific policies. Em J. Sventek e G. Coulson, editores, *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, páginas 144–163, New York, USA, Abril 2000. Springer-Verlag. (LNCS 1795).

[JWR+99] Matjaž Jurič, Tatjana Welzer, Ivan Rozman, Marjan Heričko, Boštjan Brumen, Tomaž Domajnko, e Aleš Živkovič. Performance assessment framework for distributed object architectures. Em *Proceedings of ADBIS'99*, páginas 349–366. Springer-Verlag, Setembro 1999. (LNCS 1691).

[KC99] Fabio Kon e Roy H. Campbell. Supporting Automatic Configuration of Component-Based Distributed Systems. Em *Proc. 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, páginas 175–187, San Diego, CA, Maio 1999.

[KC00] Fabio Kon e Roy H. Campbell. Dependence Management in Component-Based Distributed Systems. *IEEE Concurrency*, Fevereiro 2000.

[KCC99] Fabio Kon, Dulcineia Carvalho, e Roy Campbell. Automatic Configuration in the 2K Operating System. Em *Proceedings of the ECOOP'99 Workshop on Object Orientation and Operating Systems*, páginas 10–14, Lisboa, Portugal, Junho 1999.

[KdRB91] Gregor Kiczales, Jim des Rivières, e Daniel Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kir97] Mary Kirtland. Object-oriented software development made simple with COM+ runtime services. *Microsoft Systems Journal*, 12(11):49–59, Novembro 1997.

[Kir99] Mary Kirtland. *Designing Component-Based Applications*. Microsoft Press, 1999.

[KMN89] Jeff Kramer, Jeff Magee, e Keng Ng. Graphical configuration programming. *IEEE Computer*, 22(10):53–65, Outubro 1989. (`http://www-dse.doc.ic.ac.uk/~kn/pub.html`).

[KMNS93] Jeff Kramer, Jeff Magee, Keng Ng, e Morris Sloman. The system architect's assistant for design and construction of distributed systems. Em *Proceedings of 4th IEEE Workshop on Future Trends of Distributed Computing Systems*, páginas 284–290, Lisboa, Portugal, Setembro 1993. (`http://www-dse.doc.ic.ac.uk/~kn/pub.html`).

[Kni99] Günter Kniesel. Type-safe delegation for run-time component adaptation. Em *Proceedings of ECOOP'99*. Springer-Verlag, 1999. (LNCS series).

[KP84] Brian Kernighan e Rob Pike. *The UNIX Programming Environment*. Prentice Hall, 1984.

[KRL+00] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, e Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. Em J. Sventek e G. Coulson, editores, *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York, Abril 2000. Springer-Verlag. (LNCS 1795).

[KS00] Fabio Kon e Katia Saikoski, editores. *Proceedings of IFIP/ACM Middleware'2000 Workshop on Reflective Middleware*, IBM Palisades Executive Conference Centre, New York, USA, Abril 2000. (`http://www.comp.lancs.ac.uk/computing/RM2000`).

[KSC+98] Fabio Kon, Ashish Singhai, Roy H. Campbell, Dulcineia Carvalho, Robert Moore, e Francisco J. Ballesteros. 2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments. Em *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Bruxelas, Bélgica, Julho 1998.

[LAN00] Markus Lumpe, Franz Achermann, e Oscar Nierstrasz. A formal language for composition. Em Gary Leavens e Murali Sitaraman, editores, *Foundations of Component Based Systems*. Cambridge University Press, 2000.

[Lim98] Michele Lima. LuaMan — uma plataforma para desenvolvimento de aplicações de gerenciamento extensíveis. Dissertação de Mestrado, Dep. Informática, PUC-Rio, Rio de Janeiro, Brasil, Janeiro 1998.

[Lis97] Maria Lúcia Blanck Lisbôa. Reflexão computacional no modelo de objetos. Tutorial do II Simpósio Brasileiro de Linguagens de Programação, Campinas, Brasil, Setembro 1997.

[LK98] Cristina Lopes e Gregor Kiczales. Recent developments in AspectJ. Em Serge Demeyer e Jan Bosch, editores, *ECOOP'98 Workshop Reader*. Springer-Verlag, 1998. (LNCS 1543).

[LSN96] Markus Lumpe, Jean-Guy Schneider, e Oscar Nierstrasz. Using metaobjects to model concurrent objects with PICT. Em *Proceedings of Languages et Modèles à Objets*, páginas 1–12, Leysin, Switzerland, Outubro 1996.

[Lun89] C. Lunau. Separation of hierarchies in Duo-Talk. *Journal of Object-Oriented Programming*, 2(2):20–26, 1989.

[Lut96] Mark Lutz. *Programming Python: Object-Oriented Scripting*. O'Reilly & Associates, 1996.

[Mad99] Ole Lehrmann Madsen. COM support in BETA. Em *Proceedings of Fourth International Workshop on Component-Oriented Programming*, Lisboa, Portugal, Junho 1999. (em conjunto com ECOOP'99).

[Man95] Frank Manola. X3H7 object model features matrix. Relatório Técnico X3H7-93-007v10, Accredited Standards Committee X3, Technical Committee X3H7, Fevereiro 1995.

[Mar98] Marco Aurélio Catunda Martins. Extensão dinâmica de agentes CORBA. Dissertação de Mestrado, Dep. Informática, PUC-Rio, Rio de Janeiro, Brasil, Setembro 1998.

[McC00] John McCoy. Scripting for Pnuts. *Dr. Dobb's Journal*, 25(1):21–26, Janeiro 2000.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, e Jeff Kramer. Specifying distributed software architectures. Em *Proceeddings of the Fifth European Software Engineering Conference*, Barcelona, Espanha, Setembro 1995.

[MDK94] Jeff Magee, Naranker Dulay, e Jeff Kramer. Regis: A constructive development environment for distributed programs. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 1(5):304–312, Setembro 1994.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[Mey92] Bertrand Meyer. Design by contract. Em Dino Mandrioli e Bertrand Meyer, editores, *Advances in Object-Oriented Software Engineering*. Prentice-Hall, 1992.

[MGG96] Philippe Merle, Christophe Gransart, e Jean-Marc Geib. CorbaScript and CorbaWeb: A generic object-oriented dynamic environment upon CORBA. Em *Proceedings of TOOLS Europe'96*, Paris, França, Fevereiro 1996. Prentice-Hall.

[MGG97] Philippe Merle, Christophe Gransart, e Jean-Marc Geib. Using and implementing CORBA objects with CorbaScript. Em *OBPDC'97 – Object-Based Parallel and Distributed Computing*, Toulouse, França, Outubro 1997.

[Mic00] *Mico is CORBA, version 2.3.3*, 2000.
(`http://www.mico.org`).

[ML97] Ana Moura e Michele Lima. *A API de gerência de redes LuaMan*. PUC-Rio, 1997.
(`http://www.telemidia.puc-rio.br/~ana/luaman.html`).

[Moo98] Robert Byron Moore. An extensible architecture for distributed object system interoperability. Dissertação de Mestrado, Department of Computer Science, University of Illinois at Urbana-Champaign, Agosto 1998. Disponível em `http://choices.cs.uiuc.edu/2k`.

[Mou99] Ana Moura. Comparação do desempenho de aplicações distribuídas utilizando LuaOrb e C++, 1999.
(`http://www.inf.puc-rio.br/~noemi/alm-tad/`).

[MRI99] Marco Martins, Noemi Rodriguez, e Roberto Ierusalimschy. Dynamic extension of CORBA servers. Em *Euro-Par'99 Parallel Processing*, páginas 1369–1376, Toulouse, França, Setembro 1999. Springer-Verlag. (LNCS 1685).

[MT97] Nenad Medvidovic e Richard Taylor. A framework for classifying and comparing architecture description languages. Em Mehdi Jazayeri e Helmut Schauer, editores, *Proceedings of ESEC'97*, páginas 60–76. Springer-Verlag, Setembro 1997. (LNCS 1301).

[MUWZ96] Naftaly Minsky, Victoria Ungureanu, Wenhui Wang, e Junbiao Zhang. Building reconfiguration primitives into the law of a system. Em *Proceedings of the Third International Conference on Configurable Distributed Systems*, páginas 89–97, Annapolis, Maryland, Maio 1996.

[NH94] Farshad Nayeri e Ben Hurwitz. Generalizing dispatching in a distributed object system. Em *Proceedings of the Eighth European Conference on Object Oriented Programming ECOOP 94*, 1994.

[Nie92] Oscar Nierstrasz. Towards an object calculus. Em *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, páginas 1–20. Springer-Verlag, 1992. (LNCS 612).

[Nie93] Oscar Nierstrasz. Composing active objects. Em G. Agha, P. Wegner, e A. Yonezawa, editores, *Research Directions in Object-Based Concurrency*, páginas 151–171. MIT Press, 1993.

[NK95] Keng Ng e Jeff Kramer. Automated support for distributed software design. Em *Proceedings of the 7th International Workshop on Computer-Aided Software Engineering*, páginas 381–390, Toronto, Julho 1995.
(`http://www-dse.doc.ic.ac.uk/~kn/pub.html`).

[NKM96] Keng Ng, Jeff Kramer, e Jeff Magee. A CASE tool for software architecture design. *Journal of Automated Software Engineering*, 3(3):261–284, Agosto 1996.
(`http://www-dse.doc.ic.ac.uk/~kn/pub.html`).

[NKMD95] Keng Ng, Jeff Kramer, Jeff Magee, e Naranker Dulay. The Software Architect's Assistant – a visual environment for distributed programming. Em *Proceedings of Hawaii International Conference on System Sciences*, páginas 254–263, Janeiro 1995.
(`http://www-dse.doc.ic.ac.uk/~kn/pub.html`).

[NKMD96] Keng Ng, Jeff Kramer, Jeff Magee, e Naranker Dulay. A visual approach to distributed programming. Em A. Zaky e T. Lewis, editores, *Tools and Environments for Parallel and Distributed Systems*. Kluwer Academic Publishers, Fevereiro 1996.
(`http://www-dse.doc.ic.ac.uk/~kn/pub.html`).

[NM94] Oscar Nierstrasz e Theo Meijler. Requirements for a composition language. Em *Proceedings of the ECOOP 94 workshop on Models and Languages for Coordination of Parallelism and Distribution*. Springer-Verlag, 1994. (LNCS).

[NSL96] Oscar Nierstrasz, Jean-Guy Schneider, e Markus Lumpe. Formalizing composable software systems — a research agenda. Em *Proceedings of the First IFIP Workshop on Formal Methods for Open Object-based Distributed Systems FMOODS'96*, páginas 271–282, Paris, França, Março 1996.

[NT95] Oscar Nierstrasz e Dennis Tsichritzis, editores. *Object-oriented Software Composition*. Prentice-Hall, 1995.

[OB99] Alexandre Oliva e Luiz Buzato. The design and implementation of Guaraná. Em *Proc. 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, San Diego, CA, Maio 1999.

[OH97] Robert Orfali e Dan Harkey. *Client/Server Programming with Java and CORBA*. John Wiley & Sons, 1997.

[OMG92] OMG. *Object Management Architecture Guide 2.0*, Setembro 1992. (`http://www.omg.org`).

[OMG96] OMG. CORBA Component Model — Request For Proposal. Relatório Técnico orbos/96-06-12, OMG, 1996. (`http://www.omg.org`).

[OMG97a] OMG. CORBA Component Imperatives. Relatório Técnico orbos/97-05-25, OMG, 1997. (`http://www.omg.org`).

[OMG97b] OMG. CORBA Scripting Language — Request For Proposal. Relatório Técnico orbos/97-05-24, OMG, 1997. (`http://www.omg.org`).

[OMG97c] OMG. CORBAservices: Common object services specification. Relatório Técnico formal/97-12-02, OMG, 1997. (`http://www.omg.org`).

[OMG98] OMG, Framingham, MA. *The Common Object Request Broker Architecture and Specification; Revision 2.2*, Fevereiro 1998. (`http://www.omg.org`).

[OMG99a] OMG. CORBA Component Model Joint Revised Submission. Relatório Técnico orbos/99-07-01, OMG, 1999. (`http://www.omg.org`).

[OMG99b] OMG. CORBA Component Scripting — Revised Joint Submission. Relatório Técnico orbos/99-08-01, OMG, 1999. (`http://www.omg.org`).

[OMG99c] OMG. Java Language to IDL Mapping. Relatório Técnico ptc/99-03-09, OMG, 1999. (`http://www.omg.org`).

[OMG99d] OMG. Notification service specification. Relatório Técnico telecom/99-07-01, OMG, Julho 1999. (`http://www.omg.org`).

[OMG99e] OMG. Portable Interceptors Joint Revised Submission. Relatório Técnico orbos/99-12-02, OMG, 1999. (`http://www.omg.org`).

[OMG00] OMG. Python language mapping specification. Relatório Técnico ptc/00-01-12, OMG, 2000. (`http://www.omg.org/cgi-bin/doc?ptc/00-01-12`).

[OOC98] Object-Oriented Concepts, Inc. *ORBacus for C++ and Java, Version 3.0*, 1998. (`http://www.ooc.com`).

[Ous90] John Ousterhout. Tcl: an embeddable command language. Em *Proc. of the Winter 1990 USENIX Conference*. USENIX Association, 1990.

[Ous94] John Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[Ous98] John Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, Março 1998.

[PA98] George Papadopoulos e Farhad Arbab. Coordination models and languages. *Advances in Computers*, 46:329–400, Agosto 1998.

[PCS98] Jim Purtilo, Robert Cole, e Rick Schlichting, editores. *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, Annapolis, Maryland, Maio 1998. IEEE Computer Society.

[Pil00] Frank Pilhofer. *Combat*, Março 2000. (`http://www.informatik.uni-frankfurt.de/~fp/Tcl/Combat/`).

[PMS96] Jim Purtilo, Jeff Magee, e Karsten Schwan, editores. *Proceedings of the Third International Conference on Configurable Distributed Systems*, Annapolis, Maryland, Maio 1996. IEEE Computer Society.

[Pri99] Jason Pritchard. *COM and CORBA Side by Side: Architectures, Strategies, and Implementations*. Addison Wesley, 1999.

[PS93] Vern Paxson e Chris Saltmarsh. Glish: A user-level software bus for loosely-coupled distributed systems. Em *Proceedings of USENIX'93*, Janeiro 1993.

[RC98] Michael Rosen e David Curtis. *Integrating CORBA and COM Applications*. John Wiley & Sons, 1998.

[RIC98] Noemi Rodriguez, Roberto Ierusalimschy, e Renato Cerqueira. Dynamic configuration with CORBA components. Em *4th International Conference on Configurable Distributed Systems*, Annapolis, 1998.

[RIR93] Noemi Rodriguez, Roberto Ierusalimschy, e José Rangel. Types in School. *ACM SIGPLAN Notices*, 28(8), 1993.

[RKC99] Manuel Román, Fabio Kon, e Roy H. Campbell. Design and Implementation of Runtime Reflection in Communication Middleware: the *dynamicTAO* Case. Em *Proceedings of the ICDCS'99 Workshop on Middleware*, páginas 122–127, Austin, TX, Junho 1999. IEEE Computer Society.

[RKF92] Ward Rosenberry, David Kenney, e Gerry Fisher. *Understanding DCE*. O'Reilly, 1992.

[RMIC97] Noemi Rodriguez, Ana Moura, Roberto Ierusalimschy, e Renato Cerqueira. Aplicações de gerência com comportamento dinâmico. Em *Anais do SFBSID'97*, páginas 476–487, Fortaleza, Brasil, 1997.

[Rog97] Dale Rogerson. *Inside COM*. Microsoft Press, 1997.

[RSC+99] Manuel Román, Ashish Singhai, Dulcineia Carvalho, Christopher Hess, e Roy H. Campbell. Integrating PDAs into Distributed Operating Systems: 2K and PalmORB. Em *Intenational Symposium on Handheld and Unbiquitous Computing (HUC'99)*, volume LNCS 1707, páginas 137–149, Karlsruhe, Alemanha, Setembro 1999. Springer-Verlag.

[RT93] Ward Rosenberry e Jim Teague. *Distributing Applications Across DCE and Windows NT*. O'Reilly, 1993.

[Sam97] J. Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag, 1997.

[Sch99a] Douglas C. Schmidt. *Real-time CORBA with TAO (The ACE ORB)*, 1999. (`http://www.cs.wustl.edu/~schmidt/TAO.html`).

[Sch99b] Jean-Guy Schneider. *Components, Scripts, and Glue: A conceptual framework for software composition*. Tese de Doutorado, University of Bern, Institute of Computer Science and Applied Mathematics, Outubro 1999.

[SDZ96] Mary Shaw, Robert DeLine, e Gregory Zelesnik. Abstractions and implementations for architectural connections. Em *Proceedings of the Third International Conference on Configurable Distributed Systems*, páginas 2–10, Annapolis, Maryland, Maio 1996.

[SG96a] Mary Shaw e David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[SG96b] Narinder Singh e Mark Gisi. Coordinating distributed objects with declarative interfaces. Em *Coordination Languages and Models*, páginas 368–385. Springer, 1996. (LNCS 1061).

[Sie96] Jon Siegel. *CORBA Fundamentals and Programming*. John Wiley & Sons, 1996.

[SLP98] Marcelo Sant'Anna, Julio Leite, e Antonio Prado. A generative approach to componentware. Em *Proceedings of CBSE'98 — International Workshop on Component Based Software Engineering*, Kyoto, Japão, Fevereiro 1998.
(`http://www.les.inf.puc-rio.br/~draco/cbse98.ps.gz`).

[SLU88] Lynn Andrea Stein, Henry Lieberman, e David Ungar. A shared view of sharing: The Treaty of Orlando. Em Won Kim e Fred Lochovsky, editores, *Object Oriented Concepts, Applications, and Databases*. ACM Press, 1988.

[SN98] Jean-Guy Schneider e Oscar Nierstrasz. Scripting: Higher-level programming for component-based systems. Tutorial OOPSLA 1998, 1998.
(`http://www.iam.unibe.ch/~scg`).

[SN99] Jean-Guy Schneider e Oscar Nierstrasz. Components, scripts and glue. Em Leonor Barroca, Jon Hall, e Patrick Hall, editores, *Software Architectures – Advances and Applications*, páginas 13–25. Springer, 1999.

[Sny86] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. Em *OOPSLA'86 Proceedings*, páginas 38–45, 1986.

[SSC97] Ashish Singhai, Aamod Sane, e Roy Campbell. Reflective ORBs: Supporting Robust Time-Critical Distribution. Em *Proceedings of the ECOOP'97 Workshop on Reflective Real-Time Object-Oriented Systems*, páginas 55–61, Finlândia, Junho 1997. ECOOP'97 Workshop Reader, LNCS 1357.

[Sta91] Manfred Stadel. Object oriented programming techniques to replace software components on the fly in a running program. *ACM SIGPLAN Notices*, 26(1):99–108, 1991.

[Sta98] Scott Stanton. Tclblend: Blending tcl and java. *Dr. Dobb's Journal*, páginas 50–54, Fevereiro 1998.

[Ste94] Patrick Steyaert. *Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks*. Tese de Doutorado, Vrije Universiteit Brussel, 1994.

[SU95] Randall Smith e David Ungar. Programming as an experience: The inspiration for Self. Em *Proceedings of the Nineth European Conference on Object Oriented Programming ECOOP'95*, 1995.

[Sun88] Sun Microsystems. *RPC: Remote Procedure Call, Protocol Specification, Version 2*, Junho 1988. RFC 1057.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 1998.

[Tho96] James Thornton. Practical description of configurations for distributed systems management. Em *Proceedings of the Third International Conference on Configurable Distributed Systems*, páginas 36–43, Annapolis, Maryland, Maio 1996.

[Tho98] Anne Thomas. Enterprise JavaBeans technology. Relatório técnico, Patricia Seybold Group, Dezembro 1998. Preparado para Sun Microsystems, Inc.
(`http://java.sun.com/products/ejb/white_paper.html`).

[TT95] John Toohey e Edward Toupin. *Building OCXs*. QUE, 1995.

[Ude94] J. Udell. Componentware. *Byte*, 19(5):46–56, 1994.

[Vis96] Visigenic Software, Inc. *VisiBroker for C++ Programmer's Guide, Version 2.0*, 1996.

[vR98] Guido van Rossum. Java and Python: a perfect couple. *developer.com*, Agosto 1998.
(`http://www.developer.com/journal/techfocus/081798_jpython.html`).

[Weg87] Peter Wegner. Dimensions of object-based language design. Em *Proceedings of OOPSLA'87*, páginas 168–182, Outubro 1987.

[Win94] J. M. Wing. Proceedings of the workshop on interface definition languages. *ACM SIGPLAN Notices*, 29(8), 1994.

[WK95] Sara Williams e Charlie Kindel. The problem with implementation inheritance. *Dr. Dobb's Journal*, 19(16):18, 1995.

[WL98] Yi-Min Wang e Woei-Jyh Lee. COMERA: COM extensible remoting architecture. Em *Proc. 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'98)*, páginas 79–88, Abril 1998.

[WS91] L. Wall e R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 1991.

[WS96] Ian Warren e Ian Sommerville. A model for dynamic configuration which preserves application integrity. Em *Proceedings of the Third International Conference on Configurable Distributed Systems*, páginas 81–88, Annapolis, Maryland, Maio 1996.

[WSL00] N. Wang, D. Schmidt, e D. Levine. Optimizing the corba component model for high-performance and real-time applications, 2000. Work-in-Progress Paper apresentado em Middleware'2000.

[WUK99] Guijun Wang, Liz Ungar, e Dan Klawitter. Component assembly for OO distributed systems. *IEEE Computer*, 32(7):71–78, Julho 1999.

[YS94] Daniel Yellin e Robert Strom. Interfaces, protocols, and the semi-automatic construction of software adaptors. Em *OOPSLA'94 Conference Proceedings*, Outubro 1994.

[YS97] Daniel Yellin e Robert Strom. Protocol specification and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, Março 1997.

[ZM97] N. Zuquello e E. Madeira. A mechanism to provide interoperability between ORBs with relocation transparency. Em *Proceedings of IEEE Third International Symposium on Autonomous Decentralized Systems (ISADS'97)*, páginas 195–202, Berlim, Alemanha, Abril 1997. IEEE.