Nélio Alessandro Azevedo Cacho

Um Middleware Reflexivo e Orientado a Aspectos: Arquitetura e Implementações

Natal – RN

Dezembro de 2006

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

Nélio Alessandro Azevedo Cacho

Um Middleware Reflexivo e Orientado a Aspectos: Arquitetura e Implementações

Dissertação de mestrado submetida à banca examinadora como parte dos requisitos para obtenção do título de mestre no Programa de Pós-graduação em Sistemas e Computação da Universidade Federal do Rio Grande do Norte.

Orientador:
Profa. Dra. Thaís Vasconcelos Batista

Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Curso de Ciências da Computação

Natal - RN

Dezembro de 2006

Dissertação de mestrado, sob o título *"Um Middleware Reflexivo e Orientado a Aspectos: Arquitetura e Implementações"*, submetida à banca examinadora como parte dos requisitos para obtenção do título de mestre no Programa de Pós-graduação em Sistemas e Computação da Universidade Federal do Rio Grande do Norte.

Profa. Dra. Thais Vasconcelos Batista
UFRN

Profa. Dra. Flavia Coimbra Delicato
UFRN

Profa. Dra. Noemi de La Rocque Rodriguez
PUC-Rio

"Nosso maior desejo na vida
é encontrar alguém que nos faça fazer o melhor que pudermos."
Ralph Waldo Emerson

A minha esposa Andréa pelo amor e dedicação.
Aos meus pais Luiz e Josita e a minha avó Francisca,
cujo exemplo de honestidade e trabalho
tem sido um norteador para a minha vida.

Agradecimentos

Dedico meus sinceros agradecimentos:

–A Deus pelo dom da minha vida.

–A minha esposa Andréa, pelos incentivos dados, pela dedicação e paciência que sempre demonstrou ao longo do curso.

– Meus pais Luiz e Josita, minha avó Francisca, meus irmãos Sheyla, Ricardo e Ênio e meu cunhado Josivan que sempre me ajudaram e construíram comigo um ambiente familiar favorável ao aprendizado;

– A Profa. Thaís, pela orientação, incentivo, confiança em mim depositada e pela amizade estabelecida por todos esses cinco anos de orientação;

– Ao Dr. Alessandro Garcia, pelos preciosos comentários e sugestões que foram utilizados para enriquecer ainda mais esta dissertação;

– Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico(CNPq) pelo suporte financeiro que tornou boa parte desse trabalho possível;

– Ao Departamento de Informática e Matemática Aplicada(DIMAp), nas pessoas dos professores e funcionários, pelo acolhimento durante todo meu curso de mestrado;

– A todas as amizades construídas durante esses dois anos de mestrado, que de alguma forma contribuíram para o meu desenvolvimento.

Resumo

Plataformas de middleware têm sido utilizadas em diversos ambientes computacionais e por diferentes classes de aplicações com requisitos variados. Para atender a cada cenário específico, é necessário conferir às plataformas de middleware capacidade de adaptação da sua infra-estrutura conforme as necessidades das aplicações e do ambiente computacional. Atualmente, uma nova geração de plataformas de middleware tem explorado o conceito de reflexão computacional para possibilitar adaptação dinâmica. No entanto, esta nova geração de plataformas não endereçam os problemas introduzidos pela presença dos *conceitos transversais* entrelaçados com os conceitos básicos, que reduzem o potencial reuso do middleware em diferentes cenários e, por conseqüência, limitam a capacidade de adaptação.

No presente trabalho, combinamos reflexão computacional com programação orientada a aspectos para permitir a separação dos conceitos transversais e assim melhorar o reuso e capacidade de adaptação das plataformas de middleware. Esta combinação é usada na especificação de uma nova estrutura para a arquitetura do Open-ORB. Tal arquitetura é implementada usando duas estratégias distintas: (i) uma estratégia interpretada utilizando a linguagem Lua em combinação com AspectLua; (ii) uma estratégia compilada que emprega a linguagem Java em combinação com AspectJ. De forma a avaliar os benefícios de cada estratégia, este trabalho apresenta uma série de comparações que traduzem as diferenças em termos de modularidade, memória utilizada e tempo de execução de cada uma das estratégias.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução p. 13

1.1 Motivação . . . . p. 14

1.2 Objetivos . . . . p. 17

1.3 Estrutura do trabalho . . . . p. 18

2 Fundamentação Teórica p. 19

2.1 Desenvolvimento baseado em Componentes . . . . p. 19

2.2 Middlewares Reflexivos . . . . p. 20

2.2.1 Arquitetura do Open-ORB . . . . p. 21

2.2.1.1 Elementos do nível-base . . . . p. 22

2.2.1.2 Elementos do meta-nível . . . . p. 23

2.3 Programação Orientada a Aspectos . . . . p. 24

2.3.1 Linguagens para a definição de Aspectos . . . . p. 25

2.3.1.1 AspectJ . . . . p. 25

2.3.1.2 AspectLua . . . . p. 27

2.4 Padrões de Projeto . . . . p. 30

3 Arquitetura e Implementações do Open-ORB p. 33

3.1 Arquitetura estendida do Open-ORB . . . . p. 33

3.1.1 Funcionalidades do Nível-Base . . . . p. 33

3.1.2 Funcionalidades do Meta-nível . . . . p. 34

3.1.3 Utilizando abstrações de alto nível na construção da infra-estrutura de componentes . . . . p. 36

3.1.4 Tratando invocações . . . . p. 38

3.1.5 Realizando invocações . . . . p. 39

3.1.6 Utilizando a infra-estrutura de comunicação do Open-ORB . . . p. 41

3.2 LOpenOrb: Implementação Lua do Open-ORB . . . . p. 43

3.2.1 Arquitetura . . . . p. 43

3.2.2 Funcionalidades do Nível-Base . . . . p. 45

3.2.3 Funcionalidades do Meta-nível . . . . p. 45

3.2.4 Infra-estrutura de componentes . . . . p. 47

3.3 JOpenOrb: Implementação Java do Open-ORB . . . . p. 51

3.3.1 Arquitetura . . . . p. 51

3.3.2 Funcionalidades do Nível-Base . . . . p. 55

3.3.3 Funcionalidades do Meta-nível . . . . p. 56

4 Estratégias de Aspectização p. 58

4.1 Separação de conceitos em plataformas de Middleware . . . . p. 58

4.2 Arquitetura do Aspect Open-ORB . . . . p. 62

4.3 Estratégia para aspectização do LOpenOrb . . . . p. 63

4.4 Estratégia para aspectização do JOpenOrb . . . . p. 65

5 Avaliação p. 69

5.1 Comparando as soluções aspectizadas versus não aspectizadas . . . . . p. 69

5.1.1 Comparando LOpenOrb com Aspect Open-ORB . . . . p. 70

5.1.2 Comparando JOpenOrb com Aspect Open-ORB . . . . p. 72

5.2 Avaliando as soluções entre as diferentes linguagens . . . . p. 76

5.2.1 Comparando LOpenOrb com JOpenOrb . . . . p. 77
5.2.2 Comparando as versões Lua e Java do Aspect OpenORB . . . . p. 79

6 Trabalhos relacionados p. 82
6.1 FlexiNet . . . . p. 82
6.2 Jonathan . . . . p. 83
6.3 dynamicTAO . . . . p. 83
6.4 Lasagne . . . . p. 84
6.5 JAC . . . . p. 84
6.6 JBoss AOP . . . . p. 85
6.7 Jacobsen et al. . . . . p. 86
6.8 Avaliação . . . . p. 86

7 Considerações Finais p. 88
7.1 Contribuições . . . . p. 89
7.2 Trabalhos Futuros . . . . p. 89

Referências p. 91

Lista de Figuras

1 Camadas em um sistema de middleware . . . . p. 16

2 Arquitetura do middleware reflexivo . . . . p. 20

3 Exemplo de um componente composto . . . . p. 22

4 Um objeto e seus quatro meta-modelos (ANDERSEN, 2002) . . . . p. 24

5 Exemplo da utilização de AspectJ . . . . p. 26

6 Exemplo da utilização do AspectLua . . . . p. 29

7 Definindo ordem para invocação de aspectos . . . . p. 30

8 Padrão de Projeto *Decorator* . . . . p. 32

9 Caminho de execução do servidor . . . . p. 39

10 Caminho de invocação do cliente . . . . p. 39

11 Interfaces de um Operational Binding (ANDERSEN, 2002) . . . . p. 40

12 Arquitetura do LOpenORB . . . . p. 43

13 Definição do “pacote”LOpenORB. . . . . p. 45

14 Criando um binding local entre duas interfaces . . . . p. 46

15 Utilizando um meta-objeto encapsulation . . . . p. 46

16 Utilizando um meta-objeto composition . . . . p. 46

17 Criando os componentes Capsule e NodeMngr . . . . p. 47

18 Definindo Caminho de execução do servidor . . . . p. 48

19 Definindo Caminho de invocação do servidor . . . . p. 49

20 Código do servidor de aplicação . . . . p. 50

21 Código do cliente utilizando um Op. *binding* . . . . p. 50

22 Arquitetura JOpenOrb . . . . p. 53

23 Combinação do Padrão Proxy com Mediator . . . . p. 53

24 Código do cliente utilizando um Op. *binding* . . . . p. 54

25 Padrões de projeto na implementação da reflexão computacional . . . . p. 56

26 Exemplo de utilização do JOpenOrb . . . . p. 56

27 Exemplo de utilização do JOpenOrb . . . . p. 57

28 Evolução do JacORB em relação ao número de classes vs. suas versões p. 59

29 Ilustração da combinação dos aspectos com as camadas . . . . p. 60

30 Plataformas de software para as versões Lua e Java . . . . p. 61

31 Arquitetura de uma aplicação desenvolvida sobre o Aspect Open-ORB p. 62

32 Configurando os aspectos do middleware . . . . p. 64

33 Aspecto abstrato para a definição do padrão Chain of Responsability . p. 67

34 Definição do aspecto concreto para o padrão Chain of Responsability . p. 67

35 Aspecto abstrato usado para implementar o mecanismo de detecção . . p. 68

36 Aspecto concreto usado para definir os pontos de interceptação . . . . . p. 68

37 Gráfico de memória utilizada . . . . p. 71

38 Definição das métricas de separação de conceitos, acoplamento, coesão e tamanho . . . . p. 73

39 Quantificação da separação de conceitos . . . . p. 74

40 Quantificando acoplamento, coesão e tamanho . . . . p. 75

41 Gráfico de criação de Operational binding . . . . p. 79

Lista de Tabelas

1 *Pointcut designators* suportados pelo AspectJ. . . . . p. 27

2 Funcionalidades do Nível-Base. . . . . p. 34

3 Métodos fornecidos por um meta-objeto Encapsulation para interfaces. p. 35

4 Métodos fornecidos por um meta-objeto *Composition*. . . . . p. 36

5 Funcionalidades do componente Capsule. . . . . p. 37

6 Funcionalidades do componente NodeMngr. . . . . p. 37

7 Funcionalidades do serviço de *Naming* do Open-ORB. . . . . p. 41

8 Padrões de Projeto utilizados na implementação do JOpenOrb. . . . . . p. 52

9 Relação de dependência entre os módulos do LOpenORB. . . . . p. 63

10 Relação entre funções do JOpenORB e padrões de projeto. . . . . p. 66

11 Resultados dos testes comparativos entre o LOpenORB e o Aspect Open-ORB(Tempo em μs) . . . . p. 71

12 Resultado da comparação entre JOpenORB e Aspect Open-ORB . . . p. 76

13 Resultado da comparação entre LOpenORB e JOpenORB . . . . p. 77

14 Resultado da capacidade de customização . . . . p. 80

15 Resultado da comparação entre as duas versões do Aspect OpenORB . p. 81

16 Resumo das avaliações . . . . p. 86

1 Introdução

Plataformas de middleware (BERNSTEIN, 1996) têm sido utilizadas em diversos ambientes computacionais e por diferentes classes de aplicações com requisitos variados. Para atender a cada cenário específico, é necessário conferir ao middleware capacidade de adaptação dinâmica da sua infra-estrutura conforme as necessidades das aplicações e do ambiente computacional.

Atualmente, uma nova geração de plataformas de middleware(BLAIR et al., 2001b; KON et al., 2000) tem explorado o conceito de *reflexão computacional* (SMITH, 1982) para possibilitar adaptação dinâmica. O propósito dessas novas plataformas é superar as limitações impostas pelas arquiteturas *black-box* que impõem ao middleware a restrição de fornecer apenas um número pré-determinado de funcionalidades para seus usuários, não sendo possível assim visualizar ou alterar a implementação dessas funcionalidades. Dessa forma, a complexidade estrutural criada exatamente pela forma como o middleware é implementado impede que funcionalidades específicas sejam adicionadas ou removidas de acordo com os requisitos da aplicação alvo.

No entanto, apesar da reflexão computacional introduzir novas abstrações que permitem adicionar e remover funcionalidades do middleware, tais abstrações não resolvem o problema do entrelaçamento de um grande número de conceitos, chamados *conceitos transversais*, com os elementos básicos do middleware. Conceitos transversais tais como persistência, comunicação transacional, segurança, qualidade de serviço e sincronização não são modularizados através da reflexão computacional e, portanto, permanecem entrelaçados com os conceitos básicos, reduzindo assim o entendimento da implementação e o potencial reuso do middleware em diferentes contextos. De forma a resolver esse problema, o presente trabalho combina reflexão computacional com um paradigma emergente, a Programação Orientada a Aspectos (POA) (KICZALES et al., 1997) para simplificar a complexidade estrutural dos middlewares reflexivos através de uma clara separação dos conceitos transversais. Esses conceitos transversais são modularizados pela POA em elementos denominados *aspectos*.

Usando tal combinação para compor plataformas de middleware, o núcleo do middleware contém apenas as suas funcionalidades básicas. Outras funcionalidades que implementam requisitos específicos são representados por aspectos.

Nesse trabalho usamos a arquitetura do Open-ORB (BLAIR et al., 2001b), por ser um dos precursores em termos de middleware reflexivo. A partir de tal arquitetura, definimos uma nova estruturação para os elementos internos desse middleware, de forma a separar, no nível base, os elementos básicos (componentes) dos elementos que implementam funcionalidades específicas (aspectos). A partir da nova arquitetura, denominada de *Aspect Open-ORB* (CACHO; BATISTA, 2005b) instâncias da plataforma corresponderão à configuração de componentes e aspectos que atendem aos requisitos de aplicações de determinados domínios ou de um ambiente específico.

De forma a validar esse trabalho, implementamos dois protótipos do Aspect Open-ORB: (1) utilizando uma linguagem dinamicamente tipada com facilidades reflexivas, a linguagem Lua(IERUSALIMSCHY; FIGUEIREDO; FILHO, 1996), que dispõe de uma extensão para programação orientada a aspectos, AspectLua (CACHO; BATISTA; FERNANDES, 2005, 2006); (2) utilizando a linguagem compilada Java que dispõe de uma extensão para programação orientada a aspectos, AspectJ (TEAM, 2002). Adicionalmente, a modularidade da versão Java foi analisada por meio de um conjunto de métricas (GARCIA et al., 2003) para verificar se a POA melhorou a modularidade da versão aspectizada.

1.1 Motivação

No contexto de desenvolvimento de aplicações distribuídas, plataformas de middleware têm sido utilizadas como infra-estrutura subjacente oferecendo transparência de distribuição, de heterogeneidade e disponibilizando serviços comuns como nomes, eventos, segurança, tolerância a falhas, entre outros. Tipicamente, a estruturação das plataformas de middleware segue uma arquitetura monolítica agregando diversas características para atender a uma gama de aplicações. Tal arquitetura é inadequada face à diversidade de aplicações e de ambientes de hardware e de software com características distintas que exigem que o middleware ofereça serviços especializados determinados caso a caso. Portanto, o comportamento dinâmico é um requisito fundamental para o middleware de forma a permitir que protocolos e serviços sejam inseridos e removidos dinamicamente na arquitetura do middleware conforme as necessidades das aplicações e do ambiente de hardware e de software.

Para endereçar esse problema surgiu a idéia de *Middleware de próxima geração* (AGHA, 2002; BLAIR et al., 1998; TRIPATHI, 2002). Segundo (AGHA, 2002), existem dois aspectos fundamentais relacionados ao desenvolvimento de middleware da próxima geração: adaptação dinâmica e abstrações de alto nível. Adaptação dinâmica oferece a flexibilidade necessária para adaptar o middleware de acordo com os requisitos da aplicação enquanto que abstrações de alto nível simplificam o problema de expressar padrões complexos de interação entre os elementos que compõem o middleware.

A nova geração de plataformas de middleware (BLAIR et al., 2001b; KON et al., 2000) explora o conceito de reflexão computacional (SMITH, 1982) para possibilitar adaptação dinâmica. A reflexão é utilizada para permitir a inspeção do comportamento interno no sistema e, conseqüentemente, a adaptação de acordo com as necessidades da aplicação ou do ambiente de hardware. No modelo de middleware reflexivo, o middleware é implementado como uma coleção de componentes (SZYPERSKI, 2002) que podem ser configurados e reconfigurados pela aplicação (KON et al., 2002). Portanto, o modelo de componentes utilizado para estruturar internamente a plataforma de middleware é fundamental para viabilizar a adaptação.

Internamente a plataforma de middleware é composta por uma variedade de elementos que devem ser combinados, de diferentes maneiras, para atender à requisitos específicos de aplicações e de ambientes computacionais distintos. Para gerenciar essa complexidade, o Open-ORB (BLAIR et al., 2001b), uma arquitetura de middleware reflexivo, usa meta-modelos e meta-informação. O Open-ORB define uma arquitetura de meta-níveis composta por um *nível base*, formado por componentes que implementam os serviços do middleware, e pelo *meta-nível*, que provê reflexão computacional permitindo inspeção e adaptação do middleware.

A Figura 1 ilustra as três camadas que compõem uma aplicação que usa um middleware reflexivo. Na camada de infra-estrutura básica estão as classes que definem a estrutura de reflexão juntamente com as classes que definem o modelo de componentes. Na camada de infra-estrutura de componentes, os elementos definidos na camada básica são instanciados para compor os serviços que implementam a conectividade do middleware e suas funcionalidades adicionais. Finalmente, na camada de aplicação o usuário utiliza a API provida pelas camadas anteriores para estruturar a aplicação desejada.

Baseando-se nessa arquitetura em camadas, o Open-ORB suporta reflexão apenas para as duas camadas superiores, ou seja, apenas nas camadas que definem características como protocolos de áudio/vídeo, múltiplos dispositivos de entrada e uma variedade de

periféricos tais como MP3 players, câmeras digitais e conectividade provida por Bluetooth, CDMA, GPRS ou 802.11. Entretanto, elementos da infra-estrutura básica do Open-ORB também requerem adaptação, uma vez que muitas das plataformas não suportam os requisitos de todos os elementos dessa camada, como é o caso da plataforma Java para dispositivos moveis (CLDC) onde não é possível a utilização da interface *Iterator* ou mesmo de recursos reflexivos.

Dessa forma, o objetivo desse trabalho é utilizar duas estratégias complementares à atualmente utilizada no Open-ORB, que apenas trata da customização no nível da infra-estrutura de componentes, para também inserir customização na infra-estrutura básica do middleware. O suporte para este tipo de customização pode ser provida pela utilização da *separação de conceitos* (PARNAS, 1972), uma vez que esta é um princípio fundamental para tratar a variabilidade de sistemas. O *desenvolvimento orientado a aspectos* (AKSIT et al., 2002) é uma área emergente que provê separação avançada de conceitos (ASoC) promovendo modularização de conceitos transversais e consequentemente, reusabilidade e adaptabilidade. Portanto, pretendemos agregar o conceito de *aspectos* (ELRAD et al., 2001) no contexto da arquitetura de middleware reflexivo.

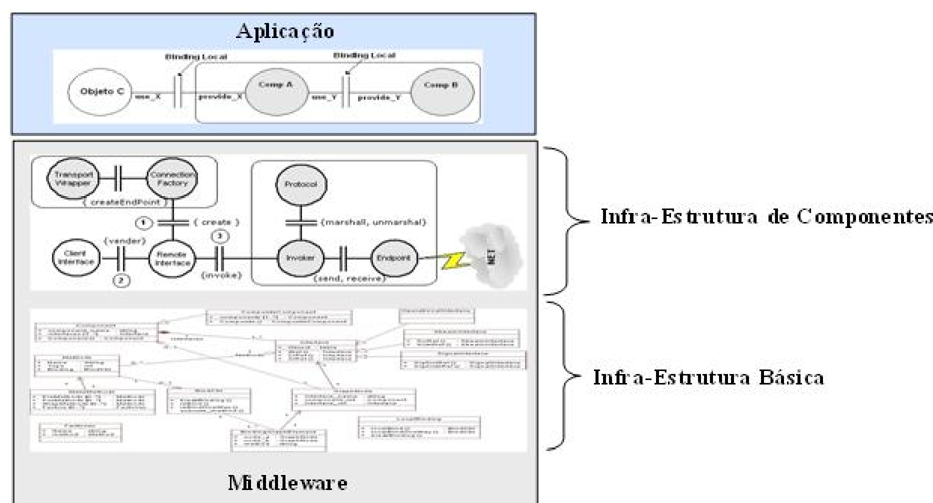

Figura 1: Camadas em um sistema de middleware

Aspecto é a abstração utilizada para centralizar um determinado conceito que, em geral, está disperso (*crosscuting concerns*) (ELRAD et al., 2001) em diversos componentes responsáveis pelas funcionalidades básicas de um sistema. Aspectos, em geral, representam funcionalidades específicas que se distinguem das funcionalidades básicas que o sistema deve oferecer. Em sistemas que não exploram o conceito de aspectos, não há essa distinção e, muitas vezes, as funcionalidades específicas estão entrelaçadas às funcionalidades básicas. Isto torna o sistema complexo, difícil de adaptar e compromete o reuso

das partes.

Visando aproveitar os benefícios que o uso de aspectos pode trazer em termos de customização, propomos uma nova arquitetura, denominada *Aspect Open-ORB*, baseada na arquitetura do Open-ORB. A partir dessa nova arquitetura, instâncias da plataforma corresponderão à configuração de componentes e aspectos que atendem aos requisitos de aplicações de determinados domínios ou de plataformas de software específicas.

Segundo (ZHANG; JACOBSEN, 2003b), apesar de ser conceitualmente intuitivo que orientação a aspectos traz benefícios para resolver os problemas da arquitetura de middleware, ainda é necessário uma análise detalhada para justificar os benefícios. De forma a realizar essa análise, implementamos neste trabalho dois protótipos de parte da arquitetura proposta. O primeiro protótipo utiliza uma linguagem dinamicamente tipada com facilidades reflexivas, a linguagem Lua (IERUSALIMSCHY; FIGUEIREDO; FILHO, 1996), e uma extensão dessa linguagem que dá suporte a programação orientada a aspectos, AspectLua (CACHO; BATISTA; FERNANDES, 2005, 2006). O segundo protótipo avalia os impactos dessa abordagem em uma linguagem compilada, como Java, utilizando padrões de projeto e uma extensão para a programação orientada a aspectos, AspectJ. O sistema de módulos suportado pela linguagem Java permitiu também analisar os benefícios da POA em termos de várias dimensões modulares, tais como: separação de conceitos, acoplamento, coesão e tamanho. A escolha por estas duas linguagens deve-se ao fato de se tratarem de paradigmas diferentes e, portanto, podem validar a solução de uma forma mais ampla.

1.2 Objetivos

Este trabalho tem como objetivo principal utilizar os conceitos de orientação a aspectos no contexto de middleware reflexivo visando promover customização da plataforma de acordo com as necessidades específicas de diferentes classes de aplicações e de ambientes de hardware e de software distintos.

Os objetivos específicos do trabalho incluem:

- O uso de modelagem orientada a aspectos na reestruturação do middleware reflexivo Open-ORB definindo a arquitetura do Aspect Open-ORB como uma combinação de componentes e aspectos. A idéia de meta-níveis existente na arquitetura original do Open-ORB, que provê reflexão computacional, será preservada.

- A implementação de um protótipo(LOpenOrb) da arquitetura proposta utilizando a linguagem Lua e AspectLua.
- A implementação de um protótipo(JOpenOrb) da arquitetura proposta utilizando a linguagem Java, AspectJ e padrões de projeto.
- A avaliação das vantagens e desvantagens conferidas pelo uso de orientação a aspectos comparando o Aspect Open-ORB com o Open-ORB através da utilização de testes de desempenho e uso de memória.
- A utilização de um conjunto de métricas que capturam importantes aspectos de modularidade de um sistema, tais como separação de conceitos, acoplamento, coesão e tamanho, para avaliar as vantagens da utilização da POA na modularização de plataformas de middleware.

1.3 Estrutura do trabalho

O restante desse trabalho está dividido em cinco capítulos. O segundo capítulo trata dos conceitos básicos que compõem a fundamentação teórica desse trabalho: middleware reflexivo, programação orientada a aspectos e padrões de projeto. O terceiro capítulo aborda a definição da arquitetura estendida do Open-ORB e a construção do LOpenOrb e JOpenOrb. O quarto capítulo descreve o uso da orientação a aspectos para a definição do Aspect Open-ORB e implementações usando AspectLua e AspectJ. O quinto capítulo apresenta a avaliação das duas soluções. O sexto capítulo comenta os trabalhos relacionados. O sétimo capítulo contém as conclusões e trabalhos futuros.

2 *Fundamentação Teórica*

O objetivo deste capítulo é introduzir alguns conceitos e áreas de pesquisa que serão relevantes para o entendimento deste trabalho. A visão geral inicia com uma breve introdução ao desenvolvimento baseado em componentes. Na seqüência, os conceitos relacionados a middleware reflexivos são apresentados. Em seguida, a programação orientada a aspectos é detalhada e exemplificada por meio de duas linguagens orientadas a aspectos: AspectJ e AspectLua. Por fim, o conceito de padrões de projeto é apresentado e discute-se como eles são utilizados no contexto de desenvolvimento orientado a objetos e como eles estão relacionados a programação orientada a aspectos.

2.1 Desenvolvimento baseado em Componentes

A abordagem de desenvolvimento baseada em componentes está fundamentada na composição de programas a partir de componentes pré-existentes. Segundo (SZYPERSKI, 2002), "Componente é uma unidade de composição com interfaces contratualmente especificadas e apenas com dependências de contexto explícitas. Um componente pode ser instalado isoladamente e estar sujeito à composição por terceiros". Dessa forma, um componente é uma unidade de software independente de aplicação, desenvolvido para um propósito específico e não para uma aplicação específica. Em uma estratégia baseada em componentes, a experiência em domínios específicos pode ser delegada para uma equipe de desenvolvedores, proporcionando economia de esforços e tempo de desenvolvimento. Para aplicações particularmente grandes, a experiência exigida para o desenvolvimento pode ser até distribuída entre várias empresas.

O desenvolvimento de software baseado em componentes tem como base o paradigma de orientação a objetos. Embora, os termos "componente"e "objeto"sejam freqüentemente usados indistintamente, um componente não é um objeto. Segundo (SZYPERSKI, 2002), um objeto é uma instância de uma classe que é criada durante a execução. Um componente pode ser uma classe, mas normalmente é uma coleção de classes e interfaces.

Durante a execução, um componente torna-se “vivo”, quando suas classes são instanciadas. Portanto, durante a execução, um componente torna-se uma teia de objetos.

Normalmente, os componentes são objetos comerciais que têm comportamentos pre-definidos e reutilizáveis. Os detalhes da implementação ficam ocultos nas *interfaces*, que isolam e encapsulam um conjunto de funcionalidades. As interfaces são o meio pelo qual os componentes conectam-se. Uma interface é um conjunto de operações nomeadas, que podem importar funcionalidades de outros componentes como também exportar funcionalidades para outros componentes. Dessa forma, interfaces bem definidas estabelecem os pontos de entrada e saída de um componente, definido assim toda a acessibilidade do componente.

2.2 Middlewares Reflexivos

O principal objetivo da reflexão(SMITH, 1982) em sistemas computacionais é permitir que um sistema possa executar algum processamento em benefício próprio, para modificar ou ajustar sua própria estrutura ou comportamento. Com base nessa idéia, a reflexão vem sendo aplicada com sucesso em áreas como linguagem de programação, sistemas operacionais e em middleware. Neste último caso, a utilização da reflexão deu origem ao termo “*middleware reflexivo*”(BLAIR et al., 1998; AGHA, 2002; TRIPATHI, 2002). O objetivo de um middleware reflexivo é dispor de um maior poder de adaptação utilizando a reflexão computacional para esse propósito.

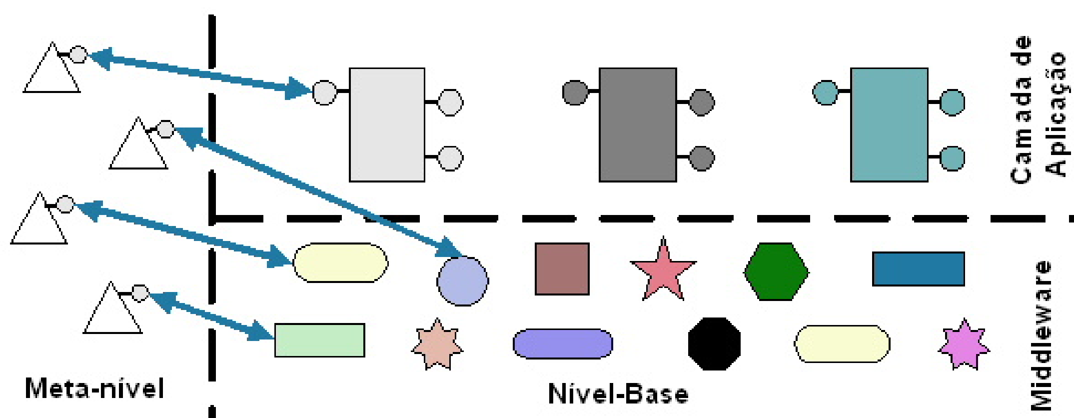

Figura 2: Arquitetura do middleware reflexivo

A reflexão está diretamente ligada aos detalhes de implementação de um sistema, uma vez que promove a separação entre as funcionalidades fornecidas pelos objetos, e os detalhes de sua implementação. A Figura 2 ilustra a separação do *middleware* em

dois níveis: *nível-base* e *meta-nível*. No *nível-base* estão as funcionalidades do sistema, disponíveis através das interfaces bases. No *meta-nível* estão os meta-objetos usados para descrever e manipular os objetos do nível-base. No meta-nível são aplicadas duas abordagens: *reflexão estrutural* e *comportamental*. A *reflexão estrutural* permite a inspeção e a manipulação da estrutura do middleware, além de expor detalhes de composição dos elementos internos. Na *reflexão comportamental* o comportamento das funções básicas do middleware pode ser monitorado e controlado no meta-nível. Dessa forma, operações como a manipulação de variáveis (leitura e escrita) e invocações de métodos podem ser interceptadas e desviadas para o meta-nível.

Middlewares construídos a partir dessa abordagem, tais como Open-ORB (BLAIR et al., 1998), OpenCOM (PARLAVANTZAS et al., 2000), dynamicTAO (KON et al., 2000) utilizam módulos compostos por *objetos base* e por *meta-objetos*. O meta-objeto possui uma interface com o objeto base, através da qual pode-se obter informações internas do objeto base ou interceptar chamadas que deveriam ir diretamente para o mesmo. Uma invocação interceptada pode ser manipulada pelo meta-objeto e ser reencaminhada para o objeto original.

Na seção seguinte detalharemos alguns dos elementos presentes no nível-base e no meta-nível de um dos principais representantes de middleware reflexivo, o *Open-ORB*, juntamente com os vários tipos de meta-objetos.

2.2.1 Arquitetura do Open-ORB

Diferentemente dos middlewares tradicionais, onde a arquitetura segue um estilo "caixa-preta", o Open-ORB propõe uma arquitetura aberta baseada em reflexão computacional. Tal arquitetura pode dar suporte a diversos tipos de aplicações, como: aplicações multimídia, tempo-real, computação móvel, etc.

A infra-estrutura da arquitetura Open-ORB é composta por dois elementos: *Capsule* e *Node manager*. O *Capsule* gerencia o espaço de endereçamento da aplicação, fornecendo serviços locais e remotos como registro de componentes, invocação de métodos, etc. O *Node manager* gerencia os recursos compartilhados pelos diferentes *Capsules* de um mesmo nó além de fornecer funções para criação de novas portas (TCP/IP e UDP/IP).

Nas sub-seções seguintes, descreveremos os elementos que compõem tanto o nível-base, quanto o meta-nível da arquitetura do Open-ORB.

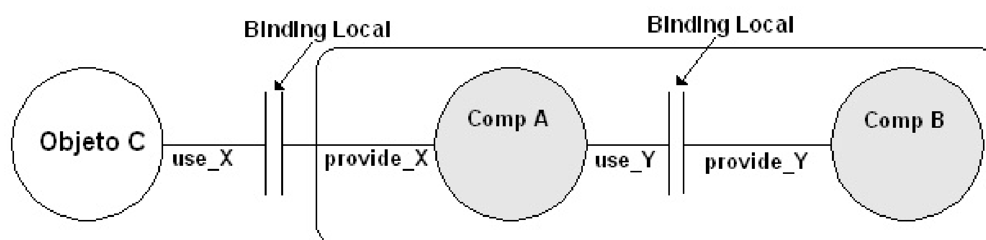

Figura 3: Exemplo de um componente composto

2.2.1.1 Elementos do nível-base

Os principais elementos do nível-base da arquitetura Open-ORB são: Interfaces, *Bindings* locais e Componentes. *Interfaces* representam os pontos de acesso para objetos e componentes. Cada interface pode exportar ou importar métodos. Os métodos exportados são os disponibilizados pelos objetos e os importados são os requisitados pelos objetos. Para estabelecer a relação entre as interfaces que exportam e as que importam é utilizado um *binding* local. Este *binding* associa interfaces compatíveis, ou seja, um método importado por uma interface deve ter seu correspondente na interface exportadora. A Figura 3 ilustra a presença de um *binding* local entre as interfaces *use_X* e *provide_X* e entre *use_Y* e *provide_Y*.

Na arquitetura Open-ORB, um componente é uma unidade de composição e distribuição independente (SZYPERSKI, 2002) que é desenvolvido para prover acesso aos seus serviços requisitados e fornecidos (métodos) através de uma ou mais interfaces. Todas as interações entre componentes são realizadas através de suas interfaces. Este é um fator que facilita o desenvolvimento, além de permitir uma maior independência. Como um componente é uma unidade de composição, existem dois tipos de componentes, o *primitivo* e o *composto*. O componente primitivo encapsula apenas um objeto e fornece acesso a uma ou mais interfaces. O componente composto contém um conjunto de outros componentes interligados, através de *bindings* locais, por suas interfaces. Cada um desses componentes que compõe um componente composto também pode ser um componente composto. Esta recursão de composição é finalizada por componentes primitivos.

A Figura 3 ilustra um componente composto por dois outros componentes primitivos: A e B. Pode-se observar que o conjunto de interfaces externas de um componente composto é um sub-conjunto das interfaces dos componentes que o compõem. Os componentes que compõem um componente composto não necessariamente devem estar no mesmo espaço de endereçamento (*Capsule*). Podem existir componentes distribuídos que, através de *bindings* implícitos/explícitos (FITZPATRICK et al., 1998), fornecem a composição de

componentes em diferentes *Capsules.* Dessa forma, esses *bindings* têm o objetivo de esconder as complexidades relativas a protocolos e conexões remotas além de ajudar a prover uma transparência de localização. Um *binding* implícito é um *binding* criado entre duas interfaces sem o conhecimento do programador. Este tipo de *binding* é normalmente usado quando uma interface é importada. O *binding* explícito, por sua vez, é criado pelo programador e pode ser de três diferentes tipos: *Operational, Signal, Stream.*

Um *binding Operational* é usado para enviar invocações de métodos de interfaces importadoras para interfaces exportadoras e, opcionalmente, retornar o resultado. Esse *binding* pode ser síncrono, assíncrono ou um *announcement.* Um *binding* síncrono bloqueia a interface importadora até que a interface exportadora forneça a resposta. O *binding* assíncrono diferencia-se do anterior por não bloquear a interface importadora enquanto aguarda pelo retorno da invocação. O *bindind announcement* realiza apenas o processo de invocação, não exige da interface exportadora o retorno da invocação e não bloqueia a interface importadora.

Um *Signal binding* suporta um conjunto de sinais *one-way* onde não é possível resposta. O fluxo de um sinal origina-se no *Source* e destina-se ao *Sink.* O *Stream binding* opera semelhantemente ao *Signal,* diferenciando apenas no suporte ao envio de dados contínuos como áudio e vídeo.

2.2.1.2 Elementos do meta-nível

Como foi visto na Seção 2.2, a reflexão é composta por aspectos estruturais e comportamentais. Para simplificar as interfaces providas por cada um desses aspectos, a arquitetura Open-ORB dividiu o meta-nível em quatro diferentes meta-modelos. Os aspectos estruturais são representados pelos meta-modelos *encapsulation* e *composition* e os aspectos comportamentais são representados pelos meta-modelos *environment* e *resource.*

A Figura 4 ilustra um objeto com seus quatro meta-modelos. Cada um desses meta-modelos é acessado através de um meta-objeto. A descrição de cada meta-modelo encontra-se a seguir:

- *Encapsulation*: O objetivo deste meta-modelo é expor o encapsulamento provido pelos objetos, uma vez que ele torna possível a inspeção, modificação e extensão da implementação de um objeto. Este meta-modelo pode ser usado para monitorar e controlar todos os acessos aos objetos, incluindo acessos aos seus atributos e métodos.

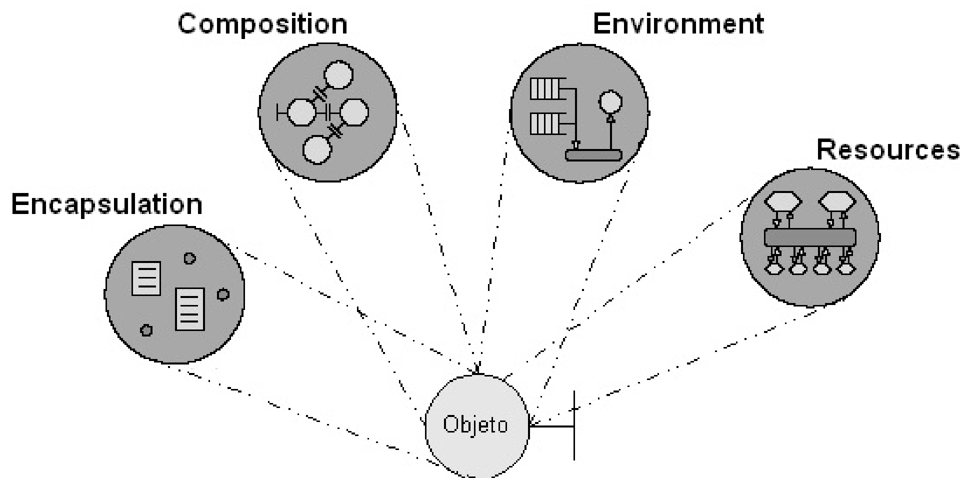

Figura 4: Um objeto e seus quatro meta-modelos (ANDERSEN, 2002)

- *Composition*: Este meta-modelo provê acesso ao grafo de componentes existente em um componente composto, ou seja, um componente primitivo não possui este meta-modelo. Através desse meta-modelo é possível remover, inserir e substituir componentes que façam parte de um grafo de componentes.

- *Environment*: Este meta-modelo expõe o ambiente de execução de cada interface. Isto inclui todas as etapas do processo de invocação de um método, tanto o lado cliente (marshall, envio) como o lado servidor (unmarshall, processamento).

- *Resource*: O objetivo deste meta-modelo é expor a alocação e gerenciar os recursos associados com cada um dos objetos ou interfaces.

2.3 Programação Orientada a Aspectos

A programação orientada a aspectos(POA)(KICZALES et al., 1997) é um paradigma emergente de desenvolvimento de software que busca melhorar a modularidade e reuso em sistemas de software. Estes benefícios são alcançados pelo modelo de separação de funcionalidades onde é possível separar conceitos transversais *crosscutting concerns*) (KICZALES et al., 1997) e junta-los posteriormente de modo a compor o sistema final. Exemplos de *crosscutting concerns* são persistência, distribuição, controle de concorrência, tratamento de exceções, e depuração. O aumento da modularidade implica em sistemas mais legíveis, os quais são mais facilmente projetados e mantidos.

AspectJ e uma das linguagens percussoras da POA. Em AspectJ os *crosscutting concerns* são separados da aplicação por meio da utilização de *join points* e *advice*. Um *join*

point (TEAM, 2002) é um ponto bem definido no fluxo de execução de um programa, onde é possível adicionar o comportamento do *crosscutting concern*. Exemplos de *join points* são: invocação e execução de métodos, inicialização de objetos, execução de construtores, tratamento de exceções, acesso e atribuição a atributos, entre outros. A definição dos *join points* permite a inserção de comportamentos adicionais denominados de *advice*. Tipicamente, um *advice* pode ser adicionado antes, depois ou durante a execução de um *join points* e permite a total inserção de *crosscutting concerns* que são definidos externamente ao código da aplicação.

A junção entre os *join points* e *advice* definidos na fase de implementação é realizada através do processo de *weaving*. Este processo, dependendo da implementação, pode ser realizado em tempo de compilação ou em tempo de execução. No AspectJ (KICZALES et al., 1997) e AspectC++ (GAL; SCHRODER-PREIKSCHAT; SPINCZYK, 2001) o *weaving* é realizado em tempo de compilação. Dessa forma, os aspectos são definidos através da inserção de novas construções à sintaxe das linguagens e o *weaver* é um compilador que, dado o código fonte de um sistema e um conjunto de aspectos, gera uma versão do sistema contendo os aspectos.

Uma outra abordagem compreende a inserção de aspectos em tempo de execução. AspectLua (CACHO; BATISTA; FERNANDES, 2005, 2006), AOPy (DECHOW, 2003) e AspectR (BRYANT; FELDT, 2002) são linguagens que seguem esta estratégia. Estas implementações representam, respectivamente, extensões para as linguagens Lua, Python (DRAKE, 2003) e Ruby (THOMAS; FOWLER; HUNT, 2000) que disponibilizam a inserção de aspectos utilizando os próprios elementos da linguagem. Dessas três abordagens, detalharemos AspectJ e AspectLua por serem as linguagens utilizadas nesse trabalho.

2.3.1 Linguagens para a definição de Aspectos

Como mencionado anteriormente, existe uma grande variedade de ferramentas e extensões que se propõem a definir aspectos. Nesta seção, iremos estudar as soluções fornecidas por extensões propostas para a linguagem Java e Lua.

2.3.1.1 AspectJ

AspectJ (KICZALES et al., 1997) é uma das extensões compatível com a plataforma Java para utilização de POA. Ela provê a modularização dos conceitos transversais através da implementação de aspectos como classes.

No sentido de facilitar o entendimento, a Figura 5 mostra um exemplo de definição do aspecto *BindStateAspect*. Nesta figura, as linhas 3 a 7 mostram um exemplo da utilização de"*introductions*"para modificar a estrutura estática de outra classe(*ConcreteBind*). Dessa forma, este mecanismo permite introduzir métodos concretos ou abstratos, construtores e atributos em diferentes classes.

Cada aspecto define uma função específica que pode afetar várias partes de um sistema, como, por exemplo, segurança. Um aspecto, como uma classe Java, pode definir membros e uma hierarquia de aspectos, através da definição de aspectos especializados. Estes aspectos são definidos como aspectos abstratos, os quais devem ser estendidos provendo a implementação do componente abstrato. Os componentes abstratos podem ter métodos, como em uma classe Java, e "pointcuts", os quais devem ser definidos em um aspecto concreto, permitindo o reuso do comportamento dos aspectos abstratos.

```
1  public aspect BindStateAspect {
2
3          private BindState ConcreteBind.bdstate;
4
5          public void ConcreteBind.rebind(Port newRecept) {
6                  bdstate.rebind(newRecept);
7          }
8
9      pointcut changeState(ConcreteBind bind) :
10         execution(public Object ConcreteBind.makeRequest(..)) &&
               target(bind);
11
12
13     before(ConcreteBind bind)  :
14         changeState(bind)
15         {
16                 bind.bdstate = (BindState)PrototypeAspect.aspectOf()
                       .createCloneFor(bind.running);
17         }
18
19 }
```

Figura 5: Exemplo da utilização de AspectJ

Para definir em quais pontos de junção um dado aspecto irá atuar, AspectJ fornece uma construção denominada *pointcut*. Os *Pointcuts* são formados pela composição de *join points*, através dos operadores &&, ——, e ! . A utilização de *pointcuts* permite obter valores de argumentos de métodos, objetos em execução, atributos e exceções dos *join points*. Para definir *pointcuts*, identificando os *joinpoints* a serem afetados, utiliza-se construtores de AspectJ chamados designadores de *pointcut*, AspectJ fornece os designadores mostrados na Tabela 1. O elemento *PadrãoTipo* é uma construção que pode definir um conjunto de tipos utilizando *wildcards*, como * e + . O *wildcard* (*)pode ser usado sozinho para representar o conjunto de todos os tipos do sistema, ou depois de caracteres, representando qualquer seqüência de caracteres. O *wildcard* (+) deve ser utilizado junto ao nome de um tipo para assim representar o conjunto de todos os seus subtipos. Portanto,

a Figura 5 mostra a definição do *pointcut changeState* que é definido pela combinação dos Os designadores *execution* e *target* para a formação do *pointcut changeState*. Neste caso, o *pointcut* é definido para interceptar todas as execuções do método *makeRequest* que tenham como alvo a classe *ConcreteBind*.

Tabela 1: *Pointcut designators* suportados pelo AspectJ.

Nome do *designator*	**Descrição**
call(Assinatura)	Invocação de método/construtor identificado por Assinatura
execution(Assinatura)	Execução de método/construtor identificado por Assinatura
get(Assinatura)	Acesso a atributo identificado por Assinatura
set(Assinatura)	Atribuição de atributo identificado por Assinatura
this(PadrãoTipo)	O objeto em execução é instância de PadrãoTipo
target(PadrãoTipo)	O objeto de destino é instância PadrãoTipo
args(PadrãoTipo,...)	Os argumentos são instâncias de PadrãoTipo
within(PadrãoTipo)	O código em execução está definido em PadrãoTipo

Após a definição dos pontos de junção, é necessário definir os *advice* que serão executados para cada *join point*. AspectJ suporta três tipos de *advice*: *before*, *around* e *after*. Dessa forma, um *advice* pode ser definido para ser executado, respectivamente, antes ou depois de um determinado conjunto de pontos de junção definidos pelo *pointcut* associado. A utilização de um *advice* do tipo *around* permite a opção pela execução ou não do ponto de junção.

As Linhas 13 a 17 da Figura 5 mostram a definição de um *advice* que será executado antes da invocação do método *makeRequest*. Este *advice* utiliza-se do parâmetro *bind* para acessar um dos atributos da classe *ConcreteBind*.

2.3.1.2 AspectLua

AspectLua é uma extensão da linguagem Lua que oferece suporte à definição de aspectos e à combinação dinâmica do código base da aplicação com o código de aspecto. Tudo isso é feito utilizando os mecanismos básicos da própria linguagem sem a necessidade de alteração no interpretador.

A linguagem Lua (IERUSALIMSCHY; FIGUEIREDO; FILHO, 1996) caracteriza-se por ser uma linguagem interpretada e dinamicamente tipada, isto é, as variáveis não possuem tipos definidos, sendo estes associados a cada valor a elas atribuídas. Existem oito tipos básicos em Lua: *nil* (correspondente ao valor nulo), *boolean*, *number*, *string*, *function*, *userdata*, *thread* e *table*.

A sintaxe e as estruturas de controle são similares as de linguagens convencionais possuindo *while* e *if*, variáveis locais e globais e definição de funções com parâmetros.

As funções são tratadas como valores de primeira classe podendo ser armazenadas em variáveis, passadas como argumentos para outras funções e retornadas como resultados. Além disso, a possibilidade de uma função retornar mais de um valor dispensa a utilização de parâmetros por referência.

Tabelas (*tables*) implementam arrays associativos, ou seja, arrays que podem ser indexados não só por números, mas por qualquer valor diferente de *nil* (valor nulo). Elas constituem a principal estrutura de dados da linguagem sendo utilizada para construção de arrays comuns, conjuntos, registros, árvores, filas, entre outros.

Lua possui um mecanismo de extensão chamado *metatables* que são tabelas comuns utilizadas para definir comportamento de tabelas e dados do usuário (chamados de *userdata* em Lua). Cada *metatable* pode possuir eventos (índices) e *metamethods* (valores). As *metatables* são declaradas pelo programador e permitem modificar o comportamento padrão do interpretador para determinadas situações como, por exemplo, a geração de um erro para chamada de métodos inexistentes. Com isso, através da definição de uma *metatable*, é possível evitar que o interpretador gere um erro inserindo um tratamento diferente do fornecido pela linguagem. Para isso, o interpretador, ao receber uma invocação de um método desconhecido por ele, verifica, antes de gerar um erro, se existe alguma *metatable* definida. Caso exista, o interpretador executa o código correspondente definido na *metatable*. Caso contrário, o erro é gerado.

Apesar de oferecer várias facilidades reflexivas tais como *metatables*, a linguagem Lua originalmente não possui uma API que gerencie as mudanças relativas ao meta-nível e ao nível-base de modo que a consistência entre estes dois níveis seja preservada. Neste sentido, utilizamos o protocolo de meta-objetos LuaMOP (FERNANDES; BATISTA; CACHO, 2004a) para prover uma camada de abstração sobre as funcionalidades reflexivas da linguagem Lua de modo que aspectos possam ser definidos para afetar tanto variáveis quanto métodos.

Com base nas funcionalidades fornecidas pelo LuaMOP, AspectLua provê um conjunto de funcionalidades que potencializam e facilitam a utilização da POA: (1) inserção e remoção de aspectos em tempo de execução; (2) definição de ordem de precedência entre aspectos; (3) a possibilidade de usar *wildcards*; (4) a possibilidade de associar aspectos com elementos não declarados (*anticipated join points*).

Para definir um aspecto, AspectLua fornece o método *new()* que cria uma instância da classe AspectLua. Esta classe fornece o método *aspect* que recebe como parâmetro três tabelas: nome do aspecto, informações do *pointcut* e informações do *advice*. As

informações do *pointcut* são compostas pelo nome do *pointcut*, pelo designador e pelas funções ou variáveis que devem ser interceptadas. AspectLua suporta cinco tipos de designadores: *call* para uma chamada de função; *callone* para aspectos que devem ser executados somente uma vez; *introduction* para introdução de funções em tabelas(objetos em Lua); *get* e *set* aplicados sobre variáveis. As variáveis e funções a serem interceptadas não necessariamente precisam estar declaradas no momento da definição do aspecto. Além disso, pode-se utilizar *wildcards* para generalizar a aplicação para uma determinada classe. Por exemplo, *Bank.** implica em dizer que um aspecto deve ser aplicado para todos os métodos da classe *Bank*. O último parâmetro do método *aspect* define os elementos do *advice*: o tipo (*after*, *before*, *around*) e a ação a ser tomada quando o *pointcut* for alcançado.

Portanto, a Figura 6 define um aspecto com nome *logaspect* que possui o *pointcut logdeposit* e que irá atuar em todas as chamadas(*call*) do método *Bank.deposit*. Com essa definição, o *advice logfunction* será sempre invocado antes(*before*) da execução do método *deposit*.

```
1  asp = Aspect:new()
2  id = asp:aspect( {name = 'logaspect'},
3     {pointcutname = 'logdeposit',
4     designator='call', list= {'Bank.deposit'}},
5     {type = 'before', action = logfunction} )
6
7  Bank = {balance = 0}
8  function Bank:deposit(amount)
9   self.balance = self.balance + amount
10 end
11 function logfunction(a)
12  print('It was deposited: ' .. a)
13 end
14
15 oldasp = asp:getAspect(id)
16 oldasp.advice.type = 'after'
17 asp:updateAspect(id, oldasp)
```

Figura 6: Exemplo da utilização do AspectLua

A invocação do método *aspect* define um pointcut para cada aspecto e retorna um identificador de aspecto(Id). Assim, para associar vários *pointcuts* para um mesmo aspecto é necessário invocar o método *aspect* para cada *pointcut*. Para gerenciar os aspectos definidos, AspectLua fornece os seguintes métodos: *getAspect(id)*, *getAll()*, *removeAspect(id)*, e *updateAspect(id, newasp)*. Os métodos *getAspect* e *getAll* podem ser utilizados para obter um ou todos os aspectos já definidos. A obtenção de um aspecto permite modificar ou atualizar seus elementos pela utilização do método *updateAspect*. Este método pode modificar *pointcuts* e *advice* para um aspecto já definido. A remoção de um aspecto pode ser feita por meio do método *removeAspect*.

Na Figura 6, o objeto *Bank*, com o método *deposit*, é declarado depois da definição do *join point*. Nas versões iniciais de AspectLua, isto não era possível uma vez que cada método alvo de *join point* necessitava ser anteriormente declarado. Para resolver essa limitação, a versão atual do AspectLua usa o conceito de *anticipated join point* (CACHO; BATISTA; FERNANDES, 2006) que permite ao programador definir um *join point* para um elemento que ainda não foi declarado pela aplicação. Um *anticipated join point* pode ser útil em muitas situações. Por exemplo, para a alocação dinâmica de recursos em sistemas embarcados, um *anticipated join points* pode ser usado para suportar uma abordagem de *lazy loading* (MILES, 2004).

Para controlar a ordem de execução dos aspectos para um dado *pointcut*, AspectLua fornece as funções *getOrder* e *setOrder*. A função *getOrder* é usada para obter a lista de aspectos associadas a uma determinada função ou variável. Ela recebe como parâmetro o nome da variável ou função. O resultado é formado por uma lista de aspecto e sua respectiva ordem de execução. A mudança desta ordem é responsabilidade da função *setOrder* que recebe os seguintes parâmetros: nome da variável ou função e a nova ordem de execução. Na Figura 7, o método *deposit* tem dois aspectos que serão executados antes da sua execução. Por padrão, a ordem de execução é a mesma da ordem de definição. No entanto, *logfunction* será indevidamente executada antes de *checkRights*. Para modificar esta ordem, *setOrder* é invocado recebendo como parâmetro o método *deposit* e a nova ordem de execução.

```
1 function checkRights() ... end
2 a:aspect( {name = 'secaspect'},
3  {pointcutname = 'verifyRights',
4  designator='call',list={'Bank.deposit'}},
5  {type ='before',action=checkRights } )
6 local order= Aspect:getOrder('Bank.deposit')
7 Aspect:setOrder('Bank.deposit',{order[2], order[1]})
```

Figura 7: Definindo ordem para invocação de aspectos

2.4 Padrões de Projeto

Os padrões de projeto realizam uma importante função no projeto das aplicações. A idéia de padrões de projeto foi inicialmente adotada para a área de arquitetura, onde Alexander e dois colaboradores (ALEXANDER, 1979; ALEXANDER; ISHIKAWA; SILVERSTEIN, 1977) formularam uma linguagem de padrões para o projeto e construção de prédios e casas. Segundo Alexander, cada padrão descreve um problema que ocorre repetidamente em nosso ambiente. Dessa modo, um padrão fornece uma solução básica

que pode ser aplicada milhões de vezes sem que para isso ela seja modificada.

Esta mesma idéia vem sendo aplicada aos padrões de projeto utilizados no desenvolvimento de software orientado a objeto. Neste caso, os padrões descrevem o problema, a solução, quando aplicar a solução, e suas conseqüências. A solução é geralmente fornecida pela organização de objetos e classes que buscam solucionar o problema. Cada solução é personalizada e implementada para resolver o problema de um contexto particular (GAMMA; HELM; JOHNSON, 1995). O principal trabalho nesta área foi feito por (GAMMA; HELM; JOHNSON, 1995) ao catalogar 23 padrões de projeto identificados recorrentemente no projeto de aplicações orientadas a objeto. Um padrão consiste em quatro principais partes: nome do padrão, problema, solução e conseqüências.

O *nome do padrão* tem como objetivo descrever o padrão em poucas palavras. Ele fornece aos projetistas um vocabulário comum para a comunicação entre projetos permitindo, assim, que um projetista descreva um projeto em um alto nível de abstração. O *problema* descreve uma situação particular que pode ocorrer recorrentemente em um projeto orientado a objeto e que deve ser resolvido. Em alguns casos, é necessário que o problema satisfaça uma lista de condições para que o padrão possa ser aplicado. A *solução* descreve um conjunto de classes e objetos que implementam a solução do padrão. Esta solução não é concreta para uma particular instância do problema, mas sim uma solução abstrata que pode ser usada como um modelo para diferentes situações. As *consequências* descrevem as implicações de aplicar uma solução a um problema. Ou seja, as conseqüências estão relacionadas aos impactos que a solução vai gerar em fatores como: flexibilidade, extensibilidade e portabilidade.

Um exemplo de padrão de projeto catalogado por (GAMMA; HELM; JOHNSON, 1995) pode ser visto na Figura 8. Este padrão tem como nome *Decorator* e é utilizado para resolver problemas relacionados a inserção dinâmica de responsabilidades a um objeto. A solução proposta por este padrão é composta pelos elementos mostrados na Figura 8. Os elementos *Component* e *ConcreteComponent* são os elementos da aplicação e devem definir respectivamente as funcionalidades básicas da aplicação. Por sua vez, os elementos *Decorator* e *ConcreteDecorator* são os responsáveis por adicionar as novas funcionalidades. As mesmas funcionalidades do elemento *Component* são fornecidas pelo elemento *Decorator*. Isto é feito por meio da referência mantida pelo elemento *Decorator* ao elemento *Component*. Dessa forma, o elemento *ConcreteDecorator* pode adicionar novos comportamentos e, ao mesmo tempo, utilizar os comportamentos já definidos pelo elemento *Component*. Uma das conseqüências da utilização do padrão *Decorator* é o ganho de flexibilidade com-

parada à solução de herança estática. Em outras palavras, a utilização de decoradores permite adicionar e remover responsabilidades em tempo de execução, enquanto que a herança estática requer a criação de novas classes para cada responsabilidade adicional.

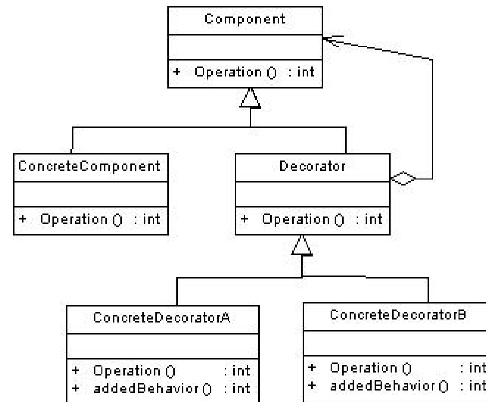

Figura 8: Padrão de Projeto *Decorator*

Como discutimos anteriormente, a aplicação de padrões é feita pela definição de uma funcionalidade para cada papel do padrão. Dessa forma, é possível construir um catálogo com 23 soluções comprovadamente úteis e que garantem uma reusabilidade e manutenabilidade das soluções. O problema de alguns desse padrões é que eles terminam por inserir as funcionalidades do padrão entre muitas classes do sistema de software. Neste sentido estudos recentes (HANNEMANN; KICZALES, 2002; GARCIA et al., 2005; CACHO et al., 2006b) têm mostrado que as abstrações orientadas a objeto não são capazes de modularizar os *concerns* relacionados a um padrão especifico, produzindo assim sistemas pouco modulares. Para resolver esse problema, os trabalhos (HANNEMANN; KICZALES, 2002; GARCIA et al., 2005; CACHO et al., 2006b, 2006a) utilizam a programação orientada a aspectos para inserir os padrões por meio de aspectos reutilizáveis.

3 *Arquitetura e Implementações do Open-ORB*

Neste capítulo descreveremos uma versão estendida da arquitetura do Open-ORB e duas implementações para esta arquitetura. As implementações do Open-ORB seguem uma abordagem onde são definidas abstrações de alto nível para se expressar a estrutura interna do middleware (Infra-estrutura básica na Figura 1) e aproveitados os mecanismos reflexivos definidos pelo Open-ORB que irão atuar sobre tal estrutura. Ou seja, é definido um conjunto mínimo de elementos com suporte a reflexão que, combinados, irão gerar um middleware reflexivo.

Portanto, a seção 3.1 descreve as funcionalidades exigidas pela arquitetura estendida do Open-ORB. Em seguida, a seção 3.2 ilustra os detalhes da arquitetura e da implementação do LOpenOrb. Por fim, a seção 3.3 descreve como as funcionalidades exigidas foram implementadas no JOpeORB.

3.1 Arquitetura estendida do Open-ORB

Como foi dito na Seção 2.2, um middleware reflexivo é formado pelos elementos do nível-base e meta-nível. Nesta seção descreveremos a API exigida por cada um desses níveis e ilustraremos como eles são combinados para formar a arquitetura do middleware reflexivo.

3.1.1 Funcionalidades do Nível-Base

As funcionalidades básicas do Open-ORB são mostradas na Tabela 2. O método *IRef* é utilizado para criar as interfaces de um componente. Para isto, é necessário o fornecimento do objeto que implementa a interface, a lista de métodos importados e exportados. A vinculação entre duas interfaces é feita por meio do método *localBind*. Este método

recebe como parâmetro dois *GraphNode*, onde cada um deve ter a referência da interface, o nome da interface e a referência do componente que fornece essa interface. Os dois últimos parâmetros são opcionais e buscam ampliar o alcance do grafo de componentes, uma vez que ao chegar em uma interface, se a mesma fizer parte de um componente, as demais interfaces desse componente também poderão ser incluídas no grafo.

No Open-ORB, um componente pode ser criado de duas maneiras. A primeira é através do método *Component(name, interfaces, objeto)*, onde são necessários o nome do componente, suas interfaces externas e o objeto container do componente. A segunda maneira é através do método *Composite*, que cria um componente composto formado por vários outros componentes. Este método recebe como parâmetro o nome do novo componente composto, as interfaces externas, os componentes que irão compor o componente composto, as interfaces internas e uma lista com a combinação das interfaces internas que serão vinculadas através de *bindings* locais. Por exemplo, o componente da Figura 3 pode ser criado pela seguinte invocação: *Composite("CompXY",{{"provide_X", {compA,"provide_X"}}}, {{compA, compB}}, {{"use_Y", {compA,"use_Y"}}, {{"provide_Y", {compB,"provide_Y"}}}, {{"use_Y","provide_Y"}})*.

Tabela 2: Funcionalidades do Nível-Base.

Nome do método	**Descrição**
IRef(o,$\{m_1\}$,$\{m_2\}$) $\longrightarrow i$	Cria uma interface i através da implementação do objeto o, da lista de métodos exportados $\{m_1\}$ e da lista de métodos importados $\{m_2\}$.
localBind(gn_1,gn_2) $\longrightarrow l$	Cria um *binding* local l, entre duas instâncias da classe *GraphNode*, gn_1 e gn_2.
localBindOneWay(gn_1,gn_2) $\longrightarrow l$	Cria um *binding* local l, que apenas vincula os métodos importados de gn_1 aos de gn_2, sem realizar o processo inverso.
l.breakBinding()	Remove o *binding* local controlado por l
l.reBind(gn_1,gn_2)	Reconstrói o *binding* local entre gn_1 e gn_2 controlado por l
Component(n,$\{\{in,i\}\}$,o) $\longrightarrow c$	Cria o componente primitivo c com nome n, objeto o e com uma ou mais interfaces externas definidas pelo par in e i, ou seja, nome da interface e sua referência.
Composite(n,$\{\{in,i\}\}$,$\{\{c\}\}$,$\{\{ic,i\}\}$, $\{cb\}$) $\longrightarrow cp$	Cria o componente composto cp com nome n, uma ou mais interfaces externas definidas pelo par in e i, por um ou mais componentes definidos por c, pelas interfaces internas ic e pela combinação das interfaces cb.

3.1.2 Funcionalidades do Meta-nível

O meta-nível do Open-ORB é formado por quatro meta-modelos, dentre os quais apenas o *encapsulation* e *composition* serão implementados. Os métodos fornecidos pelo meta-objeto *encapsulation* do Open-ORB são descritos pela Tabela 3. *inspect()* retorna

a descrição de uma interface ou componente; *addPreMethod(m, f)* adiciona o método *f*, da classe *Method*, para ser executado antes do método de nome *m*. Existem ainda os fornecidos pelo meta-objeto *encapsulation* de um componente. Entre eles, os principais são: *inspect()* retorna descrição do componente; *addIF(name, impl)* adiciona ou substitui a interface com nome *name* pela implementação *impl*; *delIF(name)* remove a interface com nome *name* do componente e *changeObject(obj)* substitui o objeto conteiner do componente por *obj*. Além desses métodos, o Open-ORB fornece ainda: *addFactoryMethod(m, fname,f)*, *delFactoryMethod(m,fname)* e *getFactoryMethod(m,fname)* que respectivamente adiciona um *Factory* com nome *fname* para o método *m* e que será executado pelo método *f*, remove um *Factory* com nome *fname* do método *m* e obtém um Factory com nome *fname* do método *m*. O *Binding* de um método também pode ser definido através da função *setBindingMethod(m,b)* ou pode ser obtido através *getBindingMethod(m)*. O primeiro recebe o nome do método *m* e uma instância da classe *BindCtrl* *b* e o segundo apenas o nome do método.

Tabela 3: Métodos fornecidos por um meta-objeto Encapsulation para interfaces.

Nome do método	**Descrição**
inspect()	Retorna descrição da interface.
addMethod(m,type)	Adiciona um método *m* do tipo *type*(import ou export) a interface.
addSMethod(m,f)	Adiciona um método *f*, da classe *Method*, para ser executado antes(Pre), depois(Pos) ou durante(Wrap) o método *m*. Para invocar esse método substitua S por Pre, Pos ou Wrap.
delSMethod(m,f)	Remove um método *f* que seria executado antes(Pre), depois(Pos) ou durante(Wrap) o método *m*. Para invocar esse método substitua S por Pre, Pos ou Wrap.
addFactoryMethod(m,fname, f)	Adiciona um Factory com nome *fname* e *f*, uma instância da classe *Method*, usado para criar o caminho de invocação do método *m*.
getFactoryMethod(m, fname)	Obtém o Factory com nome *fname* definido para o método *m*.
delFactoryMethod(m, fname)	Remove o Factory com nome *fname* definido para o método *m*.
setBindingMethod(m,b)	Define o BindCtrl *b* usada para invocar o método *m*.
getBindingMethod(m)	Obtém o BindCtrl usada para invocar o método *m*.
changeObject(o)	Modifica o objeto da interface pelo objeto *o*.

Os mesmos métodos disponibilizados pelo meta-objeto *composition* são mostrados na Tabela 4, entre eles: *inspect()* retorna o grafo de componentes; *insert(comp, bind)* insere um componente *comp* no grafo de componentes definido por *bind*; *replace(name,comp)* substitui o componente com nome *name* pelo componente *comp*, refazendo todos os *bindings*.

No Open-ORB o método *composition* pode ser aplicado a uma interface, componente ou componente composto. Isto faz com que o poder de representação dos grafos de

Tabela 4: Métodos fornecidos por um meta-objeto *Composition*.

Nome do método	**Descrição**
inspect()	Retorna um grafo que descreve as conexões dos componentes.
bind(iface1, iface2)	Conecta a interface com nome *iface1* a interface com nome *iface2* no grafo de componentes.
break(iface1,iface2)	Desconecta a interface com nome *iface1* da interface com nome *iface2*
insert(comp, bind)	Insere o componente *comp* e realiza a vinculação através dos pares de interfaces fornecidas pelo parâmetro *bind*.
addIIF(name, iface, comp)	Insere a interface *iface* do componente *comp* com nome *name* no grafo de componentes.
remove(name)	Remove o componente que possuir nome *name* do grafo de componentes.
replace(name, comp)	Substitui o componente com nome *name* pelo componente fornecido em *comp*. Todos bindings entre as interfaces do componente original são refeitos para o novo componente.

componentes não se restrinja apenas aos componentes compostos e passem a representar quase a totalidade dos vínculos(*bindings*) existentes em uma aplicação. Isso pode ser visto na própria implementação da infra-estrutura de comunicação(Figura 9) do Open-ORB, que será discutida na próxima Seção. Existem componentes compostos ligados a outros componentes através de *bindings* locais constituindo um caminho que pode ser percorrido pelo meta-objeto *composition*. Dessa forma, partindo do objeto *Capsule Local* é possível alcançar os componentes *Capsule*, *Invoke Factory*, *NodeMngr*, *Accept Factory*, *Accept*, *Thread Mngr*, *Transport Wrapper*, *Dispatcher Factory*, *Dipatcher*, *EndPoint*, *Protocol* e por fim a interface que exporta o método *vender*.

3.1.3 Utilizando abstrações de alto nível na construção da infra-estrutura de componentes

As abstrações de alto nível (Portas, Interfaces, Componentes e *Bindings*) definidas na seção anterior serão utilizadas nesta seção para definir a infra-estrutura de componentes. A utilização de abstrações que suportam reflexão computacional permite gerar componentes que também suportam reflexão e, em conseqüência, obtem-se alta capacidade de customização. Esta capacidade é obtida uma vez que os próprios elementos da infra-estrutura de componentes são componentes compostos ligados através de suas interfaces e *bindings* locais onde os meta-objetos (*encapsulation* e *composition*) podem atuar na reflexão e adaptação de forma transparente e flexível.

Os primeiros componentes criados, seguindo essa idéia, foram *Capsule* e *NodeMngr*. Estes componentes são responsáveis pelas funcionalidades básicas da infra-estrutura de

Tabela 5: Funcionalidades do componente Capsule.

Nome do método	**Descrição**
server()	Inicia o modo de espera de novas conexões.
stopserver()	Para o processo de escuta de novas conexões.
registerComponent(c, n) $\longrightarrow id$	Registra o componente c com o nome n e retorna *id*.
delComponent(id)	Remove componente registrado *id*.
getIdComp(n) $\longrightarrow id$	Retorna o *id* do componente registrado com nome n.
getIRef(c,n) $\longrightarrow i$	Retorna a interface i do componente registrado com nome c e interface de nome n.
getRemoteInterface(h,p,c,n) $\longrightarrow i$	Retorna a interface i do componente com nome c e interface n que estiverem disponíveis no host h e porta p.
callMethod(m, i, a) $\longrightarrow r$	Invoca método m da interface i com os parâmetros a.
annouceMethod(m, i, a)	Invoca método m da interface i com os parâmetros a sem esperar retorno.
sendMethod(m, i, a) $\longrightarrow v$	Invoca método m da interface i com os parâmetros a e retorna um *handle* v.
recvMethod(v) $\longrightarrow r$	Obtém o retorno do método *sendMethod*.

comunicação. O componente *Capsule* é responsável por gerenciar o ambiente local provendo para isso a interface *provide_capsule* que disponibiliza as funções descritas na Tabela 5.

Todos os componentes que podem ser acessados remotamente devem ser registrados no componente *Capsule*. Esse registro é feito através do método *registerComponent(c, n)* que recebe a referência e o nome do componente, retornando seu *id* de registro. Para obter o *id* de um componente pode-se usar o método *getIdComp(n)*. Para obter a interface de um componente registrado utiliza-se *getIRef(c, n)*, que recebe o *id* do componente e o nome da interface externa e retorna a referência da interface correspondente. Como um *Capsule* pode ser acessado remotamente, ele próprio se registra com *id* igual a 1. Isso permite que para acessar uma funcionalidade do componente *Capsule* é necessário apenas invocar *getIRef(1,"provide_capsule")*.

Tabela 6: Funcionalidades do componente NodeMngr.

Nome do método	**Descrição**
server(p)	Inicia o processo de reconhecimento de conexões. p é um parâmetro opcional que determina uma porta para ser iniciada.
stopserver(p)	Esta função para o processo de reconhecimento de conexões. p é um parâmetro opcional que determina uma porta para ser parada.
newPort($\{c\}$) $\longrightarrow a$	Cria uma ou mais portas para tratar os componentes c.
delport(a)	Remove a porta com identificador a
getPort(i) $\longrightarrow a$	Obtém uma ou mais portas através do valor i.

O componente *NodeMngr* gerencia as portas do ORB através da interface *provide_node-*

mngr com os métodos descritos na Tabela 6. O método *newPort({c})* recebe como parâmetro uma lista de componentes que poderão receber invocações em uma porta. Em seguida, são utilizadas as funções *inspect* e *getFactoryMethod*, do meta-objeto *encapsulation*, para separar os componentes que possuem um *Factory* com nome *RemoteListen*. Esse *Factory* é padronizado para utilização do componente *NodeMngr* e a sua presença determina os componentes e *bindings* que serão criados e combinados para tratar e processar as mensagens recebidas (Figura 9). Na ausência desse *Factory* utiliza-se o *Accept Factory* vinculado ao *NodeMngr*. O resultado da função *newPort* é uma lista que relaciona os componentes fornecidos com as portas criadas. O método *getPort(i)* pode ser usado para obter uma ou mais portas de um *NodeMngr*. Para isso, este método recebe o número da porta e retorna o componente *Accept* correspondente, ou recebe "*"e retorna uma lista de componentes *Accept*.

3.1.4 Tratando invocações

O processo de tratamento de invocações é iniciado pela chamada do método *server* da interface *provide_capsule*. Este método usa a interface importadora *use_nodemngr*, do componente *Capsule*, para invocar os métodos *newPort* e *server*, que criam e verificam a lista de portas criadas com sua respectiva lista de *Factories*. Para as portas que possuem *Factory* definido, esse *Factory* é invocado. Para as que não possuem, o método *create* do *Accept Factory* é invocado.

O método *create* cria um componente composto formado pelos componentes *Accept*, *Transport Wrapper*, *Dispatcher Factory* e *Thread Mngr*. Na seqüência, esse método cria um *binding* local entre o componente criado e a respectiva porta do *NodeMngr*. O último passo é invocar o método *listen* do componente *Accept*.

O método *listen* invoca o método *createListen*, do componente *Transport Wrapper*, que fica bloqueado esperando uma nova conexão. Quando uma conexão é estabelecida, o método *create* do componente *Dispatcher Factory* é invocado, retornando um componente composto pelos componentes *Dispatcher* e *Protocol*. O próximo passo é estabelecer dois *bindings* locais: o primeiro entre este componente e o componente *EndPoint*, retornado pelo método *createListen*, e o segundo entre os componentes *Dispatcher* e *Capsule*. O último passo é invocar o método *createThread* do componente *ThreadMngr* para processar em uma *Thread* separada a invocação do método *process* do componente *Dispatcher*.

Para tratar cada conexão, *process* invoca o método *receive* do componente *EndPoint* que obtém a mensagem. Esta é tratada através do método *unmarshal*, do componente

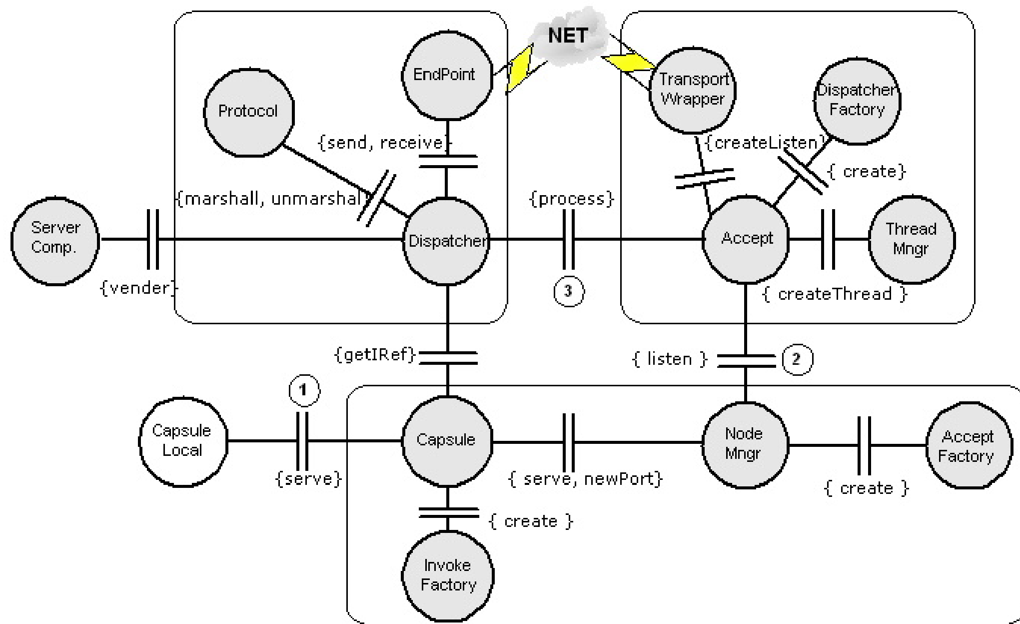

Figura 9: Caminho de execução do servidor

Protocol e através do método *getIRef* do componente *Capsule*, a interface que irá executar o método é localizada. Para invocar o método desejado, é necessário criar um *binding* local entre essa interface e o componente *Dispatcher*. Através desse *binding* pode-se invocar o método desejado, receber seus parâmetros de retorno, fazer o *marshall* da mensagem e enviar a resposta através do método *send*, finalizando o caminho de execução do servidor.

3.1.5 Realizando invocações

Descrevemos anteriormente como o Open-ORB implementa o mecanismo de recebimento e tratamento de invocações remotas. Neste momento detalharemos como as invocações são enviadas pelos clientes, utilizando para isso a invocação do método *vender*, pertencente a interface *DepVendas* do componente com *id* igual a 5.

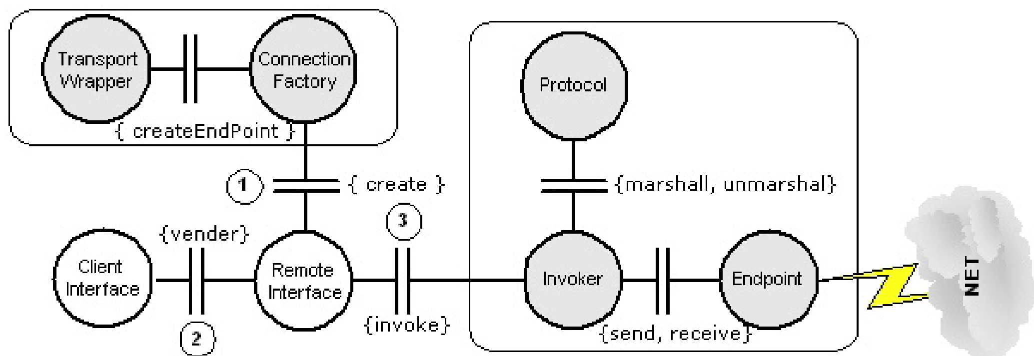

Figura 10: Caminho de invocação do cliente

A Figura 10 ilustra os componentes envolvidos na construção do caminho de execução do lado cliente. O primeiro passo é obter uma interface local que representa a interface remota. Para isso a interface *provide_capsule* fornece o método *getRemoteInterface("localhost", 8080, "CompVendas", "DepVendas").* Este método recebe o nome do host, a porta, o nome do componente e o nome da interface. Em seguida, retorna uma interface local que exporta os métodos da interface remota. Internamente o que esse método faz é criar uma interface que atua como *proxy* genérico para os métodos exportados por *DepVendas* e criar, através do componente *Invoke Factory*(Figura 9), um componente composto formado pelos componentes *Transport Wrapper* e *Connection Factory.* Na seqüência, o método cria um *binding* local entre o componente composto e a interface criada(*Remote Interface*). O *BindCtrl*, do binding local criado, é associado através da função *addFactoryMethod("vender", "FactoryInvocation",{BindCtrl,"create"})* do meta-objeto *encapsulation.*

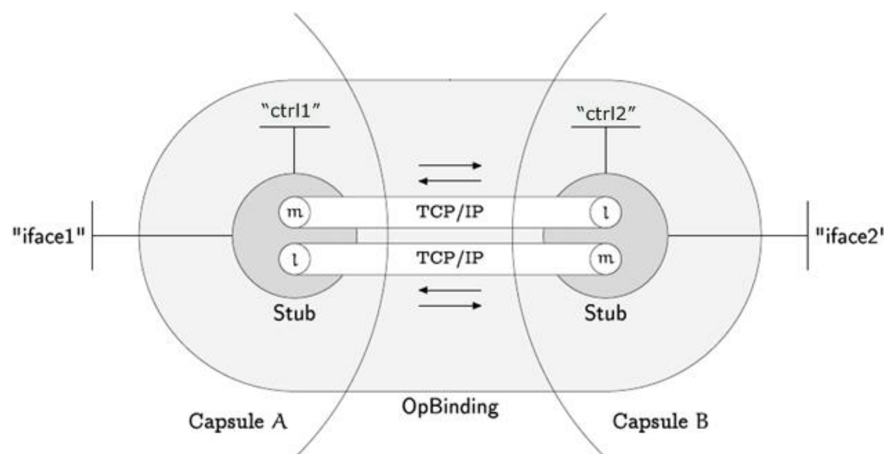

Figura 11: Interfaces de um Operational Binding (ANDERSEN, 2002)

Para invocar o método remoto a interface cliente cria um *binding* local entre sua interface e a interface remota. Quando o cliente invoca o método *vender*, o *BindCtrl* verifica que o método tem um *Factory* com nome *FactoryInvocation.* Neste momento, o método *create* do componente composto formado por *Connection Factory* é invocado, criando um componente composto formado pelos componentes *EndPoint*, *Protocol* e *Invoker.* O passo seguinte envolve a criação de um *binding* local entre esse componente e a interface remota, modificando através da função *setBindingMethod("vender",BindCtrl)*, do meta-objeto *encapsulation*, o *binding* do método *vender.* Por fim, este método invoca a função *delFactoryMethod("vender","FactoryInvocation")* para evitar que o caminho seja recriado para cada invocação. Dessa forma, quando a interface cliente invoca o método *vender*, esse método é repassado para a interface remota, que por sua vez repassa para o compo-

nente *Invoker*. Este componente realiza o *marshall* da mensagem, através do componente *Protocol*, e a envia através do componente *EndPoint*. Dependendo do componente *Invoker* utilizado, o cliente pode ficar ou não bloqueado no método *receive* esperando a resposta do servidor. No caso de uma resposta, o componente *Invoker* invoca o método *unmarshal* para tratar a mensagem e retorna para a interface cliente o resultado.

3.1.6 Utilizando a infra-estrutura de comunicação do Open-ORB

A discussão anterior tratava da definição dos mecanismos de tratamento e invocação de requisições remotas. Discutiremos nessa seção a utilização desses mecanismo para a construção de um *Operational binding*.

Tabela 7: Funcionalidades do serviço de *Naming* do Open-ORB.

Nome do método	**Descrição**
exportCaps(n, c)	Exporta *Capsule* c com nome n
exportIRef(n,i)	Exporta interface i com nome n
delCaps(n)	Remove *Capsule* com nome n.
delIRef(n)	Remove interface com nome n.
importCaps(n) $\longrightarrow c$	Obtém o *Capsule* c com nome n.
importIRef(n) $\longrightarrow i$	Obtém interface i com nome n.
listCaps() $\longrightarrow \{c\}$	Obtém uma lista de todos os *Capsules*.
listIRefs() $\longrightarrow \{i\}$	Obtém uma lista de todas as interfaces.

Um *Operational binding (Op. binding)* é um componente composto e distribuído formado por quatro interfaces (Figura 11): *iface1*,*iface2*, *ctrl1* e *ctrl2*. As interfaces *iface1* e *ctrl1* disponibilizam a interface do componente remoto e sua respectiva interface de controle. O mesmo ocorre para *iface2* e *ctrl2* que disponibilizam, respectivamente, a interface local e a interface de controle local. Dessa forma, se for necessário invocar métodos da interface remota, deve-se criar um *binding* local com a interface *iface1*. Mas, se for preciso parar a interface local, deve-se criar um *binding* local com *ctrl2*. No Open-ORB, o *Op. binding*, como os demais *bindings* explícitos (*Signal* e *Stream*), são implementados através de um componente. Dessa forma, um *Op. binding* é criado por um componente com apenas uma interface *provide_opbind* que é registrada no *Capsule* para receber invocações remotas. Esta interface possui dois métodos: *create* e *createRemote*. O método *create* é invocado pelo método *remoteBind* para criar um *Operational binding*. O método *remoteBind(iref1, iref2)*, recebe duas interfaces: *iref1* que pode ser uma interface local como *IRef(obja, {"vender","cancelarPedido"},{"entregar"})* e *iref2* que pode ser uma interface remota como *IRef({}, {"entregar"}, {"cancelarPedido"})*. O processo de construção de um *Op. binding* resume-se a criação de dois componentes, um local e outro remoto, tendo

cada um quatro interfaces. O método *create* da interface *provide_opbind*, inicialmente cria o componente *compLocal* com apenas a interface local informada, *iref1*. Esse componente é registrado no *Capsule* e através do Id desse componente é criado um outro componente para controlar a interface local. De posse das duas interfaces locais, *iface2* e *ctrl2* o método *create* invoca o *createRemote*, ao host que disponibiliza a interface remota *iref2*. São passados para esse método a interface *iface2, ctrl2* e a interface remota *iref2*. No host remoto, o método *createRemote* cria um componente *compFunc* com as interfaces *iface1*, *iface2* e *ctrl1* recebidas como parâmetro e registra o novo componente no *Capsule*. A partir do id do novo componente, um componente *compCtrl*, com interface *ctrl1* é criado e composto, através do meta-objeto *composition*, com o componente *compFunc*. Por fim, a função retorna uma lista com as interfaces *iface1* e *ctrl1*. De volta ao *host* que iniciou o processo de criação do *Op. binding*, o método *create* recebe os parâmetros de retorno do método *createRemote* e também realiza a composição do *compLocal* com as interfaces fornecidas, finalizando dessa forma o processo de criação de um *Operational binding*.

Assim como o *Operational Binding*, um *Signal* e *Stream binding* também são componentes compostos e distribuídos. Um *Signal binding* é criado pela função *sigBind(src, sink)* e é composto por três interfaces: *src*, *sink* e *ctrl*. A inteface *src* representa a origem dos eventos, *sink* o destino e *ctrl* a interface que controla a atividade da interface *sink*. A construção de um Signal Binding segue a mesma sequência da mostrada para um Op. *binding*, diferenciando apenas pela mudança dos componentes Invoker e Dispatcher, mostrados na Figura 9, que são substituídos através da função *replace* do meta-objeto *composition* por componentes que não esperem retorno no Invoker, apenas *send*, e não produzam retorno no Dispatcher, apenas *receive*.

Um *Stream binding* é criado pela função *streamBind(src, sink)* e é composto por três interfaces: *src*, *sink* e *ctrl*. A inteface *src* representa a origem dos Streams, *sink* o destino e *ctrl* a interface que controla a atividade da interface *sink*. A construção de um *Stream binding* também segue a mesma sequência da mostrada para um Op. *binding*, diferenciando apenas pela mudança dos componentes Invoker, Dispatcher e Transport Wrapper, mostrados na Figura 9, que são substituídos. Os dois primeiros pelo mesmo motivo do *Signal Binding* e o terceiro pelo fato do Transport Wrapper utilizado nos *bindings* anteriores serem TCP/IP, enquanto que o utilizado pelo *Stream Binding*, ser um UDP/IP.

3.2 LOpenOrb: Implementação Lua do Open-ORB

O LOpenOrb(CACHO; BATISTA, 2005a, 2005b) é uma implementação em Lua da arquitetura Open-ORB. Esta implementação explora as facilidades reflexivas, a natureza interpretada e o sistema de tipos dinâmico fornecidos pela linguagem Lua para implementar um middleware reflexivo com grande capacidade de adaptação dinâmica. Os detalhes da implementação e exemplos de utilização do LOpenOrb serão descritos nas sub-seções a seguir.

3.2.1 Arquitetura

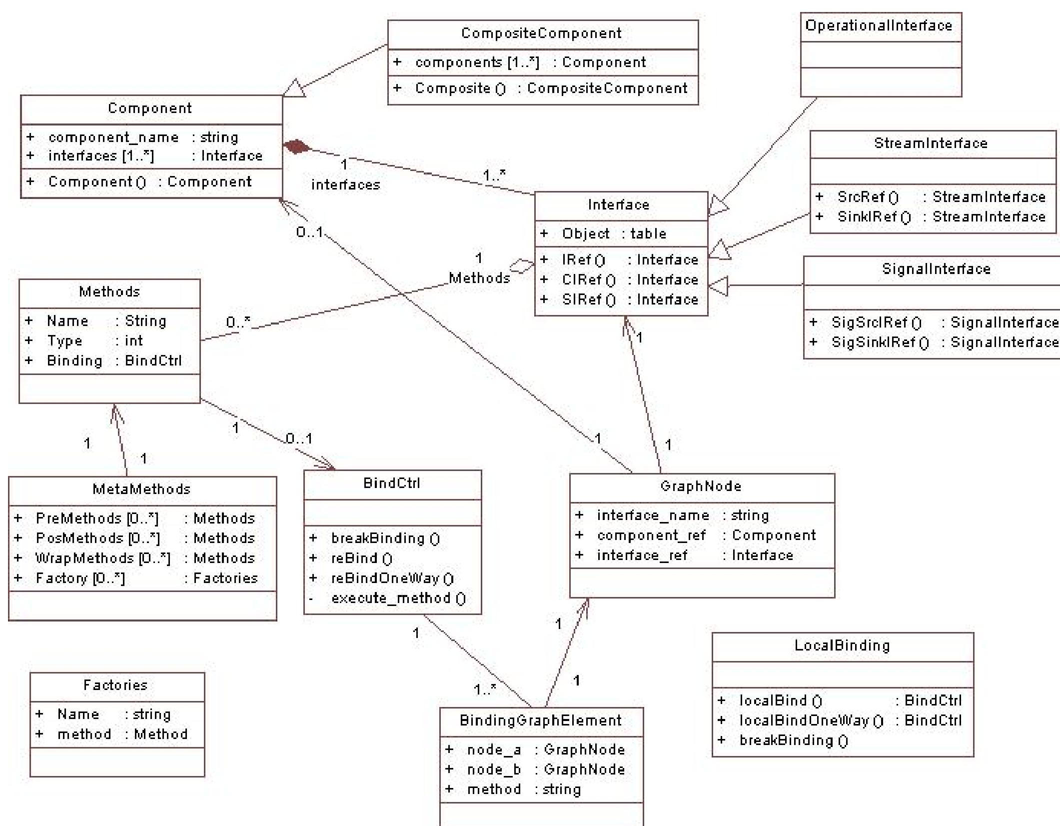

Figura 12: Arquitetura do LOpenORB

A Figura 12 ilustra os elementos que compõem a arquitetura de implementação do LOpenORB. Nela estão descritas as abstrações de alto nível fornecidas para facilitar o desenvolvimento e a adaptação de um middleware reflexivo. Ou seja, nesta seção descrevemos os elementos referentes a infra-estrutura básica ilustrada na Figura 1. Dessa forma, o primeiro elemento a compor a infra-estrutura básica é a classe *Interface* que representa o ponto de acesso para os objetos e componentes. Uma interface possui vários métodos importados e exportados representados pela classe *Methods*. Os métodos exportados são

implementados pelo elemento associado ao atributo *Object*. Cada método, representado pela classe *Methods*, possui como atributo o nome do método, seu tipo (importado e exportado) e um *Binding* usado para invocar um método relacionado a um respectivo *BindCtrl*. A classe *BindCtrl* é um elemento chave na vinculação de interfaces e na composição de componentes. Ela é criada pelo método *localBind* da classe *LocalBinding*. Um *BindCtrl* possui um *BindingGraphElement* que estabelece um vínculo entre os *node_a* e *node_b*. Cada nó é uma instância da classe *GraphNode* que possui como atributos uma referência da classe *Interface*, o nome da interface e uma referência da classe *Component*.

A composição das Interfaces é realizada pela classe *Component* e *CompositeComponent*. A classe *Component* representa um componente primitivo e armazena o nome do componente e as interfaces externas fornecidas pelo componente. A classe *CompositeComponent* representa um componente composto formado pelos componentes armazenados no atributo *components*.

A classe *MetaMethods* é criada quando uma interface possui um meta-objeto associado. Cada método da interface possui uma instância de *MetaMethods*. Essa classe registra os métodos *Pre, Pos e Wrap*, que serão executados respectivamente antes, depois e durante a execução do método original. Além disso, possui a propriedade *Factory* responsável por determinar uma lista de possíveis *Factories* que podem ser usadas para construir o caminho de invocação de cada método. Cada *Factory* é representada pela classe *Factories* que possui o nome da *Factory* e um atributo *method*, da classe *Method*, usado para invocar o construtor.

Na arquitetura Open-ORB os *bindings* são os elementos que vinculam componentes e interfaces. No LOpenORB, os *bindings* locais são representados pela classe *BindCtrl*, de modo que, quando um método é invocado em uma interface importadora, esse método é encaminhado através da função *execute_method* para o *BindCtrl* correspondente. Na classe *BindCtrl*, o *BindingGraphElement* é recuperado e verifica a existência de um meta-objeto para aquela interface. Caso exista, os métodos *Pre, Pos, Wrap* e *Factory* são invocados. Os *Factories* são um tipo especial de Pré-métodos, adicionados pela abordagem do LOpenORB que, ao invés de receber os parâmetros da função invocada, sempre recebe uma referência do *BindCtrl* utilizado. Dessa forma, é possível, dentro das funcionalidades do *Factory*, modificar os elementos que compõem o *BindingGraphElement* e assim alterar o caminho de execução da função. Podem existir vários *Factories* para um mesmo método, no caso do *BindCtrl*, é utilizado o *Factory* com nome *FactoryInvocation*.

3.2.2 Funcionalidades do Nível-Base

As funcionalidades básicas do LOpenORB são implementadas pelas classes mostradas na Figura 12. No entanto, para manter a compatibilidade com a idéia de "pacote"da linguagem Lua, a Figura 13 mostra a criação da tabela *LOpenORB* que irá representar o "pacote"com todas as funcionalidades do LOpenORB. Dessa forma, ao invés de se invocar *Interface:IRef*, deve-se invocar *LOpenORB.IRef*. Ou seja, o arquivo *lopenorb_interfaces* além de definir as funcionalidades da classe *Interface*, também faz o mapeamento entre as funções da classe *Interface* e a tabela LOpenORB.

```
1  LOpenOrb = {}
2  function LOpenOrb.init(arg)
3
4          LOpenOrb.hostname = arg[2]
5          LOpenOrb.port = arg[3]
6
7          require "lopenorb_lbind"
8          require "lopenorb_interfaces"
9          require "lopenorb_component"
10         require "lopenorb_composite"
11         require "lopenorb_encapsulation"
12         require "lopenorb_metacomposition"
13         require "lopenorb_marshaler"
14         require "lopenorb_msg"
15         require "lopenorb_transport"
16         require "lopenorb_invoker"
17         require "lopenorb_threadmngr"
18         require "lopenorb_dispatcher"
19         require "lopenorb_nodemngr"
20         require "lopenorb_capsule"
21         require "lopenorb_confinfra"
22         require "lopenorb_accept"
23         require "lopenorb_opbind"
24         require "lopenorb_sigbind"
25         require "lopenorb_streambind"
26
27 end
```

Figura 13: Definição do "pacote"LOpenORB.

A utilização de uma parte das funcionalidades mostrada na Tabela 2 pode ser vista na Figura 14. Esta figura ilustra a utilização de um *binding* local que vincula as interfaces Loja e Vendedor. Nas Linhas 9 e 10, as duas interfaces exportadora e importadora, respectivamente, são criadas e na Linha 11, o *binding* local é definido. O método *vender* da interface importadora é invocado na Linha 12.

3.2.3 Funcionalidades do Meta-nível

No LOpenOrb os meta-modelos são fornecidos por um componente composto chamado *MetaModelComponent*. Este componente possui duas interfaces externas: *EncapsulationMetaModel* e *CompositionMetaModel*. A primeira interface exporta o método *encapsulation* que recebe como parâmetro a referência de uma interface ou componente

```
1  require "LOpenOrb"
2  LOpenOrb.init(arg)
3
4  local objLoja = {}
5  function objLoja:vender(produto)
6                          . . .
7  end
8
9  objLoja.InterfaceLoja = LOpenOrb.IRef(objLoja, {"vender"}, {})
10 local InterfaceVendedor = LOpenOrb.IRef({}, {}, {"vender"})
11 LOpenOrb.localBind({objLoja.InterfaceLoja},{InterfaceVendedor})
12 InterfaceVendedor:vender("145214")
```

Figura 14: Criando um binding local entre duas interfaces

e retorna um *encapsulation* meta-objeto para o respectivo elemento. A segunda interface exporta o método *composition* que recebe como parâmetro a referência de uma interface ou componente e retorna um meta-objeto *composition* para o respectivo elemento.

```
1  local objLog = {}
2  function objLog:regvenda(...)
3                          . . .
4  end
5  objLog.InterfaceLog = LOpenOrb.IRef(objLog, {"regvenda"}, {})
6
7  local bindctrl = LOpenOrb.localBindOneWay({},{objLoja.InterfaceLog})
8
9  local meta_encaps = LOpenORB.metamodel:encapsulation(
       InterfaceVendedor)
10 meta_encaps:addPreMethod("vender", {"regvenda",bindctrl})
```

Figura 15: Utilizando um meta-objeto encapsulation

A Figura 15 ilustra a utilização de um meta-objeto *encapsulation*. Nesta figura, o *objLog* é definido com o método *regvenda*. Na seqüência, a interface *InterfaceLog* é criada para ser o alvo de um bind *one-way* definido na Linha 7. A obtenção do meta-objeto *encapsulation*, na linha 9, permite invocar o método addPreMethod que adiciona uma funcionalidade extra ao método *vender* da interface *InterfaceVendedor*.

```
1  objLog = {}
2  objLog.ProvideMyY = LOpenOrb.IRef(objLog, {"vender"}, {})
3  objLog.UseMyY = LOpenOrb.IRef(objLog, {},{"vender"})
4  complog = LOpenOrb.Component("CompLog",{{"ProvideMyY", objLog.
       ProvideMyY},{"UseMyY", objLog.UseMyY}}, objLog)
5
6  local meta_compos = LOpenORB.metamodel:composition(CompXY)
7  meta_compos:insert(complog, {{"ProvideMyY","use_Y"},{"UseMyY","
       provide_Y"}})
```

Figura 16: Utilizando um meta-objeto composition

Um código correspondente ao da utilização do meta-objeto *encpasulation* pode ser visto na Figura 16, onde o meta-objeto *composition* é utilizado para inserir um componente entre duas interfaces. O componente inserido é definido na Linha 4. Este componente é formado por duas interfaces que importam e exportam o método *vender*. Na Linha

6, o meta-objeto *composition* é obtido para o componente CompXY, e na seqüência, o componente *complog* é inserido entre as interfaces *use_Y* e *provide_Y*.

3.2.4 Infra-estrutura de componentes

Nesta seção descrevemos a infra-estrutura de componentes ilustrada na Figura 1. A infra-estrutura de componentes do LOpenOrb é formada inicialmente pelos componentes *Capsule* e *NodeMngr*. A Figura 17 ilustra o código necessário para criar e compor estes dois componentes. As três primeiras linhas mostram a definição das interfaces *useInvokeFactory*, *useNodeMngr* e *provideCapsule*. Na Linha 5, o componente *Capsule* é criado e na Linha 7 a interface *LOpenORB.localcapsule* é criada para facilitar o acesso aos métodos fornecidos pelo componente *Capsule*. Na Linha 9, é estabelecido um *binding* local entre *LOpenORB.localcapsule* e *LOpenORB.localCompCapsule*.

```
1  objCapsule.useInvokeFactory=LOpenOrb.IRef({},{},{"create"})
2  objCapsule.useNodeMngr=LOpenOrb.IRef({},{},{"serve","stopserver","
       newPort"})
3  objCapsule.provideCapsule = LOpenOrb.IRef(objCapsule,{"
       registerComponent","delComponent","getIdComp","getIRef","
       callMethod","getRemoteInterface","serve","stopserve","
       annouceMethod","sendMethod","recvMethod"},{})
4
5  LOpenOrb.localCompCapsule = LOpenOrb.Component("capsule",{{"
       provide_capsule", objCapsule.provideCapsule},{"use_NodeMngr",
       objCapsule.useNodeMngr},{"use_InvokeFactory",objCapsule.
       useInvokeFactory}}, objCapsule)
6
7  LOpenOrb.localcapsule=LOpenOrb.IRef({},{},{"registerComponent","
       delComponent","getIdComp","getIRef","callMethod","
       getRemoteInterface","serve","stopserve","annouceMethod","
       sendMethod","recvMethod"})
8
9  LOpenOrb.localBind({LOpenOrb.localcapsule}, {LOpenOrb.
       localCompCapsule.interfaces["provide_capsule"],"provide_capsule",
       LOpenOrb.localCompCapsule})
10
11 objNodeMngr.provideNodeMngr=LOpenOrb.IRef(objNodeMngr,{"serve","
       stopserver","stopPort","newPort","delport","getPort"},{})
12 objNodeMngr.useFactory = LOpenOrb.IRef({},{},{"create"})
13
14 LOpenOrb.localCompNodeMngr = LOpenOrb.Component("nodemngr",{{"
       provide_nodemngr", objNodeMngr.provideNodeMngr},{"use_factory",
       objNodeMngr.useFactory}}, objNodeMngr)
15
16 LOpenOrb.localBind({LOpenOrb.localCompCapsule.interfaces["
       use_NodeMngr"],"use_NodeMngr",LOpenOrb.localCompCapsule}, {
       LOpenOrb.localCompNodeMngr.interfaces["provide_nodemngr"],"
       provide_nodemngr",LOpenOrb.localCompNodeMngr})
17
18 LOpenOrb.localcapsule:registerComponent(LOpenOrb.localCompCapsule,"
       Capsule")
19 LOpenOrb.localcapsule:registerComponent(LOpenOrb.localCompNodeMngr,"
       NodeMngr")
```

Figura 17: Criando os componentes Capsule e NodeMngr

O componente *NodeMngr* é criado através da definição de suas duas interfaces *provideNodeMngr* e *useFactory*, nas Linhas 11 e 12, e através do componente criado na Linha

14.

A relação de dependência entre os componentes *Capsule* e *NodeMngr* é estabelecida através da criação de um *binding* local entre as interfaces *provideNodeMngr* e *useNodeMngr*. Por fim, nas Linhas 18 e 19, os componentes *Capsule* e *NodeMngr* são registrados para serem acessados remotamente.

```
1  function ObjAcceptFactorie:create()
2  local compAccept = ObjAcceptFactorie:createComponentAccept()
3
4  return  LOpenOrb.Composite("accept_factory",
5    {{"provide_accept",{compAccept,"provide_accept"}},
6     {"use_factorie_dispather",{compAccept,"use_factorie_dispather"}},
7     {"use_TransportWrapper",{compAccept,"use_TransportWrapper"}},
8     {"use_ThreadMngr",{compAccept,"use_ThreadMngr"}}
9    },
10   {{compAccept, compDispatherFactorie,compTransportWrapper,
       compThreadMngr}},
11   {{"use_factorie_dispather",{compAccept, "use_factorie_dispather"
       }},
12    {"use_TransportWrapper",{compAccept, "use_TransportWrapper"}},
13    {"use_ThreadMngr",{compAccept, "use_ThreadMngr"}},
14    {"provide_createDispather",{compDispatherFactorie, "
        provide_createDispather"}},
15    {"provide_createTransport",{compTransportWrapper, "
        provide_createTransport"}},
16    {"provideThread",{compThreadMngr, "provideThread"}}},
17   {{"use_factorie_dispather","provide_createDispather"},
18    {"use_TransportWrapper","provide_createTransport"},
19    {"use_ThreadMngr","provideThread"}})
20 end
21
22 ObjAcceptFactorie.Factorie = LOpenOrb.IRef(ObjAcceptFactorie, {"
     create"},{})
23 compAcceptFactorie = LOpenOrb.Component("factory_accept",{{"
     provide_create", ObjAcceptFactorie.Factorie}}, ObjAcceptFactorie)
24 LOpenOrb.localBind({CompNodeMngr.interfaces["use_factorie"]},{
     compAcceptFactorie.interfaces["provide_create"]})
```

Figura 18: Definindo Caminho de execução do servidor

Para tratar invocações, o *NodeMngr* deve ser conectado ao componente *Accept Factory* para poder definir os demais componentes necessários ao tratamento de uma invocação remota. A Figura 18 mostra o código correspondente ao componente *Accept Factory*. As linhas 1 a 20 são responsáveis por criar um componente composto formado por outros quatro componentes: *compAccept*, *compDispatcherFactory*, *compTransportWrapper* e *compThreadMngr*. Estes quatro componentes irão definir o caminho de execução para cada invoção remota. A Linha 23 define o componente *Accept Factory* que é vinculado ao componente *NodeMngr* na linha seguinte.

A invocação de métodos remotos é mostrada na Figura 19. As linhas 1 a 17 são responsáveis pela definição do componente composto formado pelos componentes: Protocol, Invoker e EndPoint. Estes três componentes são definidos, respectivamente, pelas linhas 2,3 e 4. Na linha 6, o componente *compCompInvoker* é definido e na linha 16, este

```
1  function ConnectionFactorie:create(iref)
2    local compInvoker = ConnectionFactorie:createComponentInvoker(iref
         .idcomp, iref.interface)
3    local compEndpoint = ConnectionFactorie.Transport:createEndpoint(
         iref.host,iref.port)
4    local compProtocol = ConnectionFactorie.Protocol:create()
5
6    local compCompInvoker =  LOpenOrb.Composite("invoker",
7          {{"provide_invoke",{compInvoker,"provide_invoke"}}},
8          {{compInvoker, compEndpoint, compProtocol}},
9          { {"use_protocol",{compInvoker, "use_protocol"}},
10           {"use_endpoint",{compInvoker, "use_endpoint"}},
11           {"provide_endpoint",{compEndpoint, "provide_endpoint"}},
12           {"provide_protocol",{compProtocol, "provide_protocol"}}
13         },{{"use_protocol","provide_protocol"},
14            {"use_endpoint","provide_endpoint"}})
15
16     LOpenOrb.localBind({iref.objref.intInvoke}, {compCompInvoker.
          interfaces["provide_invoke"]})
17 end
18
19 ConnectionFactorie.Factorie = LOpenOrb.IRef(ConnectionFactorie, {"
       create"},{})
20 compConnectionFactorie = LOpenOrb.Component("factorie",
21     {{"provide_create", ConnectionFactorie.Factorie},
22      {"use_transport", ConnectionFactorie.Transport}},
           ConnectionFactorie)
23 InvokerFactorie = LOpenOrb.Composite("invokercomp",
24     {{"provide_create",{compConnectionFactorie,"provide_create"}}},
25     {{compConnectionFactorie,compTransportWrapper}},
26     {{"use_transport",{compConnectionFactorie, "use_transport"}},
27      {"provide_createTransport",{compTransportWrapper, "
           provide_createTransport"}}},
28     {{"use_transport","provide_createTransport"}})
```

Figura 19: Definindo Caminho de invocação do servidor

componente é vinculado a interface cliente que realiza a invocação.

O componente *Connection Factory* é definido na linha 20 e composto com o componente *Transport Wrapper* na linha 23.

A utilização da infra-estrutura de comunicação será demonstrada pela utilização de um *Operational binding* para invocar métodos fornecidos por um servidor bancário. Para este exemplo são necessários três participantes: cliente, servidor de nomes e servidor de aplicação. O código do cliente e do servidor serão detalhados abaixo. O servidor de nomes é fornecido pelo LOpenORB como um serviço adicional. Dessa forma, os métodos fornecidos pelo serviço de *Naming* do LOpenORB serão baseados no modelo descrito pela Tabela 7.

O código do sevidor de aplicação está descrito na Figura 20. As duas primeiras linhas são utilizadas para inicializar e carregar as funcionalidades do *LOpenORB* com os parâmetros fornecidos pela Linha de comando. Nas quatro Linhas seguintes são definidas as funcionalidades do servidor bancário. Na Linha 9 é criada a interface que exporta os métodos do servidor e nas Linhas 10 e 12, o componente é criado e registrado.

```
1  require "LOpenOrb"
2  LOpenOrb.init(arg)
3
4  BankServer = {bal = 0}
5  function BankServer:deposit(val) self.bal = self.bal + val end
6  function BankServer:withdraw(val) self.bal = self.bal - val end
7  function BankServer:balance() return self.bal end
8
9  BankServer.intAtendeCliente = LOpenOrb.IRef(BankServer, {"deposit","
       withdraw","balance"},{})
10 compBank = LOpenOrb.Component("Bank",{{"AtendeCliente", BankServer.
       intAtendeCliente}}, BankServer)
11
12 id = LOpenOrb.localcapsule:registerComponent(compBank,"CompBank")
13
14 local irefNaming = LOpenOrb.localcapsule:getRemoteInterface("
       localhost", 9000, "NameServer", "intNameServer")
15 local proxyNaming = LOpenOrb.IRef({},{},{"exportIRef"})
16 LOpenOrb.localBind({proxyNaming},{irefNaming})
17
18 proxyNaming:exportIRef("BankServer", LOpenOrb.localcapsule:getIRef(
       id,"AtendeCliente"))
19 LOpenOrb.localcapsule:serve()
```

Figura 20: Código do servidor de aplicação

Na seqüência, são realizadas as operações para publicar o servidor bancário no serviço de *Naming*. O primeiro passo consiste em obter a referência do serviço de *Naming*. Isso é feito na Linha 14, onde o método *getRemoteInterface* obtém a referência da interface *intNameServer* do componente *NameServer* que esta disponível no servidor *localhost:9000*. Em seguida, nas Linhas 15 e 16, é criada uma interface que importa o método *exportIRef* que é vinculada através de um *binding* local à interface *irefNaming*. Por fim, a interface *AtendeCliente* do componente *compBank* é exportada como o nome *BankServer* e através do método *serve*, o servidor fica apto a receber invocações remotas.

```
1  require "LOpenOrb"
2  LOpenOrb.init(arg)
3
4  local irefNaming = LOpenOrb.localcapsule:getRemoteInterface("
       localhost", 9000, "NameServer", "intNameServer")
5  local proxyNaming = LOpenOrb.IRef({},{},{"importIRef"})
6  LOpenOrb.localBind({proxyNaming},{irefNaming})
7
8  local AtendeClienteIref = proxyNaming:importIRef("BankServer")
9  local LocalBank = LOpenOrb.IRef({}, {},{"deposit","withdraw","
       balance"})
10
11 local RemoteBank = LOpenOrb.remoteBind(LocalBank, AtendeClienteIref)
12
13 RemoteBank.interfaces["iface1"]:deposit(100)
14 local ClientBank = LOpenOrb.IRef({}, {},{"deposit","withdraw","
       balance"})
15 LOpenOrb.localBind({RemoteBank.interfaces["iface1"]},{ClientBank})
16 ClientBank:withdraw(50)
17 print(ClientBank:balance())
18 RemoteBank.interfaces["ctrl1"]:stop()
```

Figura 21: Código do cliente utilizando um Op. *binding*

A Figura 21 ilustra o código que cria um *Operational binding* entre o cliente e o servi-

dor bancário. Para criar um Op. *binding*, nas Linhas 4 a 8 obtém-se o servidor de *Naming* e importa-se a interface registrada com nome de *BankServer*. Esta interface é fornecida como parâmetro para o método *remoteBind* juntamente com a interface *LocalBank*. O método *remoteBind* retorna o componente *remoteBank* que representa o Op. *binding*. As Linhas seguintes, mostram que se pode invocar um método diretamente da interface *iface1* ou através de um *binding* local. Por fim, o método *stop*, da interface remota *ctrl1*, é invocado para desabilitar os serviços fornecidos pela *interface *iface1*.

3.3 JOpenOrb: Implementação Java do Open-ORB

O JOpenOrb é uma implementação em Java da arquitetura Open-ORB que foi projetada por meio da utilização de 21 dos 23 padrões de projeto catalogados por (GAMMA; HELM; JOHNSON, 1995). Os padrões proporcionam aos projetistas de software um vocabulário comum. Dessa forma, usamos os padrões não apenas para ajudar a alavancar ou duplicar projetos bem-sucedidos, mas também para ajudar a comunicarmos o vocabulário e formato comuns para outros desenvolvedores de middleware. Nesse sentido, um projeto que não utiliza os padrões apresenta, em geral, maior dificuldade para transmitir o seu modelo de implementação para outros desenvolvedores. Por exemplo, ao dizer que o *binding* local foi implementado por meio do padrão *Mediator*, explicitamos a estrutura do *binding local* e declaramos, a partir das características do *Mediator*, as características da implementação do *binding*. A lista completa dos padrões de projeto e o objetivo da sua utilização no JOpenOrb está descrito na Tabela 8.

A utilização de padrões também apresenta um componente que restringe o espaço da solução. Ou seja, a utilização dos padrões esta restrita a um cojunto de regras e papeis que limitam o espaço no qual um padrão pode ser aplicado. Dessa maneira, um padrão sugere fortemente a definição de interfaces, classes e relacionamentos aos quais a implementação deve se prender. Não implementar ou ultrapassar o que esta sugerido pelo padrão, representa a quebra da adesão ao padrão e pode indicar a construção de um projeto indesejado.

3.3.1 Arquitetura

Nesta seção iremos detalhar como alguns dos padrões de projeto listados na Tabela 8 foram aplicados para implementar a infra-estrutura básica ilustrada na Figura 1.

Iniciaremos a descrição pelo padrão *Mediator* que desempenha um papel funda-

Tabela 8: Padrões de Projeto utilizados na implementação do JOpenOrb.

Nome do padrão de projeto	**Local de utilização**
Builder	Na construção das mensagens de invocação e retorno.
Factory Method	Na construção dos caminhos de invocação e aceitação de chamadas remotas
Prototype	Na implementação do mecanismo de mudança de estado dos *bindings* locais
Singleton	Na implementação da classe Capsule, como elemento único no ambiente
Adapter	Na construção de uma camada de abstração para a utilização de mais de uma API de comunicação, como Socket e TLI.
Bridge	No desacoplamento de uma parte do *binding* local da hierarquia dos meta-objetos.
Composite	Na composição dos grafos de componentes.
Decorator	Na inserção dinâmica do mecanismo de reflexão aos *bindings* locais.
Facade	Na construção da interface unica do JOpenOrb.
Flyweight	No armazenamento das conexões já estabelecidas(endpoints).
Proxy	Na implementação do *binding* local.
Chain of Responsability	Na implementação do mecanismo de invocação dos pré e pós métodos associados a um *binding* local. Cada *Chain* representa um método a ser executado.
Command	Na implementação do mecanismo de tratamento das invocações remotas. Cada invocação é representada por um comando.
Iterator	Na manutenção e busca dos componentes registrados pelo componente *Capsule*.
Mediator	Na implementação do *binding* local.
Memento	Na captura do estado de um base-objeto sem que para isso seja preciso violar o encapsulamento.
Observer	Na implementação do mecanismo de reflexão responsável por manter a conexão causal.
State	Na implementação da mudança de estado de um *binding* local.
Strategy	Na implementação das estratégias de invocação de um método(Local ou Remota)
Template Method	Na construção dos vários algoritmos de reflexão
Visitor	No percurso do grafo de componentes

mental ao ser aplicado aos elementos que implementam o *binding* local. Na Figura 22 é possível observar que o padrão *Mediator* é constituído pelas classes: *ConcreteBind(ConcreteMediator)*, *BindMediator(Mediator)*, *Port(Colleague)* e *Interface/Receptacle(ConcreteColleague).* O papel da classe *BindMediator* é definir uma interface para a comunicação entre os objetos *Colleagues.* Os *Colleagues* são generalizados por meio da classe *Port* e podem assumir dois tipos: *Interface* e *Receptacle.* A Interface importa métodos fornecidos por um Receptacle, que por sua vez, mantém a referência do objeto que implementa os métodos exportados. A invocação de uma Interface é feita por meio de uma referência para a interface *BindMediator.* Esta interface é implementada pela classe *ConcreteBind* que gerencia as invocações que partem de uma Interface por meio do

método *makeRequest*. Em outras palavras, as *Interfaces* invocam o método *makeRequest* passando como parâmetro o nome do método a ser invocado e os argumentos necessários para a sua invocação. Como resultado, *makeRequest* retorna um objeto do tipo *Object* que representa o retorno da função invocada na classe *Receptacle*(ver Figura 24).

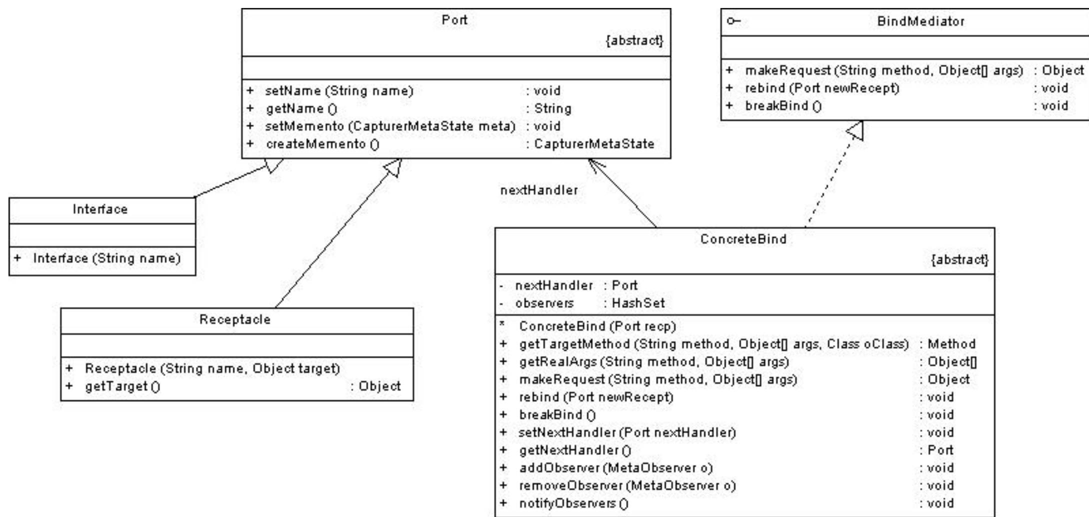

Figura 22: Arquitetura JOpenOrb

A utilização do padrão *Proxy* na implementação do *binding* local é necessária devido as características da linguagem Java que exige a definição de uma classe *Stub* para representar as funcionalidades dos métodos importados. Dessa forma, a Figura 23 ilustra a presença dos elementos que compõem o padrão Proxy, como: BankStub(Proxy), Bank(Subject) e Bank_Impl(RealSubject). Estas três classe estão preenchidas por se tratarem de classes que não pertencem a arquitetura do JOpenOrb, estando presente na figura apenas para ilustrar a relação entre as classes *Interface* e *Receptacle* com os elementos da aplicação.

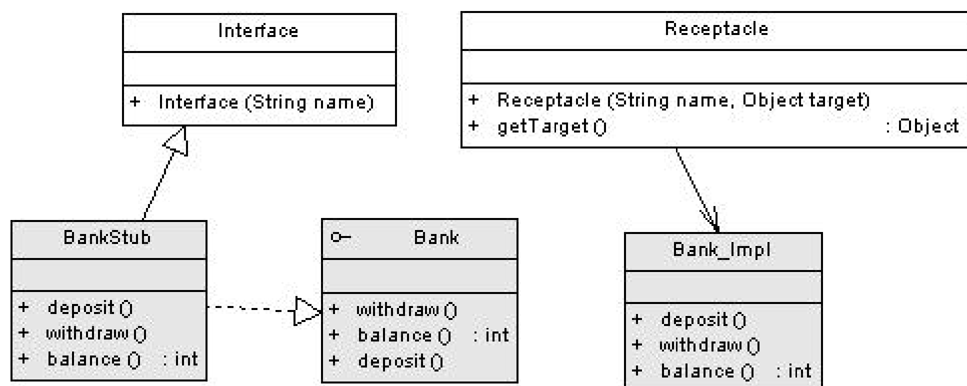

Figura 23: Combinação do Padrão Proxy com Mediator

A utilização de padrões de projeto no desenvolvimento de qualquer aplicação implica na combinação dos padrões utilizados. Segundo (ALEXANDER; ISHIKAWA; SILVERSTEIN, 1977), nenhum padrão é uma entidade isolada. Cada padrão só pode existir no

mundo na medida em que for suportado por outros padrões: os padrões maiores nos quais ele é incorporado, os padrões do mesmo tamanho que o envolvem e os menores que são incorporados a ele. Dessa maneira, a Figura 24 ilustra a combinação dos padrões *Mediator* e *Proxy*. O padrão Proxy envolve o padrão Mediator pela invocação do método *makeRequest* da classe *ConcreteBind(ConcreteMediator)*. O objetivo deste método é invocar a função correspondente ao solicitado na classe Bank_Impl(RealSubject). O atributo *bind* utilizado por meio do *casting* ((InterfaceState)this.state) revela a relação de herança entre o BankStub e a classe *Interface*. Ou seja, na versão Java do Open-ORB, o método IRef não é disponibilizado porque toda interface que importa métodos deve ser declarada, como é mostrado na Figura 24, como uma extensão da classe *Interface* e uma implementação da classe de negócio, no caso, a interface Bank.

```
1  public class BankStub extends Interface implements Bank{
2
3          public BankStub(String name) {
4                  super(name);
5          }
6
7          public void deposit(Float x) {
8                  ((InterfaceState)this.state).bind.makeRequest("
                       deposit", new Object[]{x});
9          }
10
11         public void withdraw(Float x) {
12                 ((InterfaceState)this.state).bind.makeRequest("
                       withdraw", new Object[]{x});
13         }
14
15         public Float balance() {
16                 return (Float)((InterfaceState)this.state).bind.
                       makeRequest("balance", null);
17         }
18         public void setSenha(String senha) {
19                 ((InterfaceState)this.state).bind.makeRequest("
                       setSenha", new String[]{senha});
20         }
21 }
```

Figura 24: Código do cliente utilizando um Op. *binding*

A composição entre padrões também é utilizada para implementar o mecanismo de reflexão do JOpenOrb. Neste caso, a reflexão envolve dois mecanismos complementares: *detecção de mudança* e *captura de estado*. O primeiro mecanismo detecta as mudanças nos objetos base e informa estas ocorrências aos meta-objetos apropriados. Depois disto, o meta-objeto captura o estado do objeto base para identificar as mudanças e realizar as operações necessárias. A Figura 25 mostra a composição desses dois mecanismos, onde as caixas cinzas representam o mecanismo de detecção enquanto que as caixas pontilhadas o mecanismo de captura.

A implementação do mecanismo de detecção é baseada no padrão *Observer*. Este

padrão define uma relação de dependência de umparamuitos que envolvem um objeto base com vários meta-objetos. Deste modo, quando o estado de um dos objetos base muda, todos os meta-objetos associados a este último são automaticamente notificados e atualizados. O objetivo desta decomposição é suportar o reuso dos objetos base sem reusar seus meta-objetos, e vice versa. Conseqüentemente, é possível adicionar um meta-objeto sem modificar seu objeto base ou outro meta-objeto. Estas características são compatíveis com as funcionalidades da conexão causal. Para manter os meta-objetos em estado consistente, os objetos base (Portas, Componentes e Bindings) invocam o método *update* para todas as situações em que é importante notificar os meta-objetos, por exemplo: mudança de nome de porta ou componente, definição de novo *receptacle*, etc.

Por sua vez, a captura de estado é baseada na instanciação do padrão *Memento*. Este padrão externaliza e registra o estado interno de um objeto que posteriormente pode ser recuperado. Para garantir flexibilidade aos objetos base, a classe *MetaState* representa seus dados por meio de uma classe *Map*, permitindo assim que os atributos de cada um dos objetos base sejam representados por meio de itens na estrutura da *HashMap*. Como resultado, a classe *Meta-Object* invoca o método *getState* para obter uma instância da classe *Map* populada por atributos de um respectivo objeto base. Na medida em que o meta-objeto detém o estado de um objeto base, ele pode registrar as mudanças, atualizar o objeto base e restaurá-lo para o seu estado anterior. Isto permite que o meta-objeto retorne o estado da base-objeto assim que a conexão causal for quebrada entre os dois.

3.3.2 Funcionalidades do Nível-Base

As funcionalidades do JOpenOrb são disponibilizadas por uma instância do padrão de projeto *Facade*. Este padrão provê uma interface unificada que abstrai o conhecimento dos níveis inferiores da arquitetura do JOpenOrb. Um exemplo da utilização do JOpenOrb pode ser visto na Figura 26. Nesta figura, após a inicialização do JOpenOrb, cria-se um Stub para o cliente e define-se a instância do servidor que irá receber a implementação do serviço. Na seqüência, o método *createReceptacle* é invocado para criar um *receptacle* que irá fornecer acesso a implementação do servidor *servBank*. Por fim, o *binding* local é definido e o método *deposit* invocado.

Na comparação com o exemplo mostrado na Figura 14, é possível perceber que a diferença entre as APIs do LOpenOrb e JOpenOrb esta na ausência do método *IRef* e na presença do método *createReceptacle*. O método *IRef* foi removido em decorrência

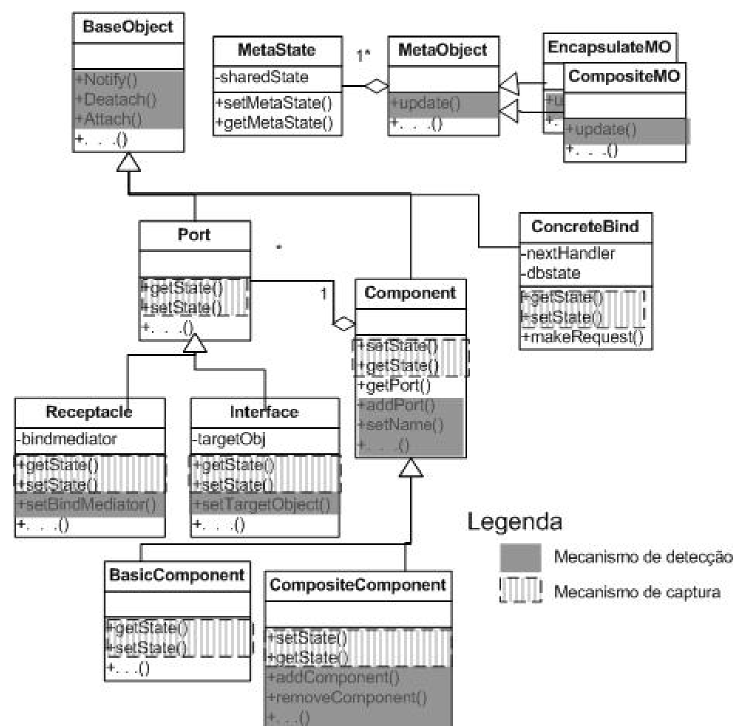

Figura 25: Padrões de projeto na implementação da reflexão computacional

das restrições dinâmicas da linguagem Java que não permitem a definição dinâmica de métodos. Por este motivo, criou-se a distinção entre Interface e Receptacle, de modo que a Interface é definida apenas pela definição do *Stub*, enquanto o *receptacle* exige a invocação do método *createReceptacle*.

```
1  JOpenOrb.init();
2  BankStub client = new BankStub("IBankClient");
3  BankServer servBank = new BankServer(new Float(1000));
4
5  Receptacle RBank = JOpenOrb.createReceptacle("BankRec",servBank);
6  ConcreteBind bd = (ConcreteBind) JOpenOrb.localbind(client,RBank);
7  client.deposit(50);
```

Figura 26: Exemplo de utilização do JOpenOrb

3.3.3 Funcionalidades do Meta-nível

Assim como no LOpenOrb, os meta-modelos do JOpenOrb são fornecidos por um componentes composto formado por duas interfaces: *EncapsulationMetaModel* e *CompositionMetaModel*. A primeira interface exporta o método *encapsulation* que recebe como parâmetro a referência de uma interface ou componente e retorna um *encapsulation* meta-objeto. A Figura 27 mostra um exemplo da utilização do *encapsulation* meta-modelo. A única diferença existente entre este exemplo e o mostrado na Figura 15, é a forma como

os parâmetros do tipo tabela, na API LOpenOrb, são passados para os métodos. Na implementação Java, cada tipo de tabela foi transformada numa classe. Isto pode ser visto na Figura 27 onde a classe *MethodArg* representa a tabela formada pelos valores "regvenda" e *bindctrl*.

```
1 LogServer servLog = new LogServer();
2 Receptacle RLog = OpenOrb.createReceptacle("BankLog",servLog);
3 ConcreteBind bindctrl = (ConcreteBind) OpenOrb.localbindOneWay(null,
      RLog);
4
5 MetaObject meta = OpenOrb.metamodel.encapsulation(client);
6 meta.addPreMethod("deposit",new MethodArg("regvenda", bindctrl));
```

Figura 27: Exemplo de utilização do JOpenOrb

Os dois exemplos descritos acima mostram que apesar da diferença entre as linguagens Java e Lua, as APIs do LOpenOrb e JOpenOrb apresentam poucas diferenças. Em outras palavras, a diferença resume-se a utilização do conceito de *Receptacle* para superar as limitações dinâmicas de Java e pela introdução de algumas novas classes que são usadas única e exclusivamente como forma de compatibilizar o número de parâmetros com a versão em Lua. Estas diferenças não modificam, no entanto, a seqüência de combinações realizadas para implementar a infra-estrutura de comunicação mostrada na Seção 3.2.4. Por este motivo, optamos por não repetir a descrição da infra-estrutura de comunicação do JOpenOrb, uma vez que se trataria de uma repetição do que já foi descrito na Seção 3.2.4.

4 Estratégias de Aspectização

A customização de uma plataforma de middleware consiste na composição das funcionalidades requeridas pela aplicação que executa no topo da plataforma. Dessa maneira, a customização é um processo que compreende a adição de novas funcionalidades assim como a remoção de funcionalidades desnecessárias para a aplicação em execução. No capítulo anterior vimos como a customização pode ser alcançada por meio da reflexão para inserir ou remover componente(s) desejado(s) ou indesejado(s). No entanto, neste capítulo iremos mostrar como a customização pode ser alcançada por meio da POA.

Dessa forma, a seção 4.1 descreve a importância da separação de conceitos nas plataformas de middleware e ilustra como essa separação pode beneficiar o desenvolvimento e reuso destas plataformas. Na seqüência, a seção 4.2 delineia a arquitetura do Aspect Open-ORB, uma versão orientada a aspectos do Open-ORB, e descreve como seus componentes interagem para a formação da plataforma customizada. Por fim, as seções 4.3 e 4.4 descrevem a implementação do Aspect Open-ORB usando a estratégia interpretada (com Lua e AspectLua) e compilada (com Java e AspectJ).

4.1 Separação de conceitos em plataformas de Middleware

Os middlewares tradicionais como CORBA(Object Management Group, 2001), COM+ (MORGENTHAL, 1999) e Java RMI(WOLLRATH; RIGGS; WALDO, 1996), foram projetados e implementados para prover uma grande quantidade de funcionalidades para múltiplos domínios de aplicação. Esta diversa gama de funcionalidades aumentou a atração pelo uso destes middlewares mas, por outro lado, contribuiu para o acréscimo de funcionalidades específicas que, às vezes, são utilizadas por pouquíssimas aplicações. Segundo (ZHANG; JACOBSEN, 2004), a aplicação de plataformas de middleware em mui-

tas aplicações emergentes, tais como roteadores[1], sistemas de controle de combate(KELLY; DIPALMA, 2001) e dispositivos *wireless* (CARON; HERSCHER; O'CONNOR,) tem provocado uma crescente busca por novos requisitos o que está gerando problemas para essas plataformas. O primeiro problema apresentado por (ZHANG; JACOBSEN, 2004) refere-se a complexidade das plataformas de middleware que tem crescido drasticamente nos últimos anos em decorrência do constante acréscimo de novas funcionalidades. Para comprovar este crescimento, a Figura 28 ilustra o número de classes do JacORB[2], uma implementação, *open source*, em Java de CORBA que triplicou de tamanho num período de quatro anos.

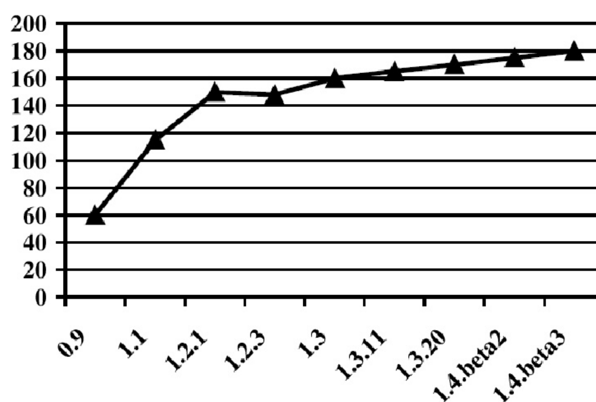

Figura 28: Evolução do JacORB em relação ao número de classes vs. suas versões

Segundo (ZHANG; JACOBSEN, 2004), além de aumentar o número de módulos, as plataformas de middlewares também requerem uma maior quantidade de recursos de CPU, memória. Como exemplo, o ORBacus Java[3], uma implementação comercial de CORBA, requer aproximadamente 10MB de memória. Uma implementação C, como é o caso do ORBit[4] requer 2MB de memória. Estes requisitos de memória dificultam a utilização de plataformas de middleware em sistemas com restrições de recursos, como em dispositivos móveis e sistemas embutidos. O problema é ainda maior para os casos dos dispositivos móveis onde não existe uma padronização das plataformas de software. Ou seja, muitos dos dispositivos usam o sistema operacional Symbian[5], enquanto outros PDA´s e organizadores pessoais usam versões de Palm OS, RIM OS, Microsoft PocketPC, etc. Esta diversidade de plataformas de software cria um grande número de configurações

[1]Cisco ONS 15454 plataforma de transporte ótico, usa CORBA para negociar comunicação entre o software de gerência e os *drivers* de hardware.
[2]JacORB http://www.jacorb.org
[3]ORBacus: http://www.orbacus.com
[4]ORBit: http://gnome.org/projects/ORBit2/
[5]http://www.symbian.com

que precisão ser suportadas. Em certos casos[6], o numero de versões pode alcançar 700 plataformas alvo que variam de acordo com o dispositivo e suas características. Para estes casos, plataformas de middleware são tipicamente re-projetadas e implementadas individualmente, de modo a permitir que todas as suas funcionalidades sejam direcionadas para as necessidades da plataforma alvo.

O grande número de plataformas alvo torna essa abordagem praticamente inviável, uma vez que um mesmo código pode estar distribuído em várias implementações e a manutenção desse código pode gerar graves problemas de confiabilidade.

A solução, portanto, para muitos autores (BERGMANS; AKSIT, 2000; ZHANG; JACOBSEN, 2004) consiste na utilização da POA. Com o uso da POA, pretende-se separar as funcionalidades básicas das funcionalidades específicas(ortogonais) e através de um processo de *weaving* juntar essas funcionalidades para atender aos requisitos da aplicação.

(BERGMANS; AKSIT, 2000) resume os problemas da utilização da POA nas plataformas de middleware através da Figura 29. Esta Figura ilustra um exemplo de separação de conceitos: primeiro uma separação em camadas, onde cada camada possui diferentes objetos; segundo, uma separação vertical em aspectos, que atravessa os limites das camadas e objetos. Dessa forma, o que normalmente é feito pelos fornecedores de plataforma de middleware é distribuir o código relacionado a um aspecto sobre o código que implementa os objetos distribuídos nas diferentes camadas. O resultado dessa mistura é o chamado código entrelaçado que pode ser implementado em diferentes camadas, como é o caso de aplicações com requisitos de QoS, que aplicam código de controle em várias camadas do middleware. Dessa forma, o desafio da utilização da POA está em conseguir separar os aspectos sem, no entanto, distribuir código entrelaçado pelas camadas.

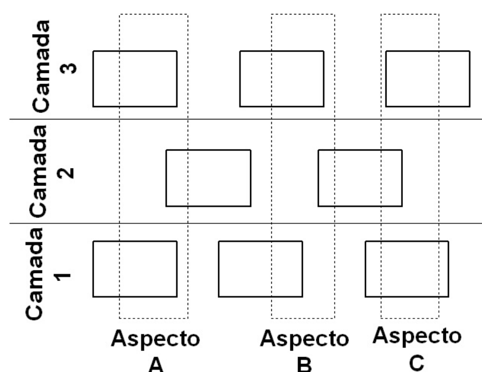

Figura 29: Ilustração da combinação dos aspectos com as camadas

Neste trabalho utilizamos POA para customizar a infra-estrutura básica do Open-

[6]http://www.gameloft.com/press_release.php?press1=333

ORB. Entretanto, no sentido de avaliar diferentes cenários, foram definimos duas estratégias de customização. A primeira é baseada nas características dinâmicas da linguagem Lua e no conjunto de elementos fornecidos pelo AspectLua e LOpenOrb. Esta estratégia baseia-se na idéia de que as funcionalidades da aplicação são definidas pelas funções invocadas pelo código do desenvolvedor. Dessa forma, AspectLua deve controlar os métodos invocados e verificar suas dependências, de modo que estas sejam carregadas e executadas de acordo com a necessidade. A segunda estratégia é baseada nas características compiladas da linguagem Java e em como AspectJ e JOpenOrb podem gerar um modelo adequado para reduzir o número de configurações para as plataformas alvo. A distinção entre as estratégias é motivada pelo diferente suporte provido por cada uma das linguagens frente a diversidade de plataformas alvo(Figura 30). Na linguagem Lua, cada plataforma possui um interpretador próprio que provê a mesma API. Isto permite desenvolver um único LOpenOrb para todas as plataformas, lidando apenas com restrições de memória. Por esse motivo, a estratégia de aspectização do LOpenOrb irá focar na questão da inserção dinâmica de novas funcionalidades, buscando assim otimizar o uso dos recursos.

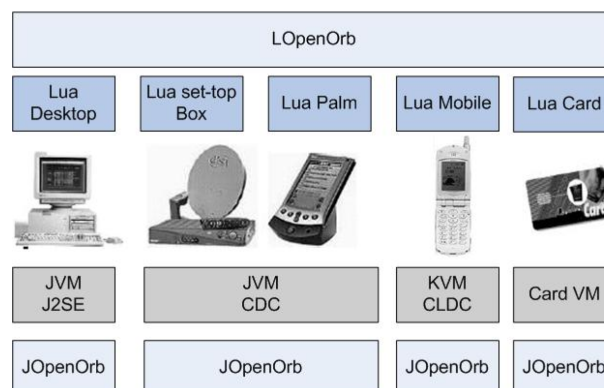

Figura 30: Plataformas de software para as versões Lua e Java

Por outro lado, a solução Java fornece um conjunto diferenciado de APIs para cada plataforma alvo. Isto ocorre porque o tamanho da Maquina Virtual Java(JVM) para um aplicação desktop é totalmente incompatível com as limitações de um dispositivo móvel. Esta distorção levou a uma redução da JVM para a então chamada Kilobyte Virtual Machine (KVM), por meio da eliminação de funcionalidades da API de Java, como reflexão, carga dinâmica de classes, etc. Sem poder contar com estas funcionalidades, importantes recursos da plataforma J2SE não podem mais ser utilizadas em dispositivos móveis, como, por exemplo, chamada de métodos remotos (RMI) e serialização de objetos. Aplicativos que utilizem CLDC *(Connected Limited Device Configuration)*, que é uma especificação

voltada para dispositivos extremamente restritos em termos de memória, largura de banda e segurança [Rig01], não podem requerer recursos avançados com capacidades reflexivas. Por exemplo, a implementação proposta do JOpenOrb é inadequada para dispositivos móveis com baixos recursos que utilizam J2ME com a configuração CLDC, uma vez que utiliza vários recursos que não são suportado por essa configuração, tal como a interface *Iterator*,o pacota java.lang.reflection, etc.

Para endereçar este problema, iremos utilizar a POA para configurar o JOpenOrb de acordo com a plataforma alvo, evitando assim, a definição e duplicação de múltiplas versões de um mesmo middleware.

4.2 Arquitetura do Aspect Open-ORB

A Figura 31 ilustra a arquitetura de uma aplicação desenvolvida sobre o Aspect Open-ORB. Primeiro são definidos os aspectos relativos ao ORB(Configuração dos elementos e dependências), seguido pela definição do código funcional da aplicação e por último pelos aspectos da aplicação(Segurança, Tolerância a falhas,etc). Estes três elementos são combinados pelo AspectLua ou AspectJ para, em tempo de execução ou compilação, gerar a implementação final baseada nos requisitos da plataforma alvo.

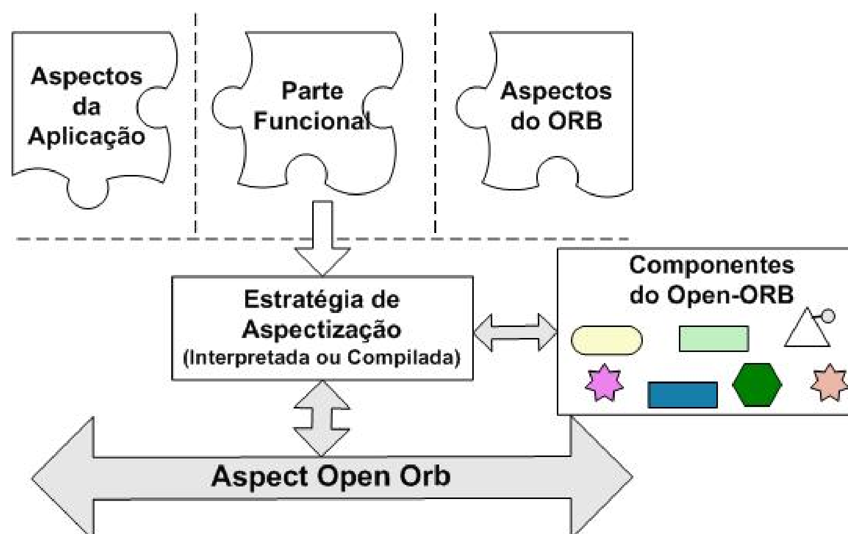

Figura 31: Arquitetura de uma aplicação desenvolvida sobre o Aspect Open-ORB

Portanto, o objetivo do Aspect Open-ORB é permitir a customização das três camadas do middleware, onde na camada de infra-estrutura básica utiliza-se POA e nas duas seguintes reflexão computacional. No entanto, apesar de descrever uma arquitetura comum para Aspect Open-ORB, a implementação dessa arquitetura é diferenciada para

cada um dos middlewares apresentados no Capítulo 3. As diferenças entre as implementações do Aspect Open-ORB para o LOpenOrb e JOpenOrb serão descritas nas duas próximas seções.

Tabela 9: Relação de dependência entre os módulos do LOpenORB.

Funcionalidade	**Designator**	**Dependências**
Criação de interfaces e *bindings* locais	LOpenORB.*IRef e LOpenORB.localBind*	7 e 8
Criação de componente	LOpenORB.Component	9
Criação de componente composto	LOpenORB.Composite	10
Criação de meta-objeto encapsulation	LOpenORB.metamodel. encapsulation	11
Criação de meta-objeto composition	LOpenORB.metamodel. composition	12
Utilização do componente *Capsule*	LOpenORB.localcapsule.*	19,20,21
Recebendo invocações remotas	LOpenORB.localcapsule.serve	13,14,15,17,18,22
Invocando métodos remotos	LOpenORB.localcapsule. getRemoteInterface	13,14,15,16
Criando Operational binding	LOpenORB.remoteBind	23
Criando Signal binding	LOpenORB.sigBind	24
Criando Stream binding	LOpenORB.streamBind	25

4.3 Estratégia para aspectização do LOpenOrb

Como ilustrado na Figura 30, as características interpretadas da linguagem Lua fornecem um mesmo conjunto de APIs para várias plataformas o que facilita a portabilidade do código do middleware. Além da portabilidade, o código interpretado permite controlar, em tempo de execução, os elementos que serão necessários para a implementação das funcionalidades. Isto otimiza o uso dos recursos e ainda permite adicionar funcionalidades dinamicamente. Portanto, nesta seção iremos descrever como utilizamos a POA para personalizar o LOpenOrb e otimizar a utilização de recursos.

O processo de personalização do LOpenOrb exige a aplicação de dois conjuntos de aspectos. O primeiro conjunto irá interceptar a invocação do método *LOpenORB.init* e o segundo irá atuar sobre os elementos definidos na Tabela 9. Esta Tabela descreve as dependências de cada um dos conjuntos de funcionalidades do Middleware. Em cada linha está representado um aspecto com um determinado *designator* e uma lista de números[7] que determinam as dependências entre os módulos. A primeira linha, por exemplo, indica que quando o método *LOpenORB.CIRef* for invocado, o *advice* relativo a esse aspecto

[7] Cada número corresponde a linha em que o módulo é descrito na Figura 13. Exe.: 14 equivale ao módulo *lopenorb_msg*

deve ser invocado para antes carregar as dependências desse método, no caso, os módulos *lopenorb_lbind* e *lopenorb_interfaces*. Essa abordagem permite limitar o código da Figura 13 a apenas as cinco primeiras linhas.

```
1  asp = Aspect:new()
2
3  function myInit(arg)
4          LOpenOrb.hostname = arg[2]
5          LOpenOrb.port = arg[3]
6  end
7  asp:aspect({name = 'skipInit'} , {name = 'skip', designator = 'call'
       , list = {'LOpenOrb.init'}}, {type ='around', action = myInit})
8
9  tblfiles = {}
10 tblfiles[7] = "lopenorb_lbind.lua"
11 tblfiles[8] = "lopenorb_interfaces.lua"
12 tblfiles[9] = "lopenorb_component.lua"
13   . . .
14
15 tblconf = {}
16 table.insert(tblconf,{key = "LOpenOrb.*IRef", dep = {7,8} }
17 table.insert(tblconf,{key = "LOpenOrb.Component", dep = {9} }
18   . . .
19 table.insert(tblconf,{key = "LOpenOrb.localcapsule.serve", dep
       = {13,14,15,17,18,22,19,20,21} }
20 table.insert(tblconf,{key = "LOpenOrb.localcapsule.
       getRemoteInterface", dep = {13,14,15,16,19,20,21} }
21 table.insert(tblconf,{key = "LOpenOrb.localcapsule.*", dep
       = {19,20,21} }
22  . . .
23
24 function loadfiles( ... )
25       local funcname = table.remove(arg,table.getn(arg))
26       loca contload = 0
27       for k,idxfile in ipairs(searchKey(funcname)) do
28               local namefile = rawget(tblfiles,idxfile)
29               if (namefile ~= nil)then
30                       dofile(namefile)
31                       deletefile(idxfile)
32                       contload = contload + 1
33               end
34       end
35
36       if contload > 0 then
37         metafunc = LuaMOP:getInstance(funcname)
38         func = metafunc:getFunction()
39         return func(unpack(arg))
40       end
41 end
42
43 for k,conf in ipairs(tblconf) do
44         asp:aspect({name = 'ConfORB'} , {name = 'laodfiles',
               designator = 'callone', list = {conf.key}}, {type ='
               around', action = loadfiles})
45 end
```

Figura 32: Configurando os aspectos do middleware

A Figura 32 mostra o código responsável pela personalização do middleware. As linhas 3 a 6 mostram a definição da função *myInit* que, através do aspecto definido na linha 7, irá atuar na função *LOpenORB.init* para evitar que todos os módulos do LOpenORB sejam carregados.

A segunda parte da Figura 32 consiste na definição das regras mostradas na Tabela 9. As linhas 9 a 12 mostram uma parte do processo de criação da lista de módulos (*tblfiles*)

indexada pela numeração apresentada na Figura 13. Na sequência, os elementos definidos na Tabela 9 são inseridos na lista *tblconf* através dos campos *Key*, que contém o nome do aspecto, e *dep* que armazena os módulos necessários para invocação do método. As linhas 43 a 45 mostram a definição de cada elemento da lista *tblconf* como um aspecto. Cada um desses aspectos invocará uma única vez(*callone*) o método *loadfiles* quando o *pointcut* definido por *tblconf.Key* for alcançado. O objetivo da função *loadfiles* é a partir do nome da função, obtido na linha 25, carregar, via o laço que retorna a variável *idxfile*, cada um dos módulos necessários para a invocação do método. A carga do módulo é feita pelo método *dofile*. Para controlar os módulos que já foram carregados, a função *deletefile* é invocada, logo após a carga, para remover da lista *tblfiles* o respectivo nome do módulo. A variável *contload* armazena o número de módulos que já foram carregados para determinar se a função alvo será executada. Isto é fundamentalmente importante para evitar que no caso dos *pointcuts LOpenOrb.localcapsule.getRemoteInterface* e *LOpenOrb.localcapsule.** não haja duplicação de invocação do método alvo. O LuaMOP é utilizado para obter a função alvo que, neste momento, já está definida. A função alvo é invocada na linha 39, recebendo como parâmetro os argumentos da invocação original(arg).

A utilização do código da Figura 32 pode ser combinada com os códigos mostrados nas Figuras 14 ou 21 para construir um middleware personalizado que disponibilize apenas o que for realmente utilizado.

4.4 Estratégia para aspectização do JOpenOrb

A estratégia de aspectização do JOpenOrb difere da utilizada pelo LOpenOrb porque as duas implementações utilizam linguagens e metodologias diferentes para a construção do mesmo middleware. Como o JOpenOrb foi implementado em Java através da utilização de padrões de projeto, estes padrões serão utilizados para inserir e remover as funcionalidades do middleware. Com base nessa idéia, a estratégia de aspectização JOpenOrb irá utilizar-se dos aspectos abstrados definidos por (HANNEMANN; KICZALES, 2002) para inserir e remover as funcionalidades requeridas pelo JOpenOrb. A Tabela 10 mostra a relação que existe entre as funcionalidades do JOpenOrb e os padrões de projeto.

Um exemplo da utilização de aspectos abstratos pode ser visto na Figura 33. O aspecto *ChainOfResponsibilityProtocol* define as interfaces *Handler* e *Request* além de um conjunto de métodos que serão utilizados pelo aspecto concreto. Esta abordagem permite a introdução do código do padrão em vários locais da arquitetura do midleware,

Tabela 10: Relação entre funções do JOpenORB e padrões de projeto.

Funcionalidade	**Designator**	**Padrão de Projeto**
Criação de interfaces e *bindings* locais	LOpenORB.*IRef e LOpenORB.localBind*	Mediator, Proxy, Facade e Singleton
Criação de componente	LOpenORB.Component	Memento
Criação de componente composto	LOpenORB.Composite	Memento e Composite
Criação de meta-objeto encapsulation	LOpenORB.metamodel. encapsulation	Decorator, Template Method, State, Prototype, Bridge e Chain of Responsability
Criação de meta-objeto composition	LOpenORB.metamodel. composition	Visitor e Observer
Utilização do componente *Capsule*	LOpenORB.localcapsule.*	Iterator e Factory Method
Recebendo invocações remotas	LOpenORB.localcapsule.serve	Builder, Adapter e Command
Invocando métodos remotos	LOpenORB.localcapsule. getRemoteInterface	Strategy

sem que para isso seja necessário inserir código duplicado. Cada local deve definir um aspecto concreto como é mostrado na Figura 34. Neste caso, o aspecto *ChainAspect* define a relação entre as classes *ConcreteMetaInvocationHandler* e *MetaObjectEncapsu* e as interfaces definidas na classe abstrata. Na seqüência, os métodos *acceptRequest* e *handleRequest* são definidos com as funcionalidades apropriadas aos locais onde serão aplicado os aspectos.

A estratégia de aspectização utilizada neste trabalho afeta igualmente as funcionalidades reflexivas. Dessa forma, o espalhamento dos conceitos relativos a detecção e captura de estado ilustrada na Figura 25 também foram aspectizados. Ou seja, apesar do método que implementa a conexão causal ser definido na classe *BaseObject*, a invocação deste método está espalhada por vários métodos na hierarquia dos objetos base. Portanto, o Aspect OpenORB define um conjunto de aspectos que modulariza o código relativo ao mecanismo de detecção, evitando assim que os módulos que implementam as abstrações de alto nível tenha conhecimento da existência de elementos do meta-nível.

A Figura 35 ilustra o aspecto abstrato utilizado para implementar o mecanismo de detecção. Este aspecto define dois métodos(*addTrigger* e *delTrigger*) que são utilizados para associar ou desassociar um base-objeto de um meta-objeto. Os base-objetos são representados pelo parâmetro *baseObj* que é passado para ambos os métodos, enquanto que a referência dos meta-objetos é passada pelo parâmetro *metaObject*. Neste caso, o meta-objeto é representado por um *ConcreteBind* o que implica que este pode estar

```
1  public abstract aspect ChainOfResponsibilityProtocol {
2    protected interface Handler {}
3    protected interface Request {}
4    private WeakHashMap successors = new WeakHashMap();
5    protected void receiveRequest(Handler handler, Request request) {
6          if (handler.acceptRequest(request)) {
7              handler.handleRequest(request);
8          } else {
9              Handler successor = getSuccessor(handler);
10             if (successor == null) {
11                         throw new ChainOfResponsibilityException("
                              request unhandled (end of chain reached)\
                              n");
12             } else {   receiveRequest(successor, request);}}
13         }
14     public boolean Handler.acceptRequest(Request request) {
15             return false;
16     }
17     public void Handler.handleRequest(Request request) {}
18     public void setSuccessor(Handler handler, Handler successor) {
19                 successors.put(handler, successor);
20         }
21     public Handler getSuccessor(Handler handler) {
22             return ((Handler) successors.get(handler));
23         }
24 }
```

Figura 33: Aspecto abstrato para a definição do padrão Chain of Responsability

```
1  public aspect ChainAspect extends ChainOfResponsibilityProtocol {
2
3      declare parents: ConcreteMetaInvocationHandler implements
           Handler;
4      declare parents: MetaObjectEncapsu  implements Request;
5      public boolean ConcreteMetaInvocationHandler.acceptRequest(
           Request request) {
6          return true;
7      }
8      public void ConcreteMetaInvocationHandler.handleRequest(Request
           request, Object[] args) {
9                  mediator.makeRequest(mediator.getNextHandler().
                       getName(), args);
10     }
11 }
```

Figura 34: Definição do aspecto concreto para o padrão Chain of Responsability

definido remotamente. Além dos métodos, o aspecto também define um *pointcut* abstrato (*subjectChange*) e seu respectivo *advice*. Este *advice* intercepta as invocações e as desvia para o respectivo meta-objeto associado. A definição dos pontos de interceptação são definidas no aspecto concreto ilustrado na Figura 36. Nesta figura, o *pointcut* abstrato é concretizado pela definição de vários pontos de junção.

A utilização da abordagem de separação de padrões sugerida por (HANNEMANN; KICZALES, 2002) em conjunto com o conceito de aspecto abstrato fornecido por AspectJ facilita o processo de identificação e inserção dos aspecto. Por outro lado, as características compiladas do AspectJ limitam a utilização dessa estratégia para a aspectização da arquitetura apenas para requisitos pré-definidos.

```
1  public abstract aspect ObserverProtocol {
2
3      private static HashMap triggerList = new HashMap();
4
5          public static void addTrigger(BaseObject baseObj, String
               operation, ConcreteBind metaObject){
6              HashMap methods = (HashMap) triggerList.get(baseObj);
7              if (methods== null )
8                   methods = new HashMap();
9
10              methods.put(operation, metaObject);
11              triggerList.put(baseObj, methods);
12         }
13
14         public static void delTrigger(BaseObject baseObj){
15                 triggerList.remove(baseObj);
16         }
17
18
19     protected abstract pointcut subjectChange(BaseObject baseObj);
20
21     after(BaseObject baseObj): subjectChange(baseObj) {
22         Object[] argss = thisJoinPoint.getArgs();
23         Signature signature = (Signature)thisJoinPoint.getSignature
               ();
24         String operation = signature.getName();
25
26         HashMap methods = (HashMap) triggerList.get(baseObj);
27         if (methods != null){
28                 ConcreteBind metaObject = (ConcreteBind) methods.get
                       (operation);
29                 bind.makeRequest(operation, argss);
30             }
31         }
32 }
```

Figura 35: Aspecto abstrato usado para implementar o mecanismo de detecção

```
1  public aspect ObserverAspect extends ObserverProtocol {
2      protected pointcut subjectChange(BaseObject baseObj):
3          (call(public void Component.setName(String))
4                  || call (public void Component.addPort(Port))
5                  || call(public void ConcreteBind.rebind(Port))
6                  || call(* Port.setName(String))
7                  || call(* Receptacle.setTarget(Object))
8                  || call(* Interface.setBindMediator(BindMediator))
9                  . . .
10                 || call(* ConcreteBind.setNextHandler(Port)))
11                 && target(baseObj);
12 }
```

Figura 36: Aspecto concreto usado para definir os pontos de interceptação

5 *Avaliação*

Como ilustrado nos capítulos anteriores, o principal propósito de uma plataforma de middleware é simplificar a construção de sistemas distribuídos a partir de um conjunto de requisitos. Estes requisitos descrevem como as funcionalidades devem se comportar para determinadas situações. Além disso, requisitos tratam de propriedades emergentes do sistema que são relacionadas à aplicação, tais como tempo de resposta e outras qualidades. Portanto, a capacidade de customização é o principal atributo que suporta a diversidade de requisitos, uma vez que representa a capacidade de personalizar sem que se introduzam inconsistências. Dessa forma, a customização é motivada pela necessidade de acomodar adequadamente a diversidade de funcionalidades com a disponibilidade do serviço, evitando assim que a diversidade influencie na qualidade do serviço. Por conseqüência, desempenho é um dos fatores que mede o nível de inconsistência de uma customização, uma vez que tradicionalmente desempenho entra em conflito com a utilização de soluções genéricas. Outro aspecto que limita a customização é a modularidade da solução. Ou seja, o nível de dependência entre os módulos contribui para uma melhor ou pior capacidade de customização, uma vez que a alta dependência entre módulos tende a limitar o poder de customização.

No sentido de avaliar o poder de customização das nossas soluções e como elas afetam o desempenho e a modularidade, nossa avaliação será divida em duas partes. A seção 5.1 compara como a solução aspectizada evoluiu em relação à versão não aspectizada enquanto que a seção 5.2 trata de como os aspectos relativos a cada linguagem de programação influenciaram nas duas soluções.

5.1 Comparando as soluções aspectizadas versus não aspectizadas

Esta seção compara as soluções LOpenORB com Lua Aspect Open-ORB e JOpenOrb com Java Aspect Open-ORB para verificar como as versões aspectizadas evoluíram em

relação as versões não aspectizadas.

5.1.1 Comparando LOpenOrb com Aspect Open-ORB

Nesta seção avaliamos as diferenças existentes entre as duas versões Lua do OpenORB. A primeira versão suporta customização da camada de infra-estrutura de componentes e aplicação, enquanto a segunda suporta a customização nas três camadas. Inicialmente analisamos se a utilização da POA cumpre com os requisitos exigidos pelas aplicações que possuem restrições de recursos. Dessa forma, avaliamos o desempenho e a memória consumida pelas duas soluções. Para isso são realizados dois conjuntos de testes. O primeiro compara a diferença de desempenho entre os elementos básicos(*bindings* locais, componentes, etc.) e em seguida a diferença existente na infra-estrutura de componentes e os elementos de reflexão. O segundo teste verifica se o consumo de memória foi otimizado pela utilização da POA.

Todos os experimentos foram conduzidos num PC Duron 1.6MHz com 512MB de RAM, executando Linux-Mandrake 9.2. O LOpenORB foi testado utilizando o Lua 5.1. Todos os testes foram realizados com os mesmos conjunto de elementos, ou seja, tudo que foi implementado em LOpenORB(objetos, métodos exportados, componentes, etc) também foi exatamente utilizado em Aspect OpenORB.

A Tabela 11 mostra a comparação entre os tempos de criação de cada um dos elementos básicos do LOpenORB. Nesta tabela é mostrada o tempo de execução no LOpenORB, o tempo correspondente no Aspect Open-ORB para uma primeira invocação, ou seja, quando o método *loadfiles* da Figura 32 é invocado e o tempo das subseqüentes invocações. Como resultado, percebemos que o tempo da primeira invocação está relacionado ao tamanho do módulo e ao número de módulos que precisaram ser carregados. A pequena diferença entre os tempos das invocações subseqüentes correspondem a influência do AspectLua na invocação de cada método.

Além de comparar o tempo de execução de cada um dos métodos foram observados também os recursos de memória requisitados por cada aplicação. Como base de comparação, temos que o ambiente de execução Lua consome 564 Kbytes sem qualquer código do usuário. A carga do módulo *AspectLua.lua* faz com que o ambiente Lua passe a ocupar 680 Kbytes. O gráfico 37 mostra a quantidade de memória utilizada para cada funcionalidade do ORB. Como o LOpenORB carrega todos os seus módulos de uma única vez, seu valor fica constante em 1075 Kbytes. Por sua vez, o Aspect Open-ORB apresenta um aumento gradual na memória consumida, compatível com um número de módulos

Tipo de teste	LOpenORB	Aspect Open-ORB: (1)Invocação	Aspect Open-ORB: (2..n)Invocação
Criando uma interface (IRef)	45.9	1825.96	46.5
Criando componente	7.08	347.08	7.27
Criando componente composto	57.1	549.25	60.04
Criando um *binding* local	17.5	45.13	17.08
Executando através de um *binding* local	19.6	19.89	19.89
Executando através de um *binding* local com meta objeto	21.75	780.48	22.15
Inserção de componente em um grafo de componentes	17.84	697.09	17.98

Tabela 11: Resultados dos testes comparativos entre o LOpenORB e o Aspect Open-ORB(Tempo em μs)

carregados.

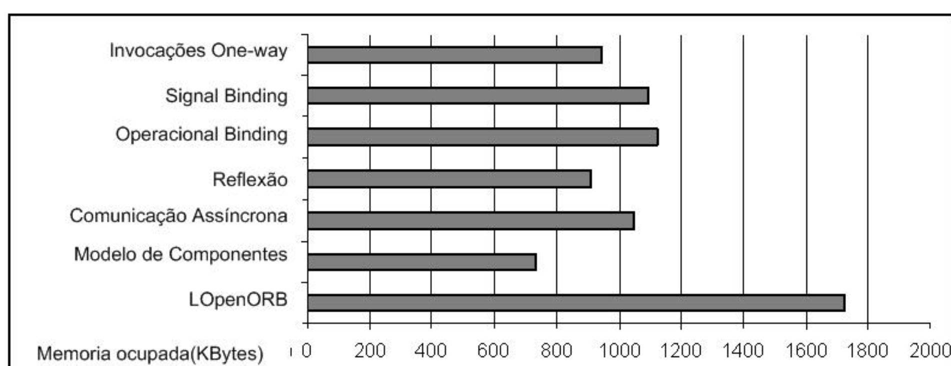

Figura 37: Gráfico de memória utilizada

Para relacionar o tempo de execução de uma aplicação com sua memória consumida, executamos os programas mostrados nas Figuras 14 e 21. O código da Figura 14 quando executado sem o Aspect Open-ORB, demorou 19.74ms e consumiu 1100 Kbytes. Com a utilização do Aspect Open-ORB, os valores diminuíram para 3.94ms e um consumo de 710 Kbyte. O código da Figura 21 foi executado sem o Aspect Open-ORB em 61.57ms e consumiu 1328 Kbytes. Com a utilização do Aspect Open-ORB o tempo de execução subiu para 67.39ms e obteve um consumo de 1302 Kbytes.

Em resumo, a utilização do Aspect Open-ORB para o código da Figura 14 resultou tanto numa redução de memória como no tempo de execução. Estes dois fatores são resultados do menor número de módulos que precisam ser carregados. Por sua vez, a utilização do Aspect Open-ORB no código da Figura 21 produziu consumos semelhantes de memória, mas uma diferença de 6ms em relação ao tempo de execução. Esta diferença está relacionada ao aumento no tempo de execução de uma invocação remota, que passou de 1.31ms para 1.44ms quando ambos os lados(cliente e servidor) utilizam o Aspect Open-ORB.

5.1.2 Comparando JOpenOrb com Aspect Open-ORB

Na comparação entre as duas versões Java do OpenORB utilizamos uma abordagem diferente, onde inicialmente verificamos como a modularidade da infra-estrutura básica do OpenORB é afetada pela utilização da POA. Na seqüência avaliamos quais os impactos dessa modularização no desempenho do ORB. A principal motivação para essa abordagem está diretamente relacionada a principal motivação do JOpenORB que não é prover um mecanismo de customização dinâmica, mas sim uma infra-estrutura básica modular que permita ser estaticamente customizada pela introdução ou remoção de funcionalidades que satisfaçam os requisitos da plataforma alvo. Portanto, neste contexto, a modularidade é um fator preponderante que deve ser avaliado.

Neste sentido, nossa avaliação comparou as duas versões Java do OpenORB, uma orientada a objetos (JOpenORB) e outra orientada a aspectos (Aspect OpenORB). Para identificar as diferenças entre as duas versões utilizamos um conjunto de métricas(SANT'ANNA et al., 2003; GARCIA et al., 2005) que capturaram importantes características de modularidade do sistema. Este conjunto de métricas é distribuído entre métricas de separação de conceitos (SoC), acoplamento, coesão e tamanho. As métricas de acoplamento, coesão e tamanho são extensões de tradicionais métricas OO adaptadas para suportar a comparação entre soluções Java e AspectJ. Por sua vez, as métricas de separação de conceitos capturam o grau que um simples conceito do sistema é mapeada para o modelo de componentes (classes e aspectos), operações (métodos e advices), e linhas de código. A Figura 38 ilustra brevemente cada uma das métricas utilizadas e as associa com os atributos medidos por cada uma. Neste trabalho optamos por este conjunto de métricas por considerá-lo extensivamente usado em vários estudos (GODIL; JACOBSEN, 2005; CACHO et al., 2006b, 2006a) e por ele ser um bom indicador da qualidade de cada uma das implementações.

Para realizarmos a coleta dos dados relativos aos valores das métricas, utilizamos uma ferramenta (FIGUEIREDO et al., 2005) que automaticamente obtém todas as métricas, exceto as métricas de separação de conceitos (CDC, CDO e CDLOC). Estas últimas foram obtidas pelo sombreamento de todas as classes, interfaces e aspectos em todas as implementações (OO e AO). O sombreamento baseou-se nas características da infra-estrutura básica que implementam o middleware reflexivo. Dessa forma, cada característica foi tratada como um conceito que pode entrelaçar outros conceitos nas duas versões. Depois de sombrear, os dados relativos às métricas de separação de conceitos foram manualmente coletados. No entanto, como as métricas são orientadas a unidades de baixa granulari-

	Métricas	Definições
Separação de conceito	Difusão do Assunto por Componentes (CDC)	Conta o número de componentes cujo propósito principal é contribuir para a implementação do assunto avaliado. Além disso, CDC conta também o número de componentes que fazem referência aos componentes principais do assunto.
	Difusão do Assunto por Operações (CDO)	Conta o número de operações cujo propósito principal é contribuir par a implementação do assunto avaliado. CDO conta também os métodos, construtores e advices que acessam alguma das operações principais do assunto.
	Difusão do Assunto por Linhas de Código (CDLOC)	Conta o número de pontos de transição existentes no código entre o assunto avaliado e os demais assuntos do sistema. O uso desta métrica requer sombreamento do código que o divide em áreas
Acoplamento	Acoplamento entre Componentes (CBC)	Conta o número de classes e aspectos que estão associados, considerando elementos como parâmetros e tipos de retorno.
	Profundidade da Árvore de Herança (DIT)	Conta o número de níveis da hierarquia até que se atinja a raiz da árvore.
	Número de Filhos (NOC)	Conta quantos filhos uma classe ou aspectos possui.
Coesão	Perda de Coesão entre Operações (LCOO)	Conta o número de pares de operações que compartilham atributos, menos o número de pares de operações que não compartilham nenhum atributo.
Tamanho	Linhas de Código (LOC)	Conta o número de linhas de código.
	Número de atributos (NOA)	Conta o número de atributos que cada classe ou aspeto possui.
	Peso das Operações por Componente (WOC)	Conta o número de métodos, construtores e advices com seus respectivos parâmetros por componentes.
	Tamanho do Vocabulário (VS)	Conta o número de classes e aspectos de um sistema.

Figura 38: Definição das métricas de separação de conceitos, acoplamento, coesão e tamanho

dade, tais como operações e linha de código, foi necessário um trabalho adicional para alinhar as duas implementações antes de finalmente realizar a coleta dos dados. Com isso, garantimos que as duas versões implementam exatamente as mesmas funcionalidades.

A Figura 39 mostra os resultados obtidos com as métricas de separação de conceitos. Os resultados são divididos entre as várias características da infra-estrutura básica que inclui a reflexão computacional através da conexão causal e os meta-modelos.

A análise da Figura 39 mostra que a versão AO do middleware teve um melhor desempenho que a versão OO para a maioria das medições. Em particular, a versão AspectJ é superior para todos os conceitos apresentados na seção 3.3.1, tal como os mecanismos de binding e conexão causal. Portanto, as métricas de SoC indicaram um significante aumento de modularidade na maior parte das características e sub-características da versão aspectizada do OpenORB, incluindo o estado dos *bindings*, gerenciamento de invocações, etc. Ainda mais importante, a separação de todas as características reflexivas claramente foram melhoradas pela utilização de aspectos. Isto fica evidente pela redução em 50% do número de elementos removidos, tais como o nível de entrelaçamento (CDLOC) na implementação de estrutura dos grafos de componentes, conexão causal, mecanismo de detecção e meta-modelo de composição.

A Figura 40 mostra os resultados obtidos para as três métricas de acoplamento -

	CDC		CDO		CDLOC		Superior
Características (Padrões)	OO	AO	OO	AO	OO	AO	**Solução**
Infra-estrutura Básica							
Composição de grafo de componentes (Composite)	5	2	11	11	10	2	AO
Binding							
➢ Estado do binding (State)	5	4	12	6	14	4	AO+
➢ Gerenciamento das invocações (Mediator)	5	2	18	10	22	8	AO
➢ Implementação do binding local (Proxy)	24	19	69	59	14	10	AO
➢ Estratégias de invocação (Strategy)	4	2	9	5	8	2	AO
Conexão causal							
➢ Mecanismo de detecção (Observer)	8	2	14	4	28	4	AO+
➢ Mecanismo de captura (Memento)	5	2	9	5	12	4	AO+
Meta Modelos							
➢ Meta modelo de composição (Composite/Visitor)	4	3	7	7	6	4	AO
➢ Meta modelo de encapsulamento (Decorator)	4	3	15	11	10	8	AO
Total de sucessos	**0 vs. 9**		**0 vs. 7**		**0 vs. 9**		**0 vs. 9**

Figura 39: Quantificação da separação de conceitos

Acoplamento entre componentes (CBC), Profundidade da Arvore de Herança (DIT) e número de filhos (NOC) - juntamente com as métricas de coesão, tais como perda de coesão entre operações (LCOO). Esta figura também mostra o resultado para as quatro métricas de tamanho - Tamanho do Vocabulário (VS), Linhas de Código (LOC), Número de atributos (NOA) e Peso das Operações por Componente (WOC). No sentido de facilitar o entendimento dos resultados, a Figura 40 divide os resultados entre classes e aspectos para as duas versões do OpenORB. Dessa forma, usamos o termo "classe"para indicar classes e interfaces. As linhas indicadas por "Diferença"representam a diferença entre as versões OO e AO para cada métrica. Um valor positivo representa que a versão OO foi melhor, enquanto que negativo indica que a versão AO obteve um melhor resultado.

Portanto, a aspectização do Open-ORB apresentou os seguintes resultados: (i) melhorou a coesão das funcionalidades do middleware e (ii) reduziu o acoplamento entre pais e filhos em mais de 13% e 24%, respectivamente de acordo com as métricas DIT e NOC. Por outro lado, a aspectização da parte reflexiva da infra-estrutura básica aparentemente não mostrou um efeito realmente positivo para a métrica CBC. Entretanto, um estudo detalhado da implementação e dos dados presentes na Figura 40 mostraram claramente a redução do acoplamento entre os componentes que implementam a infra-estrutura básica. Isto pode ser visto na Figura 40 pelo decréscimo em 40 unidades no valor apresentado pela métrica CBC para as classes. No entanto, o aumento no número de aspectos faz com que o valor final dessa métrica apresente uma perda de acoplamento. Na verdade o que ocorre é uma inversão de dependência, onde os aspectos dependem das classes sem que as classes dependam dos aspectos.

	Acoplamento entre Componentes - CBC		*Profundidade da Árvore de Herança* - DIT	
	OO	AO	OO	AO
Classes	183	143	107	84
Aspectos	-	64	-	9
Total	183	207	107	93
Diferença	+11.6%		–13.1%	
	Número de Filhos - NOC		*Perda de Coesão entre Operações - LCOO*	
	OO	AO	OO	AO
Classes	41	31	100	85
Aspectos	-	0	-	2
Total	41	31	100	87
Diferença	–24.4%		–13.0%	
	Tamanho do Vocabulário - VS		*Linhas de Código - LOC*	
	OO	AO	OO	AO
Classes	72	62	1128	830
Aspectos	-	9	-	292
Total	72	71	1128	1122
Diferença	–1.4%		–0.5%	
	Número de atributos - NOA		*Peso das Operações por Componente - WOC*	
	OO	AO	OO	AO
Classes	68	66	460	350
Aspectos	-	6	-	102
Total	68	72	460	452
Diferença	+5.6%		–1.7%	

Figura 40: Quantificando acoplamento, coesão e tamanho

Finalmente, contradizendo a idéia geral que aspectos produzem programas menores devido a modularização e reuso do comportamento entrelaçado, as versões OO e AO do Open-ORB tiveram resultados bastante similares para as quatro métricas de tamanho. Embora exista uma certa superioridade para POA em três métricas, elas deveriam ser consideradas significantes se apresentassem valores abaixo de 5%. Portanto, concluímos que a separação de conceitos no OpenORB foi alcançada mas o reuso não foi expressivo devido ao limitado número de instâncias definidas para cada aspecto abstrato.

Além de analizar a modularidade do OpenORB, este trabalho também mostra como o uso de POA na infra-estrutura básica pode influenciar ou não no desempenho do middleware. Portanto, a Tabela 12 mostra um comparativo entre os tempos de criação de cada um dos elementos básicos do JOpenORB. Neste tabela é mostrada o tempo de execução no LOpenORB e o tempo correspondente no Aspect Open-ORB. Diferentemente do esperado, as quatro primeiras linhas da Tabela 12 mostram o melhor desempenho da versão Aspect Open-ORB em relação ao JOpenORB. Esta diferença justifica-se pelas características dessas invocações, onde é criado instâncias para as classes *Receptacle*, *Compo-*

nent, *CompositeComponent* e *BindMediator*. Como todas essas classes foram refatoradas no processo de aspectização, é natural que com menos métodos e atributos estas classes passem a ser criadas em menor tempo.

Tipo de teste	Realizado em	Tempo em μs
Criando uma interface (IRef)	Aspect Open-ORB	0.47
	JOpenORB	0.94
Criando componente	Aspect Open-ORB	0.93
	JOpenORB	1.41
Criando componente composto	Aspect Open-ORB	4.53
	JOpenORB	5.31
Criando um *binding* local	Aspect Open-ORB	0.62
	JOpenORB	1.25
Executando através de um *binding* local	Aspect Open-ORB	21.4
	JOpenORB	5.69
Executando através de um *binding* local com meta objeto	Aspect Open-ORB	40.47
	JOpenORB	11.1
Inserção de component em um grafo de componentes	Aspect Open-ORB	3.28
	JOpenORB	9.22

Tabela 12: Resultado da comparação entre JOpenORB e Aspect Open-ORB

Apesar de reduzir o tempo de instanciação, o Aspect Open-ORB aumenta o tempo de invocação dos métodos. Como pode ser visto na quinta e sexta linhas, o tempo para invocação local cresceu de 5.69μs para 21.4μs e a invocação com meta-objeto associado cresceu de 11.1μs para 40.47μs. Em ambos os casos destacamos o efeito do elemento mediador. Ou seja, a presença de vários aspectos que atuam na classe *ConcreteBind* reduzem significativamente o desempenho. A mesma perda de desempenho ocorre para as invocações remotas, onde a primeira invocação passou de 78ms para 80.08ms e as demais passaram de 5.33ms para 6.02ms.

5.2 Avaliando as soluções entre as diferentes linguagens

Esta seção compara as implementações por linguagem de programação. Dessa forma, iremos comparar LOpenORB com JOpenORB e em seguida compararemos as duas versões do Aspect OpenORB.

5.2.1 Comparando LOpenOrb com JOpenOrb

A Tabela 13 mostra os resultados obtidos na realização dos testes com os elementos básicos. O primeiro teste compara o tempo de criação de uma Interface através da invocação do método IRef. A abordagem JOpenORB obteve um tempo de 0.94μs, enquanto a abordagem LOpenORB obteve 45.9μs. Essa diferença de 45μs deve-se ao fato de que a abordagem LOpenORB cria uma instância da classe *Methods* para todos os métodos importados e exportados, enquanto que o JOpenORB cria apenas uma instância da classe *Receptacle*.

O segundo e o terceiro teste comparam o tempo de criação de um componente e um componente composto. A grande diferença entre os tempos das duas abordagens é atribuída as características de cada uma das linguagens. Na linguagem Lua é necessário construir, em tempo de execução, toda a estrutura de definição dos componentes, enquanto que em Java a própria definição das classes já provê essa estruturação.

Tipo de teste	Realizado em	Tempo em μs
Criando uma interface (IRef)	LOpenORB	45.9
	JOpenORB	0.94
Criando componente	LOpenORB	7.08
	JOpenORB	1.41
Criando componente composto	LOpenORB	57.1
	JOpenORB	5.31
Criando um *binding* local	LOpenORB	17.05
	JOpenORB	1.25
Executando através de um *binding* local	LOpenORB	19.6
	JOpenORB	5.69
Executando através de um *binding* local com meta objeto	LOpenORB	21.75
	JOpenORB	11.1
Inserção de component em um grafo de componentes	LOpenORB	17.84
	JOpenORB	9.22

Tabela 13: Resultado da comparação entre LOpenORB e JOpenORB

No teste relacionado ao tempo de criação de um *binding* local, a abordagem LOpenORB apresenta um baixo desempenho em decorrência da criação um maior número de objetos, sendo eles uma instância de *BindCtrl*, uma instância de *BindingGraphElement* para cada método importado e duas instâncias de *GraphNode* para cada método importado. Conseqüentemente, o LOpenORB obteve um tempo de 17.05μs enquanto que o JOpenORB, que cria apenas uma instância de *BindCtrl*, obteve 1.25μs.

A eficiência do *binding* local é verificada pela invocação de um método usando tal infraestrutura. Para ter uma noção do tempo gasto em cada linguagem, o mesmo método,

implementado igualmente nas duas linguagens foi executado. Em Java, o método obteve um tempo de 0.31μs enquanto que em Lua o tempo foi de 5.42μs. Quando invocado por um *binding* local, os tempos subiram para 5.69μs no JOpenORB e 19.6μs no LOpenORB. Ou seja, a presença de um *binding* local reduziu a velocidade de invocação no LOpenORB em 14.18μs, enquanto que no JOpenORB essa perda foi de 5.38μs. Portanto, apesar de apresentar um melhor desempenho, a solução em Java sofreu uma perda maior do que a mesma solução em Lua. Isto ocorre porque o mecanismo de invocação em Java requer a utilização de inúmeras invocações reflexivas para a obtenção do método a ser invocado.

Para verificar o impacto dos elementos reflexivos na invocação de métodos, realizamos dois experimentos: i) associação de meta-objeto ao *binding* local e ii) substituição de componente em um grafo de componentes. O primeiro experimento mostrou que a presença do meta-objeto reduz a velocidade de execução de um respectivo métodos. No caso da solução em Java o desempenho variou de 5.69μs para 11.1μs, representando uma perda de desempenho que corresponde a 50% do valor inicial. Uma perda menor foi obtida para a solução em Lua, onde foi obtido o valor 2.8μs, o que equivale a uma redução de 30% do desempenho. A menor perda da solução Lua é justificada pelo maior processamento das invocações reflexivas utilizadas em Java e pela simplicidade provida por Lua para fazer o mesmo. O segundo experimento permitiu verificar o comportamento das duas abordagens no tratamento da troca de um componente por outro similar. Dessa forma, na solução Java, a troca foi efetivada em 9.22μs, enquanto que na versão Lua, a mesma troca foi feita em 17.84μs. Neste específico caso, a diferença esta relacionada a utilização de diferentes algoritmos que dependem das facilidades providas pela linguagem para realizar a troca dos componentes.

Na Figura 41, são comparados os tempos de criação de *Operational* bindings entre as abordagens JOpenORB e LOpenORB. Para um *Operational binding* a abordagem JOpenORB consumiu 62.13ms, enquanto que o LOpenORB consumiu 24.48ms. Comparando os valores obtidos quando são atingidos 20(vinte) *Operational bindings*, onde JOpenORB obteve um tempo de 78.95ms e o LOpenORB um tempo de 140.63ms, percebemos que a versão Java consegue escalonar melhor as invocações em comparação com a versão Lua que cresce exponencialmente. O motivo para isto decorre de usarmos em Lua o escalonador suportado pela biblioteca LuaThread, que não apresenta a mesma sofisticação da implementação suportada pela JVM.

O tempo de uma invocação remota também foi alvo de análise. No JOpenORB, a primeira invocação consome 6.15ms enquanto que as demais consomem 5.27ms. No LO-

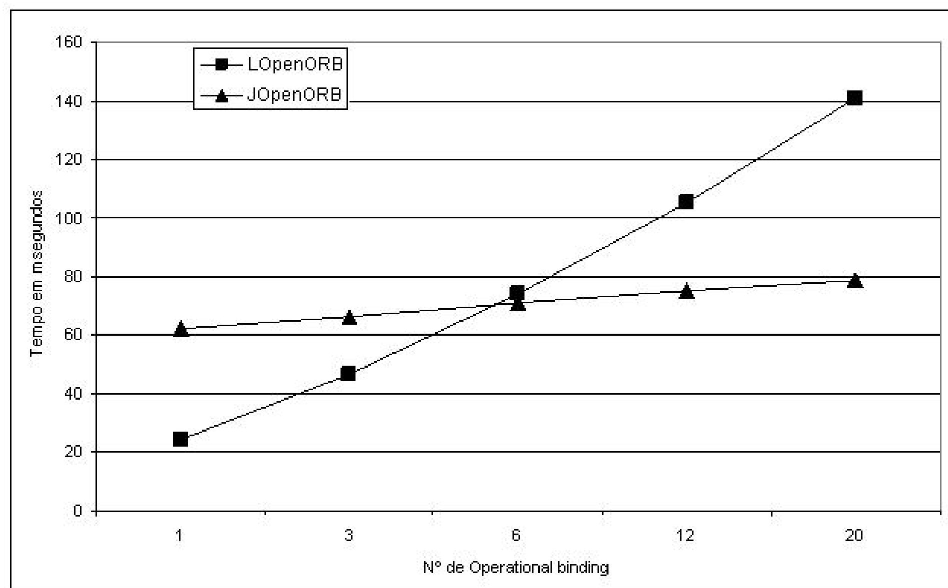

Figura 41: Gráfico de criação de Operational binding

penORB, a primeira invocação consome 3.69ms e nas demais esse tempo cai e permanece constante em 1.31ms. Esta diferença entre a primeira e as subseqüentes invocações do JOpenORB e LOpenORB é gerado pelo tempo inicial necessário para criar o componente composto responsável por tratar conexão.

5.2.2 Comparando as versões Lua e Java do Aspect OpenORB

Nas seções anteriores analisamos como a aspectização influencia no desempenho, consumo de memória e modularidade das duas implementações. Com isso, cabe a esta seção analisar como cada versão aspectizada trata da customização da plataforma OpenORB. Para tanto, listamos na Tabela 14 como cada implementação comporta-se diante dos três tipos de customização requeridas. No sentido de orientar a nossa avaliação, definimos níveis de customização que são distribuídos em quatro categorias:

1. *Durante compilação*: Esta customização requer o acesso ao código fonte e sua posterior modificação para a introdução ou remoção de uma das características requeridas pela plataforma alvo. Este nível de customização poderá ser representado na Tabela 14 pelo seguinte símbolo $\ominus$.

2. *Durante load-time*: Esta customização ocorre pela utilização de mecanismos como *hooks* que permitem postergar o carregamento de certas funcionalidades, ou mesmo pela utilização de arquivos de configuração que são utilizados para determinar as políticas de utilização dos recursos de uma determinada plataforma. Este tipo de

Implementação	Infra-Estrutura Básica	Infra-estrutura de componentes	Aplicação
Java Aspect OpenORB	$\ominus$	$\diamondsuit$	$\diamondsuit$
Lua Aspect OpenORB	$\triangle$	$\triangle$	$\triangle$

Tabela 14: Resultado da capacidade de customização

customização é bastante empregado para bibliotecas dinâmicas(DLL) e poderá ser representado na Tabela 14 pelo seguinte símbolo $\oplus$.

3. *Durante execução com restrições*: Esta customização ocorre durante a execução da plataforma de middleware mas apresenta um certo nível de restrições onde apenas operações definidas pelo protocolo de meta-objeto que não exigem modificação dinâmica de código são suportadas, tais como troca de componentes ou *bindings*. Este nível de customização poderá ser representado na Tabela 14 pelo seguinte símbolo $\diamondsuit$.

4. *Durante execução*: Esta customização caracteriza-se quando não existe qualquer restrição para inserir/remover elementos durante a execução do código da aplicação. Isto inclui adicionar atributos, métodos, classes ou mesmo substituir funções sem que para isso seja necessário realizar a carga de toda a classe. Este nível de customização é representado na Tabela 14 pelo seguinte símbolo $\triangle$.

Como pode ser visto na Tabela 14, a versão Lua do Aspect OpenORB usa as características dinâmicas da linguagem Lua para fornecer um elevado nível de customização para todas as três camadas. Por sua vez, a versão Java e AspectJ limita sua customização para a infra-estrutura básica e demais camadas para, respectivamente, customização em tempo de compilação e em tempo de execução com restrições. As restrições incluem a impossibilidade de modificar a estrutura das classes, etc. No entanto, nenhuma dessa limitações desqualifica a versão Java do OpenORB, uma vez que ela foi projetada para atender os requisitos de aplicações móveis em plataformas com grande diversidade de requisitos.

No sentido de avaliar como cada estratégia de aspectização afetou o desempenho das duas implementações, mostramos na Tabela 15 os resultados obtidos nos testes com os elementos da infra-estrutura básica. A diferença entre os valores das quatro primeiras linhas são resultado da mesma motivação mostrada na seção anterior, ou seja, a diferença entre as duas versões deve-se a necessidade da linguagem Lua de construir, em tempo de execução, toda a estrutura de definição dos componentes, enquanto que em Java a própria definição das classes já provê essa estruturação. Por outro lado, o resultado da

quinta e sexta linhas mostram a grande diferença entre as versões Java e Lua. Neste caso, a presença do elemento mediador reduz o desempenho da implementação em Java, enquanto que em Lua, a interceptação de aspectos ocorre na invocação dos métodos da API e não diretamente na infra-estrutura básica do OpenORB.

Tipo de teste	Realizado na versão	Tempo em μs
Criando uma interface (IRef)	Java	0.47
	Lua	46.5
Criando componente	Java	0.93
	Lua	7.27
Criando componente composto	Java	4.53
	Lua	60.04
Criando um *binding* local	Java	0.62
	Lua	17.08
Executando através de um *binding* local	Java	21.4
	Lua	19.89
Executando através de um *binding* local com meta objeto	Java	40.47
	Lua	22.15
Inserção de component em um grafo de componentes	Java	3.28
	Lua	17.98

Tabela 15: Resultado da comparação entre as duas versões do Aspect OpenORB

6 Trabalhos relacionados

Neste capítulo apresentamos um resumo das principais pesquisas realizadas na área de customização de middleware e como estas podem ser comparadas com a nossa abordagem. A maioria dos trabalhos apresentados estão diretamente relacionados a utilização da reflexão computacional na customização de plataformas de middleware. Entretanto, este trabalho vai além, uma vez que utiliza POA para customizar a infra-estrutura básica da plataforma, algo que não é possível de ser feito apenas com reflexão. Esta é, portanto, a principal diferença entre a nossa abordagem e as demais listadas nesse capítulo.

6.1 FlexiNet

FlexiNet (HANSSEN; ELIASSEN, 1999) é uma plataforma de middleware desenvolvida em Java que permite invocações de métodos remotos por meio de uma infra-estrutura de *binding*. Tal infra-estrutura é construída por meio de uma pilha de protocolo usando um *white-box framework* que suporta não somente as funcionalidades básicas mas também características avançadas como replicação e criptografia. Para isso, cada camada da pilha de protocolo manipula cada invocação individualmente antes que esta seja enviada para o objeto destino, sendo assim possível controlar o número de pilhas envolvidas e seus respectivos objetos para cada invocação. No sentido de suportar esta tarefa, FlexiNet permite especificar e coordenar a construção de grafos de componentes que usam *constraints* para determinar, por meio de um processo de resolução, quais objetos irão participar de um específico grafo de componentes (Ex. um grafo para implementar o protocolo de transporte). Dessa forma, quando as *constraints* são satisfeitas, o processo de resolução termina com sucesso e cada nó do grafo, correspondendo a um simples objeto, pode ser construído e inicializando; caso contrário, uma falha é reportada. FlexiNet permite que uma aplicação controle a pilha de protocolo e as *constraints* que serão usadas para construir um *binding* para cada objeto. Além disso, esta abordagem provê suporte básico para *binding* dinâmico pela definição de uma interface genérica no proxy-cliente. Como

resultado da evolução desta plataforma, FlexiNet(HANSSEN; ELIASSEN, 1999) estende este suporte básico pelo acréscimo de novas políticas que podem configurar a pilha de protocolo e meta-políticas que controlam a seleção das primeiras políticas.

6.2 Jonathan

Jonanthan (DUMANT et al., 1999) é uma plataforma de middleware baseada em um *white-box* framework que suporta a composição de um conjuntos de APIs, tais como CORBA e RMI. Jonathan é organizado por meio de quatro frameworks: *binding*, comunicação, recursos e configurações. O primeiro framework suporta o desenvolvimento e introdução de novos tipos de binding na forma de *pluggable binders*. Estes *bindings* são responsáveis por gerenciar e prover acesso ao objeto alvo da invocação remota. Na solução padrão, o *framework binding* implementa invocações remotas usando CORBA IIOP e um canal de eventos que prove comunicação no estilo *publish/subscribe*. Por sua vez, o *framework* de comunicação suporta a composição de grafos de protocolos a partir do reuso de implementações disponíveis, tais como IP *multicast*, protocolos de tempo-real e CORBA GIOP. No sentido de prover abstrações para suportar *buffers*, *threads* e conexões de rede, o *framework* de recursos gerencia tais recursos através da associação de políticas de gerenciamento. Finalmente, o *framework* de configuração permite definir genericamente através de um arquivo XML as configurações iniciais do middleware. Esta descrição XML é utilizada para instanciar e interconectar as várias classes do Jonathan, tais como *binders*, protocolos e políticas de gerenciamento.

6.3 dynamicTAO

2K(KON et al., 1999) é um sistema operacional criado para suportar o desenvolvimento de aplicações dinâmicas e heterogêneas. Este sistema adota uma arquitetura em camadas composta pelo middleware e pelo kernel. A camada de kernel corresponde a um sistema operacional tradicional onde reside o micro kernel do 2K. Por sua vez, a camada de middleware corresponde a um conjunto de serviços, tais como nomes e segurança que executam no topo de um middleware reflexivo baseado em CORBA, chamado dynamicTAO(KON et al., 2000). Este middleware é uma versão estendida e reflexiva do TAO(SCHMIDT; CLEELAND, 1998), que suporta a customização de componente para habilitar reconfiguração dinâmica enquanto preserva a consistência de seus elementos internos. Dessa forma, dynamicTAO suporta instanciar um componente baseando-se em

meta-informações que descrevem o tipo e a capacidade de recursos de hardware que o componente requer, juntamente com uma lista de outros componentes requeridos. Este primeiro conjunto de informações são fornecidas ao serviço de gerenciamento de recursos do 2K que realiza a alocação dos recursos apropriados. Por sua vez, o controle de dependência entre componentes é feito pelo configurador de componentes (component configurators). Este componente trata as dependências entre componentes pela exposição de um conjunto padrão de operações que permite manipular as dependências dinamicamente de modo que não seja necessária a interrupção do serviço para que o mesmo seja realizado.

6.4 Lasagne

Lasagne (TRUYEN et al., 2001) é um middleware orientado a aspecto para customização de aplicações dinâmicas e sensíveis a contexto. Portanto, o principal foco deste middleware é permitir a customização dinâmica de seu modelo de componentes. Para tanto, Lasagne define seu próprio modelo de componentes e de definição de aspectos que não segue nenhum dos padrões existentes. Dessa forma, o modelo de programação suportado por Lasagne é baseado na definição de *collaborations*, *wrappers*, *component descriptors* e *interceptors*. Com isso, um serviço distribuído em Lasagne é implementado pelo modelo de componentes e customizado por um conjunto de aspectos (*collaborations* em Lasagne) que podem seletivamente serem integrados as funcionalidades básicas. Assim, aspectos são modelados como *collaborations* que por sua vez definem um conjunto de *wrappers*, onde cada um desses *wrappers* pode ser aplicado a diferentes pontos da aplicação. O processo de *weaving* entre as *collaborations* e os componentes definidos pela aplicação são especificados pelos *component descriptors*. Estes descritores descrevem as interfaces fornecidas e requeridas de cada componente, juntamente com uma lista de *wrappers* que podem ser usados para adicionar novas funcionalidade aos componentes. A definição desses *wrappers* é acompanhada por uma lista de *method bindings* que especifica os pointcuts que devem ser interceptados para cada componente no sentido de inserir um wrapper.

6.5 JAC

JAC (Java Aspect Components) (PAWLAK et al., 2004) é um middleware orientado a aspectos implementado em Java. JAC provê um conjunto de conceitos que possibilita

a distribuição e a utilização de aspectos dinâmicos na customização de uma plataforma de middleware. Os três principais conceitos são: *aspect components*, *wrappers* e JAC *containers*. *Aspect components* são responsáveis por definir os *pointcuts* enquanto que os *wrappers* encapsulam o comportamento do aspecto. Dessa forma, *aspect components* represetam as propriedades que entrelaçam um conjunto de objetos que estão disponíveis no *container* do JAC. Este *container* é similar ao provido por plataformas J2EE, mas no entanto implementa seus próprios padrões. Dessa forma, a arquitetura desses *containers* incluem *class loaders* que são capazes de carregar tanto classes quanto aspectos. Isso permite que novos serviços sejam adicionados na medida em que os mesmo forem necessários. Na versão atual, JAC prove serviços para controle de persistência, balanceamento de carga, consistência, segurança, etc. Em termos de tecnologia de middleware, JAC suporta comunicação por RMI e CORBA.

6.6 JBoss AOP

JBoss AOP (BURKE; BROCK, 2003) é um framework orientado a aspectos que pode ser integrado como um dos serviços do servidor de aplicação JBoss. Aspectos e outras construções em JBoss AOP são definidos por meio de classes Java e relacionados com o código da aplicação por meio de documentos XML ou anotações Java. Na implementação JBoss AOP, o código da aplicação pode ser afetado por aspectos, *interceptadores*, *introdutores* ou *mixins*. Aspecto são utilizados para encapsular a definição de *advices*, *pointcuts*, *mixins* ou qualquer outra construção JBoss AOP. Por sua vez, interceptadores são aspectos que tem somente um método *advice* com nome *invoke*. Introdutores são usados para forçar que uma classe implemente uma interface ou para adicionar uma anotação em qualquer um dos elementos da aplicação. Finalmente, uma classe *mixin* é usado para implementar uma interface previamente introduzida. A arquitetura do servidor de aplicação JBoss é dividida em três camadas: (1) microkernel, responsável por prove um modelo de componentes que oferece *hot deployment* e características avançadas de *class-loading*; (2) serviço, consiste em uma série de serviços tais como, transação, mensagens, segurança, etc; (3) aspecto, responsável por instanciar os aspectos que são definidos nos arquivos XML ou nas anotações. Esta terceira camada permite adicionar e remover aspectos tanto em tempo de compilação como em tempo de carga. Dessa forma, embora o processo de *weaving* seja realizado em tempo de carga por meio de transformação de *bytecodes*, existe uma API dinâmica que permite adicionar e remover *advices* e interceptadores para qualquer *joinpoint* que foi definido durante a compilação ou na carregamento.

6.7 Jacobsen et al.

A idéia de aspectizar plataformas de middleware vem sendo explorada em vários artigos por Jacobsen (ZHANG; JACOBSEN, 2003b, 2004; ZHANG; GAO; JACOBSEN, 2005b) . O principio da decomposição horizontal (HD) foi proposto em (ZHANG; JACOBSEN, 2003a) para resolver o problema dos conceitos transversais. Neste sentido, (ZHANG; JACOBSEN, 2003a) define um guia para se realizar decomposição horizontal, usando como exemplo o ORBacus e a linguagem AspectJ. O guia inclui a decomposição do middleware em três camadas (IDL, mensagem e transporte/protocolo) e também as funcionalidades que podem ser representadas como aspectos: invocações *oneway*, tipo dinâmicos para implementar reflexão em invocações remotas, mecanismo de codificação e o suporte para invocações locais. No sentido de validar a decomposição horizontal, os autores aplicam esta técnica na decomposição do banco de dados Prevayler(GODIL; JACOBSEN, 2005). Um outro trabalho deste mesmo grupo propõem o Modelware(ZHANG; GAO; JACOBSEN, 2005a) como uma metodologia que combina a abordagem MDA(Model Driven Architecture) e POA para implementar a customização e melhorar o desempenho das plataformas aspectizadas.

6.8 Avaliação

Para facilitar a comparação entre os trabalhos listados acima, iremos utilizar a mesma categorização usada na Seção 5.2.2, onde existem quatro tipos de capacidade de customização que são realizadas em diferentes momentos: durante compilação($\ominus$), durante load-time($\oplus$), durante execução com restrições($\diamondsuit$) e sem restrição($\triangle$), e não suporta customização ($\emptyset$).

Abordagem	Infra-Estrutura Básica	Infra-estrutura de componentes	Aplicação
FlexiNet	$\emptyset$	$\diamondsuit$	$\diamondsuit$
Jonathan	$\emptyset$	$\diamondsuit$	$\diamondsuit$
dynamicTAO	$\emptyset$	$\diamondsuit$	$\diamondsuit$
Lasagne	$\emptyset$	$\diamondsuit$	$\diamondsuit$
JAC	$\emptyset$	$\oplus$	$\oplus$
JBoss AOP	$\emptyset$	$\diamondsuit$	$\diamondsuit$
Jacobsen et al.	$\ominus$	$\oplus$	$\oplus$

Tabela 16: Resumo das avaliações

A Tabela 16 mostra que apenas a abordagem provida por Jacobsen suporta a aspectização da infra-estrutura básica. No entanto, nossa abordagem distingui-se da provida por Jacobsen em vários aspectos: (i) a versão em Java do Aspect OpenORB prioriza a

modularidade em detrimento do desempenho fornecido pela solução do Jacobsen, (ii) por sua vez a versão Lua prioriza a capacidade dinâmica de inserir e remover aspectos, algo que as soluções do Jacobsen não suportam, e (ii) por fim a nossa versão AO é obtida pela total aspectização da versão OO, enquanto que Jacobsen aspectiza apenas os aspectos que podem melhorar o desempenho da middleware, ou seja, a utilização da versão do Jacobsen em ambientes limitados pode exigir uma nova aspectização.

Em relação a customização da infra-estrutura de componentes e aplicação, percebemos que todas as abordagem usam a mesma técnica para as duas camadas. As soluções FlexiNet, dynamicTAO, Jonathan, Lasagne e JBoss AOP suportam a customização dinâmica pela utilização de mecanismos diversos como reflexão computacional, binding factories e aspectos que permitem inserir e remover componentes da infra-estrutura de componentes e aplicação. No entanto, por limitações da própria linguagem de implementação, como Java e C++, tais abordagem estão restritas a aplicação da customização para pontos bem definidos na arquitetura ou framework utilizados. Por outro lado, as abordagens JAC e Jacobsen não suportam a customização dinâmica, uma vez que o processo de weaving implementado por essas abordagens é restrito a tempo de carga ou compilação, não permitindo assim a flexibilidade das abordagens dinâmicas.

Comparando a Tabela 14 da seção anterior com a Tabela 16, percebemos que a versão Lua da nossa abordagem supera todas as demais abordagens em termos de capacidade de customização. No entanto, consideramos para esta avaliação apenas a possibilidade de adicionar e remover elementos, tais como métodos, atributos, classes, etc. Ou seja, neste critério, a versão Lua do Aspect Open-ORB apresenta um ótimo desempenho. Entretanto, a capacidade de customização envolve outros critérios que não foram implementados por nossas abordagens, como por exemplo, o uso de políticas de utilização de recursos (Memória, CPU, Threads) para customizar a plataforma de middleware. Este tipo de customização é implementada por várias abordagens como FlexiNet, Jonathan e dynamicTAO e que não é suportada pelo Aspect OpenORB.

7 *Considerações Finais*

Neste trabalho apresentamos uma proposta para integração de reflexão computacional e Programação Orientada a Aspectos no contexto de desenvolvimento de uma plataforma de middleware. Diferentemente das abordagens tradicionais, nossa proposta explora duas estratégias: (1) utiliza POA para implementar a customização da infra-estrutura básica; (2) utiliza reflexão computacional para implementar a customização das demais camadas do middleware. No sentido de avaliar como esta abordagem é afetada pela linguagem de programação utilizada, implementamos neste trabalho dois protótipos de parte da arquitetura proposta, o primeiro protótipo utilizou uma linguagem dinamicamente tipada com facilidades reflexivas, a linguagem Lua, e uma extensão dessa linguagem que dá suporte a programação orientada a aspectos, AspectLua. O segundo prótotipo avaliou os impactos dessa abordagem em uma linguagem compilada, Java, utilizando padrões de projeto e uma extensão para a programação orientada a aspectos, AspectJ. A escolha por estas duas linguagens deve-se ao fato de se tratarem de paradigmas diferentes e que, portanto, podem validar a solução de uma forma mais ampla.

A definição da estrutura interna do LOpenORB e JOpenORB foi pautada no suporte amplo à customização da infra-estrutura de comunicação e aplicação. A descrição das implementações mostrou que, além de fornecer abstrações da alto nível com suporte a reflexão, como interfaces, componentes e *bindings* locais, é possível e necessário também aplicar essas abstrações na construção dos mecanismos internos do ORB, como na infra-estrutura de comunicação e na construção dos *bindings* explícitos. A opção pela implementação do OpenORB em duas linguagens (Lua e Java) foi importante para verificar como uma boa estruturação interna, baseada no modelo de componentes pode suportar a construção de uma implementação com alto poder de adaptação dinâmica, não importando qual linguagem seja utilizada.

A avaliação da versão Java aspectizada mostrou que apesar de não ter apresentado um bom desempenho na invocação de métodos, a modularidade foi significativamente melhorada da versão OO para a AO, o que implica em uma maior facilidade na remoção

e inserção de componentes que constituem a infra-estrutura básica e que, por sua vez, dependem dos requisitos da plataforma alvo. Na mesma direção, a versao Lua teve uma leve perda de desempenho mas por outro lado conseguiu reduzir a memória utilizada para determinadas configurações. Portanto, ambas as implementações cumpriram com o seu objetivo de promover uma solução mais modular (versão Java) e com maior capacidade de customização dinâmica (versão Lua).

7.1 Contribuições

De maneira geral, as contribuições desse trabalho foram:

- Definição e implementação de um middleware reflexivo na linguagem Lua e Java (LOpenORB e JOpenORB), que permitem utilizar toda a infra-estrutura básica para a construção de aplicações distribuídas.
- Definição da arquitetura Aspect Open-ORB através da descrição dos vários aspectos utilizados no processo de personalização de um ORB.
- A implementação do middleware reflexivo usando duas linguagem de programação, demonstrou que a arquitetura de implementação e as estratégias usadas são independentes de linguagem e podem ser empregadas usando-se outras linguagens tanto interpretadas quanto compiladas.
- Utilização de duas estratégias de aspectização (interpretada e compilada) que respeitou os limites de cada uma das linguagens para obter uma maior modularidade e customização dinâmica.
- Avaliação de desempenho das implementações: (a) LOpenORB x JOpenORB; (b) LOpenORB x Aspect Open-ORB; (c) JOpenORB x Aspect Open-ORB e Aspect Open-ORB em Lua x Java.
- Avaliação da modularidade das implementações Java por meio de um conjunto de métricas que consideram separação de conceito, acoplamento, coesão e tamanho.

7.2 Trabalhos Futuros

Como forma de dar continuidade à pesquisa desenvolvida nessa dissertação, alguns trabalhos futuros podem ser relacionados, tais como:

- Introduzir customização baseada em políticas de utilização de recursos;
- Implementar a versão Java do Aspect OpenORB por meio de uma outra linguagem de aspecto com suporte a weaving dinâmico;
- Uso do Aspect Open-Orb em cenários de aplicações dinâmicas complexas visando testar, amplamente, sua capacidade de adaptação dinâmica bem como o desempenho.

Referências

AGHA, G. A. Adaptive middleware. *Commun. ACM*, ACM Press, v. 45, n. 6, p. 31–32, 2002. ISSN 0001-0782.

AKSIT, M. et al. *Aspect-oriented Software Development*. 2002. Disponível em: <http://aosd.net>.

ALEXANDER, C. *The Timeless Way of Building*. [S.l.]: Oxford University Press, 1979.

ALEXANDER, C.; ISHIKAWA, S.; SILVERSTEIN, M. *A Pattern Language: Towns, Buildings, Construction*. [S.l.]: Oxford University Press, 1977.

ANDERSEN, A. A reflective component-based middleware in Python. In: *The Eighth International Python Conference*. Arlington, Virginia, USA: [s.n.], 2000. (short talk). Disponível em: <http://www.cs.uit.no/ aa/abstracts/andersen2000a.html>.

ANDERSEN, A. *OOPP - A Reflective Middleware Platform incluing and Quality of Service Management*. Tese (Dr. Sci. Thesis) — Department of Computer Science, University of Tromsø, Tromsø, Norway, fev. 2002.

ANDERSEN, A. et al. Security and middleware. In: *WORDS 2003*. Mexico: [s.n.], 2003.

ANDERSEN, A. et al. *A reflective component-based middleware in Python*. [S.l.], 1999. Submitted IPC8. Disponível em: <http://www.cs.uit.no/ aa/abstracts/andersen1999b.html>.

ANDERSEN, A.; BLAIR, G. S.; ELIASSEN, F. OOPP: A reflective component-based middleware. In: *NIK 2000*. Bodø, Norway: [s.n.], 2000. Disponível em: <http://www.cs.uit.no/ aa/abstracts/andersen2000c.html>.

ANDERSEN, A.; BLAIR, G. S.; ELIASSEN, F. A reflective component-based middleware with quality of service management. In: *PROMS 2000, Protocols for Multimedia Systems*. Cracow, Poland: [s.n.], 2000. Disponível em: <http://www.cs.uit.no/ aa/abstracts/andersen2000b.html>.

BATISTA, T. V.; CERQUEIRA, R.; RODRIGUEZ, N. Enabling reflection and reconfiguration in corba. In: *In Workshop Proceedings of the International Middleware Conference*. [S.l.: s.n.], 2003. p. 125–129.

BATISTA, T. V.; RODRIGUEZ, N. Dynamic reconfiguration of component-based applications. In: *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE)*. [S.l.: s.n.], 2000.

BERGMANS, L.; AKSIT, M. Aspects & crosscutting in layered middleware systems. In: *International Middleware Conference, Workshop Proceedings*. New York, USA: [s.n.], 2000. Disponível em: <http://www.comp.lancs.ac.uk/computing/rm2000/paper-list.htm>.

BERNSTEIN, P. A. Middleware: a model for distributed system services. *Commun. ACM*, ACM Press, v. 39, n. 2, p. 86–98, 1996. ISSN 0001-0782.

BLAIR, G. S. et al. Formal support for dynamic QoS management in the development of open component-based distributed systems. *IEE Proceedings – Software Engineering*, v. 148, n. 3, p. 83–92, jun. 2001. Special issue on generative and component-based software engineering. Disponível em: <http://www.cs.uit.no/ aa/abstracts/blair2001b.html>.

BLAIR, G. S. et al. The design and implementation of Open ORB v2. *IEEE Distributed Systems Online*, v. 2, n. 6, 2001. Disponível em: <http://www.cs.uit.no/ aa/abstracts/blair2001a.html>.

BLAIR, G. S. et al. An architecture for next generation middleware. In: *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*. London: Springer-Verlag, 1998. Disponível em: <citeseer.ist.psu.edu/blair98architecture.html>.

BRYANT, A.; FELDT, R. *AspectR - Simple aspect-oriented programming in Ruby*. 2002. Disponível em: <http://aspectr.sourceforge.net/>.

BURKE, B.; BROCK, A. *The Aspect Oriented Programming and JBoss tutorial*. 2003. Disponível em: <http://www.onjava.com/pub/a/onjava/2003/05/28/>.

CACHO, N.; BATISTA, T. Adaptação dinâmica no open-orb: detalhes de implementação. In: *23 Simpósio Brasileiro de Redes de Computadores*. [S.l.: s.n.], 2005. p. 495–508. ISBN 85-7669-022-5.

CACHO, N.; BATISTA, T. (Ed.). *Using AOP to Customize a Reflective Middleware: CoopIS, DOA, and ODBASE, OTM Confederated International Conferences, Agia Napa, Cyprus, 2005, Proceedings, Part I*, v. 3761 de *Lecture Notes in Computer Science*, (Lecture Notes in Computer Science, v. 3761). [S.l.]: Springer, 2005. ISBN 3-540-20498-9.

CACHO, N.; BATISTA, T.; FERNANDES, F. Aspectlua - a dynamic aop approach. *Journal of Universal Computer Science*, v. 11, n. 7, p. 1177–1197, 2005.

CACHO, N.; BATISTA, T.; FERNANDES, F. A lua-based aop infrastructure. In: *Journal of the Brazilian Computer Society - Special Issue on AOSD*. [S.l.: s.n.], 2006. v. 11, p. 7–20. ISSN 0104-6500.

CACHO, N. et al. Improving modularity of reflective middleware with aspect-oriented programming. In: *SEM '06: Proceedings of the 6rd International Workshop on Software Engineering and Middleware*. [S.l.]: ACM Press, 2006.

CACHO, N. et al. Composing design patterns: a scalability study of aspect-oriented programming. In: *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*. New York, NY, USA: ACM Press, 2006. p. 109–121. ISBN 1-59593-300-X.

CARON, J.; HERSCHER, S.; O'CONNOR, A. M. *CORBA in the palm of your hand whitepaper*. [S.l.]. Whitepaper.

COHEN, T.; GIL, J. Aspectj2ee = aop + j2ee. In: *ECOOP*. [S.l.: s.n.], 2004. p. 219–243.

COULSON, G. et al. The design of a highly configurable and reconfigurable middleware platform. *Commun. ACM*, ACM Press, v. 15, n. 2, p. 109–126, 2002.

DECHOW, D. R. Advanced separation of concerns for dynamic, lightweight languages. In: *Generative Programming and Component Engineering*. [S.l.: s.n.], 2003.

DRAKE, F. L. *Python Reference Manual*. 2003. Disponível em: <http://www.python.org/doc/current/ref/ref.html>.

DUMANT, B. et al. Jonathan: an open distributed processing environment in java. *Distributed Systems Engineering*, v. 6, n. 1, p. 3–12, 1999.

ELRAD, T. et al. Discussing aspects of aop. *Commun. ACM*, ACM Press, v. 44, n. 10, p. 33–38, 2001. ISSN 0001-0782.

FERNANDES, F.; BATISTA, T. Dynamic aspect-oriented programming: An interpreted approach. In: *Proceedings of the 2004 Dynamic Aspects Workshop (DAW04)*. [S.l.: s.n.], 2004. p. 44–50.

FERNANDES, F.; BATISTA, T.; CACHO, N. Exploring reflection to dynamically aspectizing corba-based applications. In: *ARM '04: Proceedings of the 3rd workshop on Adaptive and reflective middleware*. New York, NY, USA: ACM Press, 2004. p. 220–225. ISBN 1-58113-949-7.

FERNANDES, F.; BATISTA, T.; CACHO, N. Exploring reflection to dynamically aspectizing corba-based applications. In: *Proceedings of the 3rd workshop on Adaptive and reflective middleware*. [S.l.]: ACM Press, 2004. p. 220–225. ISBN 1-58113-949-7.

FIGUEIREDO, E. et al. Assessing aspect-oriented artifacts: Towards a tool-supported quantitative method. In: *9th ECOOP Workshop on Quantitative Approaches in OO Soft. Engineering (QAOOSE.05)*. [S.l.: s.n.], 2005.

FITZPATRICK, T. et al. Supporting adaptive multimedia applications through open bindings. In: *Proceedings of the International Conference on Configurable Distributed Systems*. [S.l.]: IEEE Computer Society, 1998. p. 128. ISBN 0-8186-8451-8.

GAL, A.; SCHRODER-PREIKSCHAT, W.; SPINCZYK, O. *AspectC++: Language Proposal and Prototype Implementation*. University of Magdeburg: [s.n.], 2001.

GAMMA, E.; HELM, R.; JOHNSON, R. *Design Patterns. Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley, 1995. (Addison-Wesley Professional Computing Series). GAM e 95:1 1.Ex.

GARCIA, A. et al. Modularizing design patterns with aspects: a quantitative study. In: *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*. New York, NY, USA: ACM Press, 2005. p. 3–14. ISBN 1-59593-042-6.

GARCIA, A. F. et al. Separation of concerns in multi-agent systems: An empirical study. In: *SELMAS*. [S.l.: s.n.], 2003. p. 49–72.

GOBEL, S. et al. The comquad component model: enabling dynamic selection of implementations by weaving non-functional aspects. In: *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. [S.l.]: ACM Press, 2004. p. 74–82. ISBN 1-58113-842-3.

GODIL, I.; JACOBSEN, H.-A. Horizontal decomposition of prevayler. In: *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*. [S.l.]: IBM Press, 2005. p. 83–100.

HANNEMANN, J.; KICZALES, G. *Design pattern implementation in Java and Aspect*. 2002. Disponível em: <citeseer.ist.psu.edu/hannemann02design.html>.

HANSSEN, O.; ELIASSEN, F. A framework for policy bindings. In: *DOA '99: Proceedings of the International Symposium on Distributed Objects and Applications*. Washington, DC, USA: IEEE Computer Society, 1999. p. 2. ISBN 0-7695-0182-6.

IERUSALIMSCHY, R. *Programming in Lua*. [S.l.]: Lua.org, 2003. ISBN 8590379817.

IERUSALIMSCHY, R.; FIGUEIREDO, L. H. de; FILHO, W. C. Lua an extensible extension language. *Softw. Pract. Exper.*, John Wiley Sons, Inc., v. 26, n. 6, p. 635–652, 1996. ISSN 0038-0644.

JRGENSEN, B. N. et al. Customization of object request brokers by application specific policies. In: *Middleware '00: IFIP/ACM International Conference on Distributed systems platforms*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2000. p. 144–163. ISBN 3-540-67352-0.

KELLY, R.; DIPALMA, L. Applying corba in a contemporary embedded military combat system. In: *OMG's Second Workshop on Real-time And Embedded Distributed Object Computing*. [S.l.: s.n.], 2001.

KICZALES, G. et al. Aspect-oriented programming. In: IT, M. A.; MATSUOKA, S. (Ed.). *Proceedings European Conference on Object-Oriented Programming*. Berlin, Heidelberg, and New York: Springer-Verlag, 1997. v. 1241, p. 220–242. Disponível em: <citeseer.ist.psu.edu/kiczales97aspectoriented.html>.

KON, F. et al. *2K: A Distributed Operating System for Dynamic Heterogeneous Environments*. Champaign, IL, USA, 1999.

KON, F. et al. The case for reflective middleware. *Commun. ACM*, ACM Press, v. 45, n. 6, p. 33–38, 2002. ISSN 0001-0782.

KON, F. et al. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In: *IFIP/ACM International Conference on Distributed systems platforms*. [S.l.]: Springer-Verlag New York, Inc., 2000. p. 121–143. ISBN 3-540-67352-0.

MILES, R. *Lazy Loading with Aspects*. 2004. Disponível em: <http://www.onjava.com>.

MORGENTHAL, J. P. *Microsoft COM+ Will Challenge Application Server Market*. 1999. Disponível em: <http://www.microsoft.com/com/wpaper/complus-appserv.asp>.

Object Management Group. *The Common Object Request Broker: Architecture and Specification*. 2.5. ed. [S.l.], September 2001.

OKAMURA, H.; ISHIKAWA, Y.; TOKORO, M. AL-1/D: A distributed programming system with multi-model reflection framework. In: *Proceedings of the Workshop on New Models for Software Architecture*. [S.l.: s.n.], 1992.

PARLAVANTZAS, N. et al. *Towards a Reflective Component Based Middleware Architecture*. 2000. Disponível em: <citeseer.csail.mit.edu/331827.html>.

PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, v. 15, n. 12, p. 1053–1058, 1972.

PAWLAK, R. et al. Jac: an aspect-based distributed dynamic framework. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 34, n. 12, p. 1119–1148, 2004. ISSN 0038-0644.

PUTRYCZ, E.; BERNARD, G. Using aspect oriented programming to build a portable load balancing service. In: *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*. [S.l.]: IEEE Computer Society, 2002. p. 473–480. ISBN 0-7695-1588-6.

SANT'ANNA, C. et al. On the reuse and maintenance of aspect-oriented software: An assessment framework. In: *Brazilian Symposium on Software Engineering (SBES'03)*. [S.l.: s.n.], 2003. p. 19–34.

SCHMIDT, D.; CLEELAND, C. *Applying Patterns to Develop Extensible ORB Middleware*. 1998. Disponível em: <citeseer.ifi.unizh.ch/schmidt98applying.html>.

SMITH, B. C. *Procedural Reflection in Programming Languages*. Tese (Doutorado) — Massachusetts Institute of Technology, 1982.

SZYPERSKI, C. *Component Software: Beyond Object-Oriented Programming*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0201745720.

TEAM, A. *The AspectJ Programming Guide*. 2002. Disponível em: <http://aspectj.org>.

TEAM, A. *The AspectJTM 5 Development Kit Developer's Notebook*. 2005. Disponível em: <http://www.eclipse.org/aspectj/doc/next/adk15notebook/index.html>.

THOMAS, D.; FOWLER, C.; HUNT, A. *Programming Ruby: A Pragmatic Programmer's Guide*. 2000. Disponível em: <http://www.rubycentral.com/book/>.

TRIPATHI, A. Challenges designing next-generation middleware systems. *Commun. ACM*, ACM Press, v. 45, n. 6, p. 39–42, 2002. ISSN 0001-0782.

TRUYEN, E. et al. *Aspects for Run-Time Component Integration*. 2000. Disponível em: <citeseer.ist.psu.edu/truyen00aspects.html>.

TRUYEN, E. et al. Dynamic and selective combination of extensions in component-based applications. In: *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2001. p. 233–242. ISBN 0-7695-1050-7.

WOLLRATH, A.; RIGGS, R.; WALDO, J. A distributed object model for the Java system. In: *2nd Conference on Object-Oriented Technologies & Systems (COOTS)*. USENIX Association, 1996. p. 219–232. Disponível em: <citeseer.ist.psu.edu/wollrath96distributed.html>.

ZHANG, C.; GAO, D.; JACOBSEN, H.-A. Generic middleware substrate through modelware. In: *Middleware*. [S.l.: s.n.], 2005. p. 314–333.

ZHANG, C.; GAO, D.; JACOBSEN, H.-A. Towards just-in-time middleware architectures. In: *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*. New York, NY, USA: ACM Press, 2005. p. 63–74. ISBN 1-59593-042-6.

ZHANG, C.; JACOBSEN, H.-A. *Aspectizing middleware platforms*. [S.l.], 2003. Technical Report CSRG-466.

ZHANG, C.; JACOBSEN, H.-A. Quantifying aspects in middleware platforms. In: *Proceedings of the 2nd international conference on Aspect-oriented software development*. [S.l.]: ACM Press, 2003. p. 130–139. ISBN 1-58113-660-9.

ZHANG, C.; JACOBSEN, H.-A. Resolving feature convolution in middleware systems. In: *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*. [S.l.]: ACM Press, 2004. p. 188–205. ISBN 1-58113-831-9.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo