Tássia Aparecida Vieira de Freitas

Métricas para Avaliação de Sistemas de Middleware Orientado a Aspectos e Aplicação em um Sistema de Monitoramento de Poços de Petróleo

Natal - RN

2009

Tássia Aparecida Vieira de Freitas

Métricas para Avaliação de Sistemas de Middleware Orientado a Aspectos e Aplicação em um Sistema de Monitoramento de Poços de Petróleo

Dissertação submetida ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como parte dos requisitos para a obtenção do grau de Mestre em Sistemas e Computação

Orientador:
Profa. Dra. Thaís Vasconcelos Batista

Programa de Pós-Graduação em Sistemas e Computação
Departamento de Informática e Matemática Aplicada
Centro de Ciências Exatas e da Terra
Universidade Federal do Rio Grande do Norte

Natal - RN

2009

Divisão de Serviços Técnicos

Catalogação da Publicação na Fonte. UFRN / Biblioteca Central Zila Mamede

Freitas, Tássia Aparecida Vieira de.
Métricas para avaliação de sistemas de middleware orientado a aspectos e aplicação em um sistema de monitoramento de poços de petróleo / Tássia Aparecida Vieira de Freitas. – Natal, RN, 2009.
149 f.

Orientadora: Thaís Vasconcelos Batista.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Departamento de Informática e Matemática Aplicada. Programa de Pós-Graduação em Sistemas e Computação.

1. Sistemas de middleware – Dissertação. 2. Métricas de software – Dissertação. 3. Desenvolvimento de software orientado a aspectos – Dissertação. I. Batista, Thaís Vasconcelos. II. Universidade Federal do Rio Grande do Norte. III. Título.

RN/UF/BCZM CDU
004.75(043.3)

Esse trabalho foi desenvolvido com o apoio da Agência Nacional do Petróleo, através do Programa de Formação de Recursos Humanos Nº22 - Formação em Geologia, Geofísica e Informática do Setor do Petróleo e Gás Natural na UFRN - com especialização em Sistemas em Tempo Real para Otimização e Automação do Setor do Petróleo e Gás Natural.

Agradecimentos

A Deus.

À professora Thaís, pelo incentivo e orientação.

Ao PRH-22/ANP, pelos auxílios concedidos.

Aos que sempre estiveram ao meu lado: minha mãe Cida, meu pai Pedro, minha madrinha Maria e minha tia Socorro e aos meus queridos colegas e amigos.

Resumo

Atualmente, há diversas implementações de sistemas de *middleware* orientado a aspectos que aproveitam o suporte a modularização do paradigma de orientação a aspectos. Apesar desses trabalhos sempre apresentarem uma avaliação do *middleware* de acordo com algum atributo de qualidade, não há ainda um conjunto de métricas especificamente definidas para avaliá-los de forma abrangente, seguindo vários atributos de qualidade.

Este trabalho tem como objetivo propor um conjunto de métricas para avaliação de sistemas de *middleware* orientado a aspectos em diferentes fases de desenvolvimento: *design*, refatoração, implementação e execução. O trabalho apresenta as métricas e como elas são aplicadas em cada uma das fases de desenvolvimento. O conjunto é composto por métricas associadas a propriedades estáticas (modularidade, manutenibilidade, reusabilidade, flexibilidade, complexidade, estabilidade e tamanho) e dinâmicas (desempenho e consumo de memória). Tais métricas são baseadas em abordagens existentes de avaliação de sistemas orientados a aspectos e a objetos.

As métricas propostas são utilizadas no contexto do OiL (*Orb in Lua*), um *middleware* baseado em CORBA e implementado em Lua, e AO-OiL, uma refatoração do OIL que segue uma arquitetura de referência para sistemas de *middleware* orientados a aspectos. O estudo de caso executado no OiL e no AO-OiL é um sistema de monitoramento de poços de petróleo. Esse trabalho apresenta ainda a ferramenta CoMeTA-Lua para automatizar a coleta das métricas de tamanho e acoplamento em código-fonte Lua.

Área de Concentração: Sistemas Distribuídos.
Palavras-chaves: Sistemas de *Middleware*; Métricas de Software; Desenvolvimento de Software Orientado a Aspectos.

Abstract

Nowadays, there are many aspect-oriented middleware implementations that take advantage of the modularity provided by the aspect oriented paradigm. Although the works always present an assessment of the middleware according to some quality attribute, there is not a specific set of metrics to assess them in a comprehensive way, following various quality attributes.

This work aims to propose a suite of metrics for the assessment of aspect-oriented middleware systems at different development stages: design, refactoring, implementation and runtime. The work presents the metrics and how they are applied at each development stage. The suite is composed of metrics associated to static properties (modularity, maintainability, reusability, flexibility, complexity, stability, and size) and dynamic properties (performance and memory consumption). Such metrics are based on existing assessment approaches of object-oriented and aspect-oriented systems.

The proposed metrics are used in the context of OiL (Orb in Lua), a middleware based on CORBA and implemented in Lua, and AO-OiL, the refactoring of OIL that follows a reference architecture for aspect-oriented middleware systems. The case study performed in OiL and AO-OiL is a system for monitoring of oil wells. This work also presents the CoMeTA-Lua tool to automate the collection of coupling and size metrics in Lua source code.

Area of Concentration: Distributed Systems.
Key Words: Middleware Systems; Software Measurement; Aspect-Oriented Software Development.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de abreviaturas e siglas

Ca	*Afferent Coupling*
CAE	*Coupling on Advice Execution*
CBO	*Coupling Between Object classes*
CDA	*Coupling Degree of an Aspect*
CDC	*Concern Diffusion over Components*
CDLOC	*Concern Diffusion over Lines of Code*
CDO	*Concern Diffusion over Operations*
Ce	*Efferent Coupling*
CIM	*Coupling on Intercepted Modules*
DAC	*Data Abstraction Coupling*
DIT	*Depth of Inheritance Tree*
DSOA	Desenvolvimento de Software Orientado a Aspectos
I	*Instability*
LCC	*Lack of Concern-based Cohesion*
LOC	*Lines Of Code*
NOA	*Number of Attributes*
NOC	*Number of Children*
RFC	*Response for a Class*
VS	*Vocabulary Size*
WOC	*Weighted Operations per Component*

1 Introdução

A construção de sistemas distribuídos apresenta grandes desafios. Um deles é garantir fatores como atomicidade, consistência, independência, durabilidade e segurança. Outro importante desafio é dar suporte à execução desses sistemas em ambientes heterogêneos, ou seja, em plataformas e sistemas operacionais diversos. Nesse contexto, surgiram os sistemas de *middleware*: plataformas que formam uma camada intermediária entre, sistema operacional, redes, e as aplicações. O *middleware* oferece um conjunto de serviços ao desenvolvedor de forma a tornar transparente aspectos referentes à distribuição e heterogeneidade que aumentam a complexidade da construção de sistemas distribuídos (BERNSTEIN, 1996). Portanto, tipicamente um *middleware* é projetado para prover um amplo conjunto de características visando auxiliar a construção de sistemas distribuídos. Entretanto, prover todas essas funcionalidades deixa o código do *middleware* excessivamente grande e complexo. Outro aspecto negativo ainda é a dificuldade de definir arquivos de configuração para esse *middleware*. Tais arquivos contêm especificações declarativas do contexto de execução requisitados pelos componentes.

A alternativa para essa questão é especializar o *middleware*. Há duas formas de se fazer isso. Na fase de *design* do *middleware*, define-se o conjunto de características que devem estar presentes. Isso faz com que o *middleware* sirva apenas a um domínio de aplicação. A segunda alternativa é construir um núcleo base para o *middleware*, contendo somente serviços que são comuns a todos os domínios de aplicação. Os serviços específicos de um domínio são então adicionados em uma segunda fase, fazendo com que características especializadas sejam incrementalmente agregadas ao núcleo para prover funcionalidades necessárias a um domínio específico.

Agregar serviços específicos de um domínio a um núcleo favorece o reuso e diminui o tempo de desenvolvimento do *middleware*. Assim, torna-se uma estratégia interessante para o desenvolvimento de sistemas de *middleware*. Para isso, uma tecnologia que tem sido bastante utilizada é o *paradigma de desenvolvimento de software orientado a aspectos* (DSOA), além do uso de reflexão e *framework* de componentes. Em (LOUGHRAN et

al., 2008), são encontradas diversas tecnologias de *middleware* que usam um modelo de programação orientado a aspectos.

O paradigma de DSOA tem como objetivo dar suporte à separação de conceitos transversais — propriedades que não podem ser encapsuladas em um único componente e entrecortam diversas partes do sistema. Nesse paradigma, os sistemas são separados em componentes e aspectos. Componentes são elementos que implementam os conceitos básicos da aplicação. Aspectos modularizam os conceitos transversais de forma a prover sua separação do código dos componentes. Componentes e aspectos são então definidos separadamente e combinados em tempo de compilação ou de execução para compor o sistema. Os aspectos são ligados aos componentes em pontos de junção previamente definidos no código dos componentes (KICZALES et al., 1997). O principal propósito em aplicar DSOA na construção de *middleware* é conseguir flexibilidade e modularidade com aceitável impacto no desempenho(LOUGHRAN et al., 2008).

Para mensurar um *middleware* construído segundo o paradigma DSOA, em geral, os trabalhos existentes avaliam apenas o desempenho do *middleware* (POPOVICI; GROSS; ALONSO, 2002), (POPOVICI; ALONSO; GROSS, 2003) e a quantidade de memória (*footprint*) ocupada pelo mesmo. Com o crescente número de implementações desse tipo de *middleware*, faz-se necessário realizar uma avaliação mais ampla, levando em conta um conjunto de métricas bem definido, para comparar e analisar diversas propriedades, de forma a avaliar os reais benefícios que a orientação a aspectos pode conferir a um sistema de *middleware*. Diversos sistemas de *middleware* apresentados na literatura como Jboss AOP (JBOSSAOP, 2008), PRISMA (PEREZ et al., 2003) e Lasagne (TRUYEN et al., 2001) não são avaliados através de métricas. Uma exceção é o trabalho de Zhang e Jacobsen (ZHANG; JACOBSEN, 2003), no qual são apresentadas métricas para avaliar a refatoração de *middleware* durante a fase de implementação. No entanto, além de não fornecer métricas para a avaliação em fase de *design*, a refatoração é avaliada somente em termos de tamanho, complexidade, acoplamento e tempo de resposta. É necessária a avaliação de propriedades como separação de conceitos e flexibilidade, por exemplo, que são características importantes em sistemas de *middleware* orientado a aspectos. Ainda não há uma solução de um conjunto de métricas que possa ser usado para avaliar o sistema na fase de *design* e na fase de implementação, além de permitir avaliar o processo de refatoração e comparar sistemas de *middleware* similares.

No contexto de *middleware* orientado a aspectos, métricas podem ser aplicadas em diversas fases:

(i) fase de *design*: para comparar soluções similares e selecionar a mais adequada aos requisitos do sistema em questão. Não há trabalhos com métricas para a fase de *design* específicas para *middleware* orientado a aspectos. Métricas para sistemas orientados a aspectos que podem servir de base para definição de métricas para essa fase podem ser encontradas em (SANT'ANNA et al., 2003);

(ii) fase de implementação: para avaliar quão boa é a solução de implementação em termos de propriedades como acoplamento, complexidade. Métricas para sistemas orientados a aspectos que podem ser utilizadas nessa fase estão presentes em (SANT'ANNA et al., 2003) e (ZHAO, 2002);

(iii) fase de refatoração: para quantificar as mudanças na complexidade da estrutura do sistema refatorado. No trabalho (ZHANG; JACOBSEN, 2003) há métricas para a fase de refatoração;

(iv) tempo de execução: para avaliar se a utilização de aspectos compromete o desempenho do sistema. Exemplos de métricas para essa fase são encontrados em (ZHANG; JACOBSEN, 2003), (POPOVICI; GROSS; ALONSO, 2002) e (POPOVICI; ALONSO; GROSS, 2003);

1.1 Objetivos

Este trabalho tem como objetivo a proposição de uma lista de propriedades associada a um conjunto de métricas para avaliação de sistemas de *middleware* orientados a aspectos. São avaliadas propostas existentes, como (ZHANG; JACOBSEN, 2003) que apresenta um conjunto de métricas específicas para o processo de refatoração de *middleware* orientado a aspectos. (DRIVER, 2002), (SANT'ANNA et al., 2003), (ZHAO, 2003), (ZHAO, 2002) apresentam métricas para sistemas de software orientados a aspectos. O conjunto de métricas proposto baseia-se nos trabalhos citados, como também em (BRIAND; DALY; WUEST, 1999) e (CHIDAMBER; KEMERER, 1994) que apresentam métricas para sistemas de software orientados a objetos. Pretende-se comparar propostas existentes sobre avaliação de software e, seja selecionando métricas de outros trabalhos ou adaptando-as a aspectos, construir um conjunto amplo para avaliação de *middleware* orientado a aspectos. O conjunto de métricas a ser desenvolvido tem como objetivo avaliar *middleware* no intuito de:

(i) comparar pares de *middleware* semelhantes;

(ii) comparar versões orientadas a objetos e a aspectos de um mesmo *middleware* e

(iii) avaliar o comportamento dinâmico – desempenho e consumo de memória – de um *middleware*.

A comparação de pares de *middleware* semelhantes vem a ser uma facilidade para desenvolvedores de aplicações a fim de escolher o *middleware* mais adequado à sua aplicação. A comparação entre versões AO e OO de um mesmo *middleware* avalia se a refatoração trouxe mais vantagens ou desvantagens e em quais aspectos. Pode-se ainda avaliar diferentes versões AO de um mesmo *middleware* para verificar se as atualizações efetuadas trouxeram benefícios ou não. O conjunto de métricas proposto deve ainda caracterizar sistemas de *middleware* quanto ao seu desempenho e ao momento da carga de aspectos: se a carga se dá em tempo de execução, de forma estática ou em tempo de carga. O desenvolvedor de *middleware* pode então melhor avaliar em que momento deve carregar os aspectos.

O conjunto de métricas deve avaliar 4 fases do ciclo de vida de um sistema de *middleware*:

(i) na fase de refatoração, processo em que um *middleware* é refatorado em componentes e aspectos, avaliam-se propriedades como separação de conceitos, acoplamento e tamanho antes e depois da refatoração para comparação de soluções;

(ii) a fase de *design*, que também é avaliada com esse conjunto de métricas para verificar fatores como separação de conceitos, acoplamento e tamanho durante a construção do projeto do *middleware*;

(iii) a fase de implementação, em que são avaliadas propriedades como separação de conceitos, acoplamento, reusabilidade, como na fase de *design*, mas através de métricas apropriadas para o nível de código-fonte, e

(iv) a fase de execução, em que se avalia o comportamento dinâmico do *middleware*.

Como a coleta de métricas é uma tarefa repetitiva, este trabalho apresenta também como objetivo a construção da ferramenta COMETA-Lua (Ferramenta para Coleta de Métricas de Tamanho e Acoplamento em Lua), para automatizar a aquisição das métricas das propriedades de tamanho e acoplamento, que demandam maior trabalho na coleta. Como os sistemas de *middleware* usados como estudo de caso foram desenvolvidos na linguagem de *script* Lua, a ferramenta foi direcionada a analisar código Lua. No entanto, o conjunto de métricas proposto neste trabalho é genérico, e pode ser aplicado no contexto de qualquer linguagem de programação.

De forma a validar a aplicabilidade do conjunto de métricas proposto, elas são aplicadas ao *middleware* OiL (*ORB in Lua*) (MAIA et al., 2006) e a sua versão OA, o AO-OiL (SILVA et al., 2008). Para a coleta das métricas dinâmicas do estudo de caso, é usado um sistema de monitoramento de poços de petróleo. Os resultados das métricas são coletados e analisados de forma a levantar pontos fracos e fortes das duas abordagens. Os resultados alcançados com a aplicação das métricas no estudo de caso comprovam a eficiência das métricas para avaliação de refatoração de *middleware*, como também a aplicabilidade da ferramenta como agente facilitador no processo de coleta das métricas. A relação entre propriedades e métricas estabelecida por este trabalho permite que se analisem melhorias nas propriedades de um sistema, a partir de valores obtidos para as métricas.

É importante observar que não é objetivo deste trabalho definir novas métricas. A partir da lista de propriedades, foram selecionadas métricas de trabalhos bem conhecidos dessa área para compor o conjunto de métricas. Cada métrica é associada a avaliação de uma propriedade específica. Como algumas dessas métricas são definidas em seus trabalhos originais para sistemas OO, elas foram adaptadas neste trabalho para poderem ser aplicadas em sistemas OA; as métricas selecionadas de trabalhos para sistemas OA apenas foram redefinidas de forma a seguir a terminologia deste trabalho.

1.2 Principais Contribuições

Este trabalho apresenta uma lista de propriedades relevantes para a avaliação de sistemas de *middleware* OA e um conjunto de métricas associado a essas propriedades, de forma a tornar o processo de avaliação mais sistemático. Baseando-se em diversas abordagens da área de métricas, o conjuntos de métricas se apresenta como meio de avaliação e comparação de sistemas de *middleware* OA em largura e profundidade.

Para validação do conjunto de métricas, ele foi aplicado à refatoração do middleware OiL. Para coletar parte das métricas, propõe-se a ferramenta CoMeTA-Lua. A validação permite ilustrar o uso do conjunto de propriedades, métricas e da ferramenta propostos, além de mostrar a utilidade de uma abordagem como essa na avaliação de sistemas de *middleware*.

1.3 Estruturação

Este trabalho está estruturado da seguinte forma. O capítulo 2 apresenta conceitos básicos sobre DSOA, *middleware* orientado a aspectos e métricas de software. O capítulo 3 discute os trabalhos relacionados. O conjunto de métricas proposto por este trabalho é apresentado no capítulo 4. A ferramenta CoMeTA-Lua é apresentada no capítulo 5. O capítulo 6 apresenta os sistemas de *middleware* OiL e AO-OiL e a aplicação utilizada para a avaliação de desempenho desses sistemas. O capítulo 6 ainda mostra a validação do conjunto de métricas proposto com sua aplicação no OiL e no AO-OiL. O capítulo 7 contém as contribuições deste trabalho e sugestões de trabalhos futuros.

2 Conceitos Básicos

Esse capítulo é destinado à fundamentação teórica. A seção 2.1 apresenta o paradigma de desenvolvimento de software orientado a aspectos. Conceitos sobre sistemas de middleware orientado a aspectos são mostrados na seção 2.2. A seção 2.3 é destinada à conceituação e classificação de métricas de software.

2.1 DSOA

O desenvolvimento de Software Orientado a Aspectos (DSOA) (KICZALES et al., 1997) trata da aplicação do paradigma de orientação a aspectos no desenvolvimento de software, empregando a idéia de separação de conceitos transversais em todos as fases de desenvolvimento de software. O DSOA oferece suporte para separar *componentes*, unidades de decomposição funcional do sistema, de *aspectos*, conceitos que afetam diversos componentes e originalmente estão espalhados e entrelaçados em vários pontos do sistema. O DSOA provê mecanismos para abstrair e compor componentes e aspectos de forma a produzir o sistema como um todo.

Segundo Elrad et al (ELRAD; FILMAN; BADER, 2001) separar os múltiplos conceitos em sistemas de programação promete sistemas com evolução mais simples, mais compreensíveis, com maior adaptabilidade, capacidade de customização e facilidade de reuso.

No contexto de DSOA, a nível de programação, há várias linguagens de programação orientada a aspectos que, geralmente, são extensões de linguagens orientadas a objetos já existentes que apresentam elementos para representar os conceitos transversais. A seguir, apresentam-se definições dos elementos de AspectJ — por ser a linguagem de programação mais amplamente utilizada e as outras linguagens basearem-se fortemente em AspectJ — retiradas de (KICZALES et al., 2001).

- *join points*: pontos bem definidos na execução de um programa. Podem ser conside-

rados como nós em um grafo de chamada de objetos em tempos de execução. Esses nós incluem pontos em que um objeto recebe uma chamada de método e pontos em que um campo de um objeto é referenciado. A localização do *join point* no texto do programa é denominada *join point shadow*;

- *pointcut*: conjunto de *join points* e, opcionalmente, alguns valores do contexto de execução desses *join points*. Programadores podem usar designadores de *pointcuts* para definir *pointcuts* de forma a casá-los com certos *join points* em tempo de execução;

- *advices*: define comportamento adicional a ser executado ao se encontrar os *join points*. *Advices* contém a funcionalidade que deve ser executada em cada *join point* de um *pointcut*;

- *weaver*: compiladores ou interpretadores de linguagens de aspectos que compõem a implementação de aspectos com outros módulos. Ele é o responsável pelo processo de composição.

2.2 *Middleware* Orientado a Aspectos

Sistemas de *middleware* consistem de um conjunto de serviços que facilitam o desenvolvimento de aplicações distribuídas em ambiente computacional heterogêneo. Esses serviços precisam atender a requisitos específicos de vários domínios de aplicação e de conceitos relativos à computação distribuída. Oferecer todos esses serviços implica em alto consumo de memória e de recursos computacionais, além de aumentar a complexidade do *middleware* (ZHANG; JACOBSEN, 2003).

Técnicas de DSOA são utilizadas em sistemas de *middleware* a fim de otimizar sua arquitetura. Define-se então um núcleo básico para o *middleware* que é estendido por meio da adição incremental de aspectos. Os aspectos tratam os conceitos ortogonais aos elementos básicos do *middleware*. O *middleware* deixa de ser uma caixa preta e passa a ser uma composição flexível de diferentes serviços (LOUGHRAN et al., 2008).

O núcleo básico do sistema de *middleware* é geralmente projetado usando o desenvolvimento orientado a componentes para conferir:

1. alto grau de abstração no *design*, implementação, distribuição e gerenciamento do sistema;

2. facilidade de configuração e reconfiguração em tempo de execução;

3. reuso de software.

DSOA pode ser utilizada em 4 camadas distintas da estrutura do *middleware* (LOUGHRAN et al., 2008). São elas:

(a) infra-estrutura hospedeira do *middleware*: utiliza-se técnicas de desenvolvimento orientado a aspectos para construir a infra-estrutura hospedeira do *middleware*. Um exemplo de infra-estrutura hospedeira são os sistemas operacionais e as máquinas virtuais. Mesmo que o sistema operacional não faça parte do *middleware*, seu desempenho tem impacto crítico nas camadas do *middleware*. Usa-se DSOA nessa camada para facilitar a flexibilidade. Exemplos de projetos nesse sentido são o LINUX e o OASIS (GIBBS; COADY, 2004), que utilizam aspectos para modularizar conceitos transversais sem descuidar do desempenho;

(b) *middleware* de distribuição: utiliza-se técnicas de DSOA para construir a camada de distribuição do *middleware*. Essa camada deve prover transparência de distribuição para a aplicação. Exemplos são ORB (*Object Request Broker*) e sistema de mensagens, que precisam de suporte a diferentes protocolos de transporte, conversão de tipos de dados e recursos relacionados a desempenho, como *caching*. O projeto DADO (*Distributed Adaplets for Distributed Objects*) (WOHLSTADTER; JACKSON; DEVANBU, 2003) implementa segurança e *caching* como aspectos dinâmicos, utilizando DSOA para conseguir flexibilidade em tempo de execução;

(c) serviços de *middleware*: o principal objetivo de plataformas de *middleware* é prover serviços para aplicações. Esses serviços incluem transação e persistência, além de segurança. Utiliza-se DSOA para construir esses serviços. Um exemplo de aplicação de DSOA nesse nível é o projeto JBOSS AOP que possui uma extensa biblioteca de aspectos usada para oferecer serviços às aplicações como também para a implementação do próprio *middleware*;

(d) serviços de *middleware* de domínio específico: nessa camada são oferecidos serviços voltados para um domínio específico. Utiliza-se DSOA nessa camada para agregar novos serviços para novas aplicações a um sistema.

Além disso, é preciso de uma linguagem de definição de *pointcuts* para determinar em quais pontos os aspectos devem ser inseridos no núcleo básico do sistema (LOUGHRAN et al., 2008). Aspectos como distribuição, persistência, segurança etc. podem ser adicionados ao núcleo básico do sistema de *middleware* para customizá-lo. As-

sim, é possível customizar o *middleware* de forma que ele apresente somente os serviços necessários a uma determinada aplicação. Aspectos podem ser adicionados em tempo de compilação, de carga ou de execução.

Aspectos adicionados em tempo de execução são chamados de aspectos dinâmicos e oferecem maior flexibilidade para a aplicação, uma vez que podem ser carregados e/ou retirados a qualquer momento da execução. No entanto, essa flexibilidade tem um custo. Aspectos dinâmicos podem resultar em uma queda de desempenho, ocasionada pelo *overhead* de adição de aspectos. É preciso avaliar a relação custo-benefício de sua utilização.

2.3 Métricas

2.3.1 Conceituação e Classificação de Elementos Relacionados ao Processo de Medição

A conceituação e classificação de elementos relacionados ao processo de medição apresentadas nessa subseção foi retirada de (FENTON, 1994).

Medição é o processo através do qual números ou símbolos são designados a atributos de entidades do mundo real de forma que as descreva de acordo com regras bem definidas. Uma entidade pode ser um objeto, uma pessoa ou uma especificação de software. Um atributo é uma propriedade da entidade, como tamanho ou funcionalidade. A atribuição numérica deve obedecer observações intuitivas e empíricas.

Um modo de tratar o problema da atribuição numérica é a construção de um modelo para as entidades sendo medidas. Um modelo reflete um ponto de vista específico. Por exemplo, mesmo uma medição simples do número de linhas de código necessita de um modelo de programa que defina número de linhas de forma não-ambígua. Na entidade de medição de software, há três classes principais de entidades de interesse:

- processos: quaisquer atividades relacionadas a software que ocorrem ao longo do tempo;
- produtos: quaisquer artefatos, *deliverables* ou documentos resultantes de processos;
- recursos: itens que servem de entrada aos processos.

Quanto a atributos de processos, produtos ou recursos, pode-se classificá-los em *externos* e *internos*. Atributos internos são medidos em termos dos processos, produtos e

recursos. Por exemplo, tamanho é atributo interno de um programa de software. Atributos externos de processos, produtos ou recursos são medidos considerando-se como esses processos, produtos ou recursos se relacionam com outras entidades do seu ambiente, como, por exemplo, a confiabilidade de um programa.

Em relação a medição, ela é classificada em dois tipos. A medição direta de um atributo é medição que não depende da medição de qualquer outro atributo. Em contrapartida, a medição indireta de um atributo é a medição que envolve a medição de um ou mais outros atributos.

Há ainda a classificação para métricas de software. Podem ser de três tipos:

(i) modelagem de custo: refere-se geralmente à predição de atributos de esforço ou tempo requerido para o processo de desenvolvimento, normalmente aplicada a especificação detalhada de implementação;

(ii) modelos de qualidade de software e modelos de confiabilidade: modelos de qualidade quebram qualidade em fatores, critérios e métricas e propõem relacionamentos entre eles. Fatores de qualidade geralmente correspondem a atributos de produtos externos; critérios, a atributos de produtos internos ou atributos de processo e as métricas, a medições de atributos internos. Na maioria dos casos, o relacionamento entre eles é subjetivo. Confiabilidade está relacionada a atributos de produto externos de alto nível que aparecem em modelos de qualidade. Esse tipo de atributo é importante para o produto software executável;

(iii) Ciência de Software de Halstead: medições de atributos de programa internos que refletem diferentes pontos de vista de tamanho, por exemplo, comprimento e vocabulário.

Classificando essa proposta de dissertação de acordo com o exposto acima, as medições são diretas e indiretas e avaliam atributos externos e internos de entidades classificadas como produtos. O trabalho proposto compõe-se de uma lista de propriedades, composta por atributos internos e externos, que se associam a outras propriedades e fatores. Portanto, nosso conjunto de métricas, embora seja semelhante, não pode ser classificado como um modelo de qualidade. As relações propriedade-propriedade, propriedade-fator e fator-fator baseiam-se em afirmações colhidas em diversos trabalhos na área de avaliação de software. Propriedades e fatores são avaliados pela aplicação de métricas diretas e/ou indiretas a atributos externos e/ou internos de produto de software. Uma das propriedades, tamanho, segue a categoria ciência de software de Halstead, pois as métricas avaliam mais de uma dimensão do tamanho de um produto como comprimento (linhas de código)

e vocabulário (número de atributos).

2.3.2 Métricas no Contexto de *Middleware* Orientado a Aspectos

Métricas de software é um modo efetivo de fornecer evidências empíricas de que o uso de uma abstração de um determinado paradigma de desenvolvimento pode melhorar o entendimento das diferentes dimensões da complexidade de software. Métricas auxiliam o processo de estudo empírico de uma abordagem levantando, através de valores atribuídos a atributos de software, pontos fortes e fracos da abordagem. Métricas são ainda mais efetivas quando estão associadas a algum framework de avaliação para que as pessoas envolvidas no estudo possam entender e interpretar os significados dos dados coletados (SANT'ANNA et al., 2003).

Nos sistemas de *middleware* orientados a aspectos, assim como em outros sistemas orientados a aspectos, as métricas também auxiliam o processo de tomada de decisões. Abaixo segue algumas das questões que podem ser melhor avaliadas com a aplicação de métricas:

1. se a refatoração de um *middleware* é satisfatória;
2. se a utilização de aspectos é recomendada;
3. em que partes do código do *middleware* utilizar aspectos;
4. em que momento deve-se carregar os aspectos junto ao código base do *middleware*.

Segundo (SANT'ANNA et al., 2003), a definição de métricas adequadas para software orientado a aspectos deve satisfazer os seguintes requisitos:

Requisito 1: medir atributos de software bem conhecidos como separação de conceitos, acoplamento, coesão e tamanho;

Requisito 2: apoiar-se tanto quanto possível em métricas tradicionais e em extensões de métricas orientadas a objetos para orientadas a aspectos, pois as abstrações da orientação a aspectos estendem o conjunto de abstrações da orientação a objetos;

Requisito 3: capturar diferentes dimensões de acoplamento e coesão do software orientado a aspectos;

Requisito 4: suportar a identificação de benefícios e desvantagens no uso de aspectos em um projeto de software quando comparado com uma solução orientada a objetos para o mesmo problema.

O conjunto de métricas proposto neste trabalho define uma lista de propriedades e fatores com métricas associadas. A associação entre métricas e propriedades facilita a avaliação das propriedades e a interpretação dos resultados. O conjunto de métricas será construído seguindo os requisitos de métricas para software orientado a aspectos do trabalho (SANT'ANNA et al., 2003), além de considerar conceitos e propostas de outros trabalhos como veremos no capítulo 4.

3 Trabalhos Relacionados

Nesse capítulo, são apresentados diversos trabalhos que propõem métricas para software. Por haver uma grande quantidade de trabalhos relacionados a esse tema, apenas os trabalhos mais relevantes para o trabalho proposto serão mostrados. Ficaram de fora deste capítulo, por exemplo, os trabalhos (BARTOLOMEI et al., 2006), (ZHAO; XU, 2004), (ZHAO, 2003) e (ZHAO, 2002).

3.1 Um Framework Unificado para Medidas de Acoplamento em Sistemas Orientados a Objetos

O trabalho de Briand et al (BRIAND; DALY; WUEST, 1999) propõe um *framework* de avaliação de acoplamento para sistemas de software orientado a objetos. Os autores fazem um apanhado das métricas sobre acoplamento para sistemas OO disponíveis na literatura e propõem um *framework* que engloba todas as propostas. Eles apresentam uma abordagem rigorosa, com definições formais e propriedades para as métricas.

As abordagens são comparadas quanto a: (i) o tipo de acoplamento; (ii) a força do acoplamento; (iii) se o acoplamento é importado ou exportado; (iv) se o acoplamento é direto ou indireto; (v) estabilidade da classe servidora; (vi) se o acoplamento é a nível de classe ou de objeto.

O *tipo de acoplamento* refere-se à forma de interação que origina o acoplamento. Os tipos mais considerados são acoplamento por invocação a métodos e referências a atributos.

A *força do acoplamento* entre duas classes é determinado por dois aspectos: a freqüência de conexões entre classes e os tipos de conexões (tabela 3.1) entre classes. A métrica para o primeiro aspecto define como contar a freqüência das conexões entre classes. O segundo justifica-se porque diferentes tipos de acoplamento têm forças diferentes. A maioria das abordagens define uma ordem de força para tipos de acoplamento diferentes.

A *direção do acoplamento* separa acoplamento entre importado e exportado, introduzindo uma relação de cliente-servidor entre as classes. A classe cliente usa (importa serviço) e a classe servidora está sendo usada (exporta serviço). A classe que principalmente importa serviços pode ser difícil de reusar em outro contexto porque ela depende de outras classes. A classe que exporta serviços é crítica. Se ela contiver falhas vai propagá-las para o restante do sistema mais facilmente.

O *acoplamento direto e indireto* leva em consideração a transitividade da relação de interação. Se uma classe c1 usa uma classe c2, que por sua vez usa uma classe c3, c1 indiretamente usa c3. Um defeito ou modificação em c3, afeta diretamente c2 e indiretamente c3. Esse acoplamento pode produzir uma grande cadeia de conexões indiretas.

A *estabilidade da classe servidora* refere-se a classes que não são propensas a sofrer modificações: uma classe que usa uma classe estável é melhor que usar uma instável, uma vez que se modifica a classe servidora, provavelmente será necessário modificar a classe cliente. Esse item é pouco explorado devido a dificuldade de se definir se uma possui estabilidade ou não. É um conceito bastante subjetivo.

O *acoplamento* pode ser analisado a nível de classe e a nível de objeto. O acoplamento a nível de classe pode ser avaliado a partir de uma análise estática dos documentos de *design* ou do código fonte. O acoplamento a nível de objeto depende também da estrutura concreta do objeto em tempo de execução, que é influenciada pelos dados de entrada do sistema. Assim, o acoplamento a nível de objeto é determinado pelo *design*, código fonte e ainda dados de entrada em tempo de execução.

Uma vez definidos os critérios de comparação das abordagens disponíveis na literatura, os autores definem o *framework*, com o objetivo de comparar e selecionar métricas para um determinado objetivo de medição. Os critérios definidos para o *framework* serão apresentados a seguir.

- Tipos de conexão: os tipos de conexão são mostrados na tabela 3.1. A segunda e a terceira colunas definem os elementos que participam do acoplamento. A quarta define os mecanismos que constituem acoplamento.

- Local de Impacto: nesse ponto, decide-se entre distinguir ou não acoplamento exportado e importado. Esse item é interessante quando se tratar de reuso de aspectos.

- Granularidade: a granularidade consiste em determinar o nível de detalhe de sua aplicação. A granularidade é determinada por dois fatores: o domínio das métricas,

Tabela 3.1: Tipos de conexão de acoplamento.

	Elemento 1	Elemento 2	Mecanismo de acoplamento
1	atributo a de uma classe c	classe d, com c $\neq$ d	classe d é do tipo de c
2	método m de uma classe c	classe d, com c $\neq$ d	classe d é do tipo de um parâmetro de m ou tipo de retorno de m
3	método m de classe c	classe d, com c $\neq$ d	classe d é o tipo de uma variável local de m
4	método m de uma classe c	classe d, com c $\neq$ d	classe d é do tipo de um parâmetro de um método invocado por m
5	método m de uma classe c	atributo a de uma classe d, com c $\neq$ d	m referencia a
6	método m de uma classe c	método m' de uma classe d, com c $\neq$ d	m invoca m'
7	classe c	classe d, com c $\neq$ d	relacionamentos de alto-nível entre classes, tais como "usa" e "consiste em"

Tabela 3.2: Contagem de conexões a nível de atributo e método.

	Descrição	Exemplo de acoplamento de importação	Exemplo de acoplamento de exportação
A	contagem de conexões individuais	para cada método, o número de referências para atributos	para cada atributo, o número de referências a ele.
B	contagem do número de itens distintos que estabelecem conexões	para cada método, o número de atributos referenciados	para caca atributo, o número de métodos que o referenciam

ou seja, quais componentes serão medidos, e como as conexões serão contadas. As tabelas 3.2 e 3.3 apresentam opções de como contar conexões para dois níveis de granularidade. A tabela 3.2 define a forma de contagem de conexões a nível de atributo e método. A tabela 3.3, a nível de classe.

- Estabilidade do Servidor: duas categorias sobre estabilidade de classes são definidas: classes estáveis e classe instáveis. As classes instáveis são classes sujeitas à modificação. Classes estáveis são as que não estão sujeitas às mudanças. São classes importadas de bibliotecas ou usadas sem modificações de outros sistemas. A estabilidade de classes ainda não foi endereçada na definição de nenhuma métrica.

Para cada item do *framework*, há um série de parâmetros a serem considerados. Associado a cada parâmetro, os autores associam métricas disponíveis na literatura. O trabalho também apresenta um caso de uso simples de aplicação de seu *framework*.

Tabela 3.3: Contagem de conexões a nível de classe.

	Descrição	Exemplo de acoplamento de importação	Exemplo de acoplamento de exportação
C	somar o número de conexões contadas como em A da tabela 3.2	o número total de referências a atributos por métodos na classe	o número total de referências a atributos da classe o número total de referências
D	somar o número de conexões contadas como em B da tabela 3.2	somar o número de atributos referenciados por cada método da classe	somar o número de métodos que referenciam cada atributo da classe
E	contar o número de itens distintos de conexões começando ou terminando em métodos ou atributos da classe	o número de atributos referenciados por cada método da classe	o número de métodos referenciando atributos da classe
F	para uma classe c, contar o número de outras classes com a qual há pelo menos uma conexão	o número de classes que têm um atributo que é referenciado por um método da classe c	o número de classes que têm um método que referencia um atributo da classe c

3.2 Um Conjunto de Métricas para *Design* Orientado a Objetos

Chidamber et al (CHIDAMBER; KEMERER, 1994) apresentam o desenvolvimento e validação empírica de um conjunto de métricas embasadas teoricamente para a fase de *design* de sistemas orientados a objetos. O trabalho possui 3 objetivos bem definidos: (i) propor métricas construídas com base em conceitos teóricos voltados para o *design* de sistemas OO; (ii) avaliar e propor métricas utilizando critérios estabelecidos pela validação e (iii) apresentar dados empíricos de projetos comerciais para ilustrar as características das métricas em aplicações reais.

Os autores constróem uma teoria base de medição fundamentando-se na ontologia de objetos. São definidas as seguintes propriedades de medição:

1. acoplamento: refere-se ao grau de interdependência entre partes de um *design*;
2. coesão: refere-se a consistência interna dentro de partes de um *design*;
3. grau de similaridade de métodos: relaciona-se à noção de coesão. Grau em que métodos do mesmo objeto desempenham diferentes operações no mesmo conjunto de instâncias de variáveis;
4. complexidade de um objeto: uma unidade é considerada complexa quando possui um grande número de propriedades. Logo, a complexidade de um objeto pode ser

definida como a cardinalidade de suas propriedades;

5. escopo de propriedades: um *designer* de sistemas desenvolve um abstração do domínio de aplicação organizando as classes em uma hierarquia. A hierarquia de herança é um grafo acíclico direcionado que representa classes como nós. Ela descreve a estrutura da aplicação de forma a expandir ou restringir o escopo de propriedades das classes. Relacionadas a essa propriedade, há duas decisões de *design*:

   (a) profundidade de herança: é a profundidade da classe na árvore de herança. A profundidade do nó é o comprimento do caminho mais longo até o nó raiz da árvore de herança;

   (b) número de filhos: número imediato de descendentes de uma classe;

6. métodos como medidas de comunicação: no paradigma OO, objetos se comunicam através da trocas de mensagens. Métodos podem então ser vistos como definições de respostas a possíveis mensagens. Assim, define-se conjunto de resposta para uma classe como sendo o conjunto de todos os métodos que podem ser invocados em resposta a uma mensagem para um objeto da classe. Isso inclui também invocações a métodos de fora da classe;

7. combinação de classes: a combinação de duas classes resulta em outra classe cujas propriedades são a união das propriedades das duas classes combinadas.

A seguir, as métricas propostas pelo autor.

Métodos Pesados por Classe (WMC - *Weighted Methods per Class*): a complexidade da classe é medida como a soma das complexidades dos métodos.

Profundidade da Árvore de Herança (DIT - *Depth of Inheritance Tree*): essa métrica é a propriedade de profundidade de herança discutida anteriormente. Em casos de herança múltipla, a DIT é a maior distância do nó para a raiz da árvore.

Número de Filhos (NOC - *Number of Children*): é o número de subclasses imediatas subordinadas a uma classe na hierarquia de classes.

Acoplamento entre Classes de Objetos (CBO - *Coupling between Object Classes*): CBO para uma classe é a contagem do número de outras classes às quais ela está acoplada. CBO está relacionada à noção de que um objeto está acoplado em outro se um age em outro, isto é, métodos de um objeto usam métodos ou variáveis de outro.

Resposta para uma Classe (RFC - *Response For a Class*): o conjunto resposta de uma classe é o conjunto de métodos que podem ser potencialmente executados em resposta a uma mensagem recebida por um objeto daquela classe. A cardinalidade desse conjunto é a medida dos atributos de objetos da classe, ou seja, o número de elementos do conjunto é a quantidade de objetos de uma classe, uma vez que os resultados são obtidos para cada objeto. Essa medida especificamente para invocações a métodos de fora da classe é também uma medida do potencial de comunicação entre a classe e outras classes.

Falta de Coesão nos Métodos (LCOM - *Lack of Cohesion in Methods*): A similaridade entre dois métodos é definida como o número de atributos usados em comum por esses dois métodos. LCOM é a contagem de pares de métodos cuja similaridade é zero menos o número de pares de métodos cuja similaridade não é zero. Quanto maior o número de métodos similares, mais coesiva é a classe. A definição formal da métrica é mostrada a seguir para maior compreensão do leitor.

Considere uma classe C_1 com n métodos M_1, M_2, ..., M_n. Seja $\{I_j\}$ conjunto de instâncias de variáveis usadas pelo método M_i. Há n conjuntos $\{I_1\}$, ..., $\{I_n\}$. Seja $P = \{(I_i, I_j) \mid I_i \cap I_j = \emptyset\}$ e $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset\}$. Se todos os n conjuntos $\{I_1\}$, ..., $\{I_n\}$ são $\emptyset$ então $P = \emptyset$.

$$LCOM = \begin{cases} \mid P \mid - \mid Q \mid, & \text{se} \quad \mid P \mid > \mid Q \mid \\ 0, & \text{caso contrário} \end{cases}$$

3.3 Avaliação do Desenvolvimento de Software Orientado a Aspectos para Sistemas Distribuídos

Driver (DRIVER, 2002) apresenta uma avaliação de técnicas de desenvolvimento de software orientado a aspectos aplicadas a sistemas de informação distribuído. O autor refatora um sistema distribuído, inicialmente orientado a objetos e escrito na linguagem de programação Java, utilizando aspectos e a linguagem AspectJ. A aplicação distribuída é apresentada em sua versão OO original e o autor então detalha os pontos em que originará aspectos: *tracing*, transações, tratamento de exceções. Em seguida, ele apresenta a nova implementação orientada a aspectos.

A etapa de análise avalia os seguintes pontos que se espera atingir com DSOA. A definição para cada item segue a definição dada pelo autor.

1. separação de conceitos alcançada: endereça o grau de separação de conceitos alcançada através do uso de DSOA durante o processo de refatoração do sistema. O autor analisa essa propriedade através da construção de gráficos. Os gráficos são construídos de forma a mostrar o número das linhas de código referentes a um conceito entrelaçadas com o código-base, antes e depois da refatoração do sistema;

2. capacidade de modificação e de extensão (*changeability/extensibility*): lida com a facilidade em modificar e estender o conceito reimplementado usando DSOA e o sistema em geral. Essa propriedade é avaliada de acordo com a experiência do autor em refatoração de códigos;

3. reusabilidade: considera se um conceito orientado a aspecto pode ser reusado em outro ambiente sem manipulação significativa. Essa propriedade é avaliada através da experiência do autor com reuso, porque, segundo o autor não há trabalhos relacionados à medição de reuso na literatura.

4. plugabilidade: analisa quanto o conceito está amarrado à aplicação e quão facilmente se pode removê-lo e adicioná-lo novamente à aplicação. Segundo o autor, não há trabalhos disponíveis na literatura que meçam a plugabilidade. Então ele faz a análise dessa propriedade de acordo com a experiência;

5. complexidade: essa propriedade considera qualquer mudança na complexidade do sistema a partir da introdução dos conceitos orientados a aspectos. O autor diz que técnicas para medição de complexidade disponíveis na literatura não são inteiramente precisas. Por esse motivo, mais uma vez, a propriedade é analisada de acordo com sua experiência levando em conta fatores como legibilidade, compreensibilidade e separação alcançada com a refatoração em aspectos;

6. produtividade: essa propriedade discute se a nova implementação promove a produtividade dos desenvolvedores do sistema. A experiência do autor em implementar as duas soluções, a AO e a OO, é a base para a análise dessa propriedade;

7. linhas de código: compara o número de linhas de código das duas soluções, a AO e a OO. Segundo o autor, essa é a única, de fato, mensurada cientificamente.

O autor diz que uma compreensiva avaliação de DSOA procura endereçar propriedades de sistema intangíveis e difíceis de medir, por isso relata que sua avaliação é mais subjetiva que científica. A única métrica de fato aplicada é linhas de código, os outros itens são

analisados subjetivamente de acordo com a experiência do autor em desenvolvimento de sistemas.

3.4 Quantificando Aspectos em Plataformas de Middleware

Esse artigo (ZHANG; JACOBSEN, 2003) propõe o uso da programação orientada a aspectos (POA) para resolver o problema da modularidade nos sistemas de *middleware* ocasionado pela presença de lógica entrelaçada. O autor identifica aspectos em um sistema legado e reimplementa-o utilizando uma linguagem de programação orientada a aspectos — processo de refatoração. Um conjunto de métricas é aplicado para medir as mudanças do sistema refatorado em relação à *complexidade da nova estrutur*a e ao *desempenho em tempo de execução*. A refatoração orientada a aspectos prova que é capaz de compor requisitos ortogonais. A configurabilidade é aumentada porque as propriedades aspectizadas podem ser inseridas ou retiradas em tempo de compilação.

Um dos principais problemas de arquitetura de *middleware* é que os sistemas de *middleware* têm de suportar vários requisitos computacionais distintos, além dos conceitos da computação distribuída. Esses requisitos computacionais adicionais são definidos como requisitos de *design* ortogonais em relação à funcionalidade fundamental do *middleware*.

Métricas de Quantificação

As métricas de quantificação são métricas de software para avaliar a qualidade do *design* de software. Medem tanto as propriedades estáticas: custo de desenvolver e de manter; quanto as propriedades de tempo de execução: custo de adotar a tecnologia. A seguir são apresentadas as métricas propostas pelos autores para avaliar a refatoração de um sistema de *middleware*.

Número de Complexidade Ciclomática (CCN - *Cyclomatic Complexity Number*): medida de caminhos de execução alternativos nos segmentos de código causados por comandos de controle de fluxo. É obtido através de heurísticas e independe de linguagem de programação. Um baixo CCN faz o programa mais fácil de entender e manter. Também melhora o tempo de execução do programa, com um maior número de *hits* na cache e mais oportunidades de otimizações dinâmicas. Refatoração POA pode diminuir o CCN por tratar propriedades *crosscutting* fora da decomposição primária.

Tamanho (*size*): o tamanho do sistema de software está diretamente ligado ao custo de desenvolvimento e manutenção. Define-se tamanho como o número total de linhas executáveis em todos os códigos fontes das classes medidas. Refatoração diminui comentários e linhas em branco, além da soma de comandos executáveis do programa primário.

Peso da Classe (*Weight of Class*): o peso da classe é a média do número de métodos por classe. Ele reflete a complexidade da classe. Refatoração AOP diminui o peso da classe.

Acoplamento entre Classe (*Coupling between Classes*): mede-se acoplamento como número médio de classes que uma classe tem como atributos ou invoca sobre. Acoplamento indica quanto os tipos do sistema estão relacionados a outros. Uma boa arquitetura é sempre menos acoplada devido à modularização. Refatoração AOP diminui acoplamento separando o programa primário do conhecimento dos tipo que implementam a lógica *crosscutting*.

Tempo de Resposta (*Response Time*): no contexto do *middleware*, pode ser definido como o tempo levado para responder a uma invocação de requisição do ORB. Divide o tempo de resposta em 4 intervalos: *"client-side marshalling"*, *"server-side unmarshalling and dispatching"*, *"server-side marshalling"* e *"client-side unmarshalling"*. É preciso que a refatoração, pelo menos, preserve a medida de desempenho em tempo de execução.

Os autores fazem uma classificação em métricas estruturais e métricas comportamentais. As estruturais são complexidade ciclomática, tamanho do código, peso da classe e acoplamento. As métricas estruturais são coletadas nas classes que são envolvidas na refatoração. As comportamentais refletem características de tempo de execução do sistema e incluem tempo de resposta.

3.5 Reuso e Manutenção de Software Orientado a Aspectos: Um *Framework* para Avaliação

Esse artigo (SANT'ANNA et al., 2003) apresenta um *framework* para avaliar reusabilidade e manutenibilidade de software orientado a aspectos. O *framework* é centrado na definição de separação de conceitos, acoplamento, coesão e tamanho. O *framework* consiste em um suite de métricas e um modelo de qualidade. O modelo de qualidade define precisamente como medir reusabilidade e manutenibilidade baseado em um conjunto de métricas proposto. O modelo também auxilia na interpretação do dados coletados. O *framework* pode ser usado para avaliar decisões de *design* em DSOA e também para

comparar soluções orientadas a aspectos a soluções orientadas a objetos.

O Suite de Métricas

O suite de métricas captura informações sobre *design* e código em termos de atributos de software fundamentais. O suite de métricas reusa e refina métricas clássicas, sendo adaptadas para refletir as abstrações introduzidas por aspectos. Ele é composto por cinco métricas de *design* e cinco métricas de código, agrupadas de acordo com o que medem: separação de conceitos, acoplamento, coesão e tamanho.

1. Separação de Conceitos: refere-se à habilidade de identificar, encapsular e manipular as partes do software que são relevantes a um conceito em particular.

   - **Dispersão de Conceito através dos Componentes** (CDC - *Concern Diffusion over Components*): métrica de *design* que conta o número de componentes primários cujo principal propósito é contribuir para a implementação de um conceito;
   - **Dispersão de Conceito através das Operações** (CDO - *Concern Diffusion over Operations*): conta o número de operações primárias cujo principal propósito é contribuir para a implementação de um conceito;
   - **Dispersão de Conceito através das Linhas de Código** (CDLOC - *Concern Diffusion over LOC*): conta o número de pontos de transição para cada conceito do começo ao fim das linhas de código. O uso dessa métrica requer um processo de sombreamento que particiona o código em áreas sombreadas e não-sombreadas. As áreas sombreadas são linhas de código que implementam um dado conceito. Pontos de transição são os pontos no código onde há transição de uma área não-sombreada para uma sombreada e vice-versa. Para cada conceito, o programa texto é analisado linha por linha para contar os pontos de transição. Quanto maior o CDLOC, mais misturado é o código de conceito dentro da implementação dos componentes. Quanto menor, mais localizado é o código do conceito.

2. Métricas de Acoplamento: acoplamento é uma indicação da força das interconexões entre componentes em um sistema. Sistemas altamente acoplados têm interconexões fortes, com unidades de programas dependentes umas das outras.

   - **Acoplamento entre Componentes** (CBC - *Coupling between Components*): definido para um componente (classe ou aspecto) como um registro de número de outros componentes ao qual ele é acoplado;

- **Profundidade da Árvore de Herança** (DIT - *Depth of Inheritance Tree*): definido como o máximo comprimento de um nó para o nó raiz da árvore. Ela conta quão longe a hierarquia de herança de uma classe ou aspecto é declarada.

3. Métrica de Coesão: coesão de um componente mede a proximidade do relacionamento entre seus componentes internos.

   - **Falta de Coesão nas Operações** (LCOO - *Lack of Cohesion in Operations*): mede a falta de coesão de um componente. LCOO mede a quantidade de (pares) método/*advice* que não acessam a mesma instância de variável.

4. Métricas de Tamanho: o tamanho do software fisicamente mede o comprimento do *design* e código de um sistema de software.

   - **Tamanho de Vocabulário** (VS - *Vocabulary Size*): conta o número de componentes do sistema, ou seja, o número de classes e aspectos dentro do sistema. Essa métrica mede o tamanho do vocabulário do sistema. Cada nome de componente é contado como parte do vocabulário do sistema. A instâncias do componente não são contadas;
   - **Linhas de Código** (LOC - *Lines of Code*): conta o número de linhas do código. Comentários e linhas em branco não são contados;
   - **Número de Atributos** (NOA - *Number of Attributes*): mede o vocabulário interno de cada componente, isto é, o número de atributos de cada classe ou aspecto. Atributos herdados não são incluídos na soma;
   - **Operações Ponderadas por Componente** (WOC - *Weighted Operations per Component*): mede a complexidade de um componente em termos de suas operações (é a soma da complexidade de todos os métodos ou *advices* de um componente). Essa métrica não especifica a medida de complexidade da operação que deve ser sob medida para cada contexto. A medida de complexidade de operação é obtida contando o número de parâmetros de uma operação, assumindo que uma operação com mais parâmetros é mais complexa.

O *Framework* de Avaliação

A medida de um atributo interno é útil se está relacionado à medida de algum atributo externo ao objeto de estudo. Em engenharia de software, as medidas de atributos internos são conceitos artificiais e, portanto, sem significado. Assim, desenvolveu-se um *framework*

para capturar o entendimento de separação de conceitos, acoplamento, coesão e tamanho em termos de suas utilidades como preditores de manutenibilidade e reusabilidade.

Os componentes do *framework* ajudam a organizar o processo de avaliação e auxilia na coleta e interpretação de dados. Os componentes básicos do *framework* são o suite de métricas e o modelo de qualidade. O modelo de qualidade estabelece os relacionamentos entre atributos externos, atributos internos e métricas. O *framework* requer como entrada do processo de medição os documentos de *design*, o código do sistema e uma descrição dos conceitos do sistema para guiar a identificação dos conceitos quando usando métricas de separação de conceitos (processo de sombreamento).

O Modelo de Qualidade

O modelo de qualidade define uma terminologia e clarifica os relacionamentos entre a reusabilidade, manutenibilidade e métricas. É uma ferramenta útil para guiar na interpretação de dados.

A noção de qualidade de software é geralmente capturada em um modelo que retrata outras qualidades intermediárias, que são chamadas fatores. Isso porque uma qualidade é composta por muitas outras qualidades. O modelo de qualidade proposto é composto por três elementos diferentes: qualidades, fatores e atributos internos. As qualidades são atributos que se quer primariamente observar no sistema de software (reusabilidade e manutenibilidade). Os fatores são os atributos de qualidade secundários que influenciam as qualidades primárias definidas. Os atributos estão relacionados às propriedades internas dos sistemas de software.

Reusabilidade é a habilidade do software de servir para a construção de diferentes elementos de software. Nesse modelo, interessa avaliar a reusabilidade de elementos de *design* e códigos de sistemas AO. Manutenção é atividade de modificar um sistema de software depois da entrega inicial.

Flexibilidade e compreensibilidade (*understandability*) são os fatores centrais para promover reusabilidade e manutenibilidade. Compreensibilidade indica o nível de dificuldade para estudar o *design* e o código de sistemas. Flexibilidade indica o nível de dificuldade para fazer mudanças drásticas a um componente de sistema sem precisar modificar os outros. Um sistema compreensível reforça sua manutenibilidade e reusabilidade, porque é preciso entender os componentes do sistema antes de fazer qualquer modificação ou extensão do sistema. Um sistema deve ser flexível suficiente para suportar adição e remoção de funcionalidades e o reuso de seus componentes com um esforço mínimo.

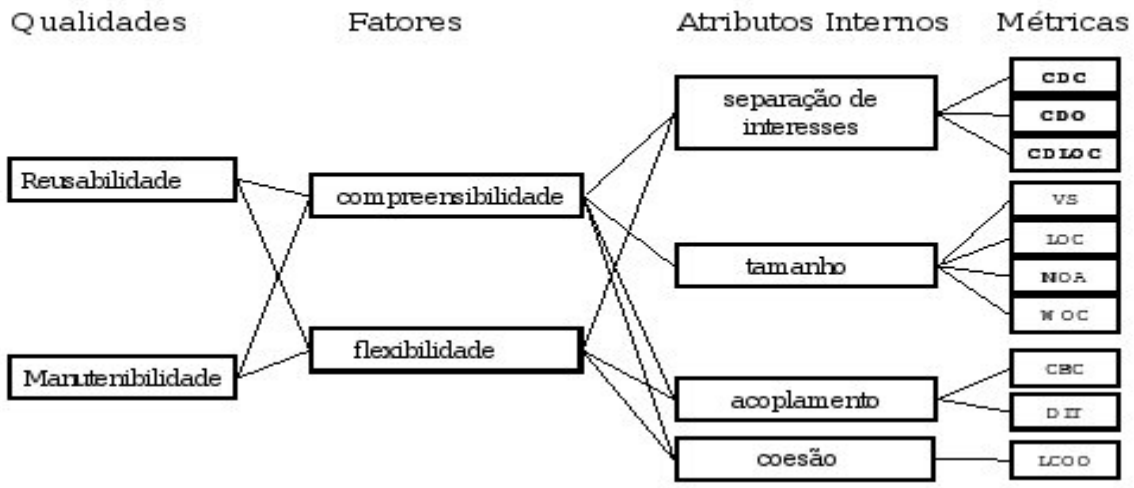

Figura 3.1: O modelo de qualidade.

Nesse modelo, o fator compreensibilidade está relacionado aos atributos internos tamanho, acoplamento, coesão e separação de conceitos. Acoplamento e coesão afetam a compreensibilidade porque um componente de um sistema não pode ser entendido sem referência a outros componentes aos quais está relacionado. O tamanho do *design* e do código pode indicar a quantidade de esforço para entender os componentes do software. A separação de conceitos é preditor de compreensibilidade porque quanto mais localizados estão os conceitos, mas fácil é entendê-los.

O fator flexibilidade é influenciado pelos atributos internos acoplamento, coesão e separação de conceitos. Alta coesão, baixo acoplamento e separação de conceitos são desejáveis porque indicam que o componente representa uma única parte do sistema e os componentes são independentes. Além disso, os conceitos não estão espalhados e entrelaçados. Se for necessário adicionar, remover ou reusar uma funcionalidade, ela está localizada em um único componente e a manutenção e reuso são flexíveis e restritos a esse componente isolado.

A figura 3.1 foi retirada do artigo e representa o que foi exposto sobre modelo de qualidade.

Como caso de uso, estenderam-se sistemas AO e OO e coletaram-se os resultados das métricas para confrontar com os resultados dos sistemas originais, visando avaliar a manutenibilidade e reusabilidade.

3.6 Micro-Medições para Sistemas Dinâmicos Orientados a Aspectos

Este artigo (HAUPT; MEZINI, 2004) está centrado em 3 contribuições: (i) analisar abordagens existentes de avaliação de desempenho de POA, (ii) construir *benchmarks* para micro-medições amplamente aplicável a sistemas POA dinâmicos e (iii) formular requisitos para conjuntos de *benchmarks* para POA dinâmica.

O autores agrupam os sistemas POA dinâmicos a partir de sua abordagem de *weaving*:

- *weaving* em tempo de carga: sistemas nesse grupo usam um *loader* de classe estendido para modificar as classes no momento da classe;
- interceptação: definem-se *breakpoints* nos *join points* da aplicação em execução. Quando um *breakpoint* é alcançado, o *advice* correspondente é chamado;
- *weaving* em tempo de execução: grupo de sistemas em que o código do aspecto passa pela etapa de *weaving* no código do aplicação em tempo de execução.

Ferramentas de POA dinâmica são então classificadas de acordo com esses grupos.

Os autores reconhecem a necessidade de um conjunto de micro-medição para avaliação de POA dinâmica, de forma a formalizar esse processo. Para isso, propõe o seguinte catálogo de micro-medições:

1. Custo de *un(weaving)* dinâmico: uma característica crucial de implementações dinâmicas é que aspectos podem ser dinamicamente adicionados e removidos. O custo dessa operação para um único aspecto é focado nesse item;
2. Custo de executar um *join point shadow*: código em *join point shadow* (invocação de métodos, acesso a membros, instanciação etc.) pode estar sujeito a decoração com funcionalidade de advice. É interessante medir o custo dos mecanismos empregados em diferentes sistemas de POA dinâmica para invocar *advices*;
3. Custo de reificação de informação do contexto do *join point*: isso inclui, por exemplo, passar parâmetros do método *advised* para o *advice*.

As medições de *join points shadows* devem ser nos seguintes contextos:

1. *unadvised*: ou seja, sem qualquer aspecto associado ao *join point shadow*;

2. *advised*: onde cada tipo possível de *advice* (*before, after, around*) deve ser aplicado e o *advice* deve existir com e sem parâmetros e ser chamado no contexto do *join point*;

3. com características avançadas: como por exemplo as instruções cflow, perthread, perthis de AspectJ.

O catálogo de micro-medições objetiva compor um *benchmark* amplamente aceitável para POA dinâmica. Entretanto, é importante lembrar que nem todas as micro-medições são aplicáveis a todas as implementações de POA dinâmica.

Uma vez definido, o catálogo é aplicado a AspectWerkz, JBOSS AOP, PROSE e Steamloom. A única operação a ser medida foi invocação de métodos. Somente as micro-medições execução de *join point shadow* e reificação no contexto do *join point* foram realizadas por serem o denominador comum das funcionalidades dos 4 sistemas.

O artigo analisa medições para avaliar o *overhead* de usar aspectos em relação à orientação a objetos e o desempenho de implementações de POA dinâmicas. A abordagem utilizada para medir o desempenho dessas implementações é micro-medições. Micro-medições são úteis para se raciocinar sobre o custo infligido em operações básicas específicas na execução de aplicações.

3.7 Medição dos Efeitos da Aspectização de Software

Ceccato e Tonella, no trabalho (CECCATO; TONELLA, 2004), propõem um método de medição para investigar a relação custo-benefício da utilização da programação orientada a aspectos. O método é baseado em um conjunto de métricas que estende métricas tradicionalmente utilizadas no paradigma orientado a objetos.

O conjunto de métricas pode ser aplicado a classes e aspectos. O termo módulo é empregado para classes e aspectos e o termo operação, para métodos, *advices* e *introduction*. As métricas são apresentas a seguir:

1. WOM (*Weighted Operations in Module*): número de operações em um dado módulo. WOM captura a complexidade interna de um módulo em termos de suas funções implementadas;

2. DIT (*Depth of Inheritance Tree*): comprimento do caminho mais longo de um dado módulo até a raiz da árvore de hierarquia. Quanto mais profunda está um módulo

na hierarquia, maior o número de operações que ele deve herdar e, portanto, mais complexo ele será de entender e manter;

3. NOC (*Number of Children*): número de subclasses ou sub-aspectos imediatos de um dado módulo. O número de filhos de um módulo indica a proporção de módulos potencialmente dependentes de propriedades herdadas do módulo pai;

4. CAE (*Coupling in Advice Execution*): número de aspectos contendo advices possivelmente disparados pela execução de operações de um dado módulo. Se o comportamento de uma operação pode ser alterado por um *advice*, há uma dependência da operação em relação ao *advice*. Assim, o módulo dessa operação está acoplado ao aspecto que contém o *advice*. Uma mudança no *advice* causa impacto na operação;

5. CIM (*Coupling on Intercepted Modules*): número de módulos ou interfaces explicitamente nomeadas nos *pointcuts* pertencentes a um aspecto. CIM captura o conhecimento direto do restante do sistema. Alto valores de CIM indicam alto acoplamento do aspecto com uma aplicação e baixa generalidade, reusabilidade;

6. CMC (*Coupling on Method Call*): número de módulos ou interfaces cujos métodos declarados são possivelmente chamados por um dado módulo. Uso de grande número de métodos de vários diferentes módulos indicam que um módulo não pode ser facilmente isolado de outros. Alto acoplamento é associado com alta dependência de funções de outros módulos;

7. CFA (*Coupling in Field Acess*): número de módulos ou interfaces cujos campos são acessados por um dado módulo. CFA mede a dependência de um módulo em relação a outros em termos de campos acessados;

8. RFM (*Response for a Module*): métodos e *advices* potencialmente executados em resposta a uma mensagem recebida por um dado módulo. RFM mede a comunicação potencial entre um módulo e os outros. A principal adaptação dessa métrica para que possa ser aplicada a aspectos está associada com as respostas implícitas que são disparadas sempre que um *pointcut* intercepta uma operação de um dado módulo;

9. LCO (*Lack of Cohesion in Operations*): pares de operações trabalhando em diferentes classes de campos menos pares de operações trabalhando em campos comuns (zero, se negativo). Operações trabalhando em conjuntos de campos de módulo separados são considerados dissimilares e contribuem para o aumento do valor da métrica;

10. CDA (*Crosscutting Degree of an Aspect*): número de módulos afetados pelos *pointcuts* e por *introductions* em um dado aspecto. Enquanto CIM considera somente módulos nomeados explicitamente, CDA mede todos os módulos possivelmente afetados por um aspecto. Isso dá uma idéia do impacto total de um aspecto nos outros módulos. A diferença entre CDA e CIM dá o número de módulos que são afetados por um aspecto sem serem referenciados explicitamente pelo aspecto, o que indica o grau de generalidade de um aspecto em termos de sua independência de classes/aspectos específicos.

Da métrica CBO (*Coupling Between Objects*) — métrica proposta por (CHIDAMBER; KEMERER, 1994), que mede acoplamento entre objetos — derivam CMC e CFA. Ela foi dividida em duas para distingüir acoplamento por operações de acoplamento por atributos.

Ceccato e Tonella, em (CECCATO; TONELLA, 2004), desenvolveram também uma ferramenta para coleta de resultados das métricas apresentadas anteriormente para código-fonte escrito em AspectJ.

3.8 Modularidade de *Design* Orientado a Aspectos: uma abordagem de medição orientada a conceitos

Sant'Anna, em (SANT'ANNA, 2008), propõe um *framework* de medição orientada a conceitos para avaliar a modularidade arquitetural. O *framework* engloba um mecanismo para documentar conceitos e um conjunto de métricas orientadas a conceitos para arquitetura. O *framework* complementa métricas existentes promovendo conceitos como uma abstração de medição. Ele visa dar suporte aos engenheiros de software para: (i) antecipar problemas de modularidade causados por conceitos relevantes para a arquitetura, (ii) comparar alternativas de soluções de *design* arquitetural com relação a habilidade de modularizar conjuntos distintos de conceitos.

O conjunto de métricas proposto avalia como conceitos particulares afetam atributos tradicionais como acoplamento, coesão e complexidade da interface. As métricas são apresentadas a seguir agrupadas por categorias de acordo com o atributo de software que avaliam:

1. difusão de conceito:

   - CDAC (*Concern Diffusion over Architectural Components*): conta o número

de componentes arquiteturais que contribuem para a realização de um dado conceito;

- CDAI (*Concern Diffusion over Architectural Interfaces*): conta o número de interfaces que contribuem para a realização de um dado conceito;
- CDAO (*Concern Diffusion over Architectural Operations*): conta o número de operação que contribuem para a realização de um dado conceito;

2. acoplamento entre conceitos arquiteturais:

   - CIBC (*Component-level Interlacing Between Concerns*): conta o número de outros conceitos com os quais o conceito avaliado divide pelo menos um componente;
   - IIBC (*Interface-level Interlacing Between Concerns*): conta o número de outros conceitos com os quais o conceito avaliado divide pelo menos uma interface;
   - OOBC (*Operation-level Overlapping Between Concerns*): conta o número de outros conceitos com os quais o conceito avaliado divide pelo menos uma operação;

3. acoplamento entre componentes:

   - AC (*Afferent Coupling between components*): conta o número de componentes que requerem serviços do componente avaliado;
   - EC (*Efferent Coupling between components*): conta o número de componentes dos quais o componente avaliado requer serviços;

4. coesão de componentes:

   - LCC (*Lack of Concern-based Cohesion*): conta o número de conceitos endereçados pelo componente avaliado;

5. complexidade de interface:

   - *Number of Interfaces*: conta de interfaces de cada componente;
   - *Number of Operations*: conta o número de operações nas interfaces de cada componente.

Esse mesmo conjunto de métricas também é apresentado em (SANT'ANNA et al., 2007).

Para suportar a aplicação das métricas orientadas a conceitos, o autor define *concern templates*. *Concern templates* capturam os elementos arquiteturais associados com conceitos chaves, possibilitando um mapeamento conceito-elemento.

Como a interpretação dos resultados das métricas é uma parte complexa do processo de avaliação, o autor define um conjunto de regras heurísticas orientadas a conceitos. Essas regras combinam os resultados de diferentes métricas e testam esses resultados contra valores de *threshold* configuráveis para suportar a identificação de potencias falhas de *design*.

4 Conjunto de Métricas para Avaliar Middleware Orientado a Aspectos

O conjunto de métricas proposto neste trabalho tem como objetivo avaliar sistemas de *middleware* orientado a aspectos. Logo, deve avaliar não somente as características comuns a sistemas orientados a aspectos e a objetos, mas também as particularidades de sistemas de *middleware*. Para isso, a construção deste trabalho baseia-se em trabalhos disponíveis na literatura nas categorias (i) métricas para sistemas orientados a objetos, (ii) métricas para sistemas orientados a aspectos, (iii) métricas de avaliação de *middleware* e (iv) métricas dinâmicas.

Vários foram os trabalhos encontrados na área de avaliação através de métricas. Os trabalhos (BRIAND; DALY; WUEST, 1999), (CHIDAMBER; KEMERER, 1994) considerados mais significativos, serviram como base para a construção das métricas voltadas à avaliação da parte orientada a objetos do *middleware*. Selecionou-se as métricas mais relevantes tendo em foco a natureza de um *middleware*. As métricas orientadas a objetos podem ainda ser adaptadas de forma a serem utilizadas na avaliação da parte do *middleware* orientada a aspectos. Os trabalhos (SANT'ANNA et al., 2003) e (BARTOLOMEI et al., 2006) apresentam métricas para avaliação de soluções orientadas a aspectos. As mais significativas para avaliação de *middleware* também foram incorporadas ao conjunto de métricas. Este trabalho se fundamenta ainda em (ZHANG; JACOBSEN, 2003) para a etapa de avaliação de refatoração de *middleware*.

O trabalho (DRIVER, 2002) é específico para avaliação de sistemas distribuídos e levanta várias propriedades fundamentais para refatoração orientada a aspectos. Apesar disso, o trabalho não define ou usa métricas. As propriedades são avaliadas subjetivamente, tendo como parâmetro a experiência do autor no desenvolvimento de sistemas.

Para a construção do conjunto de métricas dinâmicas, este trabalho está baseado em

(DUFOUR et al., 2004) que avalia o comportamento dinâmico de programas em AspectJ e o *overhead* de sua utilização em comparação com programas escritos em Java. Das medições apresentadas em (DUFOUR et al., 2004), foram selecionadas as que mais se adaptavam ao contexto de *middleware* orientado a aspectos. Um método de aplicação dessas métricas foi definido na seção 4.5 para que sirvam ao propósito de avaliação do momento de carga de aspectos que é um dos objetivos deste trabalho.

O conjunto de métricas será dividido em duas partes. A primeira parte trata de métricas para fases de *design*, implementação e refatoração dos sistemas de *middleware* orientado a aspectos. Essa parte constitui-se de um conjunto de propriedades estáticas que se relacionam entre si e métricas para avaliá-las. A segunda parte avalia o comportamento dinâmico desses sistemas, focando no desempenho.

Como arcabouço para a primeira parte do conjunto de métricas, utilizar-se-á o trabalho (DRIVER, 2002). A idéia principal é definir as propriedades relevantes para um sistema de *middleware* e associar, a cada propriedade, métricas que possam mensurá-las. A proposta é estender o trabalho (DRIVER, 2002) de forma a relacionar métricas às propriedades de sistemas distribuídos. Algumas das métricas em questão serão selecionadas e/ou adaptadas dos trabalhos mencionados para que se tenha um conjunto de métricas o mais adequado possível à realidade dos sistemas de *middleware*.

A seção 4.1 apresenta o conjunto de propriedades estáticas relevantes a sistemas de *middleware* orientado a aspectos. A seção 4.2 mostra as métricas estáticas associadas a cada propriedade do conjunto. A seção 4.3 faz um resumo das propriedades selecionadas e suas métricas estáticas. A seção 4.4 mostra o conjunto de propriedades dinâmicas para sistemas de *middleware* orientado a aspectos. Por fim, a seção 4.5 apresenta as métricas dinâmicas para avaliação das propriedades da seção 4.4.

4.1 Definição do Conjunto de Propriedades Estáticas de Sistemas de *Middleware* OA

Nessa subseção, define-se um conjunto de propriedades relevantes para sistemas distribuídos. Para cada propriedade levantada, são selecionadas métricas que contribuem para a avaliação dessa propriedade. É importante lembrar que a mensuração dessas propriedades é relevante tanto na fase de *design* quanto na fase de implementação. Esse conjunto leva em conta a proposta do trabalho (DRIVER, 2002), mas sofre algumas modificações que serão citadas durante a apresentação das propriedades. A seguir, apresenta-se o con-

junto de propriedades proposto. Lembrando que, neste trabalho, entidade é o nome dado a classes e aspectos e operações, a métodos e a *advices*.

1. **Modularidade**: em (DRIVER, 2002), separação de conceitos faz parte da lista de propriedades. Para este trabalho, optou-se por utilizar a propriedade modularidade em seu lugar, por ser mais abrangente. Em (SANT'ANNA, 2008), boa modularidade está associada à separação de conceitos, alta coesão e baixo acoplamento entre as entidades do sistema. A mesma associação também é apresentada em (CACHO et al., 2006) e em (GARCIA et al., 2005);

2. **Manutenibilidade**: em (DRIVER, 2002), há as propriedades capacidade de modificação, de extensão e produtividade. Como capacidade de modificação e extensão são influenciadas pelos mesmos fatores e é difícil definir limites precisos entre essas duas propriedades, decidiu-se por agrupá-las em um único item chamado manutenibilidade. Em (CHIDAMBER; KEMERER, 1994), os autores, avaliando suas métricas, relatam que a manutenibilidade é influenciada pelo tamanho e pelo acoplamento do sistema: quanto maior e mais acoplado, mais difícil se torna a compreensão do sistema, resultando em manutenção mais complexa. A compreensão do sistema é afetada não somente pelo acoplamento, mas também pelos outros fatores que influenciam a modularidade. Um sistema mais modular facilita o entendimento por parte do desenvolvedor. Por isso, para este trabalho, a manutenibilidade está associada às propriedades tamanho e modularidade. A propriedade produtividade é pouco citada em trabalhos de avaliação de sistemas, está mais relacionada à predição de tempo e custo de projetos. Por isso, será desconsiderada neste trabalho;

3. **Reusabilidade**: está relacionada à capacidade de codificar de forma genérica, buscando a independência de aplicação. Segundo (CHIDAMBER; KEMERER, 1994), a reusabilidade depende do tamanho dos componentes do sistema: quanto maiores os componentes, mais difícil manter independência em relação a uma aplicação específica. Outro fator que impacta a reusabilidade é o acoplamento: quanto mais uma classe estiver acoplada a outras, maior sua dependência em relação a elas e, conseqüentemente, mais difícil o reuso. Como herança é uma forma de reuso, esse também é um fator determinante para essa propriedade;

4. **Flexibilidade**: em (DRIVER, 2002), essa propriedade se chama plugabilidade. Decidiu-se pelo novo termo para tornar a propriedade mais abrangente e também deixá-la em conformidade com o trabalho (SANT'ANNA et al., 2003). Flexibilidade, no contexto de sistemas de *middleware*, refere-se a capacidade de acoplar e

desacoplar componentes à parte central do sistema, adaptando-o às necessidades da aplicação. Segundo (SANT'ANNA et al., 2003), flexibilidade é afetada pelos fatores acoplamento e separação de conceitos. Baixo acoplamento e separação de conceitos promovem a flexibilidade por facilitarem a remoção ou inclusão de funcionalidades;

5. **Complexidade**: segundo (CHIDAMBER; KEMERER, 1994), a complexidade está relacionada à herança, ao tamanho e a resposta para um classe (RFC - *Response for a Class*). RFC é a quantidade de métodos que podem ser invocados em resposta a uma mensagem recebida pelo objeto. Quanto mais altos esses fatores, maior a complexidade do sistema.

6. **Tamanho**: essa propriedade visa caracterizar o sistema de *middleware* quanto ao seu tamanho. Não somente o número de linhas de código, mas também o tamanho do vocabulário. O vocabulário pode ser interno a uma entidade — o número de atributos de uma entidade — ou relativo ao sistema — nesse caso, o número de entidades que o compõem;

7. **Estabilidade**: essa propriedade não faz parte da lista original presente em (DRIVER, 2002). Neste trabalho, ela foi incluída por ser considerada importante para avaliar reuso de entidades (MARTIN, 1994). Segundo Martin (MARTIN, 1994), quanto maior o número de classes dependendo de uma determinada classe, mais estável esta é. De forma análoga, quanto maior o número de classes das quais uma classe depende, maior é sua instabilidade, pois uma mudança em qualquer uma das classes das quais ela depende, ocasiona uma mudança nela. Baseando-se nesse raciocínio, a estabilidade de uma classe é avaliada em termos da (in)dependência de uma classe em relação às outras. Para medir a (in)dependência de uma classe, utiliza-se as métricas para acoplamento importado/exportado.

As propriedades reusabilidade, complexidade foram retiradas do conjunto original de (DRIVER, 2002) sem modificações. Modularidade, tamanho e flexibilidade foram adaptações das propriedades separação de conceitos, linhas de código e plugabilidade, respectivamente. As propriedades capacidade de modificação/extensão foram agrupadas na propriedade manutenibilidade. A propriedade estabilidade foi incluída na lista devido a sua importância na avaliação da reusabilidade. Por fim, a propriedade produtividade foi retirada.

Alguns esclarecimentos são necessários sobre a propriedade acoplamento. Para que se possa fazer uma avaliação detalhada dessa propriedade, ela foi dividida em três categorias

neste trabalho: (i) acoplamento entidade-entidade, (ii)acoplamento código base-aspecto e (iii) acoplamento exportado/importado. O acoplamento entidade-entidade analisa o acoplamento entre quaisquer duas entidades do sistema: seja entre classes, seja entre aspectos, seja entre classe e aspecto. É a categoria mais genérica das três. O acoplamento código base-aspecto, como o próprio nome diz, avalia o acoplamento entre o código-base do sistema e os aspectos que incidem nele. A última categoria — acoplamento importado/exportado — avalia a relação cliente/servidor entre entidades. Se um entidade oferece um serviço, ela é dita servidora e exporta o serviço, gerando um acoplamento exportado. No caso análogo, quando uma entidade usa um serviço, ela é dita cliente e importa um serviço, gerando um acoplamento importado.

A divisão do acoplamento em categorias foi realizada porque algumas propriedades do conjunto de métricas dependem de um tipo específico de acoplamento. Por exemplo, a propriedade flexibilidade se refere a facilidade em adicionar ou remover um aspecto ao código-base de um sistema. Nesse caso, portanto, apenas o acoplamento entre código-base e aspectos interessa.

4.2 Seleção de Métricas para o Conjunto de Propriedades Estáticas de Sistemas de *Middleware* OA

Uma vez definido o conjunto de propriedades de sistemas de *middleware* e do levantamento dos fatores que os influenciam, selecionam-se as métricas mais adequadas para a mensuração dessas propriedades no sistema. O restante dessa subseção será dedicada a esse propósito. Depois da apresentação de cada métrica, será mostrado um exemplo simples de aplicação. Ao fim da subseção, uma tabela resumo das métricas é apresentada e também a fase de desenvolvimento em que podem ser aplicadas.

4.2.1 Modularidade

Modularidade é considerado um conceito essencial em sistemas de software. Modularidade é o grau em que um sistema é composto de componentes discretos tais que uma mudança a um componente tenha impacto mínimo em outros componentes (SANT'ANNA, 2008).

Por ser um tema importante e recorrente em trabalhos sobre avaliação de sistemas, a propriedade modularidade foi incluída na lista de propriedades do conjunto de métricas.

Tradicionalmente, boa modularidade está ligada a baixo acoplamento e alta coesão nos elementos de um sistema (BRIAND; DALY; WUEST, 1999), (BRIAND; DALY; WUST, 1998). O trabalho (SANT'ANNA, 2008) faz uma ampla exploração do tema no design orientado a aspectos e considera modularidade como: (i) grau em que os conceitos do sistema (funcionalidades, características, requisitos) estão bem localizados nos módulos do sistema, ou seja, grau de separação de conceitos e (ii) grau em que os módulos do sistema são coesivos, fracamente acoplados e têm interfaces reduzidas.

Este trabalho segue a definição de (SANT'ANNA, 2008) para modularidade e associa essa propriedade a separação de conceitos, coesão e acoplamento, apresentadas nessa mesma subseção. Como o acoplamento para essa propriedade diz respeito a conexões entre as várias entidades de um sistema, o acoplamento em questão é o entidade-entidade.

Separação de Conceitos

Um conceito é qualquer propriedade importante ou área de interesse que se deseja tratar de forma modular (SANT'ANNA, 2008 apud ELRAD; FILMAN; BADER, 2001). A partir dessa informação, pode-se verificar a importância que a separação de conceitos tem para a modularidade. As técnicas de DSOA são usadas para separar os conceitos que entrecortam o restante do sistema e construir um sistema mais modularizado, mais organizado.

Para avaliar a separação de conceitos realizada em um sistemas de *middleware*, decidiu-se por utilizar as métricas do trabalho (SANT'ANNA et al., 2003): CDC, CDO e CDLOC, apresentadas na subseção 3.5. Esse é o único trabalho disponível na literatura que apresenta métricas bem definidas para essa propriedade. A maioria dos trabalhos apresenta a separação de conceitos como gráficos comparativos sobre número de linhas utilizadas para implementar o conceito e o código-base. Essas métricas serão redefinidas para se adequarem à terminologia deste trabalho.

- CDC (*Concern Diffusion over Components*): número de entidades primárias cujo principal propósito é contribuir para a implementação de um conceito;
- CDO (*Concern Diffusion over Operations*): número de operações primárias cujo principal propósito é contribuir para a implementação de um conceito;
- CDLOC (*Concern Diffusion over LOC*): número de pontos de transição para cada conceito do começo ao fim das linhas de código.

A seguir discutiremos a aplicação das métricas de separação de conceitos.

A figura 4.1 mostra um programa escrito em Java que implementa um controlador de temperatura simples para exemplificar a aplicação das métricas de separação de conceitos. Caso a temperatura ambiente esteja maior que a temperatura ideal, ele sinaliza que o ar condicionado deve ser ligado. Se a temperatura ambiente estiver abaixo da ideal, o programa informa que o aquecedor deve ser ligado. O programa mantém um arquivo para armazenar o *logging* do sistema, o conceito entrelaçado.

A aplicação das métricas CDC e CDO no sistema é direta. Para CDC, conta-se o número de entidades que têm como principal objetivo implementar um conceito no sistema. Para esse caso, CDC = 0, pois a classe não implementa somente o conceito de *logging* do sistema. Para CDO, conta-se o número de operações primárias, ou seja, métodos e/ou *advices*, cujo principal propósito é implementar um conceito no sistema. Nesse caso CDO = 1, pois o método "fecharArquivo" se refere apenas ao conceito: fechar o arquivo que armazena o *logging*.

A aplicação de CDLOC é um pouco mais trabalhosa. É necessário contar o número de pontos de transição para cada conceito através das linhas de código. Para facilitar a contagem, as partes do código referentes ao conceitos foram sombreadas. Para o programa exemplo, tem-se 9 pontos de transição. Logo, CDLOC = 9.

Bastaria uma olhada rápida pelo código e já se poderia dizer que a classe ControladorTemperatura não possui uma separação de conceitos satisfatória. O resultado das métricas corrobora com essa afirmação. Uma boa sugestão para esse caso seria a aplicação de aspectos.

Acoplamento Entidade-Entidade

O acoplamento entidade-entidade avalia o acoplamento entre qualquer par de entidades do sistema, sem fazer distinção entre aspectos e classes. Esse é o tipo de acoplamento que importa para a avaliação da modularidade do sistema, pois o que se quer é avaliar a organização do sistema, pouco interessa a abstração (classes ou aspectos) utilizada para atingir essa organização.

Para uma avaliação completa do acoplamento entidade-entidade, instancia-se o trabalho (BRIAND; DALY; WUEST, 1999), mostrado na subseção 3.1. Para endereçar

```
public class ControladorTemperatura {

        float temperaturaIdeal;

        FileWriter escritor;

        public ControladorTemperatura (float temp){
                temperaturaIdeal = temp;

                escritor = new FileWriter(new File("ArquivoLog.txt"));

        }

        public boolean ligarArCondicionado(float tempeturaAmbiente){

                boolean arCondicionado = false;

                if(temperaturaIdeal < tempeturaAmbiente){
                        arCondicionado = true;

                        escritor.write("Ar Condicionado precisa ser ligado.\n");

                }

                return arCondicionado;
        }
        public boolean ligarAquecedor(float tempeturaAmbiente){

                boolean aquecedor = false;

                if(temperaturaIdeal > tempeturaAmbiente){
                        aquecedor = true;

                        escritor.write("Aquecedor precisa ser ligado.\n");

                }

                return aquecedor;
        }

        public void fecharArquivo(){

                escritor.close();

        }

}
```

Figura 4.1: Programa exemplo para controle de temperatura.

requisitos referentes à orientação a aspectos, algumas adaptações foram necessárias. O *framework* original levanta os seguintes pontos em relação ao acoplamento:

- o tipo de acoplamento: define o que constitui acoplamento. Os tipos são conexão operação-atributo, conexão operação-operação, conexão por tipo de atributo, conexão por tipo de parâmetro e conexão por herança. Esses tipos são definidos na tabela 4.1;
- local de impacto: se o acoplamento é importado ou exportado;
- granularidade da medida: o domínio da medida e como contar as conexões do acoplamento;
- se o acoplamento é direto ou indireto;
- estabilidade da classe servidora: segundo o artigo (BRIAND; DALY; WUEST, 1999), a estabilidade da classes servidora é um item bastante subjetivo e, portanto, difícil de julgar. Por esse motivo, foi definida neste trabalho a propriedade estabilidade separadamente para que possa ser melhor analisada;
- herança: como atribuir métodos e atributos a classes. A herança será tratada de forma separada do acoplamento na seção 4.2.3.

Os tipos de conexão do trabalho original foram adaptados para refletir a realidade dos sistemas de *middleware* orientados a aspectos. Outros foram removidos por não se adequarem a esse tipo de sistema. A tabela 4.1 mostra o novo conjunto de tipos de conexões. Para facilitar a identificação, foram associados nomes a cada tipo. Para cada tipo de conexão, há um métrica retirada e adaptada de (BRIAND; DALY; WUEST, 1999) para medi-lo. Essas métricas são apresentadas na seqüência. Lembrando que neste trabalho, aspectos e classes são chamados de entidades e métodos e *advices* de operações.

- CBO' (*Coupling between Objects*): CBO' para uma entidade é a contagem do número de outras entidades as quais ela está acoplada, incluindo acoplamento devido a herança. Um objeto de uma entidade é acoplado a outro, se suas operações usam operações ou atributos de outra;
- MPC' (*Messaging Passing Coupling*): é o número de invocações estáticas a operações não implementadas em uma entidade x por operações implementadas em x. Indica quão dependente as operações de uma entidade são de operações de outras entidades;

Tabela 4.1: Tipos de conexão de acoplamento.

	Tipo de conexão	Elemento 1	Elemento 2	Mecanismos de interação que constitui acoplamento	Métricas
1	conexão operação-atributo	operação o de uma classe ou aspecto c	atributo a de uma classe ou aspecto c', com c $\neq$ c'	o referencia a	CBO'
2	conexão operação-operação	operação o de uma classe ou aspecto c	operação o' de uma classe ou aspecto c', com o $\neq$ o' e c $\neq$ c'	o invoca o'	CBO', MPC'
3	conexão por tipo de atributo	classe ou aspecto c	atributo a de classe ou aspecto c', com c $\neq c'$	c é tipo de c' (agregação)	DAC'
4	conexão por tipo de parâmetro	classe ou aspecto c	operação o de classe ou aspecto c', com c $\neq$ c'	c é tipo de um parâmetro ou retorno de o	Nenhuma

- DAC' (*Data Abstraction Coupling*): contagem do número de atributos não herdados que têm uma entidade como seu tipo.

A força do acoplamento pode ser medida de duas formas diferentes. Pela freqüência das conexões ou pelo tipo do acoplamento. Se se considerar cada tipo de acoplamento possuindo uma força diferente, cada tipo deve ser avaliado separadamente. Para este trabalho, a fim de simplificação, a força do acoplamento será determinada pela freqüência das conexões e todos os tipos possuirão forças idênticas. Assim, quanto maior a freqüência das conexões, maior será o acoplamento, não importando qual o tipo da conexão.

A direção do acoplamento, que identifica uma relação de cliente-servidor entre as classes, é um item de muito conceito para sistemas de *middleware*. Por exemplo, é importante saber a relação de um serviço, que pode ser adicionado ou removido do núcleo principal do *middleware*, com outras classes. Se esse serviço está somente sendo usado (exporta serviços) por outras classes, provavelmente ele atenderá melhor às questões de reusabilidade. Se ele usa (importa serviços) outras classes, merece maior atenção do momento de distribuição, para que as classes das quais ele depende estejam disponíveis no momento de sua execução. Esse tipo de acoplamento, neste trabalho, é chamado acoplamento importado/exportado. Foi separado do acoplamento entidade-entidade por ser relevante para a propriedade reusabilidade e não especificamente para a modularidade. O acoplamento importado/exportado será detalhado na seção 4.2.7.

A granularidade determina a que nível de detalhe as informações serão coletadas. Dos níveis de granularidade presentes em (BRIAND; DALY; WUEST, 1999), para sistemas

Tabela 4.2: Opções de contagem de acoplamento.

	Descrição	Exemplo de acoplamento de importação	Exemplo de acoplamento de exportação	Métricas correspondentes
1	soma-se o número individuas de conexões para cada operação ou atributo de uma classe/aspecto	o número total de atributos referenciados por métodos na classe	o número total de referências a atributosda classe	MPC, DAC
2	para uma classe ou aspecto c, conta-se o número de outras classes para as quais há pelo menos uma conexão	o número de classes que têm um atributo que é referenciado por um método de uma classe c	o número de classes que têm um método que referencia um atributo da classe c	CBO

de *middleware*, o acoplamento mais relevante a ser medido é o a nível de classe, visando determinar se a definição de cada classe é satisfatória. Por isso, todas as métricas previamente escolhidas, CBO', MPC' e DAC', avaliam o acoplamento a nível de classe. É importante ressaltar que a granularidade deve ser escolhida tendo em mente o que se deseja avaliar e o tipo de sistema sendo avaliado.

Em (MARTIN, 1994), Martin proprõe uma granularidade interessante: categoria de classes. Segundo ele, uma classe que é parte de um conjunto de classes colaboradoras, não pode ser separada facilmente. As classes pertencentes a esse conjunto são altamente coesivas. As classes colaboradoras estão empenhadas em realizar uma mesma funcionalidade ou atigir algum objetivo comum. Nesse caso, o acoplamento entre classes de uma categoria não é um ponto fraco, mas sim algo esperado. Afinal, por realizar uma mesma funcionalidade, as classes são interdependentes entre si. Logo, o acoplamento que se deseja avaliar é o acoplamento entre categorias de classes e não entre classes.

Ainda no que diz respeito à granularidade, é preciso determinar a forma como serão contadas as conexões entre classes. É importante lembrar que a contagem é feita somente para conexão de classes por ser o nível escolhido para a granularidade. Das várias opções de contagem fornecidas pelo *framework* original, mostradas na tabela 3.3, foram escolhidas as mais usadas em outros trabalhos. As opções de contagem selecionadas e as métricas correspondentes são apresentadas na tabela 4.2.

O acoplamento direto ou indireto determina se será considerada transitividade nas interações de acoplamento. Como geralmente a conexão indireta é utilizada para fins de rastreabilidade e esse não é o intuito aqui, somente o acoplamento direto é considerado

nas contagens das conexões.

A herança parece ser um ponto sobre o qual não há consenso nos trabalhos de métricas. A maioria dos trabalhos, segundo (BRIAND; DALY; WUEST, 1999), não consideram o acoplamento por herança. As métricas são definidas sem que se faça referência a ela. Por esse motivo, neste trabalho, optou-se por não tratar acoplamento baseado em herança. A herança é considerada de forma dissociada do acoplamento na subseção 4.2.3.

Para utilizar de maneira eficiente o *framework* de acoplamento, escolhe-se, para cada item levantado, que tipo deseja medir. Por exemplo, dos cinco tipos de conexão mostrados, o usuário deve decidir quais tipos são relevantes de acordo com a análise que ele quer fazer e o sistema a ser avaliado. Em seguida, deve utilizar as métricas associadas à sua escolha.

Para mostrar a aplicação do *framework* de acoplamento considere o código mostrado na listagem 4.1. A classe *MonitorTemperatura* é responsável por instanciar a classe *ControladorTemperatura* da figura 4.1, informando o valor da temperatura ideal. O método *monitorarTemperatura* é chamado sempre que há uma variação da temperatura ambiente. Ele é o responsável por enviar uma mensagem à instância de *ControladorTemperatura* para decidir se deve ligar/desligar o aquecedor ou o ar condicionado, bem como enviar os comandos aos equipamentos para ligar e desligá-los.

Listagem 4.1: programa exemplo para aplicação do framework de acoplamento.

```
public class MonitorTemperatura {
        ControladorTemperatura controlador;
        boolean arcond, aquecedor;
5
        public MonitorTemperatura(){
                controlador = new ControladorTemperatura(26);
                arcond = aquecedor = false;
        }
10
        //metodo chamado a cada variacao de temperatura
        public void monitorarTemperatura(float tempAmbiente){

                if(controlador.ligarArCondicionado(tempAmbiente) && !arcond
                    ){
15                      arcond = true;
                        //comandos necessarios para ligar o ar condicionado
                        if(aquecedor){
                                aquecedor = false;
```

```
                                    //comandos para desligar o aquecedor
20                              }
                        }else if(controlador.ligarAquecedor(tempAmbiente) && !
                            aquecedor){
                                aquecedor = true;
                                //comandos necessarios para ligar o aquecedor
                                if(arcond){
25                                      arcond = true;
                                        //comandos necessarios para desligar o ar
                                            condicionado
                                }
                        }
                }
30  }
```

Para cada critério, escolhe-se dentre as opções disponíveis, as que mais se adequam ao programa e ao propósito da avaliação:

- tipos de conexão: para esse programa serão considerados os tipos conexão operação-operação e conexão por tipo de atributo, números 2 e 3 na tabela 4.1, respectivamente. Esses são os dois únicos tipos de conexão entre as classes dadas como exemplo;
- força de acoplamento: seguindo a sugestão do *framework*, a força do acoplamento será medida com de acordo com a freqüência, todos os tipos de conexão têm a mesma força;
- direção de acoplamento: as classes *ControladorTemperatura* e *MonitorTemperatura* se relacionam em uma arquitetura cliente/servidor. Esta usa os serviços daquela;
- granularidade: o acoplamento será medido a nível de classe, para que se possa determinar se a definição de cada classe é satisfatória;

De acordo com as opções escolhidas, as métricas selecionadas são CBO, MPC e DAC. A granularidade utilizada para a definição das métricas é entidade.

É difícil avaliar o resultado da métrica CBO' para os programas da figura 4.1 e na listagem 4.1, dado que o sistema não é mostrado completamente. A classe *Controlador-Temperatura* provavelmente está acoplada somente a *MonitorTemperatura*, logo CBO' = 1 para ControladorTemperatura. A classe *MonitorTemperatura* está, no mínino, acoplada

a duas classes: *ControladorTemperatura* e à classe que invoca o método *monitorarTemperatura*, a cada vez que ocorre uma variação da temperatura ambiente. Assim, CBO' $\geq$ 2 para a classe *MonitorTemperatura*.

A métrica MPC' conta, para uma determinada classe, o número de chamadas a métodos implementados fora dessa classe. No caso da classe *ControladorTemperatura*, não há chamadas a métodos de outras classes do sistema, somente a métodos da biblioteca Java responsável por manipulação de arquivos. Assim, MPC' = 0, pois chamadas a métodos da biblioteca Java não são consideradas na contagem. A classe *MonitorTemperatura* contém duas invocações a métodos da classe *ControladorTemperatura*. Logo, para *MonitorTemperatura*, MPC' = 2.

A classe *MonitorTemperatura* é responsável por instanciar a classe *ControladorTemperatura*, assim aquela possui como atributo com tipo desta. Assim, DAC' = 1.

Coesão

Coesão é o grau em que os métodos e os atributos de uma classe trabalham juntos (BRIAND; DALY; WUST, 1998). Essa propriedade é avaliada em diversos trabalhos sobre métricas de software, para citar alguns: (SANT'ANNA, 2008), (BRIAND; DALY; WUST, 1998), (CHIDAMBER; KEMERER, 1994), (SANT'ANNA et al., 2003), (ZHAO; XU, 2004), (KUMAR; KUMAR; GROVER, 2008). É considerada uma das características fundamentais para a avaliação da modularidade de um sistema de software.

Aplicada ao paradigma orientado a objetos, geralmente a coesão é medida pela similaridade entre métodos de uma classe, ou seja, se eles acessam o mesmo conjunto de atributos da classe. Em (SANT'ANNA, 2008), trabalho apresentado no capítulo de trabalhos relacionados na seção 3.8, a coesão é tratada de uma perspectiva diferente: a métrica para coesão é definida baseada em conceito, uma vez que o foco do trabalho é o *design* orientado a aspectos. Conceito é definido por Sant'Anna em (SANT'ANNA, 2008) como qualquer propriedade importante ou área de interesse de um sistema que se quer tratar de forma modular. A métrica para coesão é apresentada a seguir.

- LCC (*Lack of Concern-based Cohesion*): número de conceitos endereçados por um dado componente (SANT'ANNA, 2008).

Como este trabalho têm como foco sistemas de *middleware* orientado a aspectos, em que também se deseja tratar os conceitos de forma modular, escolheu-se essa métrica para

Figura 4.2: Classe Cliente.

avaliar a coesão no conjunto de métricas proposto. Adequando a definição da métrica para os termos do trabalho, tem-se:

- LCC (*Lack of Concern-based Cohesion*): número de conceitos endereçados por uma dada entidade.

Para ilustar a aplicação da métrica LCC, considere a classe Cliente presente na figura 4.2. A classe Cliente é uma classe simples definida para requisitar serviços a uma classe servidor remota. O método conectarServidor() deve obter referência para o objeto servidor e se conectar a ele. O método criptografarMsg() criptografa as mensagens a serem enviadas ao servidor e descriptografarMsg() descriptografa as mensagens recebidas dele. O método gravarLog() armazena em um arquivo todas as ações realizadas pelo objeto da classe. Caso seja definido que a granularidade para coleta da métrica é entidade e que os conceitos do sistema são distribuição, segurança e *logging*, podemos dizer que essa classe possui os três conceitos em sua definição. Logo, LCC = 3.

4.2.2 Manutenibilidade

Em (BRIAND; MORASCA; BASILI, 1993), Briand et al. analisam o *design* de alto nível de um sistema de software com o propósito de prever e avaliar a dificuldade de mudança do ponto de vista dos projetistas. Em outras palavras, os autores avaliam a manutenibilidade de *design*. Para isso, usam métricas de coesão e acoplamento.

Em (FIGUEIREDO et al., 2008), Figueiredo et al. apresentam um *framework* orientado a conceitos que suporta a instanciação e comparação de métricas orientadas a conceitos utilizadas em estudos empíricos de manutenibilidade. O *framework* propõe uma padronização de terminologia e critérios para possibilitar a comparação das métricas. Os autores também propõem três novas métricas que medem os atributos: (i) acoplamento, (ii) coesão e (iii) espalhamento de um conceito, associando essas métricas com a avaliação da propriedade manutenibilidade. Ainda segundo Figueiredo et al., a manutenibilidade de

design de software orientado a aspectos requer que os desenvolvedores raciocinem sobre a modularidade de conceitos do sistema.

Outro trabalho que relaciona modularidade à manutenibilidade é (SANT'ANNA, 2008). Em (SANT'ANNA, 2008), Sant'Anna afirma que modularidade é um atributo que influencia manutenibilidade.

Baseando-se nesses trabalhos, o conjunto de métricas proposto neste trabalho define a propriedade manutenibilidade como dependente da modularidade. Essa associação fica em conformidade com (FIGUEIREDO et al., 2008), pois a modularidade possui como fatores: acoplamento, coesão e separação de conceitos.

A manutenibilidade está relacionada também ao tamanho do código. Conforme dito anteriormente, quanto maior um sistema, mas difícil se torna sua compreensão, resultando em manutenção mais complexa (CHIDAMBER; KEMERER, 1994).

Não foram definidas métricas particulares neste trabalho para a propriedade manutenibilidade. Portanto, ela deve ser avaliada em função das propriedades modularidade e tamanho, que serão apresentadas nas subseções 4.2.1 e 4.2.6, respectivamente.

4.2.3 Reusabilidade

Reuso de software, o uso de artefatos ou conhecimento de software existentes para criar novo software, é um método chave para melhorar a produtividade e a qualidade de software. Reusabilidade é o grau em que uma coisa pode ser reusada (FRAKES; TERRY, 1996). Em (FRAKES; TERRY, 1996), pode-se encontrar um *framework* sobre os métodos mais utilizados para a avaliação de reusabilidade de software. Os métodos utilizados são os mais diversos: desde métricas para herança até análise baseada em falhas que um componente pode apresentar.

Em (WASHIZAKI; YAMAMOTO; FUKAZAWA, 2003), Washizaki et al propõem um conjunto de métricas para avaliar a reusabilidade de componentes do tipo caixa-preta, baseando-se na informação estática limitada fornecida pela interface do componente. Em contrapartida, o conjunto de métricas proposto neste trabalho utiliza as informações obtidas do código-fonte e *design* dos componentes para analisar e prever a reusabilidade.

O trabalho (POULIN, 1994), assim como o (FRAKES; TERRY, 1996), apresenta diversas métricas existentes para a avaliação da reusabilidade de componentes. Segundo Poulin, em (POULIN, 1994), a avaliação através de métodos empíricos baseia-se em mé-

tricas de complexidade e tamanho. Como neste trabalho a definição de complexidade refere-se à complexidade interna de uma entidade, não se associa reusabilidade como dependente de complexidade. Em relação à propriedade tamanho, neste trabalho, ela é considerada um dos fatores da reusabilidade.

Para avaliar a reusabilidade, o conjunto de métricas proposto associa essa propriedade a estabilidade. Segundo Martin em (MARTIN, 1994), um design é difícil de reusar quanto as partes a serem reutilizadas são altamente dependentes de detalhes que não se quer reutilizar. Nesse caso, os projetistas terão mais trabalho para separar as partes desejáveis das indesejáveis. Quanto mais estável é uma parte de *design*, mais independente de outras partes ela é. Assim, estabelece-se a relação entre reusabilidade e estabilidade de uma classe.

A reusabilidade depende do tamanho do sistema e do acoplamento entre entidades. A relação de dependência reuso-tamanho, como dito anteriormente, baseia-se no exposto em (CHIDAMBER; KEMERER, 1994): quanto maior forem os componentes, mais difícil será manter independência em relação a uma aplicação específica. A relação de dependência reuso-acoplamento também está definida no trabalho (CHIDAMBER; KEMERER, 1994), segundo o qual quanto mais independente uma classe é em relação a outras, ou seja, quanto menor o acoplamento de uma classe, mais fácil será reusá-la em outro contexto. Nesse caso, não importa se o acoplamento é a classes ou aspectos, mas em relação a outras entidades do sistema. Assim, o acoplamento considerado aqui é o entidade-entidade.

Um outro elemento que reflete a reusabilidade é a herança: quanto maior o número de classes filhas de uma classe, maior o reuso, uma vez que herança é uma forma de reuso (CHIDAMBER; KEMERER, 1994). Por isso, utilizar-se-á métricas relacionadas a esse fator do trabalho (CHIDAMBER; KEMERER, 1994) (apresentadas na seção 3.2): NOC e DIT. Elas serão adaptadas de forma a poderem ser aplicadas a sistemas AO.

Resumindo, a reusabilidade não possui métricas próprias. Ela deve ser avaliada em função de estabilidade, tamanho, acoplamento entidade-entidade e herança, apresentadas nas subseções 4.2.7, 4.2.6, 4.2.1 e 4.2.3 respectivamente.

Herança

Nessa subseção, métricas convencionais para herança em sistemas OO são adaptadas para poderem ser aplicadas a sistemas AO.

- DIT' (*Depth of Inheritance Tree*) - Profundidade da Árvore de Herança: é o tamanho

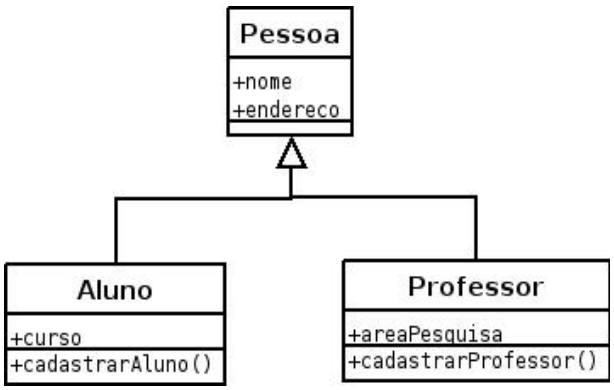

Figura 4.3: Exemplo de sistema que apresenta relacionamento de herança entre classes.

da profundidade de uma entidade na árvore de hierarquia de herança. A árvore de herança é um grafo direcionado acíclico e as entidades são representados como nós. Essa medida é o comprimento do maior caminho entre uma determinada entidade até o nó raiz da árvore. Segundo o trabalho (CHIDAMBER; KEMERER, 1994), quanto mais profunda uma entidade está em uma hierarquia de herança, maior o potencial de reuso das operações herdadas. Em outras palavras, quanto maior a profundidade de uma árvore de herança, maior a reusabilidade;

- NOC' (*Number of Children*) - Número de Filhos: é o número de entidades imediatas subordinadas a uma outra entidade. O trabalho (CHIDAMBER; KEMERER, 1994) afirma que quanto maior o número de classes filhas, maior o reuso da classe, uma vez que herança é uma forma de reuso. Portanto, pode-se dizer que, se uma classe possui um grande número de classes filhas, essa classe possui grande potencial de reuso.

Para exemplificar a aplicação das métricas para herança, considere a figura 4.3. Ela mostra um diagrama UML com 3 classes: a classe Pessoa e as classes Aluno e Professor que herdam de Pessoa. Tanto Aluno como Professor possuem DIT' = 1. Como a classe Pessoa possui somente duas classes filhas, NOC' = 2 nesse caso. É importante lembrar que avaliando somente os valores das métricas de herança, é difícil predizer o potencial de reuso da classe Pessoa. Faz-se necessário avaliar também valores das métricas referentes a acoplamento e tamanho.

4.2.4 Flexibilidade

No contexto de *middleware* orientado a aspectos, flexibilidade refere-se a facilidade de incluir ou remover um aspecto ou funcionalidade no código-base do *middleware*. Assim, o acoplamento relevante para a flexibilidade é o acoplamento código base-aspecto. Isso porque somente os aspectos é que têm a possibilidade de serem plugados ou desplugados no *middleware*. Quanto menor esse acoplamento, mais facilmente se poderá incluir e/ou remover aspectos e, portanto, mais flexível torna-se o *middleware*.

Segundo Sant'Anna et al em (SANT'ANNA et al., 2003), a propriedade de flexibilidade determina-se não só através do acoplamento, mas também em função da separação de conceitos. Baixo acoplamento e separação de conceitos promovem a flexibilidade facilitando o processo de adicionar e remover funcionalidades.

A propriedade flexibilidade, assim como reuso e manutenção, não possui métricas próprias, sendo avaliada em função do acoplamento código base-aspectos (subseção 4.2.4) e da separação de conceitos (subseção 4.2.1).

Acoplamento Código Base-Aspecto

Em algumas situações é importante avaliar o acoplamento entre o código base e os aspectos de um sistema. Nesse caso, foram definidas métricas que dissociam o sistema com relação a esses dois tipos de entidade. Com esse propósito, selecionou-se métricas apresentadas em (CECCATO; TONELLA, 2004), trabalho apresentado na seção 3.7 no capítulo sobre trabalhos relacionados. Essas métricas são adequadas para seguirem a terminologia do trabalho:

- CAE (*Coupling in Advice Execution*): número de aspectos contendo *advices* possivelmente disparados pela execução de operações de uma dada entidade. Se o comportamento de uma operação pode ser alterado por um *advice*, há uma dependência da operação em relação ao *advice*. Assim, a entidade dessa operação está acoplada ao aspecto que contém o *advice*. Uma mudança no *advice* causa impacto na operação;

- CIM (*Coupling on Intercepted Modules*): número de entidades ou interfaces explicitamente nomeadas nos *pointcuts* pertencentes a um aspecto. CIM captura o conhecimento direto do restante do sistema. Alto valores de CIM indicam alto acoplamento do aspecto com uma aplicação e baixa generalidade, reusabilidade;

- CDA (*Crosscutting Degree of an Aspect*): número de entidades afetadas pelos *pointcuts* e por *introductions* em um dado aspecto. Enquanto CIM considera somente entidades nomeadas explicitamente, CDA mede todos as entidades possivelmente afetadas por um aspecto. Isso dá uma idéia do impacto total de um aspecto nas outras entidades do sistema. A diferença entre CDA e CIM dá o número de entidades que são afetadas por um aspecto sem serem referenciadas explicitamente pelo aspecto, o que indica o grau de generalidade de um aspecto em termos de sua independência de classes/aspectos específicos.

Para exemplificar a aplicação das métricas de acoplamento código base-aspectos, considere um sistema composto pelas classes *MonitorTemperatura* (listagem 4.1) e *ControladorTemperatura* (figura 4.1) e o aspecto *ImprimirTela* da listagem 4.2. Esse aspecto foi definido usando a linguagem de programação orientada a aspectos AspectJ(ASPECTJ..., 2008). As classes fazem *logging* das ações do sistema em um arquivo, mas não imprimem nada na tela para que o usuário possa acompanhar a execução do sistema. Incluiu-se então um aspecto que imprime na tela sempre que for chamado um método para ligar algum equipamento e métodos da classe *MonitorTemperatura* forem invocados. Assim definiu-se um *pointcut* chamado pointcutImpTela (linha 1) que interceptada chamadas ao métodos que iniciem com a palavra "ligar" e todos os métodos da classe MonitorTemperatura. O *advice* (linhas 4-7) consiste em imprimir na tela a assinatura do método sendo executado.

A métrica CAE conta o número de aspectos contendo *advices* possivelmente disparados pela execução de operações de uma entidade. Como nenhum outro aspecto incide no aspecto *ImprimirTela*, CAE = 0. A métrica CIM conta o número de entidades ou interfces explicitamente nomeadas nos *pointcuts* de um aspecto. No aspecto *ImprimirTela*, somente a entidade *MonitorTemperatura* é nomeada, embora o aspecto também incida em *ControladorTemperatura*. Logo, CIM = 1. Por fim, a métrica CDA conta o número de entidade afetadas por *pointcuts* de um aspecto. O aspecto *ImprimirTela* , nesse sistema, afeta as duas outras classes. Assim, CDA = 2.

Listagem 4.2: programa exemplo para aplicação das métricas de acoplamento código base-aspectos.

```
public aspect ImprimirTela {
        pointcut pointcutImpTela(): call (* *.ligar*(..)) || call(*
            MonitorTemperatura.*(..));

        after(): pointcutImpTela(){
```

```
5           System.out.println("Executando comando: " + thisJoinPoint.
                getSignature());
            }
    }
```

4.2.5 Complexidade

Segundo (CHIDAMBER; KEMERER, 1994), quanto maior o número de métodos que uma classe herda, mais complexo é rastrear seu comportamento. Pode-se inferir dessa afirmação que o fator herança influencia diretamente a complexidade de um sistema. Logo, as métricas apresentadas na subseção 4.2.3 serão consideradas também para essa propriedade.

Uma outra medida de complexidade, mais uma vez segundo (CHIDAMBER; KEMERER, 1994), é a métrica resposta para uma classe (RFC), mostrada na subseção 3.2. O autor afirma que se um grande número de métodos podem ser invocados em resposta a uma mensagem, o sistema será mais complexo. Assim, essa métrica também é incluída para a propriedade complexidade.

A métrica RFC precisa ser adaptada aos moldes da orientação a aspectos. A definição original encontrada (CHIDAMBER; KEMERER, 1994) diz que RFC é o conjunto resposta de uma classe, ou seja, o conjunto de métodos que podem ser potencialmente executados em resposta a uma mensagem recebida por um objeto daquela classe. Adaptando essa métrica para a orientação a aspectos, tem-se:

- RFC'(*Response for a Class*): o conjunto resposta a uma entidade passa a ser o conjunto de operações que podem ser potencialmente executados em resposta a uma mensagem recebida por uma instância daquela entidade. Só serão consideradas na contagem operações pertencentes à entidade que recebeu a mensagem, para que se possa focalizar na complexidade de se compreender a entidade e desconsiderar acoplamento entre entidades.

Entram na conta *advices* invocados explicitamente ou não, no código da classe ou aspecto. Essa modificação visa refletir a condição de que quanto mais *advices* são invocados durante a execução de uma instância da entidade, mais complexa sua estrutura e mais difícil de compreender o fluxo de execução da instância da entidade.

A aplicação das métricas de complexidade é mostrada a seguir. RFC' é medido como

número de métodos que podem ser executados em resposta a uma mensagem recebida por um objeto da classe. Considere o programa mostrado na figura 4.1. Qualquer que seja mensagem recebida por um objeto da classe ControladorTemperatura, somente um método será executado, considerando que nenhum dos métodos da classe invocam outros métodos. Portanto, para esse caso, RFC' = 1, indicando que a classe possui baixa complexidade.

4.2.6 Tamanho

Para medir o software fisicamente, utiliza-se métricas de tamanho. Para essa propriedade, utiliza-se o conjunto de métricas para tamanho proposto no trabalho (SANT'ANNA et al., 2003), por ser um conjunto completo e bem definido. O conjunto em questão é composto das métricas VS, LOC, NOA e WOC, apresentado na seção 3.5. As métricas serão redefinidas para ficar em conformidade com a terminologia deste trabalho:

- VS (*Vocabulary Size*): número de entidades do sistema;
- LOC (*Lines of Code*): número de linhas de código. Comentários e linhas em branco são desconsideradas;
- NOA (*Number of Attributes*): número de atributos de uma entidade. Atributos herdados não são incluídos;
- WOC (*Weighted Operations per Component*): soma das complexidades das operações de uma entidade. A complexidade de uma operação é definida em função do número de parâmetros.

A seguir, será apresentado um exemplo simples de aplicação das métricas de tamanho.

A fim de exemplificação, as métricas de tamanho serão aplicadas ao programa da figura 4.1.

A métrica VS conta o vocabulário do sistema: o número de classes e aspectos presentes no sistema. Como esse é um programa exemplo simples e possui uma única entidade, VS = 1.

A métrica LOC conta o número de linhas de código. Nesse caso, LOC = 27. As linhas em branco são desconsideradas.

A métrica NOA conta o vocabulário interno — o número de atributos — de cada entidade. Como a classe ControladorTemperatura possui somente dois atributos, NOA = 2.

WOC mede a complexidade de uma entidade em termos de suas operações. A complexidade das operações é definida em relação ao número de parâmetros, considerando então que uma operação com mais parâmetros é mais complexa. No programa exemplo, como todos os métodos possuem 0 ou 1 parâmetro, WOC = 3.

O programa possui tamanho pequeno de acordo a aplicação das métricas, como se poderia intuir pela observação de seu código.

4.2.7 Estabilidade

Apesar de estabilidade não ser uma tema recorrentemente abordado em conjuntos de métricas, essa propriedade foi adicionada ao conjunto de métricas proposto por este trabalho por ser considerada importante para a avaliação de reuso de entidades (MARTIN, 1994).

Segundo Martin em (MARTIN, 1994), uma classe que possui outras classes dependendo dela, deve ser estável. Uma mudança nela pode ocasionar mudanças em várias outras classes. Logo, quanto mais classes dependentes de uma determinada classe, mais estável esta é. Analogamente, se uma classe depende de várias outras classes, é esperado que ela seja instável, porque qualquer mudança nas classes das quais ela depende ocasiona uma mudança nela. Baseando-se nesse raciocínio, em (MARTIN, 1994), a estabilidade de uma classe é avaliada em termos da (in)dependência de uma classe em relação às outras. Para medir a (in)dependência de uma classe, utiliza-se as métricas para acoplamento importado/exportado, apresentadas nessa mesma subseção.

Acoplamento Importado/Exportado

Martin em (MARTIN, 1994), define suas métricas sobre estabilidade baseado-se em categorias como granularidade. Segundo ele, uma categoria é um conjunto de classes que trabalham juntas para realizar um determinado objetivo. A partir dessa definição, ele define que a estabilidade de uma categoria pode ser medida pela contagem de dependências que interagem com cada categoria através das métricas:

- Ca (*Afferent Coupling*): número de classes fora de uma categoria que dependem de

classes de dentro dessa categoria.

- Ce (*Efferent Coupling*): número de classes de dentro de uma categoria que dependem de classes de fora dessa categoria;
- I (*Instability*): (Ce / (Ca + Ce)): essa métrica varia entre [0,1], fornecendo um valor normalizado para a instabilidade. I=0 indica uma categoria maximamente estável e I=1, uma categoria maximamente instável.

Em alguns sistemas, pode ser difícil determinar categorias e quais classes pertencem a cada uma delas. É possível que um objetivo seja realizado por uma única entidade. Assim, as métricas foram redefinidas de forma a mudar a granularidade de categoria para entidade, deixando-as em conformidade com as outras métricas apresentadas. Isso não impede que essas métricas sejam coletadas na granularidade categoria de classes também.

- Ca' (*Afferent Coupling*): número de entidade que usam ou dependem de uma entidade específica. Essa métrica claramente refere-se ao acoplamento exportado.
- Ce' (*Efferent Coupling*): número de entidade que uma entidade específica usa ou depende. Essa métrica refere-se ao acoplamento importado.
- I' (*Instability*): (Ce / (Ca + Ce)): essa métrica varia entre [0,1]. I=0 indica uma entidade maximamente estável e I=1, uma entidade maximamente instável.

A partir da definição de instabilidade, podemos concluir que quanto maior o acoplamento eferente, ou seja, o acoplamento importado por uma classe, menor é a sua instabilidade. Analogamente, quanto maior é a acoplamento aferente ou exportado por uma classe, maior é a sua instabilidade. Esse é um conceito importante para reusabilidade, uma vez que, quanto maior a estabilidade de uma classe, mais adequada está para o reuso.

Para dar um exemplo da aplicação das métricas de estabilidade, considere as classes *MonitorTemperatura* (listagem 4.1) e *ControladorTemperatura* (figura 4.1). Suponha que um sistema contém apenas essas duas classes, para simplificar o exemplo. Vamos aplicar as métricas primeiramente a *MonitorTemperatura*. A classe *MonitorTemperatura* invoca os métodos *ligarArCondicionado()* e *ligarAquecedor()* de *ControladorTemperatura*. Logo, para *MonitorTemperatura*, Ce' = 1. Como nenhuma classe depende de *MonitorTemperatura*, Ca' = 0. Assim, temos I' = 1. De acordo com as indicações de I', podemos dizer que *MonitorTemperatura* é maximamente instável.

Aplicando as métricas a *ControladorTemperatura*, temos exatamente o inverso. Como *ControladorTemperatura* não depende de nenhuma outra classe, Ce' = 0. A classe *MonitorTemperatura* invoca métodos dela, logo Ca' = 1. Calculando a instabilidade, I' = 0. De acordo com as indicações de I', podemos dizer que *MonitorTemperatura* é maximamente estável.

Avaliando a estabilidade pelo critério da dependência entre classes, como *ControladorTemperatura* não depende de nenhuma outra e há uma classe que depende dela, ela é classificada como estável. Da forma análoga, *MonitorTemperatura* não possui nenhuma classe que depende dela, mas ela depende de *ControladorTemperatura*, sendo então classificada como instável.

4.3 Resumo do Conjunto de Métricas Estáticas

Uma vez definidos o conjunto de métricas e as propriedades a serem avaliadas em sistemas de *middleware*, apresenta-se na tabela 4.3 um resumo da proposta. As tabelas 4.4 e 4.5 oferece um resumo das métricas e também a fase do desenvolvimento em que podem ser aplicadas. A fim de simplificação, neste trabalho, o termo entidade refere-se a classes e aspectos e o termo operações, a métodos e *advices*. A definição de em que fase de desenvolvimento cada métrica pode ser aplicada baseia-se nos trabalhos em que foram originalmente encontradas.

Em (ZHANG; JACOBSEN, 2003), trabalho com foco em refatoração de *middleware* orientado a aspectos, diz-se que métricas relacionadas a tamanho e acoplamento são coletadas nas classes envolvidas na refatoração — fase de refatoração. Assim, quando se desejar avaliar um sistema que passou por um processo de refatoração, deve-se utilizar métricas relacionadas às propriedades tamanho e acoplamento do conjunto de métricas proposto. Os autores não mencionam métricas para separação de conceitos, mas entende-se que esse é um dos objetivos principais ao se realizar um processo de refatoração. Como acoplamento e separação de conceitos são fatores da modularidade neste trabalho e a aspectização de um sistema é realizada visando melhorar sua organização, ou seja, sua modularização, as métricas das propriedades modularidade e tamanho serão consideradas relevantes para a avaliação desse processo. As tabelas 4.4 e 4.5 definem quais métricas são aplicadas na fase de refatoração.

É importante observar que as métricas não são específicas para avaliação de sistemas de *middleware*, mas foram selecionadas como meio de avaliar propriedades relevantes a

Tabela 4.3: Resumo do conjunto de métricas referente às fases de desenvolvimento design, implementação e refatoração do conjunto de métricas.

Propriedades	Relaciona-se à(s) propriedade(s)	Métricas
1. Modularidade	separação de conceitos, acoplamento entidade-entidade e coesão	
1.1. Separação de conceitos		CDC, CDO e CDLOC
1.2. Acoplamento Entidade-Entidade		CBO', MPC' e DAC'
1.3. Coesão		LCC
2. Manutenibilidade	modularidade e tamanho	
3. Reusabilidade	tamanho, estabilidade, acoplamento entidade-entidade e herança	
3.1. Herança		NOC' e DIT'
4. Flexibilidade	separação de conceitos e acoplamento código base-aspecto	
4.1. Acoplamento Código Base-Aspecto		CAE, CIM, CDA
5. Complexidade	herança	RFC'
6. Tamanho		VS, LOC, NOA e WOC
7. Estabilidade	acoplamento exportado/importado	
7.1. Acoplamento exportado/importado		Ca', Ce', I'

esse contexto de acordo com (DRIVER, 2002).

4.4 Definição de Conjunto de Propriedades Dinâmicas de Sistemas de *Middleware* OA

O conjunto de propriedades dinâmicas para avaliação de sistemas de *middleware* orientado a aspectos é composto por desempenho e consumo de memória. Considera-se que essas sejam as características mais relevantes na avaliação do comportamento dinâmico desses sistemas. Elas serão descritas a seguir.

Desempenho não faz parte do conjunto de propriedades original de (DRIVER, 2002). Ela foi incluída neste trabalho por ser de fundamental relevância para sistemas de *middleware*. Mesmo que um *middleware* apresente bons resultados para todas as outras propriedades, se apresentar um desempenho ruim, não será viável. Além disso, com essa nova propriedade o conjunto de métricas fica em conformidade com o trabalho (ZHANG; JACOBSEN, 2003).

Tabela 4.4: Resumo das métricas e fase de desenvolvimento em que podem ser aplicadas.

Métrica	Definição	Fase de Desenvolvimento
CDC (*Concern Diffusion over Components*)	número de entidades primárias cujo principal propósito é contribuir para a implementação de um conceito	fase de *design* fase de implementação fase de refatoração
CDO (*Concern Diffusion over Operations*)	número de operações primárias cujo principal propósito é contribuir para a implementação de um conceito.	fase de *design* fase de implementação fase de refatoração
CDLOC (*Concern Diffusion over Lines Of Code*)	número de pontos de transição para cada conceito em todas as linhas de código.	fase de implementação fase de refatoração
CBO' (*Coupling Between Objects classes*)	número de entidades às quais uma entidade está acoplada. Uma entidade está acoplada a outra se usa variáveis ou operações de outra entidade.	fase de *design* fase de implementação fase de refatoração
MPC' (*Messaging Passing Coupling*)	para uma entidade, número de invocações estáticas a operações não implementadas nessa entidade	fase de *design* fase de implementação fase de refatoração
DAC' (*Data Abstraction Coupling*)	número de atributos não herdados que têm uma entidade como seu tipo	fase de *design* fase de implementação fase de refatoração
LCC (*Lack of Concern-based Cohesion*)	número de conceitos endereçados por uma dada entidade.	fase de *design* fase de implementação fase de refatoração
NOC' (*Number of Children*)	número de entidades imediatas subordinados a uma entidade na hierarquia de herança	fase de *design* fase de implementação
DIT' (*Depth of Inheritance Tree*)	máximo comprimento entre um nó e o nó raiz na árvore hierárquica de herança.	fase de *design* fase de implementação
RFC' (*Response For a Class*)	número de operações que podem ser potencialmente executados em resposta a uma mensagem recebido por uma instância da entidade	fase de *design* fase de implementação
CAE (*Coupling on Advice Execution*)	número de aspectos contendo *advices* possivelmente disparados pela execução de operações de uma dada entidade.	fase de *design* fase de implementação
CIM (*Coupling on Intercepted Modules*)	número de entidades ou interfaces explicitamente nomeadas nos *pointcuts* pertencentes a um aspecto.	fase de *design* fase de implementação
CDA(*Crosscutting Degree of an Aspect*)	número de entidades afetadas pelos pelos *pointcuts* e por *introductions* em um dado aspecto.	fase de implementação

Tabela 4.5: continuação da tabela 4.4 - Resumo das métricas e fase de desenvolvimento em que podem ser aplicadas.

Métrica	Definição	Fase de Desenvolvimento
VS (*Vocabulary Size*)	número de entidades do sistema	fase de *design* fase de implementação fase de refatoração
LOC (*Lines Of Code*)	número de linhas de código do sistema	fase de implementação fase de refatoração
NOA (*Number Of Attributes*)	mede o vocabulário interno de cada entidade, ou seja, o número de atributos	fase de *design* fase de implementação fase de refatoração
WOC (*Weighted Operations per Component*)	soma da complexidade de cada operação de uma entidade. A complexidade de uma operação é determina pelo número de parâmetros	fase de *design* fase de implementação fase de refatoração
Ca' (*Afferent Coupling*)	número de entidades que usam ou dependem de uma entidade específica.	fase de *design* fase de implementação
Ce' (*Efferent Couplin*)	número de entidades que uma entidade específica usa ou depende.	fase de *design* fase de implementação
I' (*Instability*)	I = (Ce ÷ (Ca + Ce)): essa métrica varia entre [0,1]. I=0 indica uma entidade maximamente estável e I=1, uma entidade maximamente instável.	fase de *design* fase de implementação

Uma outra propriedade importante é consumo de memória. Ela é importante para avaliar se a solução de *middleware* encontrada satisfaz condições de restrição de memória. Condições de restrições de memória são comumente encontradas em computação móvel e sistemas embarcados.

O fator mais impactante no desempenho e no consumo de memória é o processo de *weaving* e o momento em que ele é realizado. Por isso, recebe atenção especial na segunda parte do conjunto de métricas.

O processo de *weaving* de aspectos pode ser estático ou dinâmico. O estático é aquele que acontece durante a compilação. Os que ocorrem em tempo de carga, de execução ou por interceptação são considerados dinâmicos (HAUPT; MEZINI, 2004). Segue a descrição de cada um desses tipos.

1. Abordagem Tempo de Compilação: para realizar o processo de *weaving* em tempo de compilação, o código do programa original e código de aspectos são combinados em uma nova versão de código que inclui as funcionalidades adicionais. Objetos e instâncias de aspectos inicializam simultaneamente na inicialização da aplicação;

2. Abordagem Tempo de Carga: é possível adicionar funcionalidades a componentes

existentes somente em formato binário. Substitui-se o *loader* de classes padrão por um que permita a modificação das classes em tempo de carga. Para implementar essa abordagem, a maioria das ferramentas decora os *join points shadows* com *hooks* ou *wrappers* que param a execução normal do programa e chamam o *advice* se necessário. O conjunto de *advices* associados com um *join point shadow* particular pode ser determinado em tempo de execução (HAUPT; MEZINI, 2004);

3. Abordagem Interceptação: nessa abordagem, uma aplicação pode ser vista como uma série de eventos que ocorrem quando um *join point* é atingido. As ferramentas baseadas em Java que a implementam usam o *debugger* da máquina virtual Java para atribuir *breakpoints* aos *join points shadows*. Toda vez que um *breakpoint* é atingido, o *debbuger* é utilizado para invocar o *advice* correspondente. Uma outra opção de implementação é modificar a máquina virtual para sinalizar os *join points* quando necessário.

   Como essa abordagem é pouco convencional, para efeito de ilustração do seu funcionamento, apresenta-se PROSE: uma plataforma dinâmica baseada em Java e que segue a abordagem de interceptação para carregar aspectos em sua primeira implementação (POPOVICI; GROSS; ALONSO, 2002). Essa implementação é uma extensão para JVM, baseada na interface de depuração JVMDI (*Java Virtual Machine Debugger Interface*) da JVM. O usuário pode registrar requisição para a execução de eventos dentro da JVM através da JVMDI e controlar a execução para cada notificação de evento. A JVMDI ainda provê meios para inspeção do estado da JVM.

   A inserção de um aspecto em PROSE, faz com que o núcleo do PROSE inspecione a JVM e a instância do novo aspecto para encontrar o conjunto de *join point*s. Para cada novo *join point*, o PROSE requisita uma notificação específica para a JVMDI. Quando uma *thread* t alcança um *join point* registrado, a execução de t é temporariamente suspensa e o *debugger* passa o controle da execução para o PROSE. Nesse ponto, PROSE chama a funcionalidade da instância do aspecto que registrou o *join point*. Depois que a execução do *advice* termina, o PROSE retorna o controle da execução para a aplicação;

4. Abordagem Tempo de Execução: essa categoria engloba os sistemas que realizam o processo de *weaving* de código de aspectos no código da aplicação em tempo de execução. Isso envolve modificar e recompilar métodos. Em tecnologias baseadas em Java, implementar essa abordagem requer a modificação da JVM

(HAUPT; MEZINI, 2004).

4.5 Métricas Dinâmicas para Avaliação de Desempenho em Sistemas de *Middleware* OA

Métricas dinâmicas são aquelas que só podem ser aplicadas exclusivamente durante a execução de um programa. São utilizadas para coletar informações sobre propriedades de execução de programas (DUFOUR et al., 2003). Em sistemas de *middleware* orientados a aspectos, elas são fundamentais para avaliar se a utilização de aspectos implica em perda significante de desempenho. São ainda úteis para avaliar o melhor momento de carga dos aspectos.

Com o objetivo de avaliar os pontos citados acima, foram definidas três métricas baseando-se em (DUFOUR et al., 2004):

1. tempo de execução: mede o *overhead* de utilizar aspectos com um determinado tipo de carga de aspectos. Para medir o *overhead*, considera-se o tempo de execução de um programa;

2. número de operações primárias: utiliza-se operações básicas ao invés de o tempo para fins de comparação. Pode-se citar como exemplos *bytecode* para Java ou operação binária para outras linguagens. Apresenta a vantagem em relação à categoria anterior de ser independente de plataforma;

3. uso de memória: mede a quantidade de memória alocada para a execução de um programa.

As métricas devem ser aplicadas de acordo com o objetivo da medição. Para facilitar o entendimento da aplicação, segue alguns exemplos. Se o objetivo é verificar o custo do uso de aspectos (adoção de DSOA): as métricas devem ser aplicadas às versões OA e OO de um mesmo sistema. Os resultados obtidos servirão como base para decidir se o impacto da utilização de aspectos no desempenho do sistema é aceitável; caso o objetivo seja verificar o custo da carga de aspectos em diferentes momentos para um mesmo sistema: faça diferentes execuções do sistema, em cada uma utilize um mecanismo de carga de aspectos distinto e colete os resultados das métricas dinâmicas. Comparando os resultados das execuções é possível ajustar a melhor combinação para flexibilidade e desempenho.

A figura 4.4 mostra uma visão geral do conjunto de métricas, tanto a parte de avaliação estática quanto a dinâmica, de acordo com o que foi exposto nesse capítulo.

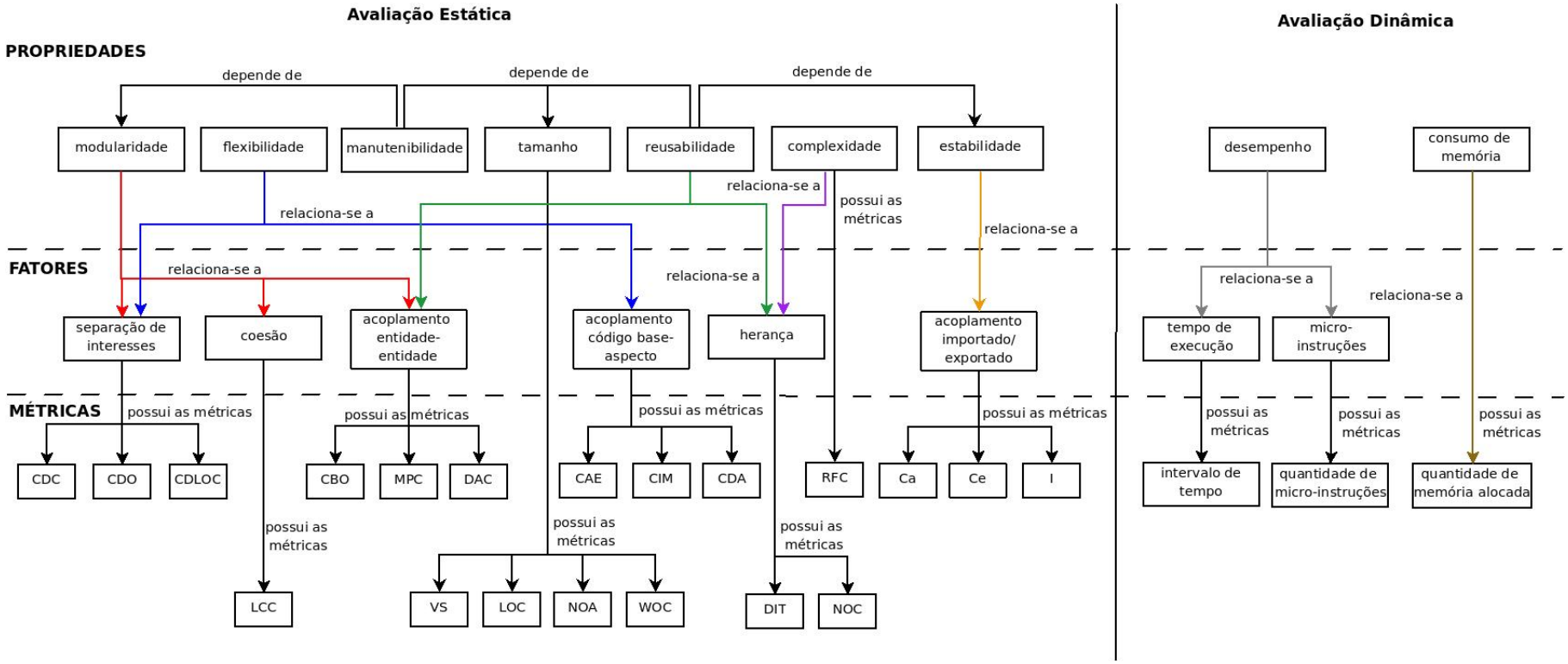

Figura 4.4: Visão geral do conjunto de métricas.

4.6 Análise Comparativa

São muitos os trabalhos que tratam sobre métricas. Nesse capítulo 3, foram descritos os mais próximos e relevantes ao trabalho sendo proposto nessa dissertação. Essa seção será destinada a fazer uma análise comparativa entre os trabalhos relacionados apresentados e o trabalho proposto. A comparação será feita em linhas gerais. Mais detalhes entre o trabalho proposto e os relacionados podem ser encontrados nas seções 4.1 e 4.2.

O trabalho de Briand et al (BRIAND; DALY; WUEST, 1999) (seção 3.1) apresenta uma definição ampla e completa sobre acoplamento. Vários tipos de acoplamento são apresentados com detalhamento. O trabalho proposto nessa dissertação instancia o *framework* de acoplamento, como por exemplo utiliza a definição dos diversos tipos de acoplamento e adapta-os para a abordagem orientada a aspectos. A instanciação do *framework* leva em conta os acoplamentos que podem ocorrer em sistemas de *middleware*. A adaptação para a análise de sistemas orientados a aspectos foi realizada porque acoplamentos que ocorrem em sistemas orientados a objetos ocorrem também em sistemas orientados a aspectos. Dessa forma, o *framework* passa a poder ser aplicado tanto ao desenvolvimento orientado a objetos quanto ao orientado a aspectos. A partir da instanciação e adaptação do *framework* de acoplamento, podemos avaliar o acoplamento entre qualquer par de entidades – na terminologia do trabalho proposto, entidade nomeia classe ou aspecto— do sistema.

O trabalho de Chidamber e Kemerer (CHIDAMBER; KEMERER, 1994) (seção 3.2) apresenta um conjunto de métricas para avaliação de propriedades como acoplamento, coesão, complexidade de sistemas orientados a objetos. Ele contém ainda um amplo embasamento teórico sobre o tema. A métrica RFC adotada pelo presente trabalho e sofreu adaptações para que pudesse também ser aplicada ao desenvolvimento orientado a aspectos. Entretanto, a parte mais importante aproveitada foi as relações entre propriedades constatadas através de estudos empíricos apresentados no trabalho de Chidamber e Kemerer: a manutenibilidade é influenciada por tamanho e acoplamento do sistema, a reusabilidade depende do tamanho dos componentes do sistema e a complexidade está relacionada à herança, ao tamanho e à métrica RFC.

O trabalho de Driver (DRIVER, 2002) (seção 3.3) serviu como base para o trabalho proposto. Driver lista um conjunto de propriedades que devem ser avaliadas em um sistema orientado a aspectos distribuídos. Ele avalia cada propriedade com base em sua experiência. Somente uma métrica é mensurada: LOC (número de linhas de código). O

presente trabalho utiliza a lista de propriedades proposta por Driver. Algumas propriedades são adaptadas (separação de conceitos, capacidade de modificação e extensão, linhas de código) e outras incluídas (modularidade, estabilidade, desempenho e consumo de memória) para se adequar à realidade dos sistemas de *middleware*. Para que o processo de avaliar as propriedades torne-se menos subjetivo e possa ser usado por desenvolvedores menos experientes, atribuiu-se as métricas CDC, CDO, CDLOC, LCC, CBO, DAC, MPC, VC, NOA, WOC, CAE, CIM, CDA, DIT, NOC, RFC, Ca, Ce e I às propriedades para avaliá-las. As relações entre propriedades e entre propriedades e métricas foram construídas com base em outros trabalhos sobre métricas, como (CHIDAMBER; KEMERER, 1994).

O trabalho de Zhang e Jacobsen (ZHANG; JACOBSEN, 2003) (seção 3.4) apresenta um conjunto de métricas para avaliação de sistemas OA, mas tratam apenas do processo de refatoração de *middleware*, com foco na implementação. Apesar de nosso trabalho aproveitar essa abordagem para a definição das métricas relacionadas a refatoração, avaliamos também outras propriedades importantes e em outras fases de desenvolvimento (*design* e implementação).

O trabalho de Sant'Anna et al (SANT'ANNA et al., 2003) (seção 3.5) apresenta um modelo de qualidade com associações entre propriedades e métricas para avaliação de reuso e manutenção de software orientado a aspectos. O conjunto de métricas do presente trabalho baseia-se nesse trabalho relacionado. As propriedades referentes à separação de conceitos e tamanho foram aproveitadas por se adequarem satisfatoriamente aos sistemas de *middleware*.

O trabalho de Haupt e Mezini (HAUPT; MEZINI, 2004) (seção 3.6) apresenta abordagens para avaliação de desempenho da programação orientada a aspectos, oferecendo o embasamento teórico necessário para o presente trabalho.

O trabalho de Ceccato e Tonella (CECCATO; TONELLA, 2004) (seção 3.7) define métricas para avaliação de sistemas orientados a aspectos. As métricas são extensões de métricas tradicionalmente utilizadas no paradigma orientado a objetos. O presente trabalho seleciona métricas de acoplamento desse trabalho para avaliar o acoplamento entre código-base e aspectos de sistemas de *middleware*.

Sant'Anna (SANT'ANNA, 2008) (seção 3.8) apresenta uma estudo aprofundado sobre a propriedade modularidade em sistemas orientados a aspectos. Suas métricas são definidas com foco nos conceitos do sistema. O presente trabalho utiliza esse trabalho relacionado como base para a definição da propriedade modularidade e definição da métrica

LCC.

5 *Ferramenta COMETA-Lua*

A coleta de métricas em código-fonte é um trabalho repetitivo e cansativo. Além disso, a coleta manual atrasa o processo de avaliação de sistemas e torna-o propenso a falhas. Nesse sentido, faz-se necessária a construção de uma ferramenta para automatizar o processo de coleta de resultados de métricas. A maioria dos *frameworks* de avaliação de sistemas trazem esse tipo de ferramenta. Elas consistem na análise do programa-fonte: elementos específicos são extraídos do programa-fonte a ser avaliado e, a partir deles, calculam-se os resultados das métricas.

A análise do programa-fonte consiste em três fases: a análise léxica, a análise sintática e a análise semântica. Na análise léxica, o fluxo de caracteres que constituem o programa é lido da esquerda para a direita e agrupado em *tokens*: seqüências de caracteres com um significado coletivo. Na análise sintática, os *tokens* são agrupados hierarquicamente em coleções aninhadas com significado coletivo. Essa fase verifica se os *tokens* formam uma expressão permitida pela linguagem. A análise semântica é o desenvolvimento das implicações das expressões validadas pela análise sintática e a realização das ações apropriadas (AHO; SETHI; ULLMAN, 1995). No caso de uma ferramenta de coleta de métricas, seria a computação dos resultados das métricas.

Um exemplo de ferramenta para coleta de métricas que utiliza o processo descrito anteriormente é AJATO(FIGUEIREDO; GARCIA; LUCENA, 2006). AJATO computa métricas orientadas a aspectos em sistemas implementados usando Java e AspectJ. Essa ferramenta define quatro módulos principais: (i) *AspectJ Model Extractor* que faz o *parsing* no código em AspectJ; (ii) *Concern Manager* que implementa o mapeamento de elementos sintáticos para elementos abstratos; (iii) *Metric Collector*, responsável por computar as métricas e (iv) *Rule Analyzer*, que aplica regras heurísticas para gerar advertências sobre possíveis problemas de *design*. Esse ferramenta coleta as métricas do *framework* de avaliação de reuso de manutenção descrito em (SANT'ANNA et al., 2003). As métricas são:

- CDC: dispersão de conceito através de componentes;
- CDO: dispersão de conceito através de operações;
- CDLOC: dispersão de conceito através das linhas de código;
- CBC: acoplamento entre componentes;
- DIT: profundidade da árvore de herança;
- LCOO: falta de coesão nas operações;
- VS: tamanho do vocabulário;
- LOC: número de linhas de código;
- NOA: número de atributos;
- WOC: operações ponderadas por componentes.

O *framework* e as métricas foram detalhados na subseção 3.5 deste trabalho.

5.1 Ferramenta COMETA-Lua

Embora Lua seja uma linguagem bastante utilizada nos meios comerciais e acadêmico, ainda não há uma ferramenta para coleta de métricas em código-fonte Lua. Por isso, propõe-se, neste trabalho, COMETA-Lua (Ferramenta para Coleta de Métricas de Tamanho e Acoplamento em Lua), uma ferramenta para a coleta das métricas das propriedades tamanho e acoplamento em Lua. As métricas de tamanho são: (i) VS (vocabulário do sistema), (ii) LOC (número de linhas de código), (iii) NOA (número de atributos) e (iv) WOC (operações ponderadas por componentes). As métricas de acoplamento são (i) CBO (acoplamento entre classes de objetos), (ii) MPC (acoplamento por passagem de mensagem) e (iii) DAC (acoplamento por abstração de dados). Todas essas métricas são detalhadas no capítulo 4 e, resumidamente, nas tabelas 4.4 e 4.5.

O estudo de caso deste trabalho visa avaliar a refatoração de um *middleware* orientado a objetos que produz uma versão orientada a aspectos. Conforme exposto nas tabelas 4.4 e 4.5, as propriedades a serem avaliadas para a refatoração são modularidade e tamanho. A aspectização de um sistema é realizada para aumentar a organização do sistema, ou seja, sua modularidade. Outro fator a ser avaliado é se a aspectização não afetou consideravelmente o tamanho do sistema. Como modularidade é avaliada em termos de separação

de conceitos, acoplamento entidade-entidade e coesão (definição presente na seção 4.2.1), as métricas referentes a esses três fatores e a tamanho devem ser coletadas.

A primeira versão da ferramenta proposta foi implementada com o objetivo principal de auxiliar a coleta das métricas nos sistemas de *middleware* orientado a objetos e aspectos do estudo de caso. Assim, até o presente momento, somente a coleta das métricas de tamanho e acoplamento foram automatizada. As métricas de separação de conceitos e coesão dependem de uma arquitetura de sistema detalhada e de um mapeamento de conceitos para componentes para que possam ter a coleta automatizada. Como o sistema do estudo de caso não disponibilizava essas informações, a coleta das métricas referentes a essas propriedades não foi implementada e, portanto, teve que ser realizada manualmente. Felizmente, as métricas de coleta mais trabalhosa e demorada, as de tamanho e acoplamento, foram automatizadas. Faz parte do planejamento para trabalhos futuros a finalização da ferramenta para coleta de todas as métricas do conjunto proposto (tabelas 4.4 e 4.5) por este trabalho.

A ferramenta COMETA-Lua, embora se assemelhe ao processo de análise de programa-fonte descrito anteriormente, não possui todas as fases de um *parser* nem a separação de fases bem definida. Somente foram implementadas as fases de análise léxica e análise semântica de forma conjunta, por motivo de simplicidade. Como o que se deseja é encontrar *tokens* significativos para a coleta das métricas e computar seus valores a partir deles, essas duas fases atendem satisfatoriamente às necessidades da ferramenta. A fase de análise sintática está associada ao processo de verificar se os *tokens* de entrada formam uma expressão permitida pela linguagem e esse tipo de verificação não é necessário para a coleta das métricas. Obviamente, a avaliação da estrutura hierárquica gerada pela análise sintática tornaria a ferramenta mais robusta, mas essa não é uma etapa essencial nesse caso.

A ferramenta compõe-se de um conjunto de *scripts* escritos em Perl (PERL, 2008). A linguagem de programação Perl foi escolhida por oferecer uma grande variedade de recursos para a definição de expressões regulares e manipulação de *tokens*. Para cada métrica a ser coletada, há um *script* que a computa. Para algumas métricas, a coleta é realizado de uma forma em código OO e de outra forma diferente em código OA. Portanto, tem-se um *script* para código-base e outro para código de aspectos. O funcionamento básico é o mesmo para todos os *scritps*. Primeiro procura-se por *tokens* relevantes para a coleta de métricas. Uma vez encontrado, computa-se o valor parcial para a métrica ou armazena-se alguma informação relevante para posterior avaliação e cálculo da métrica.

Para ilustrar o processo, considere o seguinte exemplo para cálculo da métrica de tamanho LOC (*Lines of Code*). Para cada linha do programa Lua de entrada, verifica-se se a linha não está comentada e se é não-vazia. Caso isso seja verdadeiro, o número de LOC é incrementado. Quando todas as linhas do arquivo de entrada tiverem sido avaliadas, retorna-se o valor de LOC.

De acordo com o que foi descrito, pode-se perceber que a análise léxica e semântica são implementadas conjuntamente sem a necessidade de um arquivo intermediário entre elas. Quando um *token* relevante é encontrado, toma-se logo a ação correspondente para a coleta da métrica. É importante lembrar que, na análise léxica, não são separados todos os símbolos significantes para a linguagem de programação Lua, apenas aqueles que são relevantes para o cálculo da métrica em questão.

O código-fonte em Lua de entrada para a ferramenta precisa seguir algumas restrições. Em Lua, não há obrigatoriedade da utilização do marcador de fim de comando. Por essa razão, apenas um comando poderá ser declarado por linha. Assim, o separador de comandos passa a ser o "\n", o símbolo de quebra de linha. Por ser diferente a coleta em código OO e OA para algumas métrcas, a segunda restrição da ferramenta é que código-base seja definido separadamente do código de aspectos, ou seja, em arquivos diferentes. A terceira e última restrição diz respeito à sintaxe usada no código: o código-base deve estar definido conforme a biblioteca LOOP (*Lua Object-Oriented Programming*) (LOOP..., 2008) e o código de aspectos, conforme a biblioteca RE-AspectLua (BATISTA; VIEIRA, 2007).

A dificuldade principal para a construção dessa ferramenta foi a flexibilidade da linguagem Lua. Por exemplo, na linguagem de programação Java, existe somente uma única forma de se definir uma classe: através da palavra-chave "class", precedida por modicadores como "Public" e seguida pelo nome da classe. Em Lua, segundo documentação disponível em (LOOP..., 2008) para a bibliteca de programação orientada a objetos de Lua, uma classe pode ser declarada de três formas diferentes. Para efeito de ilustração, define-se uma classe de nome "classe", com atributos "a" e "b" e um método chamado "metodo" nas três formas possíveis:

1. ```
   classe = oo.class (a=0, b=1)
   function classe:metodo()
   (...)
   end
   ```
   ```

2. ```
   module ("modulo.classe", oo.class)
   a = 0
   b = 0
   function metodo()
   (...)
   end
   ```

3. ```
   module "modulo.classe"
   oo.class(_M, require ("modulo.super_classe"))
   a = 0
   b = 1
   function metodo()
   (...)
   end
   ```

No caso 1, o tipo mais comum de definição de classe, a classe é definida usando-se uma atribuição e a palavra-chave "oo.class", recebendo uma lista com os atributos "a" e "b". O método "metodo" é definido logo em seguida com a palavra-chave "function" seguido do nome da classe, ":" e o nome do método. No caso 2, a classe é definida como um pacote através do comando "module". Esse comando recebe o nome da classe e a palavra-chave "oo.class". Os métodos e os atributos dessa classe são definidos sem nenhuma referência ao nome da classe. Tem-se uma definição bem diferente da apresentada em 1. Por fim, o caso 3, a classe também é definida como um pacote, mas a forma como isso é feito é bem diferente. Define-se o nome da classe através do comando "module". O comando "oo.class" é chamado e recebe como parâmetros _M, a variável de módulo de Lua, e o nome da super classe da qual a classe sendo definida herda. Em outras palavras, o nome da classe é "modulo.classe" e ela herda de "modulo.super_classe". Também nesse caso, os atributos da classe e seus métodos são definidos sem nenhuma referência ao nome da classe à qual pertencem.

A seguir, os *scripts* para a coleta das métricas são detalhados. (Até agora só há implementação para coleta em programas orientados a objetos. Falta adaptação para processar em aspectos.)

5.1.1 Métrica de Tamanho LOC - Número de Linhas de Código

A métrica LOC conta a quantidade de linhas de um programa. São desconsideradas, nessa contagem, as linhas em branco e as completamente comentadas. O mesmo *script* coleta a métricas em código OO e OA. Para detalhamento do funcionamento do *script* para coleta da métrica LOC, considere o algoritmo 1.

Algoritmo 1 Algoritmo para coleta da métrica LOC

```
MetricaLOC (arquivo Programa_Lua)
1: LOC ← 0
2: linha ← le_linha(Programa_Lua)
3: while linha não for nula do
4:     if (linha não for vazia) and (linha não está completamente comentada) then
5:         LOC++
6:     end if
7:     linha ← le_linha(Programa_Lua)
8: end while
9: print LOC
```

O *script MetricaLOC* recebe o programa Lua a ser avaliado como entrada. Na linha (2), a função le_linha() retorna uma nova linha do arquivo de entrada a ser analisada. Enquanto a linha não for nula, ou seja, enquanto não se encontrar o fim de arquivo, verifica-se se ele é não-vazia e se não está completamente comentada (linha 4). A linha é vazia se ela contém apenas caracteres de espaçamento. A verificação das linhas é realizada através de expressões regulares. Se essas duas condições forem verdadeiras, o valor de LOC é incrementado, sendo parcialmente calculado. Ao se encontrar o fim do arquivo de entrada, pode-se retornar o valor total de LOC (linha 9).

5.1.2 Métrica de Tamanho VS - Vocabulário do Sistema

A métrica VS fornece a quantidade de entidades do sistema, ou seja, quantas entidades há no sistema. Há dois scripts para a coleta dessa métrica: um conta a quantidade de classes no código OO e o outro conta a quantidade de aspectos no código OA. O funcionamento do *script* para coleta da métrica VS em código OO é detalhado no algoritmo 2.

O *script MetricaVS* recebe o programa em Lua a ser avaliado como entrada. Para cada linha não nula do arquivo de entrada, verifica-se se há uma definição de classe. Caso exista, o valor de VS é incrementado, calculando seu valor parcial. A verificação é feita através de uma expressão regular que procura a palavra chave “oo.class” em três formas

Algoritmo 2 Algoritmo para coleta da métrica VS em código OO

```
MetricaVS (arquivo Programa_Lua)
1: VS ← 0
2: linha ← le_linha(Programa_Lua)
3: while linha não for nula do
4:     if linha contém definição de classe then
5:         VS++
6:     end if
7:     linha ← le_linha(Programa_Lua)
8: end while
9: print VS
```

diferentes de utilização:

(i) classe = oo.class();

(ii) module ("modulo.classe", oo.class);

(iii) oo.class(_M, "modulo.super_classe").

Segundo (LOOP..., 2008), uma classe pode ser definida utilizando qualquer uma dessas formas. Ao se encontrar o fim do arquivo de entrada, o cálculo de VS está finalizado. O *script* fornece a quantidade de classes por arquivo. Para se ter o valor para todo o sistema, é necessário executá-lo para todos os arquivos do sistema e somar suas saídas.

O funcionamento do *script* para coleta da métrica VS em código OA pode ser visto no algoritmo 3. O *script MetricaVS* recebe o programa em Lua a ser avaliado como entrada. Para cada linha não nula do arquivo de entrada, verifica-se se há uma definição de aspecto. Caso exista, o valor de VS é incrementado, calculando seu valor parcial. Em RE-AspectLua, um aspecto é definido da seguinte forma:

```
nome_aspecto = Aspect:new()
```

Assim, a verificação é feita através de uma expressão regular que procura a palavra chave "Aspect:new()". Ao se encontrar o fim do arquivo de entrada, o cálculo de VS está finalizado. O *script* fornece a quantidade de aspectos por arquivo. A partir dessa breve descrição, pode-se observar que o algoritmo 2 e 3 são bem semelhantes.

5.1.3 Métrica de Tamanho NOA - Número de Atributos

A métrica NOA soma o número de atributos para uma classe para código OO. Se uma classe foi definida de forma convencional:

```
classe = oo.class(a=1, b=2),
```

os atributos são aqueles presentes na lista recebida por oo.class (nesse caso, "a" e "b") e

Algoritmo 3 Algoritmo para coleta da métrica VS em código OA

```
MetricaVS (arquivo Programa_Lua)
1: VS ← 0
2: linha ← le_linha(Programa_Lua)
3: while linha não for nula do
4:     if linha contém definição de aspecto then
5:         VS++
6:     end if
7:     linha ← le_linha(Programa_Lua)
8: end while
9: print VS
```

também os definidos na forma <nome da classe>.<nome do atributo>, por exemplo:

```
classe.a = 1
```

seria uma outra forma de declarar o atributo "a" para classe.

Caso a classe seja declarada através da função "module", seus atributos são aqueles declarados dentro do arquivo que estejam fora de operaçãoes e blocos de comandos. Os atributos declarados dentro de operações e de blocos de comandos são locais a eles e, portanto, não pertencem à classe.

Assim, o *script* para cálculo de NOA deve considerar os dois tipos de definição de classe separadamente. Outro ponto importante é que mais de uma classe pode ser declarada dentro de um arquivo. O algoritmo 4 detalha o funcionamento do *script*.

O *scritp* MetricaNOA começa definindo o vetor associativo NOA_classes (linha 1), uma estrutura de dados comum em Perl. Nele, os índices são *strings*. Enquanto não se encontra o fim do arquivo, procura-se por quatro elementos:

(i) definição convencional de classe: se encontrada, é criada uma posição no vetor associativo com índice do nome da classe encontrada. Nessa posição, é armazenado o número de elementos da lista de atributos da classe (linhas 4-6);

(ii) definição de classe através do comando module: se encontrada, é criada uma posição no vetor com índice do nome do módulo(7-9);

(iii) atributo declarado na forma <nome_classe>.<nome_atributo>: encontrado, a posição do vetor associativo com índice do nome da classe é incrementado (linhas 10-12);

(iv) atributo declarado de forma convencional fora de bloco de comando: se existe uma posição no vetor associativo com índice do nome do módulo é porque a classe foi definida através do comando module. Assim, o atributo encontrado é contado como mais um atributo da classe (linhas 13-15).

Ao fim do algoritmo, cada posição do vetor associativo tem o valor de NOA correspondente

Algoritmo 4 Algoritmo para coleta da métrica NOA para código OO

```
MetricaNOA (arquivo Programa_Lua)
 1: NOA_classes ← {}
 2: linha ← le_linha(Programa_Lua)
 3: while linha não for nula do
 4:     if linha contém definição de classe convencional then
 5:         NOA_classes{nome_classe} ← #(lista de atributos)
 6:     end if
 7:     if linha contém definição de classe através do comando module then
 8:         NOA_classes{nome_modulo} ← 0
 9:     end if
10:     if linha contém definição de atributo na forma <nome_classe>.<nome_atributo>
    then
11:         NOA_classes{nome_classe}++
12:     end if
13:     if (NOA_classes{nome_modulo} existe) and (linha contém definição de atributo
    fora de bloco de comando) then
14:         NOA_classes{nome_modulo}++
15:     end if
16:     linha ← le_linha(Programa_Lua)
17: end while
18: print NOA
```

à classe do índice.

A sintaxe para a definição de aspectos em RE-AspectLua não permite que atributos sejam associados a aspectos (BATISTA; VIEIRA, 2007), fazendo com que o NOA seja sempre zero. Por isso, não foi definido nenhum *script* para cálculo de NOA em código OA.

5.1.4 Métrica de Tamanho WOC - Operações Ponderadas por Componente

A métrica WOC calcula a soma das complexidades das operações de uma entidade. Na literatura, não se especifica como a complexidade de uma operação é medida, apenas define-se que está relacionado à quantidade de parâmetros. Para este trabalho, a complexidade de uma operação é definida da seguinte forma:

$$WOC = \sum_{i=1}^{n} 2^{p(m_i)},$$

onde n é número de operações de um componente e $p(m_i)$ é o número de parâmetros da operação i. Assim, mesmo que uma operação não possua parâmetros, ela contribui com

uma unidade para a soma do WOC da classe.

Em Lua, os construtores das classes são declarados como uma função de nome "__init". Assumimos que essa padronização estenda-se a construtores de aspectos. Os parâmetros dessa função não são considerados para o cálculo de WOC, para que o seu valor não seja superestimado. Não há uma padronização para destrutores, por isso não são considerados caso especial. Operações são declaradas como funções em Lua. Funções declaradas dentro de bloco de comando não serão consideradas por se entender que são locais ao bloco e não pertencem à classe. Em Lua, quando uma função recebe um número variável de parâmetros, usa-se o operador "...". Esse operador é desconsiderado para a soma do número de parâmetros de uma operação.

No código-base, as operações podem ser declaradas de duas formas diferentes. Se a declaração da classe é na forma convencional, a operação é declarada como:

```
function <nome_classe>:<nome_função> (<lista_parametros>)
```

Caso a classe seja declarada através do comando "module", todas as operações contidas dentro do arquivo na forma:

```
function <nome_função>(<lista_parametros>)
```

são consideradas membros da classe.

O algoritmo 5 representa o *script* para a coleta da métrica WOC no código OO. Inicialmente, define-se o vetor associativo WOC_classes para armazenar os valores parciais de WOC para cada classe do arquivo (linha 1). Enquanto não for encontrado o fim do arquivo, procura-se por quatro elementos:

(i) definição convencional de classe: se encontrada, instancia-se uma posição com índice com o nome da classe (linhas 5-7);

(ii) definição de classe através do comando "module": e encontrada, instancia-se uma posição com índice com o nome do módulo (linhas 8-10);

(iii) definição de função na forma function <nome_classe>:<nome_função> (<lista_parametros>): caso seja encontrada, calcula-se o valor parcial de WOC somando o valor atual da variável com a potência do número de parâmetros da operação na base 2. Como explicado no início dessa mesma subseção, o cálculo de WOC foi assim definido para que, mesmo que uma operação não possua parâmetros, ela contribuirá com uma unidade para o resultado da métrica;

(iv) definição de função na forma function <nome_função>(<lista_parametros>): caso seja encontrada, calcula-se o valor parcial de WOC somando o valor atual da va-

riável com a potência do número de parâmetros da operação na base 2, como no item (iii).

Ao fim do algoritmo, o vetor associativo contém os valores finais de WOC para cada classe.

Algoritmo 5 Algoritmo para coleta da métrica WOC em código OO

```
MetricaWOC (arquivo Programa_Lua)
 1: WOC_classes ← {}
 2: linha ← le_linha(Programa_Lua)
 3: while linha não for nula do
 4:     if linha contém definição de classe convencional then
 5:         WOC_classes{nome_classe} ← 0
 6:     end if
 7:     if linha contém definição de classe através do comando module then
 8:         WOC_classes{nome_modulo} ← 0
 9:     end if
10:     if linha contém definição de função na forma <nome_classe>.<nome_funcao>
    then
11:         WOC_classes{nome_classe} ← WOC_classes{nome_classe} + 2 ↑
    #(lista_parametros)
12:     end if
13:     if (WOC_classes{nome_modulo} existe) and (linha contém definição de função)
    then
14:         WOC_classes{nome_modulo} ← WOC_classes{nome_modulo} + 2 ↑
    #(lista_parametros)
15:     end if
16:     linha ← le_linha(Programa_Lua)
17: end while
18: print WOC
```

Para ilustrar a definição de um aspecto em RE-AspectLua, considere o exemplo de código a seguir.

```
1.  nome_aspecto = Aspect:new()

2.  nome_interface = AspectInterface:new({ name = 'nome_interface'},
{ { refine = 'PontoCorteAbstrato' , action = nome_advices } } )

3.  nome_aspecto:interface(nome_interface)
```

Na linha 1, define-se o aspecto *nome_aspecto*. Na linha 2, define-se a interface *nome_interface* que associa o *advices nome_advices* ao ponto de corte abstrato *PontoCorteAbstrato*. Na linha 3, a interface *nome_interface* é associado ao aspecto *nome_aspecto*. Somente nesse ponto, o *advices nome_advices* é associado ao aspecto *nome_aspecto*. Em REAspect-Lua, os *advices* são definidos como funções comuns de Lua. O que indica que uma função é um *advices* é sua associação a uma interface.

O algoritmo 6 mostra o funcionamento do *script* para cálculo de WOC em código OA. No início do algoritmo, instancia-se a variável WOC_aspectos que armazenará os valores de WOC para cada aspecto (linha 1). Enquanto não for encontrado o fim do arquivo (linha 3), procura-se por 3 elementos:

(i) definição de função: geralmente se define a função do *advices* antes de associá-la a uma interface. Por isso, para cada definição de função encontrada, calcula-se e armazena-se o seu valor de WOC usando como índice o nome da função no vetor *WOC_funcao* (linhas 4-6). Se essa função fizer parte de alguma definição de interface, o seu valor de WOC será contado para o WOC do aspecto ao qual a interface estiver associada;

(ii) definição de interface: quando o algoritmo encontra uma definição de interface, ele procura por os *advices* associados a essa interface. Para cada definição de *advices*, busca-se o valor de WOC da função associada e soma-o ao WOC da interface, armazenando no vetor *WOC_advices* (linhas 7-11);

(iii) associação aspecto-interface: quando o algoritmo encontra esse tipo de associação, o vetor *WOC_aspecto*, na posição que tem como índice o nome do aspecto em questão, recebe o WOC associado à interface (linhas 13-15).

Ao fim do algoritmo, o algoritmo imprime os valores de WOC de todos os aspectos do arquivo de entrada (linhas 16).

Algoritmo 6 Algoritmo para coleta da métrica WOC em código OA

```
MetricaWOC (arquivo Programa_Lua)
 1: WOC_aspectos ← {}
 2: linha ← le_linha(Programa_Lua)
 3: while linha não for nula do
 4:     if linha contém definição de funcao then
 5:         WOC_funcao{nome_funcao} ← 2 ↑ #(lista_parametros)
 6:     end if
 7:     if linha contém definição de interface then
 8:         WOC_advices{nome_interface} ← 0
 9:         while encontrar definição de advices dessa interface do
10:             WOC_advices{nome_interface} ← WOC_advices{nome_interface} +
    WOC_funcao{nome_advices}
11:         end while
12:     end if
13:     if linha contém associação entre aspecto e interface then
14:         WOC_aspecto{nome_aspecto} = WOC_interface{nome_interface}
15:     end if
16: end while
17: print WOC_aspecto
```

5.1.5 Métrica de Acoplamento CBO - Acoplamento entre Classes de Objetos

A métrica CBO conta a quantidade de outras entidades a que uma entidade está acoplada através de invocações a operações ou referência a atributos. Para o caso de uso deste trabalho, o OiL, classes são agrupadas em módulos. As classes que pertencem a um mesmo módulo têm um objeto e funcionalidade em comum, sendo compatível com a definição de categorias de classes. O acoplamento entre classes de uma mesma categoria não é um fator negativo, mas algo esperado. Logo, o acoplamento que interessa ser avaliado é o acoplamento entre as classes de uma categoria e classes de outras categorias. O mesmo se aplica a aspectos.

Por enquanto, o *script* para coleta de métricas de acoplamento apresentam somente a granularidade de categoria de classes, por ser essa a granularidade escolhida para a coleta do caso de uso. A granularidade a nível de classe é igualmente importante e será implementada em trabalhos futuros. A granularidade para métricas de acoplamento é descrita com mais detalhes na subseção 4.2.1.

Para o OiL e o AO-OiL, CBO será a quantidade de módulos a que um módulo está acoplado. Apenas será considerado acoplamento a outros módulos da aplicação. Acoplamento a módulos de bibliotecas de Lua serão desconsiderados.

O funcionamento do *script* para coleta da métrica CBO é detalhado no algoritmo 7. Esse algoritmo pode ser aplicado tanto a código-base quanto a código de aspectos. Em Lua, para manter uma referência a um módulo externo usa-se o comando:

```
<referencia> = require "<nome_modulo>"
```

O primeiro passo do *script* é então armazenar as referências a outros módulos da aplicação a ser avaliada. Uma vez obtidas as referências, procura-se por chamadas a operações ou referência a atributos através delas. No ínicio do algoritmo, declara-se o vetor associativo ref_CBO para armazenar informações relevantes para o cálculo de CBO (linha 1). Enquanto fim de arquivo não for encontrado, procura-se por 3 elementos:

(i) definição de referência a módulo externo: essa definição é feita através do comando <referencia> = require "<nome_modulo>". Se encontrado, instancia-se uma posição de Ref_CBO com o nome da referência como índice e recebe valor booleano "false" para indicar que ainda não foi encontrada nenhuma referência a atributo ou invocação a operação através dela (linhas 4-6);

(ii) acesso a atributo através de referência já armazenada: procura-se por comando

na forma <referencia>.<nome_atributo>, com <referencia> sendo uma referência já declarada através do comando "require". Se encontrada, a posição de ref_CBO que tenha como índice a referência recebe valor booleano "true", sinalizando que foi encontrado acoplamento através dessa referência (linhas 7-9);

(iii) invocação a operação através de referência já armazenada: procura-se por comando na forma <referencia>:<nome_operacao>, com <referencia> sendo uma referência já declarada através do comando "require". Se encontrada, a posição de ref_CBO que tenha como índice a referência recebe valor booleano "true", sinalizando que foi encontrado acoplamento através dessa referência (linhas 10-12).

Para se calcular o valor de CBO, ref_CBO é percorrido em busca das posições que tenham valor "true". Para cada valor desse encontrado, há um módulo externo para o qual há acoplamento através de acesso a atributos e invocações a operações (linhas 16-21). Logo, CBO é incrementado. Ao fim do algoritmo, o cálculo de CBO está finalizado (linha 22).

Algoritmo 7 Algoritmo para coleta da métrica CBO

```
MetricaCBO (arquivo Programa_Lua)
 1: ref_CBO ← {}
 2: linha ← le_linha(Programa_Lua)
 3: while linha não for nula do
 4:     if linha contém comando na forma <referencia> = require "<nome_modulo>" then
 5:         ref_CBO{<referencia>} ← false
 6:     end if
 7:     if linha contém referência a atributo na forma <referencia>.<atributo> then
 8:         ref_CBO{<referencia>} ← true
 9:     end if
10:     if linha contém invocação a operacao na forma <referencia>:<nome_operacao> then
11:         ref_CBO{<referencia>} ← true
12:     end if
13:     linha ← le_linha(Programa_Lua)
14: end while
15:
16: CBO ← 0
17: for cada posição de ref_CBO do
18:     if ref_CBO{posição} = true then
19:         CBO++
20:     end if
21: end for
22: print CBO
```

Em alguns casos, a referência para módulo externo é utilizada para a declaração de um atributo com tipo do módulo da referência. A partir desse atributo, são então acessados

atributos e operações não implementados no módulo em avaliação. Esses casos também são considerados para o cálculo de CBO, embora não tenha sido detalhado no algoritmo por motivo de simplificação.

É da definição de CBO considerar acoplamento por herança. Em Lua, os atributos implementados externamente ao módulo sendo avaliado, sejam eles herdados ou não, precisam ser acessados através da mesma sintaxe. O mesmo ocorre para as invocações a operações. Assim, o acoplamento por herança será contabilizado como o caso geral.

5.1.6 Métrica de Acoplamento DAC - Acoplamento por Abstração de Dados

A métrica DAC conta o número de atributos não herdados que têm como um componente como seu tipo. Assim como para a métrica CBO, a granularidade para a coleta dessa métrica é categoria de classes e só será considerado acoplamento a outros módulos da aplicação.

O funcionamento do *script* para coleta da métrica DAC em códigos OO e OA é detalhado no algoritmo 8. Em Lua, para manter uma referência a um módulo externo usa-se o comando:

```
<referencia> = require "<nome_modulo>".
```

O primeiro passo do *script* é então armazenar as referências a outros módulos da aplicação a ser avaliada. Uma vez obtidas as referências, procura-se por declaração de atributos que recebem o retorno da chamada de construtores através das referências armazenadas.

Na linha (1), é definido o vetor associativo ref_DAC. Para cada posição desse vetor, o índice será a referência a um módulo externo e o seu valor será a quantidade de atributos com tipo desse módulo. Para se ter o valor de DAC para o módulo avaliado, ao final do algoritmo, soma-se os valores de todas as posições de ref_DAC (linhas 13-17).

Enquanto não for encontrado o fim do arquivo, busca-se por referências a módulos externos. Cada referência encontrada será índice de uma posição de ref_DAC. Essa posição será depois preenchida com o número de atributos encontrados daquele módulo (linhas 4-6).

No segundo passo do algoritmo, procura-se por atributo declarado a partir de chamada a construtor de referência a módulo externo. Uma vez encontrado, a posição de ref_DAC da referência correspondente é incrementada (linhas 7-9).

Algoritmo 8 Algoritmo para coleta da métrica DAC

MetricaDAC (arquivo Programa_Lua)

```
1: ref_DAC ← {}
2: linha ← le_linha(Programa_Lua)
3: while linha não for nula do
4:     if linha contém comando na forma <referencia> = require "<nome_modulo>" then
5:         ref_DAC{<referencia>} ← 0
6:     end if
7:     if linha contém declaração de atributo que recebe retorno de construtor invocado através de referência a outro módulo then
8:         ref_DAC{<referencia>}++
9:     end if
10:    linha ← le_linha(Programa_Lua)
11: end while
12:
13: DAC ← 0
14: for cada posição de ref_DAC do
15:    DAC = DAC + ref_DAC{<posição>}
16: end for
17: print DAC
```

5.1.7 Métrica de Acoplamento MPC - Acoplamento por Passagem de Mensagem

A métrica MPC para um componente, é o número de invocações estáticas a operações não implementadas nesse componente. Assim como para a métrica CBO e DAC, a granularidade para a coleta dessa métrica é categoria de classes e somente será considerado acoplamento a outros módulos da aplicação.

O funcionamento do *script* para coleta da métrica MPC em códigos OO e OA é detalhado no algoritmo 9. Em Lua, para manter uma referência a um módulo externo usa-se o comando:

`<referencia> = require "<nome_modulo>"`.

O primeiro passo do *script* é então armazenar as referências a outros módulos da aplicação a ser avaliada. Uma vez obtidas as referências, procura-se por chamadas a operações externas através dessas referências.

Na linha (1), o vetor associativo ref_MPC é declarado para armazenar a quantidade de chamadas a operações de módulos externos. Os índices desse vetor são as referências a módulos externos.

Algoritmo 9 Algoritmo para coleta da métrica MPC

MetricaMPC (arquivo Programa_Lua)

```
1: ref_MPC ← {}
2: linha ← le_linha(Programa_Lua)
3: while linha não for nula do
4:     if linha contém comando na forma <referencia> = require "<nome_modulo>" then
5:         ref_MPC{<referencia>} ← 0
6:     end if
7:     if linha contém chamada a operação na forma <referencia>:<nome_operacao> then
8:         ref_MPC{<referencia>}++
9:     end if
10:    linha ← le_linha(Programa_Lua)
11: end while
12:
13: MPC ← 0
14: for cada posição de ref_MPC do
15:    MPC = MPC + ref_MPC{<posição>}
16: end for
17: print MPC
```

Enquanto não for encontrado o fim arquivo, procura-se por referências a módulos externos. Uma vez encontrada, instancia-se uma posição em ref_MPC com a referência como indíce (linhas 4-6). Em seguida, busca-se por invocações a operações através dessas referências. Quando encontrada, a posição de ref_MPC no índice da referência correspondente é incrementada (linhas 7-9).

Ao fim do algoritmo, soma-se todos as posições de ref_MPC para se obter o MPC total para o módulo (linhas 13-17).

6 Validação do Conjunto de Métricas

A validação do conjunto de métricas, que será apresentada nesse capítulo, foi realizada através da aplicação das métricas estáticas aos sistemas de *middleware* OiL e AO-OiL e da execução de um sistema de monitoramento de poços de petróleo no OiL e no AO-OiL para a coleta das métricas dinâmicas. Com esses resultados, será possível comparar as duas versões de um mesmo *middleware* — OiL e sua refatoração, o AO-OiL — e ainda avaliar, em cada um, as propriedades associadas às métricas.

A seguir, serão apresentados os dois sistemas de *middleware* do estudo de caso, o OiL (seção 6.1) e o AO-OiL (seção 6.2), o sistema de monitoramento de poços de petróleo (seção 6.3) e os resultados obtidos (seção 6.4).

6.1 OiL

OiL (*ORB in Lua*) (MAIA et al., 2006), (MAIA; CERQUEIRA; KON, 2005) é uma implementação da especificação CORBA (CORBA, 2008) na linguagem de *script* Lua (LUA, 2008). Devido a natureza de linguagens de *script* como Lua, OiL apresenta características como simplicidade, flexibilidade e portabilidade: a simplicidade está presente em Lua com suas estruturas simples e de alto nível de abstração; a flexibilidade é conferida pelo caráter dinâmico de Lua e portabilidade é possível porque o interpretador Lua é inteiramente escrito em ANSI C, podendo ser virtualmente compilado em qualquer plataforma, desde máquinas servidoras até PDAs e telefones celulares.

Ao contrário de outras implementações CORBA, OiL não suporta *stubs* e *skeletons*. Todo o suporte à comunicação remota é criado em tempo de execução, incluindo invocação e despacho de métodos.

Assim como outros ORBs, OiL manipula mensagens codificadas em formato CDR (*Common Data Representation*) que são enviadas e recebidas através de canais de *sockets*

de acordo com o protocolo GIOP (*General Inter-ORB Protocol*) definido na especificação CORBA.

Para construir uma mensagem GIOP corretamente, é preciso conhecer precisamente as operações e atributos disponíveis na interface de cada objeto distribuído no *middleware*. Essas informações ficam disponíveis em uma interface IDL (*Interface Description Language*) que pode ser modificada ou substituída dinamicamente. Para facilitar o desenvolvimento de aplicações, OiL provê um compilador que traduz especificações IDL em estruturas de dados Lua e as registra em um Repositório de Interface Remota.

No OiL, todas as invocações de métodos são realizadas por objetos *proxies* que funcionam como *stubs* dinâmicos e fazem a invocação de acordo com um dada definição de interface. Cada objeto *proxy* é uma instância de uma classe associada a uma determinada interface. Sempre que uma definição de classe muda, sua classe *proxy* associada também muda. Assim, cada objeto *proxy* daquela classe é adaptado para refletir a mudança.

Um processo análogo ocorre para o despacho de métodos. O despacho é feito de acordo com uma interface definida no momento da criação do objeto. Sempre que uma nova requisição chega para um determinado objeto, a definição da interface associada é usada para recuperar a assinatura da operação. Assim, quando a definição de uma interface muda, cada requisição recebida por um objeto daquela interface deve estar adaptada a nova definição.

Lua provê suporte nativo a concorrência através das co-rotinas que são usadas para criar *threads* de execução independentes. A troca de execução de uma *thread* por outra ocorre em pontos definidos explicitamente. Esse tipo de multiprogramação é chamado de concorrência cooperativa. Nesse tipo de concorrência, a sincronização é mais simples — facilitando a programação e a depuração — e a troca de contexto é quase inteiramente tratada pelo programador. Além disso, a concorrência cooperativa pode ser mais eficiente que os modelos preemptivos.

Adaptação dinâmica é a capacidade de uma implementação de uma aplicação mudar sem parar de executar. Como Lua é uma linguagem dinâmica, torna-se fácil modificar implementações Lua, incluindo o OiL. A adaptação dinâmica é favorecida pela concorrência cooperativa devido à troca de contexto ser decidida pelo programador. Ele pode então definir pontos onde as adaptações ocorreram atomicamente, evitando potenciais inconsistências.

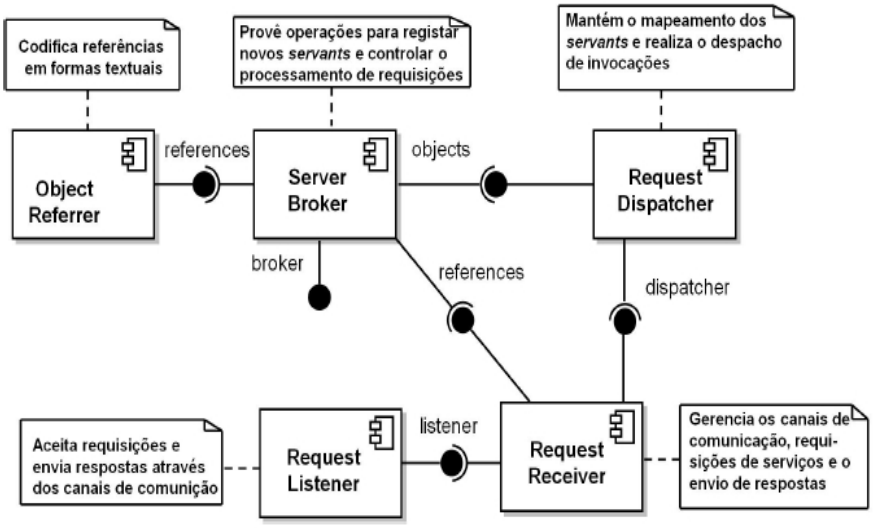

Figura 6.1: Arquitetura Servidora do OiL — figura retirada de (SILVA, 2008).

6.1.1 Arquitetura do OiL

A arquitetura de componentes do *middleware* OiL é composta por duas sub-arquiteturas: a arquitetura do servidor (figura 6.1) e a arquitetura cliente (figura 6.2). A arquitetura do servidor é a responsável por gerenciar serviços: seus ciclos de vida, conexões, interfaces expostas e objetos que os implementam. Por sua vez, a arquitetura cliente deve fornecer meios de recuperar referências para os serviços e permitir invocações de operações de forma transparente para a aplicação cliente.

A arquitetura do servidor é composta pelo componentes *Request Receiver*, *Request Dispatcher*, *Server Broker*, *Request Listener* e *Object Referrer*, mostrados na figura 6.1. Cada um desses componentes é detalhado a seguir.

O *Request Receiver* controla a aceitação das requisições, o acesso aos canais de comunicação e notifica o *Request Dispatcher* que existem requisições pendentes. Esse componente possui dois receptáculos: (i) *listener*, utilizado para a criação de canais de comunicação e obtenção de requisições, e (ii) *dispatcher*, que executa as requisições. Ele possui ainda a faceta *acceptor*, responsável por disponibilizar funções para a criação de canais de comunicação e para a gerência de execução do próprio *Request Receiver*.

O componente *Request Dispatcher* mantém um repositório de referências para os objetos servidores e realiza o despacho dos métodos. Ele possui duas facetas: (i) *object*, que registra cada objeto com um identificar único, e (ii) *dispatcher*, que recupera cada referência a um objeto a partir de seu identificador.

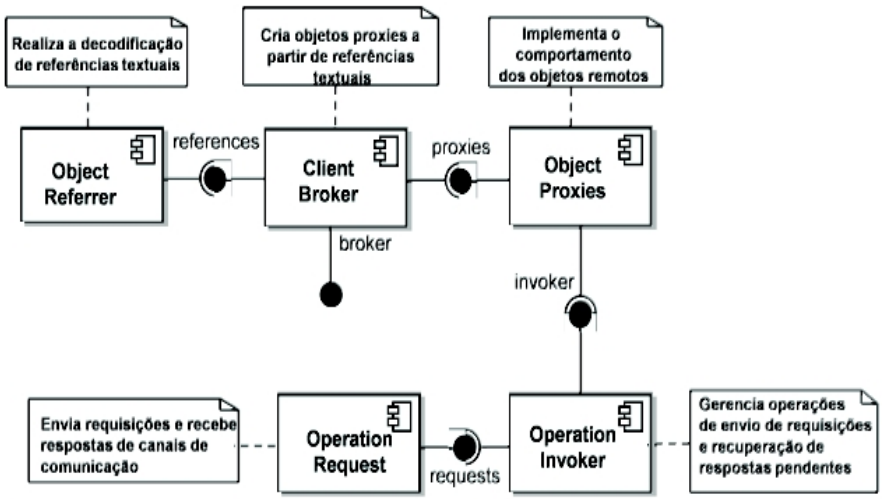

Figura 6.2: Arquitetura Cliente do OiL — figura retirada de (SILVA, 2008).

O *Server Broker* é o componente responsável por fazer o registro e controlar o processamento dos serviços. Ele possui a faceta *broker* através da qual fornece sua interface de controle e oferece operações como inicialização, execução, suspensão.

Por fim, o componente *Request Listener* tem por finalidade escrever e escutar nos canais de comunicação e o *Object Referrer* codifica as referências a objetos em referências textuais.

A arquitetura cliente possui os componentes *Client Broker*, *Object Proxies*, *Operation Invoker*, *Object Referrer* e *Operation Request*, mostrados na figura 6.2. O componente *Object Proxies* é uma fábrica de objetos *proxies*. O *Client Broker* cria estes de acordo com as referências textuais disponibilizadas pelo *Object Referrer*. O acesso de uma aplicação cliente a um serviço ocorre através de requisições de operações ao *proxy* associado a este serviço. Quando uma invocação é feita ao *proxy*, ele repassa a operação e a referência de si próprio para o *Operation Invoker*, que se encarrega de estabelecer um canal de comunicação com o objeto servidor do serviço requisitado. O *Operation Invoker* possui como atribuições gerenciar o acesso a rede e enviar requisições e receber respostas. A faceta *invoker* desse componente é responsável por realizar a chamada remota. O *Operation Request* é o componente que escreve e escuta nos canais de comunicação do lado cliente. Ele exporta a faceta *requests* que disponibiliza os métodos para gerência dos canais de comunicação.

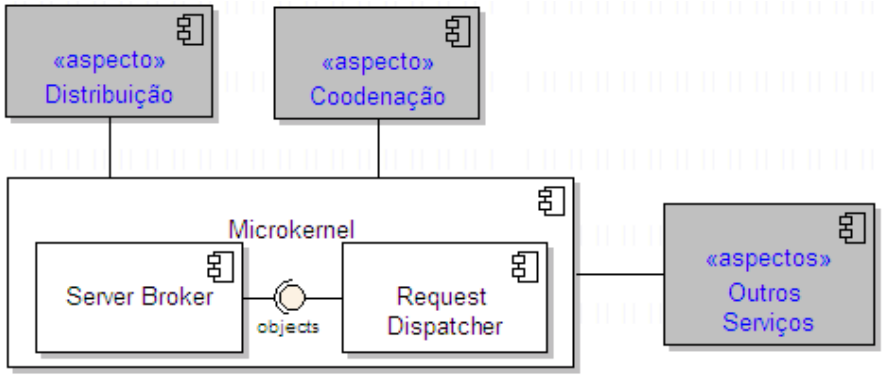

Figura 6.3: Arquitetura do AO-OiL — figura retirada de (SILVA, 2008).

6.2 AO-OiL

O *middleware* orientado a aspectos AO-OiL (SILVA, 2008) consiste em uma refatoração do *middleware* OiL, apresentado no subitem 6.1. Sua construção segue as recomendações da arquitetura de referência disponível em (LOUGHRAN et al., 2005) para sistemas de *middleware* orientado a aspectos. Aproveitando a capacidade dinâmica do OiL, o AO-OiL objetiva possibilitar a carga dinâmica de aspectos conforme as necessidades da aplicação.

6.2.1 Arquitetura do AO-OiL

A arquitetura do AO-OiL baseada na arquitetura de referência é composta de um pequeno núcleo funcional e um conjunto de extensões. As extensões são implementadas como aspectos e permitem adicionar funcionalidades ao núcleo do *middleware*.

Os serviços básicos oferecidos pelo OiL são instanciação de componentes, registro de componentes e serviços de comunicação. Como na arquitetura de referência, serviços de comunicação não fazem parte do núcleo do *middleware*, esse serviço passa a ser implementado como um aspecto no AO-OiL. Logo, o núcleo do AO-OiL trata apenas dos outros dois serviços e, por isso, passa a ser composto pelos componentes *Server Broker* e *Request Dispatcher* depois da refatoração.

A arquitetura para o AO-OiL pode ser vista na figura 6.3. O núcleo ou microkernel do AO-OiL contém os componentes *Server Broker* e *Request Dispatcher*. Serviços referentes a distribuição e coordenação são definidos dentro de aspectos que podem ser carregadas junto ao núcleo, assim como outros aspectos.

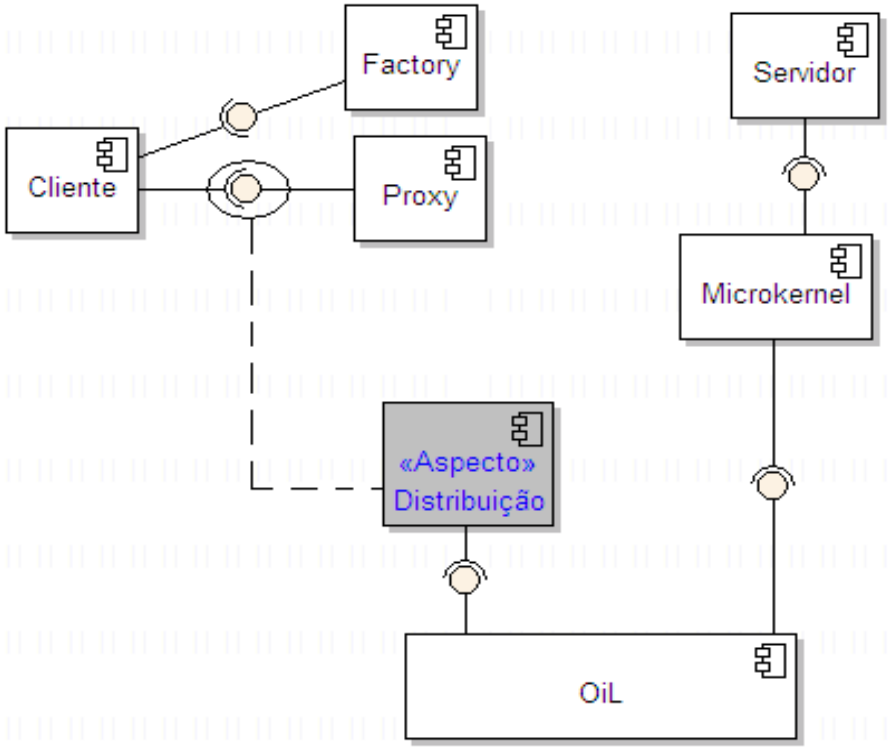

Figura 6.4: Arquitetura do AO-OiL com aspecto de distribuição — figura retirada de (SILVA, 2008).

A figura 6.4 detalha a inserção do aspecto de distribuição no AO-OiL. O serviço de distribuição é disponibilizado através da interceptação em pontos determinados da aplicação cliente pelo aspecto de distribuição. O cliente envia requisições de serviços aos *proxies* locais. O aspecto intercepta essas invocações e, através da execução de código de *advices*, transforma-as em invocações remotas. Do lado servidor, os aspectos não interceptam invocações, apenas desserializam as chamadas remotas e entrega-as ao microkernel.

A coordenação do AO-OiL implementa a abordagem *publish/subscribe*. Essa abordagem oferece um serviço de eventos em que os consumidores de eventos se registram para um determinado tópico e os produtores publicam eventos em tópicos relacionados. Quando um novo evento é publicado em um tópico, todos os consumidores registrados nele recebem uma notificação.

A figura 6.5 detalha a inserção do aspecto de coordenação no AO-OiL. O aspecto Publish/Subscribe instancia e configura o serviço de eventos de acordo com a aplicação. Quando algum componente da aplicação envia invocações de publish() ou subscribe(), o aspecto as intercepta e executa *advices* responsáveis pelo registro de ocorrência de eventos ou pela inscrição em tópicos. A gerência e armazenamento das inscrições e notificação de

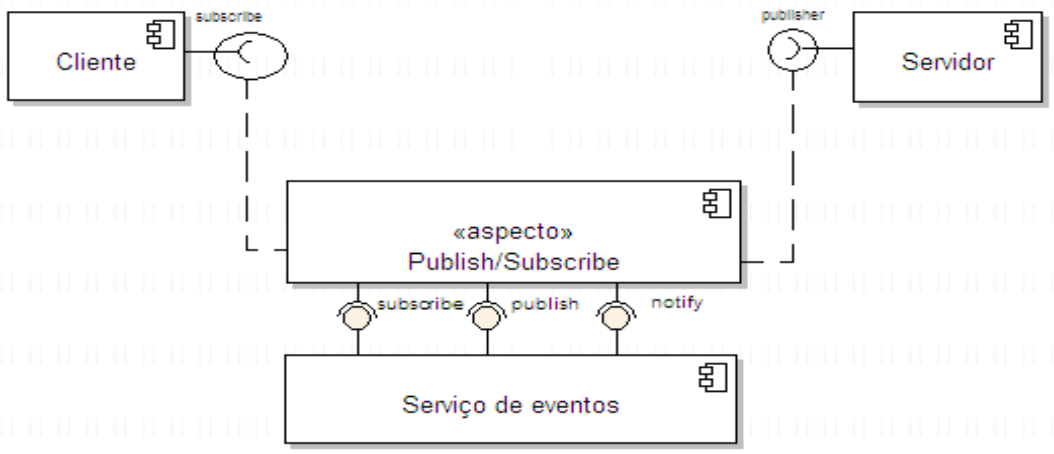

Figura 6.5: Arquitetura do AO-OiL com aspecto de coordenaçãoo — figura retirada de (SILVA, 2008).

eventos é papel do componente serviço de eventos.

6.3 Sistema de Monitoramento de Poços de Petróleo

Para coletar a métrica dinâmica tempo de execução, foi necessario desenvolver uma aplicação para ser executada nos sistemas de *middleware* OiL e AO-OiL. A aplicação escolhida foi um sistema de monitoramento de poços de petróleo, desenvolvida inicialmente para o padrão CORBA pelo aluno Diogo Salim deste mesmo departamento (SALIM, 2004). Este trabalho utilizou sua especificação, mas reimplementou o sistema na linguagem de programação Lua, a fim de poder ser executado nos sistemas de *middleware* do estudo de caso.

Essa seção apresenta uma breve introdução à elevação artificial de poços de petróleo (subseção 6.3.1) e a especificação do sistema proposto (subseção 6.3.2). A aplicação foi desenvolvida em parceria com Diego Saraiva, aluno de mestrado deste mesmo departamento.

6.3.1 Elevação Artificial de Poços Petrolíferos

Na elevação natural de petróleo, os fluidos (óleo, gás e água) jorram desde o interior do reservatório até as facilidades de produção utilizando sua própria energia natural, presente no reservatório. Caso essa energia natural seja insuficiente para fazer jorrar os fluidos até a superfície, faz-se necessário utilizar um método de elevação artificial. Existem diversos

métodos de elevação artificial, sendo o bombeio mecânico com hastes o mais aplicado na indústria petrolífera mundial. Este trabalha apresenta apenas o método bombeio mecânico com hastes por ser o método a interagir com o sistema de monitoramento de poços desenvolvido.

O bombeio mecânico com hastes consiste em transformar o movimento rotativo de um motor elétrico ou de combustão interna em movimento alternativo por uma unidade de bombeio localizada próxima à cabeça do poço. Uma coluna de hastes transmite o movimento alternativo para o fundo do poço, acionando uma bomba que eleva os fluidos produzidos pelo reservatório para a superfície (THOMAS, 2001).

O processo de bombeamento é realizado pela bomba de subsuperfície. Sua função é fornecer energa ao fluido, elevando-o até a superfície. Ela é composta das seguintes partes: colunas de hastes e de produção, bomba de fundo, pistão, válvulas de passeio e de pé, conforme mostrado na figura 6.6. O processo de bombeamento é dividido em dois ciclos: o ciclo ascendente e o ciclo descendente. No ciclo ascendente, o peso do fluido que está dentro da coluna da bomba abaixo do pistão e acima da válvula de pé faz com que esta se abra, permitindo a passagem do fluido para o interior da bomba. Todo o fluido que está acima do pistão é elevado com as hastes. O fluido que está mais próximo à cabeça do poço atinge a superfície. No ciclo descendente, os fluidos que estão acima da válvula de pé são comprimidos e esta é fechada. Como o pistão está em fase de descida, as pressões acima e abaixo da válvula de passeio se igualam e ela abre, permitindo a passagem do fluido para cima do pistão. Ao atingir o final do ciclo descendente e iniciar o ciclo ascendete, a válvula de passeio fecha e a de pé abre, iniciando um novo ciclo de elevação de fluidos (SALIM, 2004).

É possível que o interior da bomba não se encha completamente no movimento ascendente, deixando a bomba em condição de enchimento parcial ou *pump-off*. Atualmente, a detecção de *pump-off* é realizada pela instalação de dois sensores na unidade de bombeio. Um conjunto de 128 pares de dados desses sensores constituem a carta dinamométrica. A carta dinamométrica consiste na principal ferramenta para avaliação das condições em que está ocorrendo o bombeio (SALIM, 2004).

6.3.2 Especificação da Aplicação

O sistema de monitoramento de poços consiste em dados enviados por um cojunto de poços perfurados em reservatórios de petróleo que utilizam o bombeio mecânico com hastes como método de elevação artificial para a extração de petróleo e gás para uma central

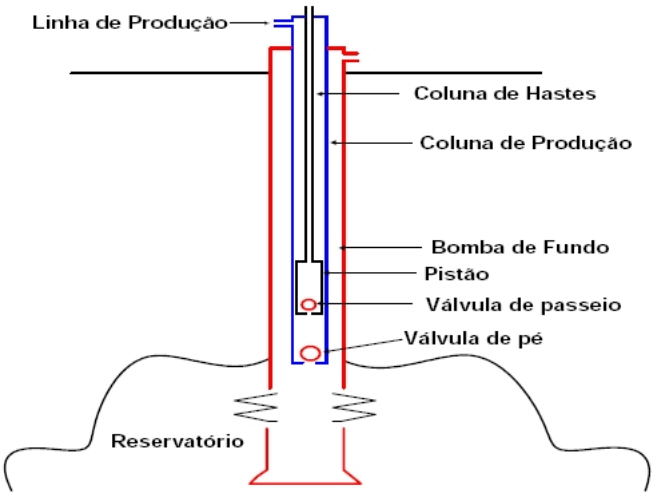

Figura 6.6: Ilustração da Bomba de subsuperfície — figura retirada de (SALIM, 2004).

de monitoramento. Esses dados são fornecidos por um par de sensores em cada poço. Na central de monitoramento, esses darão serão agrupadas para dar gerar as cartas dinamométricas. A carta permite que especialistas possam avaliar as características funcionais de cada poço para durante um ciclo completo de subida e descida da bomba. Quando a vazão do poço é menor que a vazão da bomba, efeito *pump-off*, a bomba deve ficar parada por um determinado período de tempo (*idle time*). Quando ela volta a operar, o período é chamado *runtime*.

Em cada poço, há um *Controlador Lógico Programável* (CLP) para armazenar os dados gerados pelos sensores e controlar o processo. Um *rede de comunição de campo* realiza a comunição com os CLPs e a central de monitoramento, geralmente instalada em um computador responsável por efetuar comandos de supervisão e armazenar informações enviadas pelos CLPs, através de uma arquitetura cliente-servidor.

Em (SALIM, 2004), é proposta uma nova abordagem: distribuir os dados da central de monitoramento através da rede mundial de computadores, para que todo usuário que tiver acesso a esta possa acessar os dados da central de monitoramento. Essa abordagem permite que especialistas possam acompanhar remotamente as informações da elevação dos poços, sem precisar de deslocar até a central de monitoramento. O monitoramento dos poços não é alterada: o processo de captação de informações dos sensores e sua comunicação com a central de monitoramento permanece igual. A contribuição dessa implementação está na distribuição e transparência para acesso às informações. A figura 6.7 ilustra a estrutura do sistema proposto.

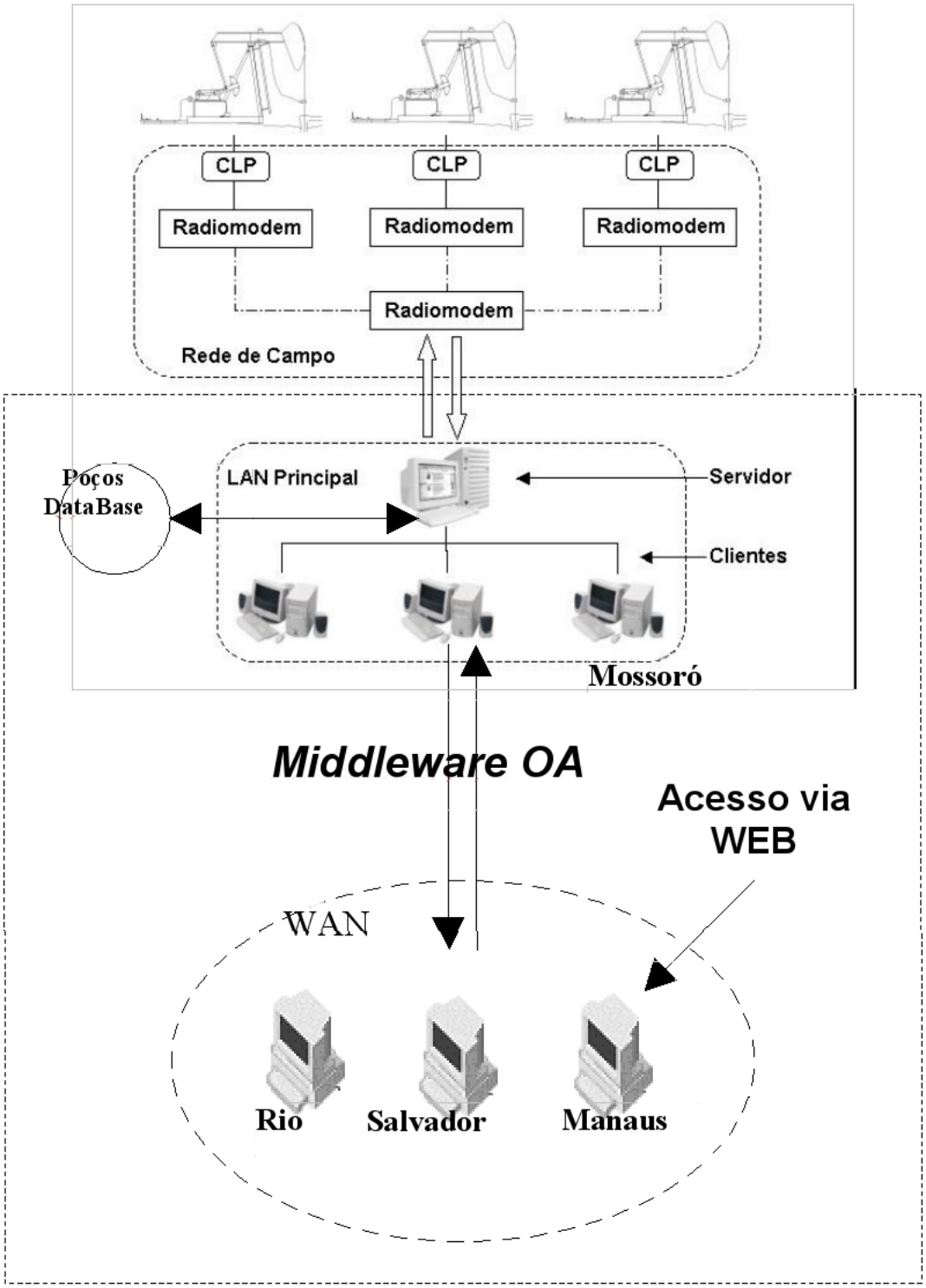

Figura 6.7: Ilustração da estrutura do sistema de monitoramento de poços — figura retirada de (SALIM, 2004).

A implementação foi definida em uma arquitetura cliente-servidor. O servidor executa na central de monitoramento para prover a distribuição das informações. Os usuários que desejam receber essas informações cadastram-se como clientes do servidor através da rede mundial de computadores. A comunicação entre cliente e servidor se dá por meio do serviço de eventos, para que se utilize esse recurso provido pelos sistemas de *middleware* OiL e o AO-OiL do estudo de caso.

A comunicação cliente-servidor ocorre da seguinte forma. Inicialmente, o cliente sinaliza ao servidor que deseja receber dados sobre os poços. O servidor gera um identificador único para o cliente e o envia. O cliente, ao receber a resposta, fica em modo de espera de eventos. O servidor avisa ao canal de eventos que há mais um consumidor. Quando há a ocorrência de um evento, o canal de eventos envia-o a todos os clientes cadastrados.

Interface IDL A listagem 6.1 apresenta a interface IDL que define os parâmetros do sistema proposto. Cada parâmetro é descrito a seguir.

- idle_time: tempo em que o bombeio mecânico de um poço fica parado;
- ultima_atualizacao: momento em que ocorreu a última atualização dos dados de um poço;
- status_bombeio: indica se o bombeio está ligado ou desligado;
- campo: número de identifição do campo monitorado;
- poco: número de identificação do poço monitorado;
- bsw: representa o percentual de água existente no petróleo sendo produzido. Varia de 99,99% a 0% (100% é água pura e 0% é petróleo puro). O ideal é que o BSW seja menor que 40%;
- nomes_campos: armazena os nomes dos campos de uma central de monitoramento;
- task: identificador de tarefas a serem realizadas;
- tag_: auxilia no processo de troca de mensagens;
- ID_Remetente: número único de identificação do cliente cadastrado;

Listagem 6.1: Definição de parâmetros para o sistema de monitoramento de poços.

```
struct Dados {
    long idle_time;
    long ultima_atualizacao;
    float carta_dinam[2][128];
    boolean status_bombeio;
};

struct Atributos {
    long task;
    long campo;
    long poco;
    Dados poco_dados;
    float bsw;
};
struct Evento {
    string tag_;
    long id_remetente;
    Atributos poco_atributo;
    string nome_campo;
};
```

A interface IDL do servidor pode ser vista na lista 6.2. O servidor, ao inicializar, carrega as interfaces IDL no repositório de interfaces, publica o objeto servidor no serviço de nomes e instancia o canal de eventos. Quando o cliente deseja receber dados, ele procura pelo objeto servidor no serviço de nomes e invoca o método *subscribe()* do servidor. O servidor, ao receber a requisição, gera um identificador único para esse cliente, envia-o ao cliente e avisa ao canal de eventos sobre o novo consumidor. Esse processo de *handshake* permite que somente clientes autorizados pelo servidor recebam as informações. O canal de eventos passa a enviar os eventos com os dados para esse novo consumidor sem que o servidor precise tomar conhecimento disso. Quando o cliente não deseja mais receber eventos, ele invoca o método *unsubscribe()* ao servidor e este notifica o canal de eventos.

Listagem 6.2: Interface IDL do servidor.

```
struct Proxy{
    unsigned long id;
    Object ref;
};
exception CannotSubscribe {
    string reason;
};
```

```
exception CannotUnsubscribe {
    string reason;
};
interface CentralMonitoramento {
    Proxy subscribe(in Object servant ) raises (CannotSubscribe);
    void unsubscribe(in unsigned long id) raises (CannotUnsubscribe);
};
```

De forma a utilizar todos os recursos dos sistemas de *middleware* do estudo de caso, o servidor é executado em duas *threads*: uma espera por requisições de cadastramento de novos clientes e a outra gera dados e produz eventos.

Simulação do Ambiente de Execução

Considerando que o foco do trabalho não está na coleta de dados dos poços pela central de monitoramento, mas sim sua distribuição, essa parte não foi implementada neste trabalho. Para simular esses dados, o servidor gera dados aleatoriamente para preencher as estruturas definidas na listagem 6.1.

Sempre que o valor do atributo *bsw* estiver fora do valor ideal (maior 40%), um evento é gerado e enviado ao canal de eventos. A simulação da aplicação define que esse evento será gerado a cada 1 segundo.

6.4 Avaliação Usando as Métricas

Para a validação do conjunto de métricas escolheu-se a refatoração do *middleware* OiL. As métricas estáticas referentes à fase de refatoração e as métricas dinâmicas relacionadas a desempenho serão aplicadas inicialmente ao OiL e, em seguida, ao AO-OiL. Os valores serão coletados e comparados com a finalidade de analisar as melhorias e os prejuízos advindos da refatoração.

A refatoração do *middleware* OiL em AO-OiL foi escolhida como alvo por estar sendo desenvolvida por um aluno desse mesmo departamento, Diego Saraiva. Isso facilita o entendimento e o acesso ao código-fonte dos sistemas de *middleware*.

(ZHANG; JACOBSEN, 2003) afirmam que a avaliação da refatoração de sistemas deve ser feita com base nas propriedades tamanho e acoplamento. Os autores não mencionam métricas para separação de conceitos, mas entende-se que esse é um dos objetivos principais ao se realizar um processo de refatoração. Como acoplamento e separação de

conceitos são fatores da modularidade neste trabalho e a aspectização de um sistema é realizada visando melhorar sua modularização, as métricas das propriedades modularidade e tamanho serão consideradas relevantes para a avaliação do processo de aspectização.

O desempenho é uma propriedade fundamental de ser avaliada na fase de refatoração. É preciso verificar se uma possível melhoria na modularidade pelo emprego de aspectos no sistema não penalizou de modo significativo o desempenho. Por isso, será aplicada a métrica dinâmica de tempo de execução. Para medir o tempo de execução dos dois sistemas de *middleware*, executou-se nos mesmos um Sistema de Monitoramento de Poços composto por uma Central de Monitoramento (CM) executado em uma estação servidora que recebe dados de sensores e fornece informações para os clientes através de um sistema de eventos. Quando um dado é atualizado, o servidor verifica quais clientes estão interessados no tipo de dado e envia a notificação. A partir da execução desse sistema no OiL e no AO-OiL é possível coletar a métrica dinâmica de tempo de execução e comparar o desempenho dos dois sistemas de *middleware*.

No restante dessa subseção, apresentam-se os resultados. Os resultados de acoplamento e tamanho, foram coletados com a ferramenta CoMeTA-Lua. Os resultados de separação de conceitos e coesão foram coletados manualmente com a ajuda valiosa do Diego Saraiva, o desenvolvedor do AO-OiL.

6.4.1 Métricas de Separação de Conceitos

As métricas de separação de conceitos coletadas são CDC, CDO e CDLOC. A granularidade escolhida para essas métricas foi entidade, para fornecer informações mais detalhadas.

Os resultados das métricas de separação de conceitos serão agrupadas em (i) CORBA: módulos referentes à implementação da especificação CORBA, (ii) Kernel: módulos referentes à implementação do kernel do *middleware* e (iii) definição: módulos referentes à definição e inicialização do *middleware*.

6.4.2 Métrica CDC

A métrica CDC conta o número de entidades cujo principal propósito é implementar um conceito. Para o OiL e o AO-OiL, conta o número de conceitos considerados para a refatoração é dois, esse é o número máximo que a métrica pode obter.

Tabela 6.1: Resultados das métricas de separação de conceitos para o middleware OiL.

Módulos	**CDC**	**CDO**	**CDLOC**
CORBA	21	213	0
Kernel	5	29	77
Definição	0	0	0

6.4.3 Métrica CDO

A métrica CDO conta o número de operações cujo principal propósito é implementar um conceito. Para o OiL e o AO-OiL, uma operação pode implementar, como principal propósito, os conceitos distribuição e/ou coordenação.

6.4.4 Métrica CDLOC

A métrica CDLOC conta o número de pontos de transição para cada conceito em todas as linhas de código. No caso do OiL e do AO-OiL, conta o número de pontos de alternância entre o código-base e os códigos dos conceitos distribuição e coordenação.

Resultados da Métrica de Separação de Conceitos para o OiL

A tabela 6.1 mostra os resultados para das métricas CDC, CDO e CDLOC para o OiL. Como todos os módulos CORBA implementam o conceito distribuição, os valores de CDC e CDO são altos para o OiL. Como não há alternância entre o código-base e o código do conceitos, CDLOC = 0 para esses módulos. Os módulos Kernel, como há implementação de código-base e códigos dos conceitos distribuição e coordenação, apresentam um valor alto para CDLOC. Os módulos Kernel, por implementam principalmente código-base, apresentam valores para CDC e CDO baixos quando comparados aos valores para módulos CORBA. Os módulos Definição apresentam resultado zero para todoas as métricas, uma vez que não é propósito desses módulos implementarem ou contribuirem para a implementação dos conceitos.

Resultados da Métrica de Separação de Conceitos para o AO-OiL

A tabela 6.2 apresenta os resultados das métricas de separação de conceitos para o AO-OiL. Os resultados para os módulos CORBA permaneceram iguais ao do OiL. Isso porque, nesses módulos do OiL, o conceito de distribuição estava bem modularizado, fazendo com que não houvesse a necessidade de separá-lo no AO-OiL. Os módulos do

Tabela 6.2: Resultados das métricas de separação de conceitos para o middleware AO-OiL.

Módulos	**CDC**	**CDO**	**CDLOC**
CORBA	21	213	0
Kernel	6	52	27
Definição	0	0	0

Kernel apresentaram maiores valores das métricas CDC e CDO para o AO-OiL em relação ao OiL. Nesses módulos do OiL, havia o entrelaçamento dos conceitos no código-base. No AO-OiL, foi necessário criar novas entidades e operações de forma a modularizar esses conceitos. Essa afirmação é confirmada pelo valor mais baixo de CDLOC, indicando que houve uma diminuição na alternância entre o código-base e os códigos dos conceitos no AO-OiL. Assim como o OiL, o AO-OiL apresenta resultado zero para todas as métricas aplicadas aos módulos Definição.

6.4.5 Métricas de Acoplamento

A granularidade utilizada para a coleta das métricas de acoplamento foi módulo. No OiL, os módulos representam componentes. Um módulo pode ser definido como uma classe, pode conter definição de uma ou mais classes ou ainda não ter nenhuma classe definida dentro de si. Para efeito de ilustração, a listagem 6.3 contém um trecho do arquivo *event.lua* do OiL. A linha 1 contém a definição do módulo "oil.corba.services.event". Na linha 3, a classe *EventChannel* é declarada conforme LOOP: a biblioteca para definição de objetos de Lua. O construtor da classe *EventChannel* é definido nas linhas 5 a 16 e o destrutor, nas linhas 18 a 28.

A granularidade módulo foi escolhida porque alguns módulos não apresentam definição de classe, mas mesmo assim apresentam acoplamento a outros módulos, sendo, portanto, relevante para a avaliação do acoplamento do sistema. Entretanto, o que mais motivou essa escolha foi o fato de as classes dentro de um mesmo módulo, na maioria das vezes, apresentarem um alto acoplamento entre si e baixo acoplamento em relação a classes definidas em outros módulos. Isso indica que classes dentro de um mesmo módulo fazem parte de uma categoria, como definiu (MARTIN, 1994). Segundo (MARTIN, 1994), classes dentro de uma categoria são interdependentes e têm função e objetivo comuns. Esse é o caso dos módulos do OiL.

Listagem 6.3: trecho do arquivo event.lua do OiL para ilustrar a definição de um módulo em Lua.

```
module "oil.corba.services.event"

local EventChannel = oo.class()

function EventChannel.__init(class)
  self = oo.rawnew(class, {
    push_consumer_count = 0,
    push_supplier_count = 0,
    event_queue = EventQueue(),
    event_factory = EventFactory()
  })
  self.event_dispatcher = EventDispatcher(self.event_queue)
  self.consumer_admin = ConsumerAdmin(self)
  self.supplier_admin = SupplierAdmin(self)
  return self
end

function EventChannel:destroy()
  self.event_queue = nil
  self.event_factory = nil
  self.consumer_admin:destroy()
  self.consumer_admin = nil
  self.supplier_admin:destroy()
  self.supplier_admin = nil
  self.event_dispatcher = nil
  self.push_consumer_count = 0
  self.push_supplier_count = 0
end
```

Somente será considerado acoplamento entre componentes da aplicação, ou seja, entre os módulos do OiL. Acoplamento às bibliotecas da linguagem de programação Lua ou a LOOP, a biblioteca Lua para orientação a objetos, serão desconsiderados.

As métricas coletadas para avaliar acoplamento entre os módulos do OiL e do AO-OiL são CBO, DAC e MPC. Elas serão detalhadas a seguir. Para melhor apresentar os resultados das métricas, eles serão agrupados da seguinte forma: (i) módulos referentes à implementação de CORBA, (ii) módulos referentes à implementação do kernel do OiL e, por fim, (iii) módulos referentes à definição e inicialização da arquitetura do OiL.

Métrica CBO

A métrica CBO conta o número de entidades às quais uma entidade está acoplada. Uma entidade está acoplado a outra se usa variáveis ou métodos de outro. É da definição de CBO incluir também o acoplamento por herança.

No OiL e no AO-OiL, quando um módulo precisa obter referência a outro, ele usa o comando "require" seguido do nome do módulo. Assim, ele consegue acessar variáveis e métodos de outro módulo. Dessa forma, serão considerados para o resultado de CBO, referências a módulos do OiL definidas dentro de comandos "require" que, no restante do código, são utilizadas para acessar atributos e métodos de outros módulos. Chamadas a construtores de classes não são contadas para CBO.

No que diz respeito à herança, se houver referência a atributos herdados ou invocações a métodos herdados, esses casos são contados para CBO.

Métrica DAC

A métrica DAC conta, para uma entidade, o número de atributos não herdados que têm uma entidade como seu tipo. No caso do OiL e do AO-OiL, conta-se o número de atributos cujo tipo é um módulo não definido dentro do módulo que se está avaliando. Como Lua é uma linguagem dinamicamente tipada, define-se que um atributo é do tipo de um módulo quando recebe o retorno do construtor de um módulo definido fora do módulo que se está avaliando.

Métrica MPC

A métrica MPC conta, para uma entidade, o número de invocações estáticas a operações não implementadas nessa entidade. Para cada módulo do OiL foram contadas todas as chamadas a métodos não implementadas no módulo em questão. Invocações a métodos realizadas a partir de atributos com tipo de módulo também foram contadas.

Resultados das Métricas de Acoplamento para o OiL

Os resultados das métricas CBO, DAC e MPC para os módulos do OiL referentes à implementação CORBA são apresentados na tabela 6.3.

Analisando os valores de DAC, pode-se perceber que há poucos módulos contendo atributos com tipos de outros módulos. Os módulos que apresentam os maiores valores

Tabela 6.3: Resultados das métricas de acoplamento para os módulos do OiL referentes à implementação CORBA do OiL.

Módulos	**CBO**	**DAC**	**MPC**
oil.corba.giop	1	0	1
oil.corba.idl	0	0	0
oil.corba.giop.CodecGec	2	0	0
oil.corba.giop.Codec	2	0	3
oil.corba.giop.Exception	1	0	0
oil.corba.giop.Indexer	1	0	0
oil.corba.giop.Listener	2	2	0
oil.corba.giop.Messenger	1	6	0
oil.corba.giop.ProxyOps	3	0	5
oil.corba.giop.Referrer	1	3	0
oil.corba.giop.Requester	2	12	0
oil.corba.giop.ServantOps	1	0	1
oil.corba.idl.Compiler	1	0	2
oil.corba.idl.Importer	2	0	0
oil.corba.idl.Indexer	1	0	0
oil.corba.idl.ir	1	0	0
oil.corba.idl.Registry	2	2	1
oil.corba.idl.sysex	2	0	1
oil.corba.iiop.Profiler	1	3	0
oil.corba.interceptors.ClientSide	1	0	0
oil.corba.interceptors.ServerSide	1	0	0
oil.corba.services.event	3	5	4
oil.corba.services.naming	0	0	0
oil.corba.services.event.ConsumerAdmin	0	1	0
oil.corba.services.event.EventFactory	0	0	0
oil.corba.services.event.EventQueue	0	0	0
oil.corba.services.event.ProxyPushConsumer	0	0	0
oil.corba.services.event.ProxyPushSupplier	0	0	0
oil.corba.services.event.SingleDeferredDispatcher	0	0	0
oil.corba.services.event.SingleSynchronousDispatcher	0	0	0
oil.corba.services.event.SupplierAdmin	0	1	0

Tabela 6.4: Resultados das métricas de acoplamento para os módulos da implementação do kernel do OiL.

Módulos	**CBO**	**DAC**	**MPC**
oil.kernel.base.Acceptor	0	3	0
oil.kernel.base.Client	0	0	0
oil.kernel.base.Connector	0	0	0
oil.kernel.base.Dispatcher	0	0	0
oil.kernel.base.Invoker	0	0	0
oil.kernel.base.Proxies	0	0	0
oil.kernel.base.Receiver	0	0	0
oil.kernel.base.Server	0	0	0
oil.kernel.base.Sockets	0	0	0
oil.kernel.cooperative.Invoker	0	0	0
oil.kernel.cooperative.Mutex	0	0	0
oil.kernel.cooperative.Receiver	0	0	0
oil.kernel.typed.Client	0	0	0
oil.kernel.typed.Dipatcher	1	0	1
oil.kernel.typed.Proxies	1	0	1
oil.kernel.typed.Server	1	0	1

de DAC são oil.corba.giop.Messenger e oil.corba.giop.Requester. Esses valores referem-se a instâncias de oil.corba.giop.Exception criadas para tratar exceções, mostrando que o código de tratamento de exceção está entrelaçado ao código dos módulos. Uma alternativa viável para reduzir esse acoplamento seria definir exceções como um conceito e tratá-las através de aspectos.

Um outro módulo que apresenta um valor considerável para essa métrica é oil.corba.services.event. Isso ocorre porque, nesse módulo, há a instanciação de todos os módulos necessários para criação de eventos. São instanciados os módulos referentes à fila de eventos (oil.corba.services.event.EventQueue), à fábrica de eventos (oil.corba.services.event.EventFactory), ao encaminhamento de eventos (oil.corba.services.event.EventDispatcher) e ao consumidor (oil.corba.services.event.ConsumerAdmin) e produtor (oil.corba.services.event.SupplierAdmin).

Contribui para o baixo valor de DAC para a maioria dos módulos, a granularidade em que a métrica foi coletada. Utilizando o conceito de categoria de classes, atributo com tipo de classe que esteja definida dentro do mesmo módulo não conta para DAC.

Os resultados das métricas CBO, DAC e MPC para os módulos do OiL referentes à implementação do Kernel do OiL são apresentados na tabela 6.4.

Tabela 6.5: Resultados das métricas de acoplamento para os módulos de definição e inicialização do OiL.

Módulos	**CBO**	**DAC**	**MPC**
oil.arch.base	1	0	2
oil.arch.cooperative	2	0	2
oil.arch.corba	2	0	2
oil.arch.typed	2	0	2
oil.builder.base	2	0	1
oil.builder.cooperative	2	0	1
oil.builder.corba	3	0	1
oil.builder.gencode	2	0	1
oil.builder.intercepted	3	0	1
oil.builder.typed	2	0	1
oil.arch	0	0	0
oil.assert	0	0	0
oil.builder	0	0	0
oil	1	2	1
oil.properties	0	0	0
oil.verbose	0	0	0

Os valores das métricas para os módulos do kernel indicam baixo acoplamento. Esse comportamento é percebido pelas três métricas e mostra que, no tocante ao acoplamento, a modularização foi bem realizada. O módulo oil.kernel.base.Acceptor apresenta valor maior para a métrica DAC. Esse valor refere-se à criação de exceções através da instanciação do módulo oil.corba.giop.Exception, assim como em alguns módulos referentes à implementação CORBA. Mais uma vez, o código de tratamento de exceções encontra-se entrelaçado ao código dos módulos.

Os resultados das métricas CBO, DAC e MPC para os módulos de definição e inicialiazação da arquitetura do OiL na tabela 6.5.

Quase todos os módulos apresentam valores diferentes de zero para CBO. Esse comportamento é justificado por tratarem da definição e inicialização da arquitetura do OiL. Tanto os módulos de definição da arquitetura (cujos nome iniciam com oil.arch) quanto os de inicialização (cujos nomes iniciam com oil.builder) estão acoplados aos módulos referentes à implementação do kernel do OiL.

Para definir um valor médio do acoplamento para o sistema como um todo, pode-se fazer uma relação entre a soma dos valores de cada métrica para todos os módulos e a quantidade de módulos do sistema. O que resulta em um valor médio de CBO = 0.90, valor médio de DAC = 0.63 e valor médio para MPC de 0.37.

Resultados das Métricas de Acoplamento para o AO-OiL

Os resultados serão agrupados da seguinta forma: (i) módulos CORBA: módulos referentes à implementação de CORBA; (ii) módulos kernel: módulos referentes à implementação do kernel do AO-OiL e, por fim, (iii) módulos de inicialização: módulos referentes à definição e inicialização da arquitetura do AO-OiL.

Os resultados das métricas CBO, DAC e MPC para os módulos do AO-OiL referentes à implementação CORBA são apresentados na tabela 6.6.

Os valores de acoplamento para os módulos de implementação CORBA do AO-OiL são, no geral, baixos. Esses valores são praticamente iguais aos apresentados para os mesmos módulos do OiL, uma vez que foram feitas poucas modificações nesses módulos durante a refatoração. Somente a classe ao_oil.corba.services.event apresentou resultados menores para o AO-OiL do que para o OiL, indicando que a refatoração conseguiu diminuir o acoplamento nessa entidade.

Os resultados das métricas CBO, DAC e MPC para os módulos do AO-OiL referentes à implementação do Kernel são apresentados na tabela 6.7.

Os valores da tabela 6.7 apresentam baixos valores para as métricas de acoplamento. Somente a entidade ao_oil.distribution.Acceptor valor para DAC um pouco maior, DAC = 3. Os 3 atributos com tipo de uma entidade são relativas a instanciação de exceções. Assim como no OiL, o código de tratamento de exceções encontra-se entrelaçado no restante do código.

Os resultados das métricas CBO, DAC e MPC para os módulos de definição e inicialiazação da arquitetura do AO-OiL na tabela 6.8.

Os dados da tabela 6.8 mostram que o acoplamento nos módulos de definição e inicialização do AO-OiL é baixo. Fazendo uma média dos resultados para todo o sistema, tem-se CBO = 0,75, DAC = 0,54 e MPC = 0,48. Comparando aos valores obtidos para o OiL, o AO-OiL apresenta menores valores de acoplamento para CBO (0,75 contra 0,90) e DAC (0,54 contra 0,63) e maior valor para MPC (0,48 contra 0,37).

6.4.6 Métricas de Coesão

Assim como para o acoplamento, as métricas de coesão serão coletadas na granularidade categoria de classes. Observou-se que entidades pertencentes a um mesmo módulo implementam uma mesma funcionalidade e estão, portanto, correlacionadas. Assim, en-

Tabela 6.6: Resultados das métricas de acoplamento para os módulos referentes à implementação CORBA do AO-OiL.

Módulos	**CBO**	**DAC**	**MPC**
ao_oil.corba.giop	1	0	1
ao_oil.corba.idl	0	0	0
ao_oil.corba.giop.CodecGec	2	0	0
ao_oil.corba.giop.Codec	2	0	3
ao_oil.corba.giop.Exception	1	0	0
ao_oil.corba.giop.Indexer	1	0	0
ao_oil.corba.giop.Listener	2	2	0
ao_oil.corba.giop.Messenger	1	6	0
ao_oil.corba.giop.ProxyOps	3	0	5
ao_oil.corba.giop.Referrer	1	3	0
ao_oil.corba.giop.Requester	2	12	0
ao_oil.corba.giop.ServantOps	1	0	1
ao_oil.corba.idl.Compiler	1	0	2
ao_oil.corba.idl.Importer	2	0	0
ao_oil.corba.idl.Indexer	1	0	0
ao_oil.corba.idl.ir	1	0	0
ao_oil.corba.idl.Registry	2	2	1
ao_oil.corba.idl.sysex	2	0	1
ao_oil.corba.iiop.Profiler	1	3	0
ao_oil.corba.interceptors.ClientSide	1	0	0
ao_oil.corba.interceptors.ServerSide	1	0	0
ao_oil.corba.services.event	1	3	2
ao_oil.corba.services.naming	0	0	0
ao_oil.corba.services.event.ConsumerAdmin	0	1	0
ao_oil.corba.services.event.EventFactory	0	0	0
ao_oil.corba.services.event.EventQueue	0	0	0
ao_oil.corba.services.event.ProxyPushConsumer	0	0	0
ao_oil.corba.services.event.ProxyPushSupplier	0	0	0
ao_oil.corba.services.event.SingleDeferredDispatcher	0	0	0
ao_oil.corba.services.event.SingleSynchronousDispatcher	0	0	0
ao_oil.corba.services.event.SupplierAdmin	0	1	0

Tabela 6.7: Resultados das métricas de acoplamento para os módulos da implementação do kernel do AO-OiL.

Módulos	**CBO**	**DAC**	**MPC**
ao_oil.kernel.Client	0	0	0
ao_oil.kernel.Dispatcher	0	0	0
ao_oil.kernel.Proxies	0	0	0
ao_oil.kernel.Referrer	0	0	0
ao_oil.kernel.Server	0	0	0
ao_oil.cooperative.Invoker	0	0	0
ao_oil.cooperative.Mutex	0	0	0
ao_oil.cooperative.Receiver	0	0	0
ao_oil.typed.Client	0	0	0
ao_oil.typed.Dispatcher	1	0	1
ao_oil.typed.Proxies	0	0	0
ao_oil.typed.Server	1	0	1
ao_oil.distribution.Acceptor	0	3	0
ao_oil.distribution.Connector	0	0	0
ao_oil.distribution.Invoker	0	0	0
ao_oil.distribution.Proxies	0	0	0
ao_oil.distribution.Receiver	0	0	0
ao_oil.distribution.Sockets	0	0	0
ao_oil.aspects.Cooperative	1	0	1
ao_oil.aspects.Corba	1	0	1
ao_oil.aspects.Event	0	0	0
ao_oil.aspects.Proxy	0	0	0
ao_oil.aspects.Typed	0	0	0

Tabela 6.8: Resultados das métricas de acoplamento para os módulos de definição e inicialização do AO-OiL.

Módulos	**CBO**	**DAC**	**MPC**
ao_oil.arch.base	1	0	2
ao_oil.arch.cooperative	3	0	2
ao_oil.arch.corba	2	0	2
ao_oil.arch.typed	1	0	2
ao_oil.builder.base	2	0	1
ao_oil.builder.cooperative	2	0	1
ao_oil.builder.corba	3	0	1
ao_oil.builder.typed	2	0	1
ao_oil.arch	0	0	0
ao_oil.assert	0	0	0
ao_oil.builder	0	0	0
oil.properties	0	0	0
ao_oil.verbose	0	0	0

tidades definidas dentro de um mesmo módulo fazem parte de uma mesma categoria.

A métrica coletada para avaliar coesão dentro dos módulos é LCC, que será detalhada a seguir.

Métrica LCC

A métrica LCC conta o número de conceitos endereçados por uma entidade. No caso do OiL e do AO-OiL, devido à modularidade escolhida, conta-se, para cada módulo, o número de conceitos que o módulo implementa. Como os conceitos considerados na refatoração são apenas 2 (distribuição e coordenação), esse é o número máximo que a métrica LCC pode atingir.

Resultados da Métrica de Coesão para o OiL

Os resultados para a métrica LCC referentes aos módulos de implementação CORBA do OiL são apresentados na tabela 6.9.

Para os módulos referentes à implementação CORBA, quase todos os módulos possuem LCC = 1. Somente o módulo oil.corba.services.naming possui LCC = 0, por que não contribuir para a implementação nem do conceito distribuição nem de coordenação. Esses resultados mostram que há uma boa modularização dos conceitos, embora estejam espalhados por diversos módulos do sistema.

Os resultados para a métrica LCC referentes aos módulos de implementação do kernel do OiL são apresentados na tabela 6.10.

Quase todos os módulos da implementação do kernel do OiL estão envolvidos na implementação dos conceitos de distribuição, por esse motivo, LCC = 1 para a grande maioria deles.

Para todos os módulos de definição e de inicialização do OiL, o LCC = 0, uma vez que esses módulos não têm como principal propósito implementar um conceito.

Resultados da Métrica de Coesão para o AO-OiL

Os resultados da métrica LCC para os módulos de implementação de CORBA do AO-OiL podem ser vistos na tabela 6.11.

Assim como para o OiL, a grande maioria os módulos de implementação de CORBA

Tabela 6.9: Resultados da métrica de coesão para os módulos da implementação kernel do OiL.

Módulos	**LCC**
oil.corba.giop	1
oil.corba.idl	1
oil.corba.giop.CodecGec	1
oil.corba.giop.Codec	1
oil.corba.giop.Exception	1
oil.corba.giop.Indexer	1
oil.corba.giop.Listener	1
oil.corba.giop.Messenger	1
oil.corba.giop.ProxyOps	1
oil.corba.giop.Referrer	1
oil.corba.giop.Requester	1
oil.corba.giop.ServantOps	1
oil.corba.idl.Compiler	1
oil.corba.idl.Importer	1
oil.corba.idl.Indexer	1
oil.corba.idl.ir	1
oil.corba.idl.Registry	1
oil.corba.idl.sysex	1
oil.corba.iiop.Profiler	1
oil.corba.interceptors.ClientSide	1
oil.corba.interceptors.ServerSide	1
oil.corba.services.event	1
oil.corba.services.naming	0
oil.corba.services.event.ConsumerAdmin	1
oil.corba.services.event.EventFactory	1
oil.corba.services.event.EventQueue	1
oil.corba.services.event.ProxyPushConsumer	1
oil.corba.services.event.ProxyPushSupplier	1
oil.corba.services.event.SingleDeferredDispatcher	1
oil.corba.services.event.SingleSynchronousDispatcher	1
oil.corba.services.event.SupplierAdmin	1

Tabela 6.10: Resultados das métricas de acoplamento para os módulos da implementação do kernel do OiL.

Módulos	**LCC**
oil.kernel.base.Acceptor	1
oil.kernel.base.Client	0
oil.kernel.base.Connector	1
oil.kernel.base.Dispatcher	1
oil.kernel.base.Invoker	1
oil.kernel.base.Proxies	1
oil.kernel.base.Receiver	1
oil.kernel.base.Server	1
oil.kernel.base.Sockets	1
oil.kernel.cooperative.Invoker	2
oil.kernel.cooperative.Mutex	1
oil.kernel.cooperative.Receiver	2
oil.kernel.typed.Client	1
oil.kernel.typed.Dipatcher	1
oil.kernel.typed.Proxies	1
oil.kernel.typed.Server	1

possuem LCC = 1. Eles módulos estão envolvidos na implementação de coordenação ou de distribuição.

Os resultados da métrica LCC para os módulos do AO-OiL referentes à implementação do kernel são apresentados na tabela 6.12.

Os resultados da tabela 6.12 mostram que a maioria dos módulos possuem LCC = 0 ou LCC = 1, indicando boa modularidade no que diz respeito à coesão.

Todos os módulos de definição e inicialização do AO-OiL, assim como para o OiL, possuem LCC = 0, porque não é o propósito principal deles implementar um conceito.

6.4.7 Métricas de Tamanho

As métricas de tamanho utilizadas são LOC, VS, NOA e WOC. Diferentemente das métricas de acoplamento, a granularidade escolhida para as métricas de tamanho é entidade. Essa granularidade foi escolhida para fornecer informações mais detalhadas sobre o *middleware*.

Os resultados para as métricas NOA e WOC foram coletados por entidade. Lembrando que alguns módulos são definidos como classes. Dessa forma, há classes que contêm definições de outras classes dentro de si. Nesse caso, elas são consideradas classes distintas.

Tabela 6.11: Resultados das métricas de coesão para os módulos referentes à implementação CORBA do AO-OiL.

Módulos	**LCC**
ao_oil.corba.giop	1
ao_oil.corba.idl	1
ao_oil.corba.giop.CodecGec	1
ao_oil.corba.giop.Codec	1
ao_oil.corba.giop.Exception	1
ao_oil.corba.giop.Indexer	1
ao_oil.corba.giop.Listener	1
ao_oil.corba.giop.Messenger	1
ao_oil.corba.giop.Referrer	1
ao_oil.corba.giop.Requester	1
ao_oil.corba.giop.ServantOps	1
ao_oil.corba.idl.Compiler	1
ao_oil.corba.idl.Importer	1
ao_oil.corba.idl.Indexer	1
ao_oil.corba.idl.ir	1
ao_oil.corba.idl.Registry	1
ao_oil.corba.idl.sysex	1
ao_oil.corba.iiop.Profiler	1
ao_oil.corba.interceptors.ClientSide	1
ao_oil.corba.interceptors.ServerSide	1
ao_oil.corba.services.event	1
ao_oil.corba.services.naming	0
ao_oil.corba.services.event.ConsumerAdmin	1
ao_oil.corba.services.event.EventFactory	1
ao_oil.corba.services.event.EventQueue	1
ao_oil.corba.services.event.ProxyPushConsumer	1
ao_oil.corba.services.event.ProxyPushSupplier	1
ao_oil.corba.services.event.SingleDeferredDispatcher	1
ao_oil.corba.services.event.SingleSynchronousDispatcher	1
ao_oil.corba.services.event.SupplierAdmin	1

Tabela 6.12: Resultados das métricas de coesão para os módulos da implementação do kernel do AO-OiL.

Módulos	**LCC**
ao_oil.kernel.Client	0
ao_oil.kernel.Dispatcher	0
ao_oil.kernel.Proxies	0
ao_oil.kernel.Referrer	0
ao_oil.kernel.Server	0
ao_oil.cooperative.Invoker	1
ao_oil.cooperative.Mutex	1
ao_oil.cooperative.Receiver	2
ao_oil.typed.Client	1
ao_oil.typed.Dispatcher	0
ao_oil.typed.Proxies	0
ao_oil.typed.Server	1
ao_oil.distribution.Acceptor	1
ao_oil.distribution.Connector	1
ao_oil.distribution.Invoker	1
ao_oil.distribution.Proxies	1
ao_oil.distribution.Receiver	1
ao_oil.distribution.Sockets	1
ao_oil.aspects.Cooperative	1
ao_oil.aspects.Corba	1
ao_oil.aspects.Event	0
ao_oil.aspects.Proxy	1
ao_oil.aspects.Typed	0

Há outros casos em que o módulo não contém definição de classe. Para esse segundo caso, não há valores de NOA e WOC.

Assim como as métricas de acoplamento, as métricas de tamanho serão agrupadas em 3 categorias: (i) entidades contidas nos módulos referentes à implementação CORBA, (ii) entidades dos módulos referentes à implementação do kernel do OiL/AO-OiL e (iii) entidades contidas nos módulos de definição e inicialização da arquitetura do OiL/AO-OiL.

Métrica LOC

A métrica LOC conta o número de linhas de código válidas. No caso do OiL e do AO-OiL, número de linhas que contenham código a ser interpretado. Linhas completamente comentadas e linhas em branco não entram nessa contagem.

Métrica VS

A métrica VS conta o número de entidades de um sistema. No caso do OiL, o número de classes que o constituem. Para o AO-OiL, o número de classes e aspectos que o compõem.

Métrica NOA

A métrica NOA conta o número de atributos de uma entidade. São contados como atributos de uma entidade aqueles definidos dentro da lista de atributos no comando de definição da entidade e os atributos definidos com o nome da entidade, seguido de ponto e do nome do atributo. Atributos definidos dentro de método são considerados locais a ele e, portanto, não entram na contagem de NOA. Nos casos em que o módulo é definido como classe, todos os atributos definidos fora de métodos são considerados para a contagem de NOA.

Métrica WOC

A métrica WOC mede a complexidade de uma entidade em termos de suas operações. O resultado para essa métrica é dado pela soma das complexidades das operações da entidade. Em (SANT'ANNA et al., 2003), o trabalho do qual a métrica foi retirada, assume-se que uma operação com mais parâmetros é mais complexa do que uma com

menos, mas não se define como essa complexidade é medida, ficando a cargo do avaliador do sistema.

Para este trabalho, definiu-se que a complexidade de uma operação é igual ao seu número de parâmetros elevado à base 2. Dessa forma, mesmo que uma operação não tenha parâmetros, seu peso será considerado para a contagem do WOC do componente.

Construtores não entram na contagem de WOC, porque, na maioria dos casos, recebe como parâmetros todos os atributos de uma classe para inicializá-los. Isso forçaria o valor de WOC para cima.

Resultados das Métricas de Tamanho para o OiL

A métrica LOC para o OiL somam um total de 7700 linhas de código válidas. A métrica VS para o OiL é igual a 84 classes. É importante ressaltar que alguns módulos do OiL não contêm definição de classe. Relacionando o total de linhas de código e a quantidade de classes do sistema, pode-se chegar a um número médio de 91,67 linhas de código por classe.

Os resultados para as métricas NOA e WOC para os módulos referentes à implementação CORBA são apresentados nas tabelas 6.13 e 6.14.

Pelos resultados de NOA e WOC, pode-se observar que as classes não seguem um tamanho padrão aproximado, elas são bem distintas em relação a esse quesito. Tome como exemplo a classe DefaultDefs que possui WOC e NOA iguais a zero. Em contrapartida, Request possui NOA=16 e WOC=60.

As métricas NOA e WOC não parecem estar relacionadas, uma vez que os resultados não apresentam proporcionalidade. Somando todos os atributos e dividindo pelo número de classes, tem-se um total de 3,48 atributos por classe. Analogamente, somando o WOC de todas as classes e dividindo pelo número de classes, encontra-se um valor médio de WOC por classe de 18,31.

Os resultados para as métricas NOA e WOC para os módulos referentes à implementação do kernel do OiL são apresentados na tabela 6.15.

A partir dos resultados apresentados na tabela 6.15, pode-se observar que as classes dos módulos que implementam o kernel do OiL apresentam, em sua maioria, poucos atributos e alta complexidade das operações.

Somando o valor de NOA para todos as classes e dividindo pelo número de classe,

Tabela 6.13: Resultados das métricas de tamanho para os módulos de implementação CORBA do OiL.

Classes	**NOA**	**WOC**
EncoderGenerator	13	28
Decoder	8	8
oil.corba.giop.CodecGen	5	8
Encoder	8	6
DecoderGenerator	11	20
CodeGenerator	3	13
oil.corba.giop.Codec	8	56
Decoder	15	35
Encoder	16	68
oil.corba.giop.Exception	4	0
oil.corba.giop.Indexer	1	8
oil.corba.giop.Listener	18	60
Request	0	1
oil.corba.giop.Messenger	6	36
oil.corba.giop.ProxyOps	0	20
oil.corba.giop.Referrer	1	24
oil.corba.giop.Requester	13	44
oil.corba.giop.ServantOps	2	32
oil.corba.idl.Compiler	2	16
oil.corba.idl.Importer	3	20
DefaultDefs	0	0
oil.corba.idl.Indexer	3	36
StructDef	4	9
ArrayDef	3	9
ExceptionDef	4	9
IDLType	0	1
oil.corba.idl.Registry	1	36
Repository	3	12
AliasDef	3	8
MemberDef	0	8
StringDef	6	0
TypedefDed	2	1
IRObject	0	10
InterfaceDef	5	40
EnumDef	3	5
ObjectRef	0	0
AttributeDef	3	11
Container	0	132
SequenceDef	6	7
OperationDef	10	20
ModuleDef	3	1
Registry	0	2
UnionDef	6	13
Contained	2	16
PrimitiveDef	1	0

Tabela 6.14: Continuação - resultados das métricas de tamanho para os módulos de implementação CORBA do OiL.

Classes	**NOA**	**WOC**
oil.corba.iiop.Profiler	5	56
oil.corba.interceptors.ClientSide	2	12
CanceledRequest	0	2
oil.corba.interceptors.ServerSide	2	20
EventChannel	0	11
NamingContext	0	26
NamingContextExt	0	10
BindingIterator	0	4
oil.corba.services.event.ConsumerAdmin	0	20
oil.corba.services.event.EventFactory	0	4
oil.corba.services.event.EventQueue	0	6
oil.corba.services.event.ProxyPushConsumer	0	10
oil.corba.services.event.ProxyPushSupplier	0	6
oil.corba.services.event.SingleDeferredDispatcher	1	15
oil.corba.services.event.SingleSinchronousDispatcher	0	14
oil.corba.services.event.SupplierAdmin	0	20
Contents	1	10

Tabela 6.15: Resultados das métricas de tamanho para os módulos de implementação do Kernel do OiL.

Classes	**NOA**	**WOC**
oil.kernel.base.Acceptor	4	28
port	0	2
oil.kernel.base.Client	1	12
oil.kernel.base.Connector	2	18
SocketCache	0	0
oil.kernel.base.Dispatcher	2	32
oil.kernel.base.Invoker	3	40
Request	0	2
FailedFuture	0	2
Results	0	1
oil.kernel.base.Proxies	5	57
oil.kernel.base.Receiver	1	20
oil.kernel.base.Server	1	42
oil.kernel.base.Sockets	0	28
oil.kernel.cooperative.Invoker	1	24
oil.kernel.cooperative.Mutex	1	29
oil.kernel.cooperative.Receiver	1	32
oil.kernel.typed.Client	1	24
oil.kernel.typed.Dispatcher	1	16
oil.kernel.typed.Proxies	5	22
oil.kernel.typed.Server	2	24

Tabela 6.16: Resultados das métricas de tamanho para os módulos de definição e inicialização do OiL.

Classes	NOA	WOC
oil.properties	2	4

tem-se um valor médio de NOA de 1.48 e fazendo o mesmo para WOC, tem-se um valor médio de 21.67. Comparando aos valores médios das classes contidas nos módulos de implementação de CORBA, o número médio de atributos é menor (1.48 contra 3.48) e o WOC médio é maior (21.67 contra 18.31).

Os resultados para as métricas NOA e WOC para os módulos referentes à implementação de definição e inicialização do OiL são apresentados na tabela 6.16. Em todos esses módulos há definição de apenas uma classe.

Como esses módulos apresentam uma única classe, o valor médio de NOA é 2 e o WOC médio é 4. O valor médio de NOA é maior do que o dos módulos do kernel (2 conta 1.48) e menor do que o NOA médio dos módulos da implementação CORBA (2 contra 3.48). Já o valor médio do WOC é bem menor do que o das outras categorias.

Fazendo um valor médio para todo o sistema, ou seja, a soma dos valores de NOA para todas as classes dividido pelo número total de classes e soma dos valores de WOC para todas as classes dividido pelo número de classes do sistema, tem-se NOA=2.96 e WOC=18.98.

Resultados das Métricas de Tamanho para o AO-OiL

A métrica LOC para o AO-OiL somam um total de 8283 linhas de código válidas. A métrica VS para o AO-OiL é igual a 99 entidades. Assim como o OiL, alguns módulos do AO-OiL não contêm definição de entidade. Relacionando o total de linhas de código e a quantidade de entidades do sistema, pode-se chegar a um número médio de 83,67 linhas de código por classe.

Os resultados para as métricas NOA e WOC para os módulos referentes à implementação CORBA do AO-OiL são apresentados nas tabelas 6.17 e 6.18.

A partir dos dados das tabelas 6.17 e 6.18, pode-se observar que, em relação às métricas de tamanho NOA e WOC, o OiL e o AO-OiL apresentam os mesmos resultados para os módulos referentes à implementação CORBA. Isso se deve ao fato de que a maior parte do código de implementação de CORBA do AO-OiL foi reaproveitado do OiL. Portanto,

Tabela 6.17: Resultados das métricas de tamanho para os módulos de implementação CORBA do AO-OiL.

Classes	**NOA**	**WOC**
EncoderGenerator	13	28
Decoder	8	8
ao_oil.corba.giop.CodecGen	5	8
Encoder	8	6
DecoderGenerator	11	20
CodeGenerator	3	13
ao_oil.corba.giop.Codec	8	56
Decoder	15	35
Encoder	16	68
ao_oil.corba.giop.Exception	4	0
ao_oil.corba.giop.Indexer	1	8
ao_oil.corba.giop.Listener	18	60
Request	0	1
ao_oil.corba.giop.Messenger	6	36
ao_oil.corba.giop.ProxyOps	0	20
ao_oil.corba.giop.Referrer	1	24
ao_oil.corba.giop.Requester	13	44
ao_oil.corba.giop.ServantOps	2	32
ao_oil.corba.idl.Compiler	2	16
ao_oil.corba.idl.Importer	3	20
DefaultDefs	0	0
ao_oil.corba.idl.Indexer	3	36
StructDef	4	9
ArrayDef	3	9
ExceptionDef	4	9
IDLType	0	1
ao_oil.corba.idl.Registry	1	36
Repository	3	12
AliasDef	3	8
MemberDef	0	8
StringDef	6	0
TypedefDed	2	1
IRObject	0	10
InterfaceDef	5	40
EnumDef	3	5
ObjectRef	0	0
AttributeDef	3	11
Container	0	132
SequenceDef	6	7
OperationDef	10	20
ModuleDef	3	1
Registry	0	2
UnionDef	6	13
Contained	2	16
PrimitiveDef	1	0

Tabela 6.18: Continuação - resultados das métricas de tamanho para os módulos de implementação CORBA do AO-OiL.

Classes	**NOA**	**WOC**
ao_oil.corba.iiop.Profiler	5	56
ao_oil.corba.interceptors.ClientSide	2	12
CanceledRequest	0	2
ao_oil.corba.interceptors.ServerSide	2	20
EventChannel	0	11
NamingContext	0	26
NamingContextExt	0	10
BindingIterator	0	4
ao_oil.corba.services.event.ConsumerAdmin	0	20
ao_oil.corba.services.event.EventFactory	0	4
ao_oil.corba.services.event.EventQueue	0	6
ao_oil.corba.services.event.ProxyPushConsumer	0	10
ao_oil.corba.services.event.ProxyPushSupplier	0	6
ao_oil.corba.services.event.SingleDeferredDispatcher	1	15
ao_oil.corba.services.event.SingleSinchronousDispatcher	0	14
ao_oil.corba.services.event.SupplierAdmin	0	20
Contents	1	10

temos também para o AO-OiL, um valor médio de 3,48 atributos por entidade e um valor médio de WOC por entidade de 18,31.

Os resultados para as métricas NOA e WOC para os módulos referentes à implementação do kernel do AO-OiL são apresentados na tabela 6.19.

Pode-se observar, a partir dos dados da tabela 6.19, que a maioria desses módulos apresentam poucos atributos e alta complexidade de operações. Fazendo uma média dos valores das métricas, tem-se um total de 1,25 atributos por entidade e um valor médio de WOC de 30,46 por entidade. Os módulos de implementação do kernel do AO-Oil apresentam uma média menor de NOA que os do OiL (1,25 contra 1,48) e uma média de WOC maior (30,46 contra 21,47). Em relação à média dos módulos de implementação CORBA do AO-OiL, os módulos de implementação do kernel do AO-OiL apresentam uma média menor de NOA (1,25 contra 3,48) e uma média de WOC maior (30,46 contra 18,31).

Os resultados para as métricas NOA e WOC para os módulos referentes à implementação de definição e inicialização do AO-OiL são apresentados na tabela 6.20. Em todos esses módulos há definição de apenas uma classe.

Como esses módulos apresentam uma única classe, o valor médio de NOA é 2 e o

Tabela 6.19: Resultados das métricas de tamanho para os módulos de implementação do Kernel do AO-OiL.

Classes	**NOA**	**WOC**
ao_oil.kernel.Client	1	12
ao_oil.kernel.Dispatcher	2	48
Results	0	1
ao_oil.kernel.Proxies	2	50
ao_oil.kernel.Referrer	3	16
ao_oil.kernel.Server	1	42
ao_oil.distribution.Acceptor	4	28
ao_oil.distribution.Connector	2	18
ao_oil.distribution.Invoker	3	108
ao_oil.distribution.Proxies	5	57
Results	0	1
FailedFuture	0	2
ao_oil.distribution.Receiver	1	44
ao_oil.distribution.Sockets	0	28
ao_oil.typed.Client	1	24
ao_oil.typed.Dispatcher	1	16
ao_oil.typed.Proxies	5	69
ao_oil.typed.Server	2	24
Cooperative	2	69
cooperative_aspect	0	69
Thread	0	5
CorbaServer	0	42
CorbaClient	0	13
corba_server	0	41
corba_client	0	11
event_aspect	0	5
aproxy	0	1
typed_aspect	0	8

Tabela 6.20: Resultados das métricas de tamanho para os módulos de definição e inicialização do AO-OiL.

Classes	**NOA**	**WOC**
oil.properties	2	4

WOC médio é 4. Esses valores são iguais aos obtidos para os módulos de definição e inicialização do OiL. O valor médio de NOA é maior do que o dos módulos do kernel do AO-OiL (2 conta 1.25) e menor do que o NOA médio dos módulos da implementação CORBA do AO-OiL (2 contra 3.48). Já o valor médio do WOC é bem menor do que o das outras categorias.

6.4.8 Métrica Tempo de Execução

A métrica dinâmica tempo de execução é útil para comparar o desempenho dos sistemas de *middleware*. Para coletar essa métrica, executou-se no OiL e no AO-OiL o sistema de monitoramento de poços de petróleo, apresentado no capítulo 6.3. Para cada *middleware*, executou-se a aplicação com número de eventos = 20, 50 e 100 e número de clientes = 1, 5, 10 e 50, para que se tivesse números variados de eventos e clientes. O tempo de execução será o tempo decorrido desde o pedido de cadastro no servidor (*subscribe*) do cliente até o momento que ele envia o pedido de desligamento do servidor (*unsubscribe()*). O valor apresentado como resultado é uma média desses tempos de todos os clientes. Para tornar essa explicação mais clara, segue um exemplo: na situação com 5 clientes e 20 eventos, cada cliente chama *subscribe()* do servidor, recebe 20 eventos e em seguida chama *unsubscribe()*. O tempo decorrido entre o *subscribe* e o *unsubscribe* é o tempo de execução. Como são 5 clientes, o tempo de execução dessa situação será uma média dos tempos de todos os clientes. Foram realizadas três execuções de cada situação. Os menores resultados para cada situação são apresentados, embora as três execuções apresentem resultados muito próximos.

A tabela 6.21 apresenta o tempo de execução em segundos para cada uma dessas situações para o OiL. A tabela 6.22 apresenta o tempo de execução em segundos para as mesmas situações no AO-OiL. A partir dessas duas tabelas, é possível observar que os valores de execução da aplicação nos dois sistemas de *middleware* são muito próximos. Somente na situação com 1 cliente e 20 eventos, o AO-OiL apresentou menor tempo de execução, uma melhoria de 1,43%. A maior diferença de tempo de execução entre o OiL e o AO-OiL ocorreu na situação com 20 eventos e 50 clientes: o AO-OiL demorou 5,35% a mais que o OiL.

Tabela 6.21: Resultados da métrica dinâmica tempo de execução para o OiL, onde ne significa número de eventos.

número de clientes	ne=20	ne=50	ne=100
1	21,05	50,98	102,61
5	23,84	52,26	106,27
10	28,33	59,04	110,11
50	59,85	90,93	142,35

Tabela 6.22: Resultados da métrica dinâmica tempo de execução para o AO-OiL, onde ne significa número de eventos.

número de clientes	ne=20	ne=50	ne=100
1	20,75	51,24	102,62
5	24,35	55,18	106,42
10	29,14	60,07	110,79
50	63,23	93,79	144,18

6.4.9 Análise Comparativa

Essa subseção faz uma análise comparativa entre os sistemas de *middleware* OiL e AO-OiL quanto aos resultados das métricas de modularidade, tamanho e tempo de execução. A tabela 6.23 mostra os resultados das métricas de tamanho VS e LOC para os dois sistemas de *middleware*. O AO-Oil apresentou um aumento de 17.9% no número de entidades em relação ao OiL. Em relação ao número de linhas, o AO-OiL possui 7.6% mais linhas de código que o OiL. É possível dizer que a refatoração acarretou em um pequeno aumento nessas duas medidas de tamanho. Para separar o código-base do código de conceitos que estavam entrelaçados no OiL, foi necessário criar novas classes e aspectos no AO-OiL, de forma a prover a separação de conceitos. Por isso o aumento no número de entidades e o aumento no número de linhas de código na versão refatorada.

A apresentação dos resultados restantes, os módulos do OiL e AO-OiL foram divididos em três categorias: (i) módulos referentes a definição e inicialização do middleware, (ii) módulos referentes à implementação do kernel do middleware e (iii) módulos referentes à implementação da especificação CORBA. Gráficos comparativos para as métricas NOA

Tabela 6.23: Resultados das métricas de tamanho VS e LOC.

Middleware	**VS**	**LOC**
OiL	84	7700
AO-OiL	99	8283

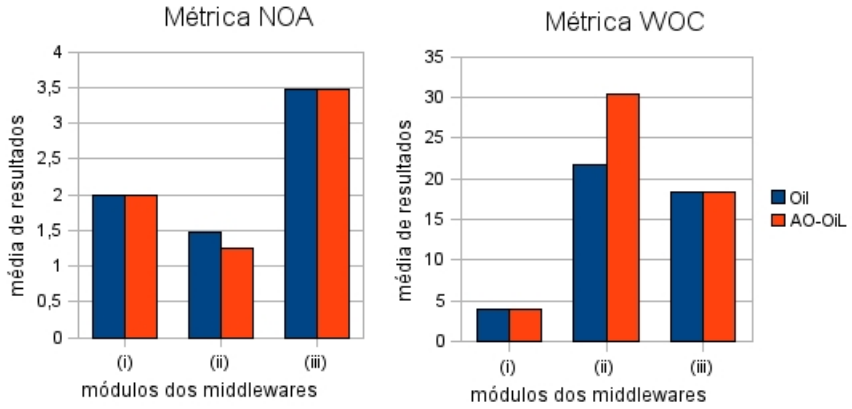

Figura 6.8: Resultados das métricas de tamanho NOA e WOC.

e WOC podem ser vistos na figura 6.8, respectivamente nos lados esquerdo e direito da figura. Para facilitar a comparação dos resultados, apresenta-se então uma média calculada da seguinte forma: soma dos resultados para a métrica de todas as entidades dividida pelo número de entidades, para cada uma das categorias.

Como a refatoração do OiL contemplou distribuição e coordenação, as mudanças foram feitas nos módulos de kernel do *middleware*. Poucas mudanças foram necessárias nos módulos referentes à definição e inicialização do middleware e à implementação do CORBA. Por isso, para as três categorias avaliadas, somente os resultados para os módulos de implementação do kernel apresentam diferenças entre o OiL e o AO-OiL. Enquanto a média do número de atributos diminuiu no AO-OiL em 15,5% em relação ao OiL, a média do número de operações ponderadas por parâmetros subiu 40,5% no AO-OiL. O aumento considerável de WOC no AO-OiL deve-se à definição de novas operações para que fosse possível separar código-base do código dos conceitos dentro das operações.

Os resultados para as métricas de separação de conceitos são apresentados na tabela 6.24. Esses resultados foram obtidos pela soma dos resultados de cada métrica para todas as entidades do sistema. CDC e CDO obtiveram resultados um pouco melhor para o OiL do que para o AO-OiL. Isso significa que os conceitos estão implementados em um maior número de módulos e operações. Em contrapartida, o AO-OiL apresentou um resultado significativamente menor para CDLOC, indicando que o seu código encontra-se melhor organizado no que diz respeito à implementação dos conceitos. Com a refatoração, foram criadas novas entidades e novas operações destinadas aos códigos dos conceitos, separando-os do código-base. Por esse motivo, assim como houve um aumento de VS e WOC, registrou-se maiores valores para CDC e CDO para o AO-OiL. O valor consideravelmente

Tabela 6.24: Resultados das métricas de separação de conceitos.

Middleware	**CDC**	**CDO**	**CDLOC**
OiL	26	242	77
AO-OiL	27	265	27

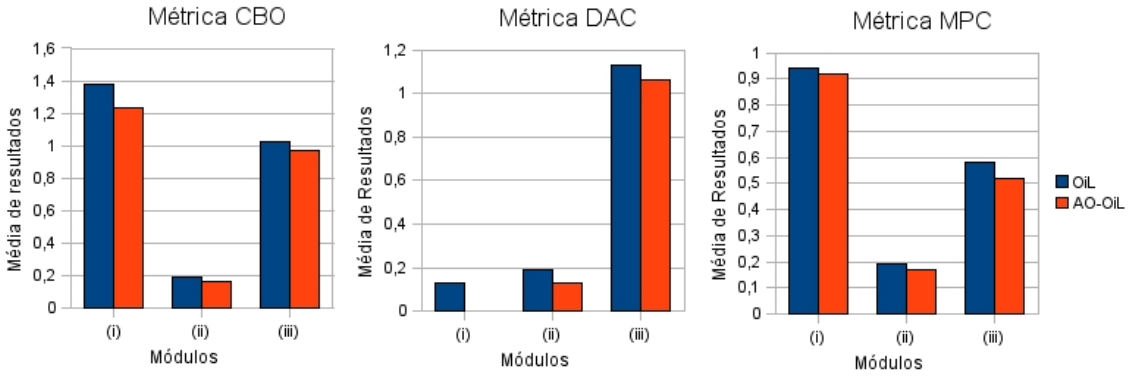

Figura 6.9: Resultados das métricas de tamanho NOA e WOC.

baixo para CDLOC indica que há menos entrelaçamento dos códigos de conceitos e base dentro de uma mesma entidade na versão refatorada.

Assim como para as métricas de tamanho NOA e WOC, os resultados das métricas de acoplamento são apresentados como uma média: soma dos valores de cada métrica para todos os módulos dividido pelo número de módulos. Gráficos comparativos para essas métricas são apresentados na figura 6.9, respectivamente no lado esquerdo, centro e lado direito da figura. Pode-se observar que o OiL apresentava baixo acoplamento entre seus módulos. Mesmo assim, os valores das métricas para o AO-OiL foram ainda mais baixos, indicando que houve uma diminuição do acoplamento entre os módulos.

Como os conceitos considerados na refatoração são apenas 2 (distribuição e coordenação), esse é o número máximo que a métricas LCC pode atingir. O número de módulos com LCC = 1 no OiL é 43 e no AO-OiL é 40. O número de módulos com LCC = 2 no OiL é 2 contra apenas 1 no AO-OiL. A partir desses resultados, observa-se que houve melhoria na coesão dos módulos na versão refatorada. Pode-se fazer a seguinte avaliação sobre a modularidade. No quesito separação de conceitos, é difícil afirmar qual o melhor middleware: o OiL apresentou melhores resultados para as métricas CDC e CDO e o AO-Oil, melhores resultados para CDLOC. Quanto à coesão e acoplamento, o AO-OiL apresentou melhores resultados para todas as métricas. Posto isso, pode-ser afirmar que o AO-OiL apresenta melhor modularidade que o OiL.

Quanto ao tamanho, o OiL apresenta menor número de entidades, maior número de linhas de código e maior soma de operações ponderadas por parâmetros do que o AO-OiL. O AO-OiL obteve resultado menor apenas para o número de atributos. Assim, pode-se concluir que, com relação ao tamanho, o AO-OiL é maior do que o OiL. O processo de refatoração acarretou em melhorias na modularidade, mas também aumento no tamanho. Essa relação entre melhorias na modularidade (maior separação de intesses, menor acoplamento e maior coesão) e maior tamanho após a refatoração, foi também observada nos trabalhos (KULESZA et al., 2006) e (CACHO et al., 2006).

Segundo (KULESZA et al., 2006), um maior número de componentes (VS) na versão OA não pode ser visto como um ponto negativo, mas como uma evidência do aumento da modularidade. O mesmo pode ser concluído para o maior número de entidades no AO-OiL, em comparação ao OiL. O aumento nos resultados da métrica WOC para o AO-OiL é também reflexo da melhoria na modularidade. Para melhor organizar o sistema, foi necessário definir mais operações, de forma a separar melhor os conceitos. Como conseqüência de maior VS e maior WOC, os valores de CDC e CDO também foram superiores para o AO-OiL, indicando que há mais entidades e mais operações cujo principal propósito é implementar um conceito. O código referente a um conceito que, no OiL, fazia parte de uma entidade cujo principal propósito era implementar funcionalidades do código-base, passa a ser, no AO-OiL, um componente separado, incrementando o valor de CDC. O mesmo acontece com trechos de código dentro de operações cujo propósito era implementar funcionalidades do código-base no OiL e passam a ser implementados separadamente no AO-OiL, incrementando o valor de CDO. Os valores coletados para CDLOC argumentam a favor desse raciocínio: no AO-OiL, o CDLOC é consideravelmente menor que no OiL, uma vez que há menor alternância entre código de conceitos e código base dentro de uma mesma entidade.

No que diz respeito à métrica tempo de execução, os dois sistemas de *middleware* apresentaram resultados semelhantes. O que significa que a refatoração não teve impacto no desempenho. Isso pode ser observador nos gráficos comparativos da figura 6.10. Os gráficos da esquerda, central e da direita mostram, respectivamente, os resultados para um número variado de clientes e 20, 50 e 100 eventos.

Podemos concluir a análise afirmando que o AO-OiL apresentou melhor modularidade, maior tamanho e desempenho semelhante em relação ao OiL.

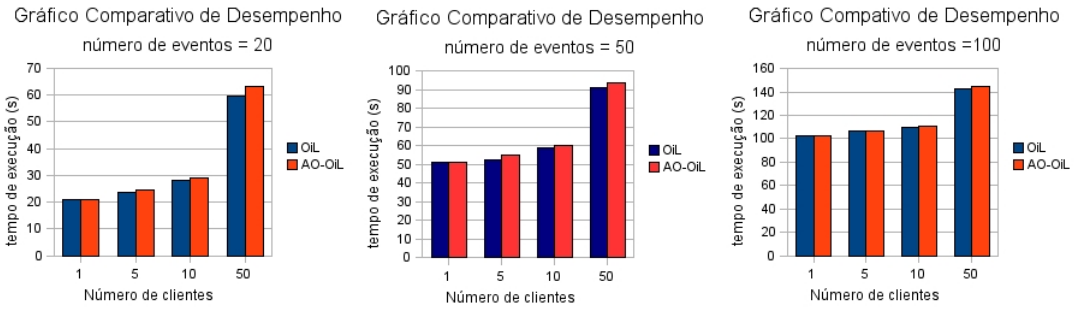

Figura 6.10: Resultados da métrica tempo de execução.

7 Considerações Finais

Esse capítulo apresenta as considerações finais deste trabalho. A seção 7.1 apresenta as contribuições e a seção 7.2, os trabalhos futuros.

7.1 Contribuições

Este trabalho apresentou um conjunto de métricas para avaliação estática e dinâmica de *middleware* OA. Propomos um conjunto de propriedades importantes para *middleware* OA (modularidade, flexibilidade, manutenibilidade, tamanho, complexidade, desempenho e consumo de memória) e as métricas associadas a cada propriedade. As maioria das métricas adotadas no trabalho são adaptações de métricas bem conhecidas para avaliação de sistemas OO. A lista de propriedades baseia-se no trabalho de (DRIVER, 2002). As métricas de separação de conceitos CDC, CDO e CDLOC e as de tamanho VS, LOC, NOA e WOC foram retiradas de (SANT'ANNA et al., 2003). A métrica de coesão LCC, de (SANT'ANNA, 2008). As métricas de acoplamento entidade-entidade foram adaptadas para o paradigma OA das apresentadas em (BRIAND; DALY; WUEST, 1999), assim como as métricas de herança DIT e NOC, de complexidade RFC de (CHIDAMBER; KEMERER, 1994) e as métricas de estabilidade Ca, Ce e I de (MARTIN, 1994). As métricas de acoplamento código base-aspecto foram retiradas de (CECCATO; TONELLA, 2004). As métricas dinâmicas desempenho e consumo de memória foram baseadas em (DUFOUR et al., 2004). A partir de todas essas abordagens, foi possível compor um conjunto de métricas para avaliar diversas características de um sistema de *middleware* OA, em largura e profundidade, nas fase de *design*, implementação, refatoração e execução.

O conjunto de métricas foi aplicado na avaliação do *middleware* OiL e sua refatoração, o AO-OiL. Aplicando métricas estáticas na fase de implementação foi possível avaliar os pontos positivos e negativos da aplicação do paradigma OA. A implementação em Lua do sistema de monitoramento de poços de petróleo permitiu a coleta de uma métrica dinâmica no OiL e no AO-OiL, possibilitando a avaliação do impacto no desempenho

decorrente do uso de aspectos. O conjunto de métricas se mostrou um meio efetivo para auxiliar a avaliação e comparação dos sistemas de *middleware*.

A ferramenta CoMeTA-Lua permitiu agilizar o processo de avaliação por automatizar a coleta de métricas de tamanho e acoplamento. A ferramenta permite coletar métricas não somente no OiL e no AO-OiL, mas também em todo *middleware* construído em Lua, por exemplo, como trabalho futuro aplicaremos as métricas no AspectOpenOrb (CACHO et al., 2006).

A avaliação realizada nos sistemas de *middleware* permitiu ilustrar o uso do conjunto de propriedades, métricas e da ferramenta proposta no âmbito da comparação de duas versões de um *middleware*: uma OO e outra OA. Além de comprovar a utilidade da abordagem proposta como agente facilitar da avaliação da refatoração de um *middleware*.

É importante observar que o conjunto de métricas também pode ser aplicado a outros tipos de sistemas, pois as métricas não são restritas ao contexto de *middleware*. Diz-se que sua finalidade é a avaliação de sistemas de *middleware* porque a lista de propriedades é baseada na lista de propriedades levantadas por Driver (DRIVER, 2002) para avaliação de sistemas distribuídos. Em outras palavras, as métricas não são específicas para avaliação de sistemas de *middleware*, mas foram selecionadas como meio de avaliar propriedades relevantes a esse contexto.

7.2 Trabalhos Futuros

Diversos trabalhos futuros podem ser derivados dessa dissertação. É interessante aplicar o conjunto de métricas em outras fases de desenvolvimento bem como em outros sistemas de *middleware*. Assim como definir um estudo de caso para comparar sistemas de *middleware* OA similares, como por exemplo duas implementações de uma mesma especificação.

A ferramenta CoMeTA-Lua precisa ser melhorada. Alguns pontos são importantes para deixá-la mais completa:

- tornar a implementação da ferramenta mais robusta;
- implementar a coleta das métricas na granularidade de classe para as métricas de coesão e acoplamento, de forma a facilitar a coleta em sistemas que precisem de uma avaliação mais detalhada;

- implementar a automatização da coleta das outras métricas propostas por este trabalho;
- testar a ferramenta em outros sistemas de *middleware* em Lua, pois até o momento ela foi aplicada somente ao OiL e ao AO-OiL;
- implementar uma interface gráfica, tornando a interação entre ferramenta e o avaliador mais amigável.

Referências Bibliográficas

AHO, A.; SETHI, R.; ULLMAN, J. *Compiladores: princípios, técnicas e ferramentas*. Rio de Janeiro: LTC, 1995.

ASPECTJ Programming Guide. 2008. Disponível em: http://www.eclipse.org/aspectj/doc/released/progguide/index.html. Acesso em 04/11/2008.

BARTOLOMEI, T. T. et al. Towards a unified coupling framework for measuring aspect-oriented programs. In: *SOQUA '06: Proceedings of the 3rd international workshop on Software quality assurance*. Portland, Oregon: ACM, 2006. p. 46–53.

BATISTA, T.; VIEIRA, M. Reaspectlua - achieving reuse in aspectlua. *Journal of Universal Computer Science*, v. 13, n. 6, p. 786–805, 2007.

BERNSTEIN, P. Middleware: a model for distributed system services. *Communications of the ACM*, v. 39, n. 2, p. 86–98, 1996.

BRIAND, L. C.; DALY, J.; WUEST, J. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, v. 25, n. 1, p. 91–121, 1999.

BRIAND, L. C.; DALY, J. W.; WUST, J. A unified framework for cohesion measurement in object-orientedsystems. *Empirical Softw. Engg.*, v. 3, n. 1, p. 65–117, 1998.

BRIAND, L. C.; MORASCA, S.; BASILI, V. R. Measuring and assessing maintainability at the end of high level design. In: *ICSM '93: Proceedings of the Conference on Software Maintenance*. [S.l.]: IEEE Computer Society, 1993. p. 88–97.

CACHO, N. et al. Improving modularity of reflective middleware with aspect-oriented programming. In: *SEM '06: Proceedings of the 6th international workshop on Software engineering and middleware*. Portland, Oregon: ACM, 2006. p. 31–38.

CECCATO, C.; TONELLA, P. Measuring the effects of software aspectization. In: *Proceedings of the 1st Workshop on Aspect Reverse Engineering*. [S.l.: s.n.], 2004.

CHIDAMBER, S.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, v. 20, p. 476–493, 1994.

CORBA. 2008. Disponível em: http://www.corba.org. Acesso em 07/07/2008.

DRIVER, C. *Evaluation of Aspect-Oriented Software Development for Distributed Systems*. Dissertação (Mestrado) — Universidade de Dublin, 2002.

DUFOUR, B. et al. Dynamic metrics for java. In: *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*. [S.l.]: ACM SIGPLAN, 2003. p. 149–168.

DUFOUR, B. et al. Measuring the dynamic behaviour of AspectJ programs. In: *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*. Vancouver, Canada: ACM SIGPLAN, 2004. p. 150–169.

ELRAD, T.; FILMAN, R.; BADER, A. Aspect-oriented programming. *Communications of ACM*, v. 44, n. 10, p. 29–32, 2001.

FENTON, N. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, v. 20, n. 3, p. 199–206, 1994.

FIGUEIREDO, E.; GARCIA, A.; LUCENA, C. AJATO: an AspectJ assessment tool. In: *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*. Nantes, France: [s.n.], 2006. (Demo Session).

FIGUEIREDO, E. et al. On the maintainability of aspect-oriented software: A concern-oriented measurement framework. In: *CSMR 2008: 12th European Conference on Software Maintenance and Reengineering*. Athens: [s.n.], 2008.

FRAKES, W.; TERRY, C. Software reuse: metrics and models. *ACM Comput. Surv.*, v. 28, n. 2, 1996.

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*. Chicago, Illinois: ACM, 2005. p. 3–14.

GIBBS, C.; COADY, Y. OASIS: Organic aspects for system infrastructure software - easing evolution and adaptation through natural decomposition. In: *Proceedings ECOOP 2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution*. [S.l.: s.n.], 2004.

HAUPT, M.; MEZINI, M. Micro-measurements for dynamic aspect-oriented systems. In: *Net.ObjectDays*. [S.l.: s.n.], 2004. p. 81–96.

JBOSSAOP. 2008. Disponível em: http://www.jboss.org/jbossaop. Acesso em: 09/04/2008.

KICZALES, G. et al. An overview of AspectJ. In: *Proceedings of the 15th European Conference on Object-Oriented Programming*. [S.l.]: Springer-Verlag, 2001. p. 327–353.

KICZALES, G. et al. Aspect oriented programming. In: *Proceedings of the 11th European Conference on Object-Oriented Programming*. [S.l.]: Springer-Verlag, 1997. v. 1241, p. 220–242.

KULESZA, U. et al. Quantifying the effects of aspect-oriented programming: A maintenance study. In: *ICSM '06: 22nd IEEE International Conference on Software Maintenance*. [S.l.: s.n.], 2006. p. 223–233.

KUMAR, A.; KUMAR, R.; GROVER, P. S. Towards a unified framework for cohesion measurement in aspect-oriented systems. In: *ASWEC '08: Proceedings of the 19th Australian Conference on Software Engineering*. [S.l.]: IEEE Computer Society, 2008. p. 57–65.

LOOP: Lua Object-Oriented Programming. 2008. Disponível em: http://loop.luaforge.net. Acesso em 07/10/2008.

LOUGHRAN, N. et al. *Requirements and definition of aspect-oriented middleware reference architecture*. 2005. Technical Report.

LOUGHRAN, N. et al. *Survey of Aspect-Oriented Middleware*. 2008. Disponível em: http://www.comp.lancs.ac.uk/computing/aop/AOSD_Europe.php. Acesso em 12/03/2008.

LUA. 2008. Disponível em: http://www.lua.org. Acesso em 07/07/2008.

MAIA, R. et al. Oil: An object request broker in the Lua language. In: *Proceedings of the 5th Tools Session of the Brazilian Simposium on Computer Networks (SBRC2006)*. Curitiba, Brazil: [s.n.], 2006.

MAIA, R.; CERQUEIRA, R.; KON, F. A middleware for experimentation on dynamic adaptation. In: *Proc. 4th Workshop on Adaptive and Reflective Middleware (ARM2005), co-located with 6th International Middleware Conference*. Grenoble, France: [s.n.], 2005.

MARTIN, R. Oo desgin quality metrics. In: *OOPSLA'94: Proc. Workshop Pragmatic and Theoretical Directions on Object-Oriented Software Metrics*. [S.l.: s.n.], 1994. Position paper.

PEREZ, J. et al. PRISMA: towards quality, aspect oriented and dynamic software architectures. In: *QSIC '03: Proceedings of the 3rd International Conference on Quality Software*. [S.l.]: IEEE Computer Society, 2003.

PERL. 2008. Disponível em: http://www.perl.org. Acesso em 31/10/2008.

POPOVICI, A.; ALONSO, G.; GROSS, T. Just-in-time aspects: Efficient dynamic weaving for java. In: *Proceedings of the 2nd international conference on Aspect-oriented software development*. [S.l.]: ACM Press, 2003. p. 100–109.

POPOVICI, A.; GROSS, T.; ALONSO, G. Dynamic weaving for aspect-oriented programming. In: *Proceedings of the 1st international conference on Aspect-oriented software development*. [S.l.]: ACM Press, 2002. p. 141–147.

POULIN, J. S. Measuring software reusability. In: *Proc. of the Third International Conference on Software Reuse: Advances in Software Reusability*. Rio de Janeiro, Brazil: [s.n.], 1994. p. 126–138.

SALIM, D. *Um Sistema Distribuído para Monitoramento de Poços de Petróleo com Elevação Artificial*. 2004. Monografia de conclusão de curso (Graduação em Engenharia da Computação. Universidade Federal do Rio Grande do Norte).

SANT'ANNA, C. et al. On the modularity assessment of software architectures: do my architectural concerns count? In: *First Workshop on Aspects in Architectural Description*. [S.l.: s.n.], 2007.

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

SANT'ANNA, C. N. *On the Modularity of Aspect-Oriented Design: A Concern-Driven Measurement Approach*. Tese (Doutorado) — Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, 2008.

SILVA, J. et al. Um middleware orientado a aspectos baseado em uma arquitetura de referência. In: *Workshop de Teses e Dissertações em Engenharia de Software*. [S.l.: s.n.], 2008.

SILVA, J. D. S. *AO-OiL: um middleware orientado a aspectos baseado em um arquitetura de referência*. 2008. Exame de Qualificação (Mestrado) - Universidade Federal do Rio Grande do Norte.

THOMAS, J. E. *Fundamentos da Engenharia do Petróleo*. 2. ed. Rio de Janeiro: Interciência, 2001.

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*. [S.l.: s.n.], 2001.

WASHIZAKI, H.; YAMAMOTO, H.; FUKAZAWA, Y. A metrics suite for measuring reusability of software components. In: *Proceedings of the ninth international software metrics symposium*. [S.l.: s.n.], 2003. p. 211–223.

WOHLSTADTER, E.; JACKSON, S.; DEVANBU, P. DADO: enhancing middleware to support crosscutting features in distributed, heterogeneous systems. In: *Proceedings of the 25th International Conference on Software Engineering*. [S.l.]: IEEE Computer Society, 2003. p. 174–186.

ZHANG, C.; JACOBSEN, H. Quantifying aspects in middleware platforms. In: *AOSD '03: Proceedings of the 2nd International Conference on Aspect-oriented Software Development*. [S.l.: s.n.], 2003. p. 130–139.

ZHAO, J. *Towards a Metrics Suite for Aspect-Oriented Software*. 2002. Technical Report SE-136-25. Information Processing Society of Japan (IPSJ).

ZHAO, J. *Measuring Coupling in Aspect-Oriented Systems*. 2003. Technical Report SE-142-6. Information Processing Society of Japan (IPSJ).

ZHAO, J.; XU, B. Measuring aspect cohesion. In: *Proceedings of Fundamental Approaches to Software Engineering (FASE'04)*. [S.l.]: Springer-Verlag, 2004. p. 54–68.

Livros Grátis

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

Milhares de Livros para Download:

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