PUC

ISSN 0103-9741

Monografias em Ciência da Computação

n° 14/05

Avaliação de um Modelo de Qualidade para Implementações Orientadas a Objetos e Orientadas a Aspectos

Eduardo Magno Lages Figueiredo

Arndt von Staa

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 14/05 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena Abril, 2005

Avaliação de um Modelo de Qualidade para Implementações Orientadas a Objetos e Orientadas a Aspectos

Eduardo Magno Lages Figueiredo Arndt von Staa

emagno@inf.puc-rio.br, arndt@inf.puc-rio.br

Abstract: This paper aims at assessing, identifying weaknesses and proposing enhancements to the quality model used by Sant'Anna to compare properties of object-oriented and aspect-oriented software systems. The assessment includes an empirical study and a qualitative analysis of the metrics used by Sant'Anna's in his model. We present a revised quality model attempting at reducing the weaknesses of the original one. The new model contains a minimal set of metrics, some of which complement the original model. All metrics are defined by means of a process that could be automated.

Keywords: Quality Model, Software Measurement, Aspect Oriented, Object Oriented.

Resumo: Neste artigo o nosso principal objetivo é avaliar, identificar as fraquezas e propor revisões ao modelo de qualidade utilizado por Sant'Anna em estudos que comparam sistemas orientados a objetos com sistemas orientados a aspectos. A avaliação inclui um estudo empírico e uma análise qualitativa feita sobre o conjunto de métricas que compõe este modelo de qualidade. A seguir apresentamos um modelo de qualidade revisado que procura eliminar os pontos fracos identificados no modelo original. Este novo modelo contém um conjunto mínimo de métricas, algumas das quais adicionais ao modelo original. As métricas do modelo revisado são definidas de modo que possam vir a ser automatizadas.

Palavras-chave: Modelo de Qualidade, Medição de Software, Orientação a Aspectos, Orientação a Objetos.

Responsável por publicações:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br

Sumário

1 Introdução 1

2 Qualidade de Software 1

2.1 Primeiro Modelo de Qualidade de Software 2

2.2 Métricas de Software 3

2.2.1 Medidas de Separação de Assuntos 5

3 Estudo de Caso: Medições envolvendo Padrões de Projetos 7

3.1 Resultados de Separação de Assuntos 8

3.2 Resultados de Acoplamento, Coesão e Tamanho 9

3.3 Discussão sobre os Resultados 10

4 Estrutura de Decomposição de Sistemas 11

4.1 Decomposição de Sistemas Java 12

4.2 Decomposição de Sistemas AspectJ 13

5 Avaliação do Modelo de Qualidade 14

5.1 Avaliação Qualitativa do Conjunto de Métricas 14

5.2 Novas Métricas Propostas no Modelo 17

5.2.1 Número de Filhos (NOC) 17

5.2.2 Número de Comandos (NOS) 18

5.2.3 Número de Operações (NOO) 19

5.2.4 Fator de Difusão do Assunto sobre Componentes (FCDC) 19

5.2.5 Fator de Difusão do Assunto sobre Operações (FCDO) 20

5.3 Modelo de Qualidade Adaptado 21

6 Conclusões 22

Referências Bibliográficas 22

Nomes das métricas utilizadas

Relacionamos a seguir os acrônimos das métricas utilizadas neste trabalho, e seus respectivos nomes em inglês e português.

CBC (*Coupling Between Components*): Acoplamento entre Componentes

CDC (*Concern Diffusion over Components*): Difusão do Assunto por Componentes

CDLOC (*Concern Diffusions over Lines of Code*): Difusão do Assunto por Linhas de Código

CDO (*Concern Diffusion over Operations*): Difusão do Assunto por Operações

DIT (*Depth Inheritance Tree*): Profundidade da Árvore de Herança

FCDC (*Factor of Concern Diffusion over Components*): Fator de Difusão do Assunto sobre Componentes

FCDO (*Factor of Concern Diffusion over Operations*): Fator de Difusão do Assunto sobre Operações

LCOO (*Lack of Cohesion in Operations*): Perda de Coesão em Operações

LOC (*Lines of Code*): Número de Linhas de Código

NOA (*Number of Attributes*): Número de Atributos

NOC (*Number of Children*): Número de Filhos

NOO (*Number of Operations*): Número de Operações

NOS (*Number of Statements*): Número de Comandos

VS (*Vocabulary Size*): Tamanho do Vocabulário

WOC (*Weighted Operations per Component*): Peso das Operações por Componentes

1 Introdução

Este documento tem como principal objetivo avaliar o modelo de qualidade proposto por Sant´Anna [Sant´Anna, 2004] utilizado em uma série de estudos [Garcia et al., 2004] [Garcia et al., 2005] [Garcia, 2004] [Sant´Anna et al, 2004] para comparar sistemas orientados a objetos com sistemas orientados a aspectos. A avaliação deste modelo divide-se em duas partes principais: um estudo empírico que procura identificar pontos do modelo que possam ser melhorados e uma análise qualitativa do conjunto de métricas. No estudo empírico as medições propostas no modelo de qualidade são utilizadas para comparar implementações Java [Java, 2004] [Deitel e Deitel, 2000] e AspectJ [AspectJ, 2004] de padrões de projeto *GoF* [Gamma et al., 1995]. A partir dos resultados obtidos com este estudo, são feitas análises qualitativas sobre o conjunto de métricas que compõe este modelo de qualidade utilizando uma estrutura de decomposição [Staa, 2000]. Medidas de software alternativas são propostas para preencher as lacunas identificadas no conjunto original de métricas do modelo.

Ao final deste trabalho, é argumentada a necessidade de um modelo de qualidade mais adequado para avaliar programas orientados a objetos e orientados a aspectos. Além de avaliar a qualidade, este modelo deve também efetuar a comparação quantitativa entre softwares, mesmo que estes sejam implementados utilizando diferentes paradigmas e linguagens de programação. As métricas para compor o modelo devem ter definições precisas, sem ambigüidades e devem ser ortogonais. O conjunto de métricas deve permitir fácil automatização por ferramentas de suporte a medições.

O restante deste documento está organizado da seguinte forma. Na seção 2 é feita uma breve revisão dos principais conceitos que envolvem a qualidade de software e é apresentado um primeiro modelo de qualidade que é trabalhado no restante do documento. Nessa seção também são discutidas métricas de software, especialmente as métricas orientadas a objetos e as orientadas a aspectos. Na seção 3 são apresentados os resultados de um estudo empírico envolvendo implementações de padrões de projetos. Estruturas de decomposição de sistemas Java e AspectJ são tratadas na seção 4 , sendo ilustrado como as métricas se distribuem sobre estas estruturas. Na seção 5 é feita uma avaliação das métricas que compõem o modelo de qualidade original, são propostas novas métricas e é apresentado um modelo revisado. Finalmente, na seção 6 são colocadas as conclusões do trabalho e propostas direções para trabalhos futuros.

2 Qualidade de Software

A definição de qualidade de software não é simples, pois o termo qualidade é bastante subjetivo e também porque software é intangível. Qualidade, de uma maneira geral, pode ser definida como "atendimento aos requisitos" ou "adequado para uso". Qualidade para o produto software requer uma definição mais específica do que as definições tradicionais. Assim, alguns autores como Stephen Kan [Kan, 2003] relacionam qualidade de software à ausência de *bugs*, podendo ser medida pela taxa de erros (erros por milhares de linhas de código) ou pela confiabilidade (horas de operação sem falhas). Em [Staa, 2000], qualidade de um artefato de software é um conjunto de propriedades a serem satisfeitas em determinado grau, de modo que este satisfaça as necessidades de seus usuários e clientes. Uma outra definição é

apresentada em [Pressman, 1995] que coloca qualidade de software como "conformidade a requisitos funcionais e de desempenho explicitamente declarados, a padrões de desenvolvimento claramente documentados e a características implícitas que são esperadas de todo software profissionalmente desenvolvido".

Pelas duas últimas definições, [Staa, 2000] e [Pressman, 1995], podemos destacar um ponto importante. Há um conjunto de propriedades, ou requisitos implícitos, que freqüentemente não é mencionado (por exemplo, desejo de dispor-se de elevada manutenibilidade). Se o software se adequar aos seus requisitos explícitos, mas deixar de cumprir seus requisitos implícitos, a qualidade do sistema será suspeita [Pressman, 1995]. Desta forma várias características implícitas devem estar presentes em um software para que este seja considerado um produto de qualidade. Algumas destas características que podem ser mencionadas [Kan, 2003] [Peters e Pedrycz, 2001] [Pressman, 1995] são: manutenibilidade, flexibilidade, testabilidade, reusabilidade, concisão, modularidade, simplicidade (ou facilidade de compreensão [Garcia, 2004] [Sant´Anna, 2004]).

2.1 Primeiro Modelo de Qualidade de Software

Nesta seção é apresentado um modelo de qualidade [Garcia, 2004] [Sant´Anna, 2004] que mapeia características implícitas de qualidade em atributos mensuráveis a partir do código fonte. Este modelo tem como finalidade avaliar atributos implícitos como manutenibilidade, reusabilidade, simplicidade e flexibilidade. Ele é construído em forma de árvore composta por quatro níveis como ilustrado na Figura 1. Este modelo de qualidade é fundamental para o restante do documento, sendo utilizado no estudo de caso da seção 3 e avaliado qualitativamente na seção 5 .

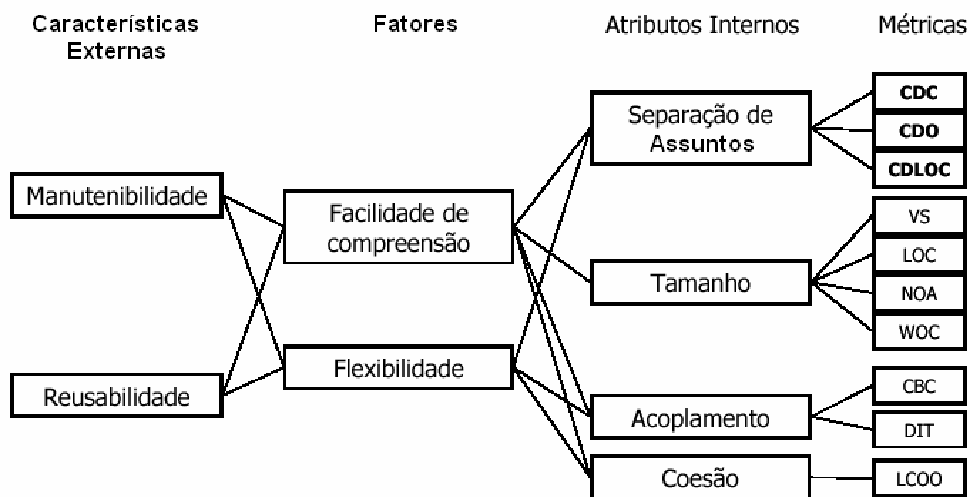

Figura 1: **Modelo de qualidade original.**

Nos dois primeiros níveis do modelo de qualidade apresentado na Figura 1 são colocadas características implícitas de qualidade. A manutenibilidade e a reusabilidade, presentes no primeiro nível, são denominadas características externas. Simplicidade e Flexibilidade, no segundo nível, são denominadas fatores. Esta distinção é feita porque, segundo os autores [Garcia, 2004] [Sant´Anna, 2004], simplicidade e flexibilidade influenciam as duas características de qualidade do nível anterior. Além disso, o modelo de qualidade é originalmente proposto para avaliar a manutenibilidade e a reusabilidade de software orientado a aspectos e orientado a objetos.

No terceiro nível do modelo de qualidade são colocados quatro atributos internos que se baseiam em princípios bem estabelecidos da engenharia de software. Separação de assuntos, tamanho, acoplamento e coesão são os atributos considerados neste modelo. O termo separação de assuntos é utilizado como modularidade do código referente a um determinado assunto ou característica, sendo melhor discutido na seção 2.2.1 . Os atributos internos se conectam aos fatores do nível anterior e a um conjunto de métricas do nível seguinte.

No nível mais baixo da árvore estão as métricas a serem aplicadas ao sistema com o intuito de avaliar seus atributos internos. De acordo com o atributo que se propõe a medir, as métricas se dividem em quatro categorias: separação de assuntos, tamanho, acoplamento e coesão. A separação de assunto é medida por ***Difusão do Assunto por Componentes*** (CDC), ***Difusão do Assunto por Operações*** (CDO) e ***Difusão do Assunto por Linhas de Código*** (CDLOC). O tamanho do sistema é medido em função do ***Tamanho do Vocabulário*** (VS), ***Número de Linhas de Código*** (LOC), ***Número de Atributos*** (NOA) e ***Peso das Operações por Componentes*** (WOC). O grau de acoplamento entre os componentes do sistema é medido por ***Acoplamento entre Componentes*** (CBC) e ***Profundidade da Árvore de Herança*** (DIT). Finalmente, a coesão de cada módulo do sistema é medida pela ***Perda de Coesão em Operações*** (LCOO). As medidas de software, incluindo as presentes neste modelo de qualidade, são abordadas na próxima subseção.

2.2 Métricas de Software

O desenvolvimento de grandes sistemas é uma atividade que consome muito tempo e recursos. Sabendo-se que cada etapa deste processo requer um esforço, é preciso prover informações para que a equipe tome as decisões, trace os planos, agende as atividades e aloque os recursos. Desta forma as métricas de software tornam-se necessárias para identificar onde são necessárias as pesquisas de melhoria do processo de desenvolvimento, sendo ainda uma fonte crucial para a tomada de decisões [Kan, 2003]. É sabido que diversas empresas têm desenvolvido seus modelos para prever custo, qualidade e pesquisa baseada em métricas de software.

Muitas métricas têm sido propostas na literatura [Chidamber e Kemerer, 1994] [Harrison et al., 1998] [Henderson-Sellers, 1996] [Lorenz e Kidd, 1994] [Sant´Anna, 2004]. As métricas consideradas mais relevantes são referentes ao código fonte porque esta é a forma mais confiável de informação. Métricas de código fonte devem ser fáceis de serem obtidas, pois com diferentes versões de um software, há bastante informação a analisar. Nesta seção são abordadas as principais métricas obtidas a partir do código fonte e que podem ser aplicadas a sistemas implementados sobre o paradigma de desenvolvimento orientado a objetos ou orientado a aspectos.

Algumas métricas são utilizadas para identificar a concisão, ou tamanho de um sistema, como por exemplo o número de subsistemas existentes (pacotes em Java ou AspectJ) ou o número de classes em cada subsistema. Note que o número de classes pode ser generalizado para o ***Tamanho do Vocabulário*** (VS) [Sant´Anna, 2004], o que inclui classes, interfaces e aspectos. Outra métrica de tamanho utiliza a contagem do número de membros [Kan, 2003] de um componente. Nesta contagem pode ser feita distinção de um tipo específico de membros, é o que ocorre nas métricas Número de Atributos (NOA) [Sant´Anna, 2004] e ***Peso das Operações por Componentes*** (WOC) [Sant´Anna, 2004].

O ***Número de Linhas de Código*** (LOC) [Kan, 2003] [Peters e Pedrycz, 2001] [Pressman, 1995] é uma das medidas mais simples que pode ser obtida de um software. Entretanto, vários fatores podem gerar uma diferença significativa no resultado de sua contagem. Na programação em *assembler*, em que cada linha de código corresponde a um comando, a definição desta métrica é clara. Com as linguagens de alto nível, a contagem de linhas de código pode ser influenciada, dentre outras coisas, pela consideração ou não de comentários, linhas em branco e declarações que não são executadas – como delimitadores. Portanto, dependendo da forma de contar, o número de linhas de código pode ser fortemente influenciado pelo estilo de programação adotado. Existem padrões para contagem de linhas de código que diminuem a influência do estilo do programador, entretanto, estes padrões nem sempre são implementados pelas ferramentas. Uma métrica menos influenciada pelo estilo de programação e que pode ser utilizada como alternativa a LOC é a contagem do número de comandos (*statements*) [Kan, 2003] a ser apresentada na seção 5.2.2 .

Métricas de acoplamento indicam o número de componentes que se relacionam a um determinado componente. A forma mais genérica de se medir o acoplamento é feita pela métrica de ***Acoplamento entre Componentes*** (CBC) [Garcia, 2004] [Sant´Anna, 2004] estendida de ***Acoplamento entre Objetos*** (CBO) [Chidamber e Kemerer, 1994]. Em CBC são considerados todos os tipos de relacionamentos entre classes e aspectos como referências por atributos, operações, herança, *pointcuts* e *inter-type declaration*. Um tipo especial de acoplamento que ocorre através da hierarquia pode ser medido pela ***Profundidade da Árvore de Herança*** (DIT) [Chidamber e Kemerer, 1994] [Sant´Anna, 2004] ou pelo ***Número de Filhos*** (NOC) [Chidamber e Kemerer, 1994]. DIT mede o número de níveis da hierarquia até que se atinja a raiz da árvore, por exemplo, na Figura 2 a classe *Hashtable* tem nível 2 e a classe *Dictionary* tem nível 1 em relação a raiz da árvore de herança, classe *Object* em Java. O ***Número de Filhos*** conta subtipos imediatos de um componente, e pela Figura 2, o número de filhos conhecidos da classe *Dictionary* é 1.

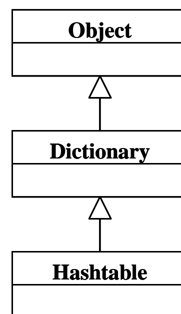

Figura 2: **Hierarquia de classes de *Hashtable*.**

A coesão de um componente mede quão interligadas estão as operações em relação ao compartilhamento de atributos [Kan, 2003]. A ***Perda de Coesão em Operações*** (LCOO) [Chidamber e Kemerer, 1994] [Sant´Anna, 2004] pode ser medida pelo contagem do número de pares de operações que compartilham atributos, menos o número de pares de operações que não compartilham nenhum atributo. A medida de perda de coesão em métodos é ilustrada através do código abaixo. Na classe do Exemplo 2.1 há dois pares de métodos que não acessam nenhum atributo em comum (`metA`, `metB`) e (`metA`, `metC`), enquanto exatamente um par de métodos (`metB`, `metC`) compartilha o atributo `v3`. Portanto, a medida de perda de coesão da classe `MyClass` é 1 (2 – 1 = 1)

```
class MyClass {
 int v1, v2, v3, v4;
 void metA() { . . .  usa v1, v2  . . . }
 void metB() { . . .  usa v3  . . . }
 void metC() { . . .  usa v3, v4  . . . }
}
```

Exemplo 2.1: **Classe ilustrativa de perda de coesão.**

Na comparação de sistemas orientados a aspectos com sistemas orientados a objetos é interessante avaliar a modularidade do código relacionado aos assuntos. A seguir são apresentadas algumas medidas definidas para quantificar o grau de espalhamento de um determinado assunto através do código. Estas medidas têm como característica a possibilidade de serem aplicadas tanto em sistemas orientados a aspectos como em sistemas orientados a objetos. Isto viabiliza a comparação dos resultados obtidos a partir de sistemas implementados segundo os dois tipos de paradigmas de desenvolvimento.

2.2.1 Medidas de Separação de Assuntos

A motivação para o desenvolvimento orientado a aspectos é modularizar assuntos que não são bem capturados por outras metodologias. Desta forma, são propostas [Lopes, 1997] três métricas que indicam o nível de espalhamento de determinado assunto através do código. A primeira mede o espalhamento através de componentes, a segunda o espalhamento por operações e a terceira métrica procura identificar o espalhamento por linhas de código. Como foram definidas, estas três métricas podem ser aplicadas tanto em programas orientados a aspectos como em programas orientados a objetos.

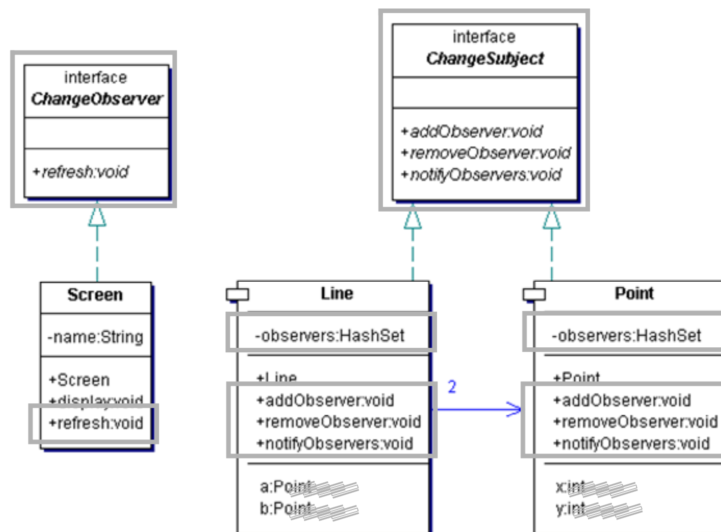

Figura 3: **Diagrama de classes orientado a objetos de um editor de figuras.**

A métrica ***Difusão do Assunto por Componentes*** (CDC) [Lopes, 1997] [Garcia, 2004] [Sant´Anna, 2004] conta o número de componentes cujo propósito principal é contribuir para a implementação do assunto avaliado. Além disso, CDC conta também o número de componentes que fazem referência aos componentes principais do assunto. Para melhor entender esta métrica, utilizamos o diagrama de classe da Figura 3 que apresenta a implementação orientada a objetos de um editor de figuras. Os retângulos cinzas destacam o código utilizado para implementar o padrão de projeto *Observer* [Gamma et al., 1995] nos componentes do sistema. Destacado por retângulos estão as duas interfaces `ChangeObserver` e `ChangeSubject`; o atributo `observers`

e os métodos `addObserver`, `removeObserver` e `notifyObservers` das classes `Line` e `Point`; e o método `refresh` da classe `Screen`. Os métodos do diagrama que apresentem uma mancha (rabiscado) fazem referências a algum código cujo propósito principal é contribuir para a implementação do padrão *Observer*, ou seja, se encontra dentro de algum retângulo. Os métodos rabiscado são `setA` e `setB` da classe `Line`, e `setX` e `setY` da classe `Point`. Supondo que se queira verificar o espalhamento do assunto referente ao padrão *Observer* neste sistema, a medida de ***Difusão do Assunto por Componentes*** irá indicar valor cinco, uma vez que todas as classes e interfaces do diagrama possuem algum código referente a este assunto.

Difusão do Assunto por Operações (CDO) [Lopes, 1997] [Garcia, 2004] [Sant´Anna, 2004] conta o número de operações cujo propósito principal é contribuir par a implementação do assunto avaliado. CDO conta também os métodos, construtores e *advices* que acessam alguma das operações principais do assunto. A contagem do espalhamento do assunto referente ao padrão *Observer* sobre as operações no sistema da Figura 3 resulta em 15. As quinze operações que possuem algum código para implementação do padrão *Observer* são: `refresh` na interface `ChangeObserver` e na classe `Screen`; `addObserver`, `removeObserver` e `notifyObservers` na interface `ChangeSubject` e nas classes `Line` e `Point`; `setA` e `setB` na classe `Line`; e `setX` e `setY` na classe `Point`.

```
public class Line
                  implements ChangeSubject {
   private HashSet observers;
   private Point a, b;

   public Line(Point x, Point y) {
      this.a = x;
      this.b = y;
      this.observers = new HashSet();
   }

   public Point getA() { return a; }
   public Point getB() { return b; }

   public void setA(Point x) {
      this.a = x;
      notifyObservers();
   }

   public void setB(Point y) {
      this.b = y;
      notifyObservers();
   }

   public void addObserver(ChangeObserver o) {
      this.observers.add(o);
   }

   public void removeObserver(ChangeObserver o) {
      this.observers.remove(o);
   }

   public void notifyObservers() {
      for (Iterator e = observers.iterator(); e.hasNext(); )
         ((ChangeObserver)e.next()).refresh(this);
   }
}
```

Código 1: **Sombreamento da Classe `Line` do editor de figuras.**

A terceira métrica apresentada nesta seção é ***Difusão do Assunto por Linhas de Código*** (CDLOC) [Lopes, 1997] [Garcia, 2004] [Sant´Anna, 2004] que conta o número de pontos de transição existentes no código entre o assunto avaliado e os demais assuntos do sistema. O uso desta métrica requer sombreamento do código que o divide em áreas

sombreadas e não sombreadas. Um exemplo de sombreamento é apresentado na classe `Line` do Código 1, onde á área sombreada destaca o código necessário para implementar o padrão *Observer* nesta classe. Os pontos de transição são os pontos em que ocorre a mudança entre áreas sombreadas e não-sombreadas ou vice-versa. No caso da classe `Line` o número de pontos de transição é 10, ou seja, a contagem de CDLOC para esta classe resulta em dez.

3 Estudo de Caso: Medições envolvendo Padrões de Projetos

Estudos recentes [Garcia et al., 2004] [Garcia et al., 2005] [Hannemann e Kiczales, 2002] mostraram que implementações orientadas a objetos de padrões de projeto envolvem assuntos espalhados. Desta forma, é importante verificar se abordagens orientadas a aspectos permitem uma melhor modularização destes assuntos. Nesta seção é apresentado um estudo quantitativo que compara implementações orientadas a objetos e orientadas a aspectos dos 23 padrões de projeto descritos pela *Gang of Four* (GoF) [Gamma et al., 1995]. Este estudo utiliza atributos rigorosos da engenharia de software como acoplamento, coesão, tamanho e separação de assuntos em seu critério de avaliação.

O estudo de caso explorado nesta seção complementa o trabalho de Hannemann e Kiczales [Hannemann e Kiczales, 2002]. Estes autores desenvolveram em [Hannemann e Kiczales, 2002] um estudo em que implementam os 23 padrões de projeto GoF em Java [Java, 2004] e AspectJ [AspectJ, 2004] e então comparam suas implementações. Para cada um dos padrões, Hannemann e Kiczales desenvolveram um exemplo representativo que utiliza o padrão e o implementa de forma orientada a objetos em Java e orientada a aspectos em AspectJ. Entretanto, eles fazem apenas comparações qualitativas entre suas implementações e estas comparações são baseadas em atributos da engenharia de software que não são bem conhecidos, tais como, "componibilidade"[1] e "plugabilidade"[2].

Para efetuar um estudo quantitativo que complementasse o trabalho de Hannemann e Kiczales [Hannemann e Kiczales, 2002], o conjunto de métricas definido no modelo de qualidade da seção 2.2 é aplicado sobre as implementações Java e AspectJ dos padrões. Com o objetivo de permitir que as distintas implementações de um mesmo padrão fossem comparáveis foram feitas pequenas alterações no código original. Dois exemplos destas alterações são a garantia de um mesmo estilo de programação e a adição (ou remoção) de funcionalidades que ocorre em uma das implementações e não ocorre em outra. Esta etapa de garantir implementações comparáveis é chamada de alinhamento do código [Garcia et al., 2005].

Após o alinhamento do código, as implementações Java e AspectJ são alteradas com a inclusão de novos componentes (Classes e Aspectos) desempenhando os mesmos papéis no padrão. Estas alterações são necessárias porque os exemplos de Hannemann e Kiczales possuem poucos componentes desempenhando cada papel, na maioria dos casos apenas um. Note que em implementações pequenas, não é possível efetivamente investigar assuntos espalhados sobre a estrutura do padrão. Semelhante ao estudo de Hannemann e Kiczales, cada papel do padrão é tratado como um assunto, visto que, os papéis são a principal fonte de espalhamento de código. A Tabela 1 [Garcia et al., 2005] apresenta os 23 padrões GoF, seus papéis e o número de componentes desempenhando

---

1 De componível, que se pode compor.

2 Do verbo plugar (inglês *plug*), ligar.

cada papel introduzidos por padrão. Esta etapa de inclusão de novos componentes para ampliar o tamanho das implementações é chamada cenário de evolução [Garcia et al., 2005].

Padrão	Papéis	Componentes Adicionados
Abstract Factory	Factory e Product	4 Factories
Adapter	Target, Adapter e Adaptee	4 métodos adaptados
Bridge	Abstraction e Implementor	2 Abstractions e 2 Implementors
Builder	Builder e Director	4 Builders
Chain Of Responsibility	Handler	4 Handlers
Command	Command, Commanding e Receiver	4 Commands e 2 Invokers
Composite	Component, Composite e Leaf	2 Composites e 2 Leafs
Decorator	Component e Decorator	4 Decorators
Façade	Façade	Nada
Factory Method	Product e Creator	4 Creators
Flyweight	Flyweight Factory e Flyweight	4 Flyweights
Interpreter	Context e Expression	4 Expressions
Iterator	Iterator e Aggregate	2 Iterators e 2 Aggregates
Mediator	Mediator e Colleague	4 Mediators e 4 Colleagues
Memento	Memento e Originator	2 Mementos e 2 Originators
Observer	Subject e Observer	4 Observers e 4 Subjects
Prototype	Prototype	4 Prototypes
Proxy	Proxy e Real Subject	4 Proxies e 2 Real Subjects
Singleton	Singleton	4 Singletons e 4 subclasses não Singleton
State	State e Context	4 States
Strategy	Strategy e Context	4 Strategies e 4 Contexts
Template Method	Abstract Class e Concrete Class	4 Concrete Classes
Visitor	Visitor e Element	4 Elements e 2 Visitors

Tabela 1: **Padrões e componentes incluídos no cenário de evolução.**

As medições são feitas nas implementações dos padrões antes e após o cenário de evolução e os resultados serão apresentados nas próximas subseções. Na subseção 3.1 o objetivo é verificar o espalhamento de código nas distintas implementações e para isso são usadas as medidas de separação de assuntos (seção 2.2.1). Na subseção 3.2 são feitas as avaliações com respeito a acoplamento, coesão e tamanho das implementações dos padrões. Uma análise do resultados comparando o espalhamento dos assuntos com características de acoplamento, coesão e tamanho é feita na seção 3.3 .

3.1 Resultados de Separação de Assuntos

Em relação às medidas de separação de assuntos, cada papel no padrão é considerado como um assunto e neles são aplicadas medições para verificar seu espalhamento sobre o código. As três medidas ***Difusão do Assunto por Componentes*** (CDC), ***Difusão do Assunto por Operações*** (CDO) e ***Difusão do Assunto por Linhas de Código*** (CDLOC) (seção 2.2.1) são utilizadas para avaliar o espalhamento do código de um determinado papel. Pelos resultados obtidos no processo de medição, os 23 padrões de projeto *GoF* são classificados em três grupos [Garcia et al., 2005]. No primeiro grupo são colocados os padrões em que as implementações orientadas a aspectos modularizam melhor seus papéis. No segundo grupo estão os padrões cujas implementações orientadas a objetos são as que melhor modularizam o código dos papéis. Os padrões em que nenhuma das implementações apresenta vantagem relevante quanto a modularização são colocados no terceiro grupo.

Os padrões que melhor modularizam o código relativo aos papéis dos padrões nas implementações orientadas a aspectos são *Decorator*, *Adapter*, *Prototype*, *Visitor*, *Proxy*, *Singleton*, *Mediator*, *Composite*, *Observer*, *Command*, *Iterator*, *Chain of Responsibility*, *Strategy* e *Memento*. Os quatorze padrões deste primeiro grupo estão listados em ordem decrescente em relação à separação dos assuntos verificada pelas medições, ou seja, o padrão que apresenta o melhor resultado é o *Decorator*.

O segundo grupo inclui seis padrões de projeto em que as implementações orientadas a aspectos mostram pior modularidade em relação à separação dos assuntos dos papéis. Neste grupo estão os padrões *Template Method*, *Abstract Factory*, *Factory Method*, *Bridge*, *Builder* e *Flyweight* listados em ordem decrescente sobre o grau de modularidade da implementação `AspectJ`. Apesar de alguns valores medidos apresentarem resultados semelhantes entre as versões Java e `AspectJ`, a maioria destes valores é melhor na implementação orientada a objetos. Isto ocorre porque os papéis destes seis padrões já são bem modularizados nas implementações orientadas a objetos de tal forma que as soluções orientadas a aspectos não conseguem melhores resultados.

O terceiro grupo possui padrões em que as distintas implementações não influenciam a modularidade dos códigos relativos aos papéis dos padrões. Este grupo inclui os três padrões *Façade*, *Interpreter* e *State*. Quanto aos padrões *Interpreter* e *State*, há uma pequena diferença entre as implementações, entretanto irrelevante por ser menor que 5%. O padrão *Façade* não apresenta nenhuma diferença porque as implementações Java e `AspectJ` são idênticas.

3.2 Resultados de Acoplamento, Coesão e Tamanho

Diferente da subseção anterior na qual a classificação é feita em relação a separação de assuntos, nesta seção são apresentados os resultados obtidos pelas medições feitas nos padrões de projeto *GoF* com respeito a acoplamento, coesão e tamanho. Os 23 padrões são classificados em cinco grupos [Garcia et al., 2005] de acordo com a semelhança dos resultados das medições. Nos grupos 1 e 2 aparecem os padrões que possuem melhores resultados na implementação orientada a aspectos, entretanto, os padrões do grupo 2 possuem exceções para algumas métricas. Nos grupos 3 e 4 estão os padrões com melhores resultados na implementação orientada a objetos, o grupo 3 apresenta exceções para algumas métricas. Os padrões classificados no grupo 5 não demonstram vantagem em favor de uma ou outra implementação.

O primeiro grupo inclui os padrões *Composite*, *Observer*, *Adapter*, *Mediator* e *Visitor* que apresentam significativas melhorias com respeito a atributos de acoplamento, coesão e tamanho nas implementações orientadas a aspectos. As melhorias apresentadas pelos padrões deste grupo estão diretamente relacionadas aos ganhos de modularidade vistos na seção 3.1 . Ou seja, as medidas de acoplamento, coesão e tamanho tiveram melhores resultados devido à melhor separação dos assuntos nas implementações orientadas a aspectos.

As implementações `AspectJ` dos padrões do segundo grupo apresentam melhores resultados para a maioria das métricas de acoplamento, coesão e tamanho, exceto em uma. Neste grupo estão os padrões *Decorator*, *Proxy*, *Singleton* e *State*. As implementações orientadas a aspectos dos três primeiros padrões deste grupo mostram melhorias em todos os valores medidos, exceto para métrica ***Acoplamento entre Componentes***. Por outro lado, o padrão *State* não mostra melhoria de resultado no ***Número de Atributos***. Como no primeiro grupo, a melhor separação dos assuntos

(identificada na seção 3.1) conduz a melhores resultados de acoplamento, coesão e tamanho.

Os padrões *Chain of Responsibility*, *Command*, *Prototype* e *Strategy* são classificados no terceiro grupo por apresentarem melhores resultados na solução orientada a aspectos para no máximo duas métricas. Em geral, as implementações orientadas a objetos são melhores no que diz respeito a acoplamento, coesão e tamanho. As implementações orientadas a aspectos dos padrões *Chain of Responsibility*, *Command* e *Strategy* vencem apenas no ***Número de Atributos***. A versão do padrão *Prototype* orientada a aspectos possui um valor menor apenas para ***Peso das Operações por Componentes***.

No quarto grupo encontram-se os padrões cujas implementações orientadas a aspectos conduzem a resultado pior em termos de acoplamento, coesão e tamanho. Oito padrões estão classificados neste grupo: *Template Method*, *Abstract Factory*, *Bridge*, *Interpreter*, *Factory Method*, *Builder*, *Memento* e *Flyweight*. Como mencionado na seção 3.1 , os seis primeiros padrões deste grupo já são bem modularizados nas implementações orientadas a objetos e este fato tem impacto direto nas medidas de acoplamento, coesão e tamanho. Os padrões *Memento* e *Flyweight* são os que mostram priores resultados para estas métricas nas implementações `AspectJ`. A remoção do código relacionado ao padrão *Memento* para o aspecto torna esta solução mais complexa, e, no caso do *Flyweight* o código adicional piora a solução orientada a aspectos como mencionado na seção 3.1 .

Finalmente, os padrões do quinto grupo, *Iterator* e *Façade*, não demonstram vantagem em favor de uma ou outra implementação. Como mencionado na seção 3.1 , as implementações orientada a objetos e orientada a aspectos do padrão *Façade* são as mesmas. Na implementação orientada a aspectos do padrão *Iterator*, os métodos que criam o iterador são movidos para o aspecto. Apesar desta solução melhorar a modularidade do código relativo ao padrão, ela não tem impacto significativo em termos de acoplamento, coesão e tamanho.

3.3 Discussão sobre os Resultados

Esta seção apresenta uma análise comparativa entre os resultados de separação de assuntos, acoplamento, coesão e tamanho discutidos nas seções 3.1 e 3.2 . As implementações `AspectJ` mostram melhores resultados em termos de separação de assuntos para 14 padrões enquanto que as implementações Java apresentam melhores resultados para apenas 6 padrões. Como apresentado na seção 3.1 , três padrões de projeto apresentam resultados semelhantes em termos de modularidade do código dos papéis para ambas implementações.

A Tabela 2 mostra um comparativo entre todos os padrões e os resultados das medições. Os campos identificados com OO indicam que a valor medido foi melhor para a implementação Java, e a identificação OA é dada quando a implementação `AspectJ` é melhor para a referida métrica. Quando o valor medido não apresenta vantagem para uma ou outra implementação (ou a diferença é desprezível), o campo é marcado com um traço "–". Com base nesta tabela e nos resultados das seções 3.1 e 3.2 pode-se verificar que o uso de aspectos leva a melhores resultados de acoplamento e coesão em padrões com elevada interação entre os papéis. Os padrões *Mediator*, *Observer*, *State*, *Composite* e *Visitor* são exemplos deste tipo de padrão.

Padrão	CDC	CDO	CDLOC	CBC	DIT	LCOO	LOC	NOA	WOC
Abstract Factory	*OO*	–	–	*OO*	–	–	*OO*	–	*OO*
Adapter	OA	OA	OA	–	OA	–	OA	OA	OA
Bridge	*OO*	*OO*	–	*OO*	–	–	*OO*	–	*OO*
Builder	*OO*	*OO*	–	*OO*	–	OA	*OO*	–	*OO*
Chain Of Resp.	OA	*OO*	OA	OA	–	*OO*	*OO*	OA	*OO*
Command	OA	OA	OA	*OO*	–	*OO*	*OO*	OA	*OO*
Composite	OA	OA	OA	OA	–	OA	OA	OA	OA
Decorator	OA	OA	OA	*OO*	OA	–	OA	OA	OA
Façade	Mesma implementação OO e OA.								
Factory Method	*OO*	–	–	*OO*	–	OA	*OO*	–	–
Flyweight	*OO*	*OO*	*OO*	*OO*	–	*OO*	*OO*	–	*OO*
Interpreter	–	–	*OO*	*OO*	*OO*	–	*OO*	–	–
Iterator	OA	–	OA	*OO*	–	–	OA	–	*OO*
Mediator	OA	OA	OA	OA	–	OA	OA	OA	OA
Memento	OA	–	OA	*OO*	*OO*	–	*OO*	–	–
Observer	OA	OA	OA	OA	–	OA	OA	OA	OA
Prototype	OA	OA	OA	*OO*	–	–	*OO*	–	OA
Proxy	–	OA	OA	*OO*	–	–	OA	OA	OA
Singleton	–	OA	OA	*OO*	–	OA	*OO*	OA	OA
State	*OO*	–	–	OA	–	OA	*OO*	*OO*	OA
Strategy	–	OA	–	*OO*	–	–	*OO*	OA	OA
Template Method	*OO*	–	–	*OO*	–	–	*OO*	–	–
Visitor	OA	OA	OA	OA	–	OA	OA	–	OA

Tabela 2: **Comparativo entre as implementações de todas as métricas.**

Ainda pela Tabela 2, é notado o considerável impacto do uso de aspectos em relação ao tamanho das implementações. Para dez padrões as implementações `AspectJ` levam a um número de atributos menor (NOA) do que as correspondentes implementações Java. Em relação à métrica WOC, para onze padrões as implementações `AspectJ` reduzem o número de operações (com respectivos parâmetros) em relação às implementações Java. Para maiores informações sobre o estudo quantitativo que compara padrões de projetos orientados a objetos e orientados a aspectos consulte as referências [Garcia et al., 2004] [Garcia et al., 2005].

4 Estrutura de Decomposição de Sistemas

Organizar um programa em termos dos módulos que o constituem é conhecido como arquitetura [Staa, 2000]. O processo de organizar um módulo em termos de suas funções e as funções em termos das suas estruturas de código é conhecido por projeto. Na realidade a arquitetura não deixa de ser uma forma de projeto utilizada para estabelecer a organização macroscópica do sistema. Neste texto, o termo arquitetura é utilizado com significado semelhante a projeto. Preferencialmente, o termo ***arquitetura*** é adotada para evitar a ambigüidade inerente à palavra ***projeto***.

Decomposição é uma técnica adotada para vencer barreiras de complexidade. Uma estrutura de decomposição é organizada em forma de grafo dirigido no qual cada patamar corresponde a um nível de abstração[Staa, 2000]. Uma vez completa, esta estrutura registra todas as decomposições, desde a idéia original até o detalhe mais elementar de sua implementação. Nesta seção são utilizadas estruturas de decomposição para representar organizações adotadas para sistemas implementados em Java (seção 4.1) e `AspectJ` (seção 4.2). Em ambos os casos, é verificada uma decomposição semelhante. No nível mais abstrato, ou seja, mais alto da estrutura observa-se o sistema como um todo. O sistema pode ser decomposto em uma estrutura

de sub-sistemas. Cada sub-sistema é composto por diversos programas ou módulos. Os módulos compostos por diversas classes, estas por funções e assim por diante.

Uma abstração raiz é um elemento complexo demais para que se possa dar uma solução direta. No caso da decomposição de um sistema, todos os vértices internos (não folhas) da árvore são considerados como abstração raiz. Em uma decomposição utilizando recursão, estamos na realidade particionando o problema da abstração raiz em uma parte a ser resolvida diretamente e em uma parte restante a ser resolvida através de recorrência [Staa, 2000]. A Figura 4 apresenta uma decomposição recursiva sobre o vértice "Classes/Interfaces" da estrutura de decomposição de um sistema Java.

4.1 Decomposição de Sistemas Java

A arquitetura de um sistema orientado a objetos implementado em Java segue a estrutura de decomposição apresentado na Figura 4. No nível mais abstrato da estrutura aparece o sistema Java como um todo. Um nível abaixo, estão localizados os elementos que representam pacotes existentes no sistema. Em um sistema implementado na linguagem Java é obrigatório a existência de pelo menos um pacote. Os pacotes podem ser decompostos em classes ou interfaces. Classes (e Interfaces) podem conter outras classes (ou Interfaces) internas (decomposição recursiva) e podem conter ainda métodos, construtores e variáveis. Os métodos e construtores são decompostos em blocos e estes em linhas de código. A decomposição pode ser continuada a partir das linhas de código, porém, não é relevante para este trabalho.

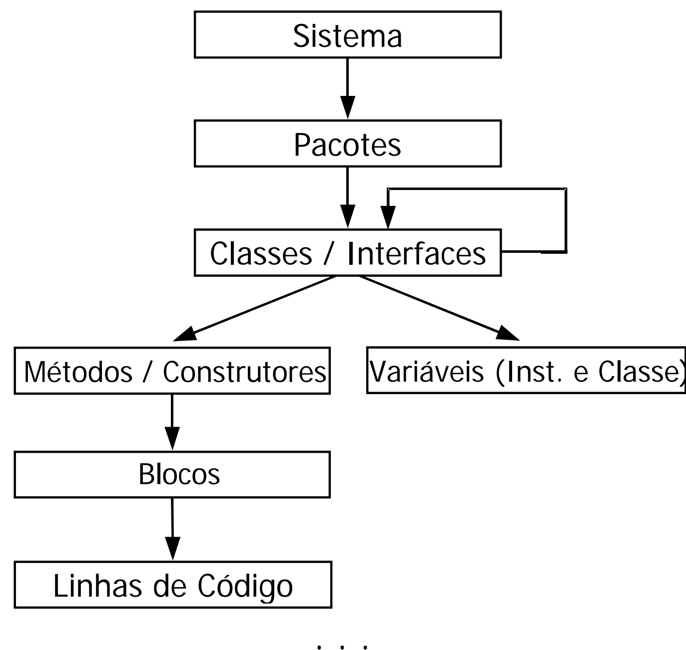

Figura 4: **Estrutura de decomposição de um sistema Java.**

As medições de um sistema implementado em Java, como apresentado no estudo de caso da seção 3 , são feitas sobre diferentes níveis de abstração de uma estrutura de decomposição (Figura 4). Como exemplo, a medição do número de classes de um sistema é feita sobre os três níveis mais abstratos da estrutura de decomposição: "Sistema", "Pacotes", "Classes/Interfaces". O ***Número de Atributos*** (NOA) é medido nos níveis de "Classes/Interfaces" e "Variáveis". E a medição do ***Número de Linhas de***

Código (LOC) é feita nos níveis mais baixos da estrutura de decomposição apresentada na Figura 4.

4.2 Decomposição de Sistemas AspectJ

Para um sistema orientado a aspetos implementado em `AspectJ`, a arquitetura da estrutura de decomposição deve ser genérica o suficiente para abranger tanto elementos da linguagem Java como da linguagem `AspectJ`. Note que a linguagem `AspectJ` estende Java com a adição de novos elementos. A estrutura de decomposição apresentada na Figura 5 ilustra a arquitetura de um sistema implementado em `AspectJ`. Esta estrutura é bastante semelhante ao da Figura 4, e também apresenta o sistema no nível mais abstrato da estrutura. Em ambas as estruturas de decomposição (Java e `AspectJ`), no segundo nível estão os pacotes do sistema. O nível de abstração seguinte é generalizado para "***Componentes***", pois em um software implementado em `AspectJ` pode haver aspectos, classes e interfaces neste nível. Da mesma forma a generalização para "***Operações***" e "***Atributos***" é feita para suportar estruturas de `AspectJ` como *advices*, *pointcuts* e *inter-type declarations*. Os dois níveis mais baixos da estrutura de sistemas `AspectJ` "***Blocos***" e "***Linhas de Código***" permanecem como na estrutura de decomposição de sistemas Java.

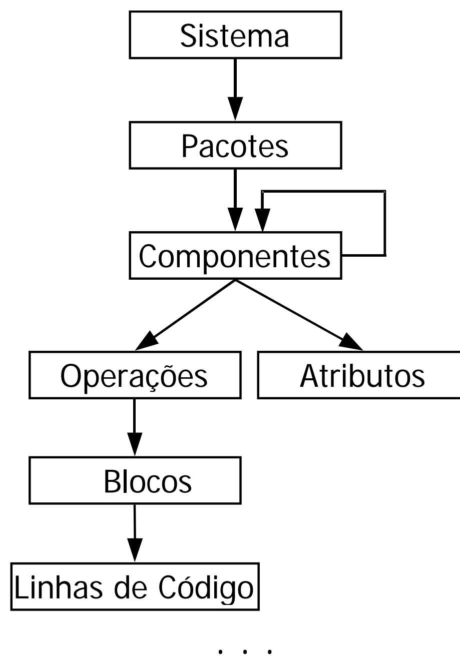

Figura 5: **Estrutura de decomposição de um sistema AspectJ.**

Ao observar as métricas da seção 2.2 é verificado que elas se distribuem sobre a estrutura de decomposição da Figura 5. As métricas ***Tamanho do Vocabulário*** e de ***Difusão do Assunto por Componentes***, por exemplo, são aplicadas nos três primeiros níveis da estrutura. Sobre os níveis de "***Componentes***", "***Operações***" e "***Atributos***" incidem outras seis métricas: ***Número de Atributos***, ***Peso das Operações por Componentes***, ***Acoplamento entre Componentes***, ***Profundidade da Árvore de Herança***, ***Perda de Coesão em Operações*** e ***Difusão do Assunto por Operações***. E nos dois níveis mais baixos da estrutura "***Linhas de Código***" e "***Blocos***" são aplicadas as medições de ***Número de Linha de Código*** e ***Difusão do Assunto por Linhas de Código***.

A característica que torna o processo de medição distribuído sobre a estrutura de decomposição é importante para que se faça uma avaliação da qualidade do conjunto de métricas. Mais adiante neste documento (seção 5.1), são utilizados critérios de avaliação da estrutura agregada para avaliar as métricas do modelo de qualidade da seção 2.1 e adotadas no estudo de caso da seção 3 . Estes critérios de avaliação são importantes para identificar características como necessidade e redundância de determinadas medições pois uma estrutura de decomposição permite que algumas medições sejam redundantes, ou seja, uma mesma característica pode estar sendo medida mais de uma vez.

5 Avaliação do Modelo de Qualidade

Nesta seção é feita uma avaliação qualitativa do modelo de qualidade proposto por Sant'Anna [Sant´Anna, 2004] e brevemente apresentado na seção 2.1 . Este modelo é utilizado para comparar implementações Java e `AspectJ` de padrões de projeto no estudo de caso da seção 3 . A partir deste estudo de caso foram identificados pontos do modelo propícios a serem melhorados, especialmente no que diz respeito ao conjunto de métricas. Não é objetivo deste trabalho argumentar sobre os dois primeiros níveis de abstração do modelo de qualidade apresentado na Figura 1 (seção 2.1). Entretanto, é evitada a subdivisão original entre características externas e fatores visto que ambos se tratam de características implícitas de qualidade [Kan, 2003] [Peters e Pedrycz, 2001] [Pressman, 1995] (seção 2).

5.1 Avaliação Qualitativa do Conjunto de Métricas

Como foi discutido na seção 4 , as medidas de software orientadas a aspectos e orientadas a objetos estão distribuídas sobre uma estrutura de decomposição. Todas as métricas que compõem o modelo de qualidade da Figura 1 (seção 2.1) são distribuídas sobre a estrutura de decomposição de sistemas Java (Figura 4) e `AspectJ` (Figura 5), conforme apresentado na seção 4 . Desta forma, nesta seção são feitas avaliações das métricas do modelo de forma individual segundo critérios de qualidade da estruturas de decomposição [Staa, 2000].

A avaliação das métricas ocorre com relação aos três critérios: ***definição***, ***necessidade*** e ***ortogonalidade***. O critério de definição procura verificar se está claramente definida a intenção do elemento, ou seja, se é definido sem ambigüidade a característica a ser medida. O segundo critério, necessidade, verifica se o elemento efetivamente contribui para resolver a abstração raiz, isto é, verifica a necessidade da métrica para o respectivo atributo interno do modelo. O terceiro critério de qualidade é a ortogonalidade que verifica se cada elemento do conjunto de solução resolve uma parte da abstração raiz que não é resolvida por qualquer outro elemento do conjunto. Caso fossem estritamente ortogonais, uma mesma característica não seria medida por mais de uma métrica.

A métrica ***Difusão do Assunto por Componentes*** (CDC) e ***Difusão do Assunto por Operações*** (CDO) possuem características semelhantes. Ambas têm como objetivo investigar o espalhamento de um determinado assunto sobre o código. A primeira métrica (CDC) conta o número de componentes afetados pelo assunto, sendo que componentes são classes, interfaces ou aspectos. E a segunda (CDO) conta o número de operações afetadas pelo assunto, neste caso operações são métodos, construtores e *advices*. Não há ambigüidade nas definições e ambas as métricas são necessárias para

investigar o espalhamento de código. Quanto à ortogonalidade, elas não são totalmente ortogonais visto que a mesma característica (espalhamento do assunto) é medida tanto por CDC quanto por CDO em um mesmo trecho código.

Difusão do Assunto por Linhas de Código (CDLOC) procura investigar o espalhamento de determinado assunto através do código do sistema. CDLOC conta o número de pontos de transição entre o assunto avaliado e os demais assuntos do sistema. A definição desta métrica é ambígua, pois códigos equivalentes podem apresentar valores distintos na contagem. Observe as duas versões do aspecto `CoordinateObserver` apresentadas nos Código 2 e Código 3 nos quais é sombreado o código referente ao papel *Observer* do padrão de mesmo nome. As duas versões são equivalentes, têm exatamente as mesmas funcionalidades e com implementações idênticas, variando apenas a ordem em que as funções foram colocadas no aspecto. Entretanto, ao se medir a espalhamento do assunto através do código (CDLOC) obtemos valor 2 para o Código 2 e valor 4 para o Código 3. Ou seja, uma pequena variação do estilo de programação pode comprometer o resultado da medição. Esta variação na contagem de CDLOC também pode ocorrer em outras situações, como por exemplo, internamente a uma operação quando a ordem dos comandos for diferente.

```
public aspect CoordinateObserver extends ObserverProtocol
{
   declare parents: Point implements Subject;

   pointcut subjectChange(Subject s): ( call(void
        Point.setX(int)) || call(void Point.setY(int)) ) &&
        target(s);

   declare parents: Screen implements Observer;

   protected void updateObserver(Subject s, Observer o)
   {
      ((Screen)o).display(
             "Screen updated: coordinates changed.");
   }
}
```

Código 2: **Aspecto CoordinateObserver com valor 2 para CDLOC.**

```
public aspect CoordinateObserver extends ObserverProtocol
{
   declare parents: Point implements Subject;
   declare parents: Screen implements Observer;

   pointcut subjectChange(Subject s): ( call(void
        Point.setX(int)) || call(void Point.setY(int)) ) &&
        target(s);

   protected void updateObserver(Subject s, Observer o)
   {
      ((Screen)o).display(
           "Screen updated: coordinates changed.");
   }
}
```

Código 3: **Aspecto CoordinateObserver com valor 4 para CDLOC.**

Ainda com respeito à métrica de ***Difusão do Assunto por Linhas de Código***, é questionável a necessidade desta métrica. Note que o espalhamento do código também é capturado pelas outras duas métricas CDC e CDO. O principal espalhamento apontado por CDLOC que não é medido pelas outras duas métricas é quando este ocorre internamente a uma operação, no nível de comandos. Pode-se afirmar que este

grau de granularidade na separação de assuntos não é importante por dois motivos: i) um bom projeto orientado a objetos (ou aspectos) não possui blocos com grande número de comandos; e ii) a ordem dos comandos pode interferir na contagem da métrica. Pelos problemas de definição e necessidade mencionados, ***Difusão do Assunto por Linhas de Código*** parece desnecessário ao modelo de qualidade como apresentado na seção 5.3 . Com respeito à ortogonalidade, esta métrica não é ortogonal às outras duas que medem separação de assuntos.

As métricas ***Acoplamento entre Componentes*** (CBC) e ***Profundidade da Árvore de Herança*** (DIT) têm por objetivo investigar o grau de acoplamento entre os componentes do sistema. A primeira conta qualquer tipo de referência que um componente faça a outro e a segunda mede o número de componentes acoplados através da herança. DIT mede também os acoplamentos indiretos que ocorrem na hierarquia. Estas métricas estão bem definidas e são necessárias para investigar os vários tipos de referências que caracterizam o acoplamento. Por medirem acoplamentos em comum, visto que CBC também conta referências de herança, estas duas métricas não são totalmente ortogonais.

A medida de coesão é bem difícil de ser obtida, por isso a literatura propõe uma série de métricas que contribuem para este objetivo [Hannemann e Kiczales, 2002] [Harrison et al., 1998] [Henderson-Sellers, 1996]. A métrica ***Perda de Coesão em Operações*** (LCOO) é bem definida, entretanto, não mostra bons resultados em relação à coesão semântica. Seus resultados adversos levam a questionar a necessidade desta métrica para contribuir com a medida de coesão. Uma alternativa é a substituição desta métrica por outra ou por um conjunto de métrica que apresentassem resultados mais satisfatórios. Como esta é a única medida de coesão existente no modelo de qualidade, LCOO é estritamente ortogonal em relação às outras métricas.

O ***Tamanho do Vocabulário*** (VS) é uma métrica para avaliar o tamanho do sistema em termos de número de classes, interfaces e aspectos. Além de medir o tamanho do sistema, VS também é útil para verificar o grau de espalhamento de um assunto sobre os componentes, como discutido mais adiante (seção 5.2.4). Esta métrica está bem definida e é necessária ao modelo de qualidade. Em relação à medição do tamanho do sistema, VS não é totalmente ortogonal às outras métricas que também medem este atributo.

Pode-se dizer que a contagem do ***Número de Linhas de Código*** (LOC) é ambígua, pois seu valor é comumente influenciado pelo estilo de programação. Existem padrões para contagem de linhas de código que diminuem a influência do estilo do programador, mas estes padrões não são implementados na ferramentas mais utilizadas. E pela utilização destas ferramentas, programas semelhantes podem apresentar valores diferentes para esta métrica. LOC tem como objetivo principal medir o tamanho das operações do sistema e uma alternativa à esta métrica é a contagem do número de comandos (seção 5.2.2) [Eclipse Metrics Plugin, 2005]. É necessário uma métrica que meça o tamanho das operações, entretanto, LOC não parece ser essencial e pode ser substituída por outra métrica com função semelhante. A contagem do ***Número de Linhas de Código*** também não é totalmente ortogonal às outras métricas de tamanho.

As outras duas métricas de tamanho definidas do modelo de qualidade são ***Número de Atributos*** (NOA) e ***Peso das Operações por Componentes*** (WOC). NOA conta o número de variáveis de instâncias, variáveis de classes e atributos introduzidos por *inter-type declaration* dos componentes. WOC conta o número de métodos, construtores e *advices* com seus respectivos parâmetros por componentes. Estas duas métricas estão bem definidas. As contagens de atributos e operações são necessárias ao modelo, entretanto,

é questionável a necessidade de se contar o número de parâmetros das operações. Não parece relevante ao tamanho do sistema o número de parâmetros que os métodos recebem. Desta forma, WOC pode ser substituída por uma métrica que permite apenas a contagem de operações (seção 5.2.3). ***Número de Atributos*** e ***Peso das Operações por Componentes*** não são totalmente ortogonais às outras métricas de tamanho, pois o tamanho do sistema pode ser medido pelo menos por três formas: LOC, VS ou (NOA+WOC).

	CDC	CDO	CDLOC	CBC	DIT	LCOO	VS	LOC	NOA	WOC
Definição	Sim	Sim	Não	Sim	Sim	+ou-	Sim	+ou-	Sim	Sim
Necessidade	Sim	Sim	+ou-	Sim	Sim	+ou-	Sim	+ou-	Sim	+ou-
Ortogonalidade	+ou-	+ou-	+ou-	+ou-	+ou-	Sim	+ou-	+ou-	+ou-	+ou-

Tabela 3: Avaliação individual das métricas do modelo de qualidade.

A Tabela 3 resume a análise qualitativa feita nesta seção em relação às métricas do modelo de qualidade definido na seção 2.1 . Nas colunas da tabela são apresentadas todas dez métricas avaliadas e nas linhas os três critérios de qualidade. Na próxima seção são apresentadas alternativas às métricas com problemas identificados nos critérios de definição e necessidade. Quanto à ortogonalidade, é observado a dificuldade em se conseguir um conjunto de métricas que sejam totalmente ortogonais.

5.2 Novas Métricas Propostas no Modelo

O estudo de caso utilizado para comparar implementações Java e `AspectJ` de padrões de projeto (seção 3) mostra pontos no modelo de qualidade adotado que podem ser melhorados. A avaliação qualitativa apresentada na subseção anterior também sugere melhorias a este modelo, especialmente no que diz respeito ao conjunto de métricas. Nesta seção são apresentadas cinco novas métricas para compor o modelo de qualidade, seja para melhor cobrir algum atributo que apresenta deficiências nas medições ou como substituição a alguma métrica existente.

Para cada uma das cinco métricas desta seção é apresentada uma definição e a justificativa de sua utilização no modelo de qualidade. As três primeiras métricas são conhecidas na literatura e utilizadas em metodologias tradicionais de desenvolvimentos de software (especialmente nas orientadas a objetos). Estas métricas de nomes ***Número de Filhos***, ***Número de Comandos*** e ***Número de Operações*** são definidas nesta seção de tal forma que possam ser também aplicadas a programas orientados a aspectos. As outras duas métricas desta seção, ***Fator de Difusão do Assunto sobre Componentes*** e ***Fator de Difusão do Assunto sobre Operações***, complementam os resultados das métricas de separação de assuntos apresentadas na seção 2.2.1 .

5.2.1 Número de Filhos (NOC)

Número de Filhos é baseada na métrica orientada a objetos [Chidamber e Kemerer, 1994] de mesmo nome e apresentada na seção 2.2 . Para que possa ser aplicada ao desenvolvimento orientado a aspectos, sua definição original foi generalizada suportando as novas estruturas deste paradigma. Esta métrica conta o número de

componentes que herdam direta e explicitamente do componente a ser medido. Desta forma, NOC não considera filhos indiretos (netos do componente) ou filhos criados implicitamente, por exemplo, pela estrutura de *inter-type declaration* de `AspectJ`.

O ***Número de Filhos*** e ***Profundidade da Árvore de Herança*** (DIT) são geralmente analisados em conjunto para verificar os relacionamentos de generalização. Um problema identificado no modelo de qualidade da seção 2.1 é a não inclusão da métrica NOC. Este fato dificulta a análise dos resultados medidos, podendo até mesmo levar a interpretações equivocadas. Observe os resultados de DIT medidos para o padrão *Observer* na Tabela 4. Estes resultados levam a interpretação de que as implementações Java e AspectJ deste padrão possuem o mesmo nível de acoplamento por herança. Na verdade a maior profundidade da hierarquia é 2 para ambas as implementações, entretanto, ao medir o ***Número de Filhos***, obtemos valor 10 para a implementação Java contra 3 para a implementação AspectJ. Desta forma, se a análise for feita utilizando as duas métricas NOC e DIT, é verificado um maior acoplamento por herança na implementação orientada a objetos. O maior acoplamento por herança resulta em mais métodos herdados, o que dificulta a previsão de comportamento do componente [Peters e Pedrycz, 2001].

A observação acima feita para o padrão *Observer* também é válida para outros dez padrões: *Chain of Responsability*, *Command*, *Composite*, *Flyweight*, *Iterator*, *Mediator*, *Prototype*, *Proxy*, *Singleton* e *Strategy*. A Tabela 4 apresenta os resultados da medição de ***Profundidade da Árvore de Herança*** e ***Número de Filhos*** para as implementações Java e AspectJ destes onze padrões. Os resultados desta tabela refletem valores após o cenário de evolução, e por estes resultados podemos claramente perceber que a medição de NOC revela um acoplamento por herança não revelado por DIT.

	DIT		**NOC**	
	Java	**AspectJ**	**Java**	**AspectJ**
Chain Of Resp.	2	2	7	1
Command	2	2	6	1
Composite	2	2	6	1
Flyweight	2	2	6	7
Iterator	2	2	6	3
Mediator	2	2	10	1
Observer	2	2	10	3
Prototype	2	2	6	1
Proxy	2	2	9	6
Singleton	2	2	5	10
Strategy	2	2	6	1

Tabela 4: **Valores de DIT e NOC para onze padrões de projeto.**

5.2.2 Número de Comandos (NOS)

O ***Número de Comandos*** (NOS) pode ser utilizado como alternativa à métrica ***Número de Linhas de Código*** (LOC). A vantagem da contagem de comandos sobre a contagem de linhas de código é que a primeira é menos sensível ao estilo de programação. NOS é geralmente utilizado para medir o tamanho das operações do sistema. No desenvolvimento orientado a aspectos com AspectJ, são considerados como operações os métodos, construtores e *advices*. Mesmo os métodos e construtores introduzidos pela estrutura de *inter-type declaration* são operações.

Na seção 3 é relatado que uma das etapas do processo de medição é o alinhamento das implementações. Este alinhamento inclui a árdua tarefa de garantir um mesmo estilo

de programação. O estilo de programação pouco (ou nada) influencia as demais medições efetuadas no código, no entanto, causa grande impacto no ***Número de Linhas de Código*** como ilustrado no exemplo abaixo (Código 4). Neste exemplo, as duas versões do método `returnGreater` possuem diferentes valores para LOC, quatro e dez respectivamente. Caso a medição seja feita utilizando ***Número de Comandos***, o resultado será o mesmo: três comandos.

```
int returnGreater(int x, int y) {
  if (x > y) return x;
  else return y;
}

int returnGreater(int x, int y)
{
  if (x > y)
  {
    return x;
  } else
  {
    return y;
  }
}
```

Código 4: **Ilustrativo do impacto do estilo de programação sobre LOC.**

A métrica que conta o número de comandos tem como vantagem reduzir o esforço de alinhamento das implementações. Por outro lado, a contagem de linhas de código parece mais fácil. Este documento não entra neste mérito, mas vale ressaltar que ambos os valores têm a mesma facilidade em serem obtidos se utilizada uma ferramenta adequada.

5.2.3 Número de Operações (NOO)

O ***Número de Operações*** surge como alternativa à métrica ***Peso das Operações por Componentes*** (WOC). Como discutido na seção 5.1 , a contagem de parâmetro feita por WOC não parece contribuir para a medida de tamanho do sistema. Desta forma, WOC pode ser substituída pela métrica NOO, que conta apenas os métodos, construtores e *advices* do sistema, sejam eles abstratos ou concretos. Além de medir o tamanho, ***Número de Operações*** também é útil para identificar o ***Fator de Difusão do Assunto sobre Operações***, apresentado na seção 5.2.5 .

5.2.4 Fator de Difusão do Assunto sobre Componentes (FCDC)

Em certos casos, o número de componentes do sistema varia entre as diferentes implementações a serem comparadas. Um valor absoluto na medição de Difusão do Assunto sobre Componentes (CDC) pode levar a interpretações erradas sobre o real espalhamento do assunto. O ***Fator de Difusão do Assunto sobre Componentes*** (FCDC) mede a percentagem de componentes afetados por um determinado assunto. FCDC é calculado pela razão entre o número de componentes afetados pelo assunto (CDC) e o número de componentes total do sistema (VS).

$$\mathrm{FCDC} = \frac{\mathrm{CDC}}{\mathrm{VS}}$$

No estudo de caso da seção 3 , as medições de CDC induzem à uma interpretação equivocada em alguns padrões. Por exemplo, ao avaliar o espalhamento do código do papel *Memento* (padrão *Memento*) na implementação original do padrão utilizando

CDC, obtemos valor igual a três para as duas implementações (Java e AspectJ). Este resultado leva a acreditar que o espalhamento sobre os componentes do sistema é equivalente em ambas implementações. Entretanto, a implementação orientada a objetos deste padrão tem um número diferente de componentes em relação à versão orientada a aspectos. Este fato justifica a medição do percentual de componentes afetados pelo assunto referente ao papel *Memento*. A versão Java tem três componentes contra cinco da versão AspectJ, e calculando o FCDC temos valor igual a 1,0 e 0,6 respectivamente. Ou seja, 100% dos componentes da versão Java são afetados pelo assunto enquanto apenas 60% dos componentes da versão AspectJ são afetados pelo mesmo assunto.

5.2.5 Fator de Difusão do Assunto sobre Operações (FCDO)

Semelhante à métrica FCDC, o ***Fator de Difusão do Assunto sobre Operações*** (FCDO) indica o percentual de operações afetadas por determinado assunto. Quando utilizada em conjunto com CDO, a métrica FCDO reduz distorções de interpretação quando o número de operações varia nas implementações avaliadas. O valor desta métrica é calculado pela razão entre o número de operações afetadas pelo assunto (CDO) e o número total de operações do sistema (NOO).

$$\text{FCDO} = \frac{\text{CDO}}{\text{NOO}}$$

O ***Fator de Difusão do Assunto sobre Operações*** pode ser calculado tanto para o sistema como um todo, como para qualquer componente do sistema. Quando aplicada apenas sobre um componente (classe, interface ou aspecto), seu valor indica a percentagem de operações deste componente que possui código de um determinado assunto. Voltando ao exemplo do Código 1 (seção 2.2.1), é verificado que a classe `Line` contêm 8 operações, sendo 6 delas afetadas pelo assunto referente ao padrão *Observer*. Pelo cálculo de FCDO, temos que 6/8, ou seja, 75% das operações da classe `Line` contém código relativo a este assunto.

Papéis	Implem. Original		Cenário de Evolução	
	Java	AspectJ	Java	AspectJ
Builder	13	15	27	29
Director	2	2	2	2

Tabela 5: Valores de CDO para o padrão Builder.

Em outro exemplo, a medição de CDO feita para o padrão *Builder* ilustrada na Tabela 5 sugere que a implementação orientada a objetos melhor modulariza o assunto em relação às operações. Note que para o papel *Builder* os valores obtidos são 13 (original) e 27 (cenário) para a implementação Java contra 15 (original) e 29 (cenário) para a implementação AspectJ. Por outro lado, o número total de operações nestas implementações é diferente. A implementação original Java possui 13 operações, enquanto a equivalente implementação AspectJ possui 15. Da mesma forma, o cenário de evolução Java possui 27 operações enquanto o cenário AspectJ possui 29. Sabendo destas informações, podemos calcular o ***Fator de Difusão do Assunto sobre Operações*** que apresenta valor igual a 1,0 (ou 100%) para ambas implementações. Estes números levam a conclusão que nenhuma das implementações é capaz de modularizar o papel *Builder*, e portanto o espalhamento é equivalente.

5.3 Modelo de Qualidade Adaptado

Nesta seção é apresentado o modelo de qualidade modificado que atende as observações feitas nas duas subseções anteriores (5.1 e 5.2). Este modelo, ilustrado na Figura 6, se baseia na versão preliminar proposta em [Sant´Anna, 2004] e discutida na seção 2.1 . Comparando os dois modelos, é verificado que as principais alterações ocorreram em seus extremos. Os dois primeiros níveis (características externas e fatores) existentes no modelo de qualidade original foram fundidos em uma única abstração que generaliza os conceitos. A decisão de efetuar esta alteração se justifica no fato de que a classificação original de características internas (manutenibilidade e reusabilidade) e os fatores (simplicidade e flexibilidade) são vistas como características implícitas de qualidade [Kan, 2003] [Peters e Pedrycz, 2001] [Pressman, 1995]. Os três atributos internos ilustrados na Figura 6 são os mesmos do modelo de qualidade original, ou seja, separação de assuntos, acoplamento, coesão e tamanho. Não é objetivo deste trabalho descrever como as características implícitas de qualidade são mapeadas para os atributos internos do software, mas é assumido tal fato mesmo que por critérios intuitivos.

O conjunto de métricas presente na Figura 6 não é o mesmo que compõe o modelo de qualidade da seção 2.1 . As alterações são resultados da avaliação das métricas ocorrida na seção 5.1 e pela contribuição de novas métricas descritas na seção 5.2 . Com avaliação negativa para os critérios de qualidade da seção 5.1 (definição, necessidade e ortogonalidade), a métrica ***Difusão do Assunto por Linhas de Código*** é excluída do modelo. Duas outras métricas são substituídas, ***Peso das Operações por Componentes*** é substituída por ***Número de Operações***, e, ***Número de Linhas de Código*** é substituída por ***Número de Comandos***. Além disso, três métricas são adicionadas ao modelo de qualidade. As métricas adicionadas são ***Número de Filhos***, ***Fator de Difusão do Assunto sobre Componentes*** e ***Fator de Difusão do Assunto sobre Operações***.

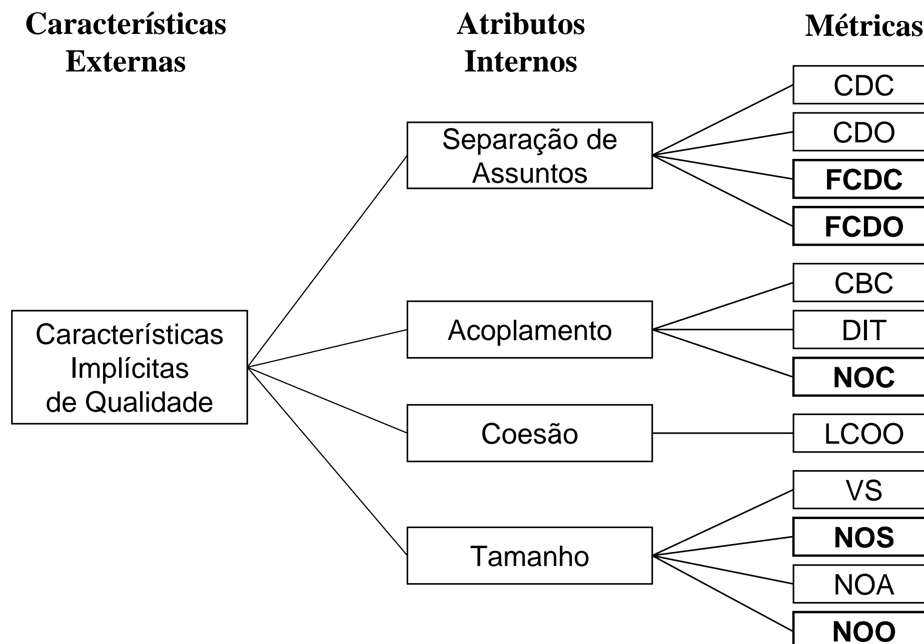

Figura 6: **Modelo de qualidade estendido para novas métricas.**

Um conjunto de doze métricas compõe o modelo de qualidade da Figura 6. Quatro contribuem para verificar o espalhamento do código relativo a um determinado assunto: ***Difusão do Assunto por Componentes*** (CDC), ***Difusão do Assunto por***

Operações (CDO), ***Fator de Difusão do Assunto sobre Componentes*** (FCDC) e ***Fator de Difusão do Assunto sobre Operações*** (FCDO). Três métricas procuram identificar o grau de acoplamento entre componentes: ***Acoplamento entre Componentes*** (CBC), ***Profundidade da Árvore de Herança*** (DIT) e ***Número de Filhos*** (NOC). A única métrica utilizada para verificar a coesão dos componentes do sistema é ***Perda de Coesão em Operações*** (LCOO). Finalmente, o modelo possui outras quatro métricas com o objetivo de mensurar o tamanho do sistema. A medida de tamanho é feita em função do ***Tamanho do Vocabulário*** (VS), ***Número de Comandos*** (NOS), ***Número de Atributos*** (NOA) e ***Número de Operações*** (NOO).

6 Conclusões

Neste documento foram apresentados resultados de um estudo empírico que comparam implementações orientadas a objetos e orientadas a aspectos de padrões de projeto. Este estudo utilizou um modelo de qualidade que envolve um conjunto de métricas propostos por Sant´Anna [Sant´Anna, 2004]. Pelos resultados obtidos no estudo empírico foi possível detectar pontos que pudessem ser melhorados no modelo de qualidade, especialmente em seu conjunto de métricas. Desta forma, foram feitas análises qualitativas sobre as métricas e propostas novas métricas para compor ao modelo. Utilizando uma estrutura de decomposição de sistemas e pela característica de distribuição das métricas sobre esta estrutura, tornou-se possível avaliação das métricas utilizando critérios de avaliação da estrutura agregada [Staa, 2000].

Este trabalho permitiu a criação de um modelo de qualidade mais adequado para avaliar programas orientados a objetos e orientados a aspectos. Além de obter resultados relativos à qualidade do software, este modelo permite efetuar comparações quantitativas entre sistemas. Mesmo sistemas implementados utilizando diferentes linguagens e paradigmas de programação podem ser avaliados e/ou comparados pelo modelo de qualidade proposto. O conjunto de métricas definidas para o modelo foi avaliado de tal forma que o processo de medição pudesse ser facilmente automatizado por uma ferramenta de suporte a medições.

Como direções para trabalhos futuros, são sugeridos novos estudos empíricos para validar o modelo de qualidade e seu conjunto de métricas resultantes deste estudo. Tais estudos empíricos podem abranger tanto avaliações da qualidade de sistemas, como trabalhos de pesquisa que comparam sistemas implementados em diferentes linguagens e/ou paradigmas. Outro trabalho futuro proposto é a implementação de uma ferramenta que dê suporte automatizado às medições. Esta ferramenta deve também disponibilizar avaliações semânticas que reflitam a qualidade do sistema baseado no modelo de qualidade.

Referências Bibliográficas

AspectJ – The AspectJ Programming Guide. Disponível em: http://eclipse.org/aspectj/ Acesso em: 10 de Dezembro de 2004.

BASILI V., CALDIERA G., ROMBACH H. Goal question metric paradigm. Marciniak, J.J. **Encyclopedia of software engineering**. John Wiley & Sons, 1994. p. 528-532.

CHIDAMBER, S.; KEMERER, C. A metrics suite for object oriented design. IEEE TRANSACTIONS ON SOFTWARE ENGENEERING, Proceedings... 1994, p. 476-493.

DEITEL, H. M.; DEITEL, P. J. **Java como programar**. 3º Edição, Editora Bookman, 2000, 1200 p.

Eclipse Metrics Plugin – Team in a Box. Disponível em: http://www.teaminabox.co.uk/downloads/metrics/ Acesso em: 11 de Abril de 2005.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns: elements of reusable object-qriented software**. Addison-Wesley, 1995, 364 p.

GARCIA, A. F. et al. **Aspectizing design patterns: rewards and pitfalls**. Rio de Janeiro: PUC-Rio, Departamento de Informática, 2004. 21 p. (Monografia de Ciência da Computação).

GARCIA, A. F. et al. Modularizing design patterns with aspects: a quantitative study. In: 4th ASPECT ORIENTED SOFTWARE DEVELOPMENT – AOSD'05, 2005, Chicago. Proceedings... 2005. Disponível (também) em: http://www.teccomm.les.inf.puc-rio.br/alessandro/GoFpatterns/empiricalresults.htm Acesso em: 03 de Fevereiro de 2005.

GARCIA, Alessandro. **From objects to agents: an aspect-oriented approach**. 298 f. Tese (Doutorado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, Rio de Janeiro, 2004.

GARCIA, A. F.; SILVA, V.; CHAVES, C.; LUCENA, C. J. P. Engineering multi-agent systems with aspects and patterns. Jornal da Sociedade Brasileira de Computação, vol. 8, no. 1, p. 57-72. Julho de 2002.

HANNEMANN, J.; KICZALES, G. Design pattern implementation in Java and AspectJ. In: 17th ANNUAL ACM CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS – OOPSLA´02, 2002, Seattle. Proceedings... 2002, p. 161-173.

HARRISON, R.; COUNSELL S. J.; NITHI, R. V. An evaluation of the MOOD set of object-oriented software metrics. In: IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 1998. Proceedings... 1998, vol. 24, no. 6, p. 491-496.

HENDERSON-SELLERS, B. Object-oriented metrics: measures of complexity. Upper Saddle River, New Jersey. Prentice Hall, 1996.

Java Reference Documentation. Disponível em: http://java.sun.com/reference/docs/index.html Acesso em: 10 de Dezembro de 2004.

KAN, S. H. **Metrics and models in software quality engineering**, 2nd ed. Pearson Education, 2003, 344 p.

KICZALES, G. et al. Aspect-oriented programming. EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING– ECOOP'97, 1241, Springer, Finland. Proceedings... 1997, p. 220-242.

LOPES, Cristina Videira. **D: A Language Framework for Distributed Programming**. 278 f. PhD Thesis, College of Computer Science, Northeastern University. Boston, USA, 1997.

LORENZ, M.; KIDD J. **Object-oriented software metrics, a practical guide**. Englewood Cliffs, N.J.: PTR Prentice-Hall, 1994.

PETERS, J. F.; PEDRYCZ, W. **Engenharia de software: teoria e prática**. Rio de Janeiro: Campus, 2001. 602p.

PRESSMAN, Roger S. **Engenharia de software**, 3ª edição. São Paulo: Makron Books, 1995.

SANT´ANNA, C. et al. Design patterns as aspects: a quantitative assessment. SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE – SBES'04, 2004, Brasília. Proceedings..., 2004, p. 113-129.

SANT´ANNA, Cláudio. **Manutenibilidade e reusabilidade de software orientado a aspectos: um framework de avaliação**. 109 f. Dissertação (Mestrado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, Rio de Janeiro, 2004.

STAA, Arndt v. **Programação modular**. Rio de Janeiro: Campus, 2000, 720 p.