5
As Métricas

O conjunto de métricas proposto neste trabalho captura informações sobre o projeto e o código de software orientado a aspectos em termos de atributos de software fundamentais como separação de *concerns*, acoplamento, coesão e tamanho. Foram definidas novas métricas e reutilizadas e refinadas métricas clássicas (por exemplo, LOC [19]) e métricas orientadas a objetos (por exemplo, as métricas de Chidamber & Kemerer [6]). Apesar de grande parte da pesquisa sobre métricas de software ter como foco o software funcional e orientado a objetos, até este momento não existem métricas para software orientado a aspectos. Por isso, este trabalho procurou adaptar métricas orientadas a objetos de forma que elas pudessem ser aplicadas a software orientado a aspectos. De fato, as definições de algumas métricas orientadas a objetos existentes foram refeitas para permitir a comparação entre projeto e código orientado a aspectos com projeto e código orientado a objetos. Cada definição de métrica foi estendida para ser aplicada de maneira independente do paradigma, apoiando a geração de resultados comparáveis. Além disso, foram propostas novas métricas de separação de *concerns*, inspiradas em métricas apresentadas na tese de doutorado de Lopes [33].

O critério para seleção das métricas usadas no framework levou em consideração demandas teóricas e práticas. Por exemplo, as métricas de Chidamber & Kemerer são baseadas em uma sólida teoria de medição e têm sido amplamente usadas e validadas empiricamente [8]. Mesmo assim, algumas métricas de Chidamber & Kemerer não deixam de ser alvo de críticas, como a crítica feita por Henderson-Sellers [7] sobre a métrica de coesão proposta por eles. A escolha das métricas se baseou em análises anteriores sobre métricas existentes [34, 35, 36, 37, 38, 39] e outros trabalhos já realizados para a escolha de métricas para a predição de manutenibilidade e reusabilidade [40]. A partir daí, foi definido um conjunto de métricas orientadas a aspectos que procura atender os seguintes requisitos: (i) mede atributos de software bem conhecidos: separação de *concerns*,

acoplamento, coesão e tamanho, (ii) depende o máximo possível de métricas tradicionais ou da extensão de métricas orientadas a objetos, já que as abstrações orientadas a aspectos estendem o conjunto de abstrações orientadas a objetos, (iii) captura as diferentes dimensões dos princípios e atributos de qualidade introduzidos pelas novas abstrações e mecanismos de composição e decomposição do DSOA (Seção 4.3) e (iv) apóia a identificação das vantagens e desvantagens do uso de aspectos no desenvolvimento de software quando comparado com uma solução orientada a objetos para o mesmo problema.

O conjunto de métricas do framework de avaliação é composto por dez métricas. Algumas delas podem ser aplicadas tanto no projeto quanto no código do sistema. Outras só podem ser aplicadas no código. Essas métricas são agrupadas de acordo com os atributos que elas medem: (i) separação de *concerns*, (ii) tamanho, (iii) acoplamento e (iv) coesão. A Tabela 1 associa cada métrica ao atributo interno definido pelo modelo de qualidade (Seção 4.1) e à pergunta gerada com o uso da abordagem *GQM* (Seção 4.2). A Tabela 1 apresenta também a fonte das métricas nas quais foram baseadas as métricas definidas neste trabalho.

A maioria das métricas do framework de avaliação é descrita em termos de classes, aspectos, métodos e *advices*. Por isso, antes de apresentar a definição das métricas, vale a pena lembrar que, neste trabalho, como já foi dito no Capítulo 2, classes, interfaces e aspectos são chamados de componentes, e métodos e *advices* são chamados de operações. Pelo fato de serem construções semelhantes entre si, classes e aspectos são tratados e contados da mesma forma na definição das métricas do framework. O mesmo acontece com métodos e *advices*.

A seguir, as métricas são apresentadas em termos de sua definição e sua relevância em relação à manutenibilidade e à reusabilidade. A relevância refere-se a suposições, complementares àquelas apresentadas nas seções 4.1.1, 4.1.2, 4.1.3, que justificam a influência de cada métrica nos atributos de manutenibilidade e reusabilidade. Além disso, é apresentado um exemplo da aplicação de cada métrica. O conjunto de métricas é apresentado nesta dissertação separadamente da descrição do framework (Capítulo 4), porque as métricas podem ser reutilizadas também por outros frameworks de avaliação (por exemplo, frameworks cujo objetivo é medir outros atributos externos de qualidade, como confiabilidade).

Métrica	Pergunta Respondida	Atributo	Baseada em
Tamanho do Vocabulário (VS)	Quantos componentes existem no sistema?	Tamanho	Papaioannou [41]
Linhas de Código (LOC)	Quantas linhas de código existem no sistema?	Tamanho	Fenton [19]
Número de Atributos (NOA)	Quantos atributos existem no sistema?	Tamanho	Fenton [19]
Peso de Operações por Componente (WOC)	Quantos métodos e *advices* existem no sistema?	Tamanho	Chidamber [6]
Acoplamento entre Componentes (CBC)	Quão elevado é o acoplamento entre os componentes?	Acoplamento	Chidamber [6]
Falta de Coesão nas Operações (LCOO)	Quão elevada é a coesão dos componentes do sistema?	Coesão	Chidamber [6]
Profundidade da Árvore de Herança (DIT)	Quão elevado é o acoplamento entre os componentes?	Acoplamento	Chidamber [6]
Difusão do *Concern* por Componentes (CDC)	Quão bem localizados estão os *concerns* do sistema?	Separação de *Concerns*	Lopes [33]
Difusão do *Concern* por Operações (CDO)	Quão bem localizados estão os *concerns* do sistema?	Separação de *Concerns*	Lopes [33]
Difusão do *Concern* por Linhas de Código (CDLOC)	Quão bem localizados estão os *concerns* do sistema?	Separação de *Concerns*	Lopes [33]

Tabela 1 – Métricas: Perguntas do *GQM*, Atributos e Fontes.

5.1. Métricas de Separação de *Concerns*

Neste trabalho, foram definidas três métricas de separação de *concerns*: Difusão do *Concern* por Componentes (CDC[4]), Difusão do *Concern* por Operações (CDO[5]), Difusão do *Concern* por Linhas de Código (CDLOC[6]). Essas métricas geram valores por *concern*, diferentemente da maioria das métricas de acoplamento, coesão e tamanho, que geram valores por componente, como será visto nas próximas seções.

[4] Do inglês *Concern Diffusion over Components*.

[5] Do inglês *Concern Diffusion over Operations*.

[6] Do inglês *Concern Diffusion over Lines of Code*.

5.1.1.
Difusão do *Concern* por Componentes (CDC)

Definição: CDC conta o número de componentes principais do *concern* avaliado, ou seja, componentes (classes, interfaces e aspectos) cujo propósito principal é contribuir para a implementação do *concern* avaliado. Além disso, conta também os componentes que fazem referência aos componentes principais do *concern*, ou seja, conta o número de componentes que acessam os componentes principais os usando em declaração de atributos, parâmetros de operações, tipos de retorno ou variáveis locais, ou chamam seus métodos.

Relevância: Essa métrica mede o grau com o qual um *concern* está espalhado pelos componentes do projeto do software. Quanto menos espalhado estiver o *concern*, mais fácil será para entendê-lo. Quanto menos espalhado estiver o *concern*, menos componentes terão que ser alterados em atividades de manutenção, e menos componentes terão que ser compreendidos e estendidos em atividades de reutilização.

Exemplo: A Figura 8 mostra parte do código de um sistema que foi avaliado com o uso do framework num estudo experimental, que será descrito mais adiante na Seção 6.1. Ela mostra as classes que contribuem para a implementação de um dos *concerns* do sistema. É um *concern* específico do domínio do sistema chamado de propriedade de adaptação. O propósito principal da classe abstrata *Property* e da classe *Adaptation* é contribuir para implementação do *concern*, por isso elas devem ser contadas na métrica CDC. A classe *Agent*, apesar de não ter o propósito principal de implementar o *concern*, também deve ser contada, pois usa a classe *Adaptation* na declaração de um de seus atributos, além de realizar chamadas a seus métodos. Então, o valor da métrica CDC para esse *concern* (propriedade de adaptação) é três, o que significa que o *concern* está espalhado por três componentes do sistema.

```
public abstract class Property
{
   protected Agent theAgent;

   public Property() {
   }
}
```

```
public class Adaptation extends Property
{
   public Adaptation(Agent agent) {
       System.out.println("<* Adaptation *> aspect created.");
       theAgent = agent;
   }

   public boolean comparePlan(Goal goal, Plan plan) {
      Goal planGoal = plan.getGoal();
      if (planGoal.getClass() == goal.getClass()) return true; else return false;
 ...
 ...
```

```
public class Agent {
...
...
    protected Interaction theInteraction;
    protected Autonomy    theAutonomy;
    protected Adaptation  theAdaptation;

    public Agent(String aName, Vector pl) {
        agentName = aName;
        theInteraction = new Interaction(this);
        theAutonomy = new Autonomy(this);
        theAdaptation = new Adaptation(this);
        planList = pl;
        System.out.println(" Name == " + agentName);
   }

   public Agent(String aName) {
        agentName = aName;
        theInteraction = new Interaction(this);
        theAutonomy = new Autonomy(this);
        theAdaptation = new Adaptation(this);
        System.out.println(" Name == " + agentName);
   }

   public Agent(String aName, Goal initialGoal) {
        agentName = aName;
        theInteraction = new Interaction(this);
        theAutonomy = new Autonomy(this);
        theAdaptation = new Adaptation(this);
        System.out.println(" Name == " + agentName);
        setGoal(initialGoal);
   }
...
...
   public void setGoal(Goal goal) {
      toAchieveGoals.addElement(goal);
      theAdaptation.adaptPlan(goal);
   }
...
...
   public void receiveMsg(Message msg)
   {
       theAutonomy.makeDecision(msg);
       theAdaptation.adaptBeliefs(msg);
   }
}
```

Figura 8 – Exemplo do uso das métricas de separação de *concerns* em parte do código do sistema do primeiro estudo experimental (Seção 6.1).

5.1.2.
Difusão do *Concern* por Operações (CDO)

Definição: CDO conta o número de operações principais do *concern* avaliado, ou seja, métodos e *advices* cujo propósito principal é contribuir para a implementação do *concern* avaliado. Em adição, conta também o número de operações que acessam qualquer operação principal. Além disso conta também as operações que acessam qualquer componente principal (descrito na Seção 5.1.1) ou chamando seus métodos ou o usando na declaração de variáveis locais, parâmetros ou tipos de retorno. Vale ressaltar que construtores e métodos abstratos também são contados.

Relevância: Essa métrica mede o espalhamento do *concern* em termos de operações. Quanto mais operações forem afetadas pelo *concern*, mais difícil será para entendê-lo. Quanto mais operações forem afetadas pelo *concern*, mais difícil será para reutilizá-lo, e mais operações serão modificadas durante atividades de manutenção.

Exemplo: Ainda analisando o código da Figura 8 sobre o *concern* de propriedade de adaptação, todos os métodos das classes *Property* e *Adaptation* devem ser contados pela métrica CDO, pois todos eles têm o propósito principal de contribuir com a implementação do *concern*, já que pertencem a classes cujo único e principal propósito é implementar o *concern*. Já na classe *Agent*, só os métodos onde existem referências para a classe ou instâncias da classe *Adaptation*, devem ser contados. Portanto, são contados o único método construtor da classe *Property* e os dois métodos da classe *Adaptation* (considerando que ela tem apenas os métodos que aparecem na figura, pois na realidade ela tem mais). Soma-se a isso os cinco métodos da classe *Agent* onde há referências para a classe *Adaptation*. Então, o valor de CDO para o *concern* de propriedade de adaptação é igual a oito, o que quer dizer que ele está espalhado por oito operações.

5.1.3.
Difusão do *Concern* por Linhas de Código (CDLOC)

Definição: CLOC foi definida com o intuito de capturar quão o código que implementa um *concern* está misturado com o código dos outros *concerns*.

CDLOC conta o número de pontos de transição entre o *concern* avaliado e os outros *concerns* do sistema através das linhas de código. O uso dessa métrica requer que as partes do código do sistema que implementam o *concern* avaliado sejam sombreadas. Esse processo de sombreamento divide então o código do sistema em áreas sombreadas e áreas não sombreadas. Os pontos de transição são os pontos do código em que há uma transição de uma área sombreada para uma área não sombreada ou vice-versa. A intuição por trás dessa métrica é que os pontos de transição são pontos do texto do programa onde há uma "troca de *concern*". Para cada *concern*, o código do programa é analisado linha por linha para contar os pontos de transição. Quanto maior o valor de CDLOC, mais misturado e emaranhado o código do *concern* avaliado está com o código de outros *concerns*.

O processo de sombreamento do código relativo ao *concern* avaliado deve seguir as seguintes regras:

a) Classes cujo único propósito é contribuir para a implementação do *concern* são tratadas de maneira especial: tanto sua declaração quanto seus atributos e métodos são sombreados como um bloco único. Invocações de métodos de instâncias dessas classes são sombreadas.
b) Aspectos cujo único propósito é contribuir para a implementação do *concern* são tratados de maneira especial: tanto sua declaração quanto seus atributos, métodos e *advices* são sombreados como um bloco único. Invocações de métodos desses aspectos são sombreadas.
c) Métodos cujo único propósito é a implementação do *concern* são sombreados. Chamadas a esses métodos também são sombreadas. Note que esses métodos não fazem parte das classes ou aspectos cujo único propósito é implementar o *concern* (regras *a* e *b*).
d) *Advices* cujo único propósito é a implementação *concern* são sombreados. Note que esses *advices* não fazem parte dos aspectos cujo único propósito é implementar o *concern* (regra *b*).
e) Declarações de variáveis do tipo das classes cujo único propósito é implementar o *concern* (regra *a*) são sombreadas. O uso dessas variáveis também é sombreado.
f) Assinatura de métodos ou *advices* cujos parâmetros ou tipo de retorno contêm referências para objetos das classes cujo único propósito é

implementar o *concern* (regra *a*) são sombreadas. Note que esses métodos e *advices* não fazem parte das classes ou aspectos cujo único propósito é implementar o *concern* (regras *a* e *b*).

g) *Inter-type declarations* que fazem referência a classes ou aspectos cujo único propósito é implementar o *concern* (regras *a* e *b*) são sombreadas. Note que essas *inter-type declarations* não fazem parte dos aspectos cujo único propósito é implementar o *concern* (regra *b*).

h) A aplicação das regras *a-g* pode gerar dois ou mais blocos sombreados para o mesmo *concern* que apareçam em seqüência, sem código de outros *concern* entre eles. Então, neste caso, esses blocos devem ser unidos em um bloco único.

i) Se dois blocos de código de um mesmo *concern* não aparecem seguidos, mas outra arrumação do código, que não influenciasse nem a compilação nem a execução do sistema, permitisse que esses blocos ficassem em seguida um do outro, então eles devem ser contados como apenas um bloco.

Os resultados de CDLOC são relativos ao *concern* que está sendo avaliado e ao processo de sombreamento. Qualquer pequena variação nesses dois fatores pode resultar em mudanças drásticas nos números.

Relevância: Quanto mais o código de um *concern* está misturado e emaranhado com o código de outros *concerns*, mais difícil será para compreendê-lo, mais pontos distintos do código serão afetados durante atividades de manutenção, e mais difícil será para reutilizá-lo.

Exemplo: Pode-se observar ainda na Figura 8 o sombreamento relativo ao *concern* de propriedade de adaptação do sistema avaliado em um dos estudos de casos a serem apresentados no próximo capítulo. As áreas sombreadas são as partes do código relativas ao *concern* avaliado. As classes *Property* e *Adaptation* são totalmente sombreadas, pois seu único propósito é contribuir para a implementação do *concern* (regra *a*). Na classe *Agent*, estão sombreadas as linhas de código onde um atributo é declarado do tipo da classe *Adaptation* e onde esse atributo é usado (regra *e*). Apesar de parecer que as classes *Property* e *Adaptation* constituem dois blocos distintos de sombreamento, deve-se considerá-las como parte de um único bloco, pois elas estão dispostas uma em seguida da outra (regra *h*). Conta-se, então, os pontos de transição entre as áreas sombreadas e as áreas

não sombreadas. CDLOC começa a ser contada do valor um, dentro da área sombreada formada pelas classes *Property* e *Adaptation*. Ao sair dessa área, ela é incrementada de mais um. Ao entrar na próxima área sombreada (declaração do atributo do tipo *Adaptation*), CDLOC é incrementada de mais um, e assim por diante. No final o valor de CDLOC é 14, o que significa que existem 14 pontos de transição entre o código relativo ao *concern* de propriedade de adaptação e os outros *concerns* do sistema.

5.2. Métricas de Acoplamento

Acoplamento entre Componentes (CBC[7]) e Profundidade da Árvore de Herança (DIT[8]) são as duas métricas de acoplamento que fazem parte do framework de avaliação.

5.2.1. Acoplamento entre Componentes (CBC)

Definição: CBC é definida para um componente como sendo o número de outros componentes com os quais ele está acoplado. Essa métrica é uma extensão da métrica de acoplamento entre objetos (CBO) proposta por Chidamber & Kemerer [6]. No sentido de definir a métrica CBC, a definição da métrica CBO foi alterada para ser capaz de tratar das novas dimensões de acoplamento do DSOA (Seção 4.3.1). Para cada componente (classe ou aspecto), conta-se o número de outras classes que são usadas em declarações de atributos, parâmetros de operações, tipos de retorno ou variáveis locais, isto é, captura os acoplamentos C2, C3 e C5 apresentados na Figura 7 (Seção 4.3.1). Além disso, para cada aspecto, conta-se: as classes nas quais são introduzidos atributos e métodos por meio de *inter-type declarations* (acoplamento C4), as classes ou aspectos que são interceptados pelo aspecto por meio de definições de *pointcuts* (acoplamentos C6 e C8) e os componentes cujos atributos e métodos introduzidos via *inter-type declaration* são acessados pelo aspecto (acoplamentos C7 e C9). Se um componente X se relaciona a um mesmo componente Y várias vezes da mesma

[7] Do inglês *Coupling between Components*.

[8] Do inglês *Depth of Inheritance Tree*.

forma ou de formas diferentes de acoplamento, o valor de CBC do componente X é incrementado de apenas um, pois essa métrica conta o número de componentes com o qual determinado componente está acoplado e não o número de vezes com que esse acoplamento se dá. Essa métrica compreende oito dimensões de acoplamento apresentadas na Figura 7 (de C2 a C9).

Relevância: A compreensão de um componente envolve o entendimento dos componentes aos os quais ele está acoplado. Então, quanto maior for o número de acoplamentos de um componente, mais difícil será para ele ser compreendido. No sentido de melhorar a modularidade e promover o encapsulamento, acoplamentos entre componentes devem ocorrer o mínimo possível. Quanto maior o número de acoplamentos, maior é a sensibilidade a mudanças em outras partes do projeto e, portanto, a manutenção será mais difícil. Acoplamento excessivo entre componentes é prejudicial ao projeto modular e previne a reutilização. Quanto mais independente for um componente, mais facilmente ele poderá ser reutilizado em outra aplicação.

Exemplo: A Figura 9 mostra o código do aspecto *ColorObserver*. Esse código ilustra as duas formas de acoplamento mais interessantes e diferentes que a orientação a aspectos introduz. É o acoplamento pelo uso de *inter-type declarations* (acoplamento C4) e o acoplamento pela definição de *pointcuts* onde são especificados *join points* em que o aspecto intercepta classes (acoplamento C6). As linhas 17 e 18 mostram que o aspecto *ColorObserver* está acoplado as classes *Point1* e *Point2*, pois, por meio do mecanismo de *inter-type declarations*, ele introduz nessas duas classes a declaração de que elas implementam a interface *Subject*. O mesmo acontece com as classes *Screen1* e *Screen2* nas linhas 25 e 26, só que, nesse caso o aspecto introduz a declaração de implementação da interface *Observer*. A linha 34 apresenta outra forma de acoplamento. Nela é definido o *pointcut* que especifica que as chamadas ao método *setColor* de qualquer classe cujo nome comece com *Point* (*Point**) deve ser interceptada pelo aspecto. Nesse caso está caracterizado mais uma forma de acoplamento do aspecto com as classes *Point1* e *Point2*. Mas o fato do aspecto está acoplado a essas classes de duas maneiras diferentes não influencia no valor da métrica, pois ela conta o número de componentes acoplados e não o número de formas de acoplamento. Existe mais um detalhe nesse exemplo. Apesar do aspecto fazer referência as interfaces *Subject* e *Observer*, elas não são consideradas como acopladas ao aspecto, porque

são interfaces definidas dentro dele (linha 07 e linhas 13-15). Portanto o aspecto *ColorObserver* está acoplado as classes *Point1*, *Point2*, *Screen1* e *Screen2*. Então o valor de CBC para esse aspecto é quatro.

```
01 public aspect ColorObserver {
02
03  /**
04   * This interface is used to say what types can be subjects.
05   */
06
07  protected interface Subject  { }
08
09  /**
10   * This interface is used to say what types can be observers.
11   */
12
13  protected interface Observer {
14     public void display (String s);
15  }
16
17  declare parents: Point1  implements Subject;
18  declare parents: Point2  implements Subject;
19
20  /**
21   * Assings the <code>Observer</code> role to the <code>Screen</code> class.
22   * Roles are modeled as (empty) interfaces.
23   */
24
25   declare parents: Screen1 implements Observer;
26   declare parents: Screen2 implements Observer;
27
28  /**
29   * Specifies the joinpoints that constitute a change to the
30   * Subject. Captures calls to Point*.setColor(Color)
32   */
33
34  protected pointcut subjectChange(Subject s): call(void Point*.setColor(Color))
    && target(s);
...
...
```

Figura 9 – Código para exemplificar o uso da métrica de acoplamento CBC

5.2.2. Profundidade da Árvore de Herança (DIT)

Definição: DIT é definida como o comprimento máximo de um nó para a raiz da árvore. Ela mede a profundidade na hierarquia de herança em que um componente é declarado. DIT é uma extensão de uma métrica de mesmo nome proposta por Chidamber & Kemerer, que passa a considerar a herança entre aspectos. Essa métrica compreende os acoplamentos C1 e C10 ilustrados na Figura 7.

Relevância: Quanto mais profundo na hierarquia um componente for declarado, provavelmente, maior será o número de métodos, *advices* e atributos herdados por ele, fazendo com seja mais difícil de compreendê-lo. Componentes que herdam atributos e operações estão acoplados aos seus supercomponentes. As

mudanças realizadas nos supercomponentes devem ser feitas com cuidado, pois serão propagadas para todos os componentes que herdam suas características. Quanto mais profundo na hierarquia um componente for declarado, mais difícil será para reutilizá-lo, pois todos os seus supercomponentes precisam ser compreendidos.

Exemplo: A Figura 10 mostra parte do código do aspecto abstrato *MediatorProtocol* e parte do código do aspecto *MediatorImplementation*. O valor de DIT para aspecto *MediatorProtocol* é igual a um, já que ele não estende nenhum outro aspecto. O aspecto *MediatorImplementation* estende o aspecto *MediatorProtocol*, e, portanto, o valor de DIT para ele é igual a dois. Esses aspectos fazem parte da implementação orientada a aspectos do padrão de projeto *Mediator* proposta por Hannemann & Kiczales [5], que foi avaliada em um estudo experimental que será apresentado na Seção 6.2.

```
public abstract aspect MediatorProtocol {

...
...
        protected abstract pointcut change(Colleague c);

        after(Colleague c): change(c) {
                notifyMediator(c, getMediator(c));
        }

        protected abstract void notifyMediator(Colleague c, Mediator m);
}
```

```
public aspect MediatorImplementation extends MediatorProtocol {

        declare parents: Button implements Colleague;

        declare parents: Label implements Mediator;

        protected pointcut change(Colleague c):
            (call(void Button.clicked()) && target(c));
...
...
}
```

Figura 10 – Código para exemplificar o uso da métrica de acoplamento DIT

5.3. Métrica de Coesão

O conjunto de métricas possui uma única métrica de coesão: Falta de Coesão nas Operações (LCOO[9]).

[9] Do inglês *Lack of Cohesion in Operations.*

5.3.1.
Falta de Coesão em Operações (LCOO)

Definição: Essa métrica mede a falta de coesão de um componente. Se um componente C_1 tem *n* operações (métodos e *advices*) O_1,..., O_n então $\{I_j\}$ é o conjunto de atributos usados pela operação O_j. Seja |P| o número de interseções vazias entre conjuntos de atributos. Seja |Q| o número de interseções não vazias entre conjuntos de atributos. Então: LCOO = |P| – |Q|, se |P| ≥ |Q|, ou LCOO = 0, se |P| < |Q|. LCOO mede a quantidade de pares de métodos/*advices* que não acessam pelo menos um mesmo atributo. Portanto, é uma medida de falta de coesão. Assim como na métrica de tamanho WOC, LCOO leva em consideração, além dos *advices* e métodos internos ao aspecto, também os métodos que o aspecto introduz nas classes que ele afeta por meio de *inter-type declarations*. Essa métrica estende a métrica LCOM de Chidamber & Kemerer. Ela trata dos *advices* e métodos dos aspectos da mesma forma que a métrica de deles trata dos métodos das classes.

Relevância: Componentes com baixa coesão sugerem um projeto inadequado, pois isso significa o encapsulamento de entidades de programa não relacionadas entre si e que não deveriam estar juntas [8]. Quanto maior for o grau com o qual diferentes ações executadas por um componente contribuam para funcionalidades distintas, mais difícil será para manter ou reutilizar o componente ou uma de suas funcionalidades.

Exemplo: A Figura 11 mostra o código da classe *Point* que faz parte da implementação em Java do padrão de projeto *Observer* proposta por Hannemann & Kiczales [5]. Essa classe possui 4 atributos e 10 métodos. Para calcular o valor de LCOO para qualquer classe ou aspecto, deve-se começar pelo primeiro método ou *advice* e verificar se ele manipula atributos em comum com cada método ou *advice* abaixo dele. Depois, concentra-se no método ou *advice* seguinte, o compara com os outros abaixo dele, e assim por diante. Nesse caso, começa-se a calcular o valor de LCOO da classe *Point* pelo construtor (linhas 11-16). Ele acessa pelo menos um atributo em comum com todos os outros nove métodos, então Q = 9, e P = 0. O método *getX* (linha 18) usa atributos em comum apenas com o método *setX* (linhas 22-25) dentre os oito métodos restantes, então Q = 9 + 1 = 10, e P = 0 + 7 = 7. O método *getY* (linha 20) também só usa atributos em

comum com o método *setY* (linhas 27-30) dentre os sete métodos restantes, então Q = 10 + 1 =11, e P = 7 + 6 = 13.

```
01  public class Point implements Subject {
02
03      private HashSet observers;
04
05      private int x;
06
07      private int y;
08
09      private Color color;
10
11      public Point(int x, int y, Color color) {
12          this.x=x;
13          this.y=y;
14          this.color=color;
15          this.observers = new HashSet();
16      }
17
18      public int getX() { return x; }
19
20      public int getY() { return y; }
21
22      public void setX(int x) {
23          this.x=x;
24          notifyObservers();
25      }
26
27      public void setY(int y) {
28          this.y=y;
29          notifyObservers();
30      }
31
32      public Color getColor() { return color; }
33
34      public void setColor(Color color) {
35          this.color=color;
36          notifyObservers();
37      }
38
39      public void attach(Observer o) {
40          this.observers.add(o);
41      }
42
43      public void detach(Observer o) {
44          this.observers.remove(o);
45      }
46
47      public void notifyObservers() {
48          for (Iterator e = observers.iterator() ; e.hasNext() ;) {
49              ((Observer)e.next()).update(this);
50          }
51      }
52  }
```

Figura 11 – Código para exemplificar o uso da métrica de coesão LCOO

A seguir é a vez do método *setX* (linhas 22-25) que não manipula nenhum atributo comum aos seis métodos restantes, logo Q = 11 + 0 = 11, e P = 13 + 6 = 19. Da mesma forma, o método *setY* (linhas 27-30) não manipula nenhum atributo comum aos cinco métodos restantes, logo Q = 11 + 0 = 11, e P = 19 + 5 = 24. O método *getColor* (linha 32) usa o mesmo atributo que o método *setColor* (linhas

34-37), mas os outros três métodos usam atributos diferentes, por conseguinte Q = 11 + 1 = 12, e P = 24 + 3 = 27. O método *setColor* (linhas 34-37) não acessa nenhum atributo comum aos três métodos que sobram, assim Q = 12 + 0 = 12, e P = 27 + 3 = 30. O método *attach* (linhas 39-41) acessa o atributo *observer*, igualmente aos dois últimos métodos, portanto Q = 12 + 2 = 14, e P = 30 + 0 = 30. E, finalmente, os métodos *detach* (linhas 43-45) e *notifyObservers* (linhas 47-51) acessam também o atributo *observer*, por conseqüência Q = 14 + 1 = 15, e P = 30 + 0 = 30. Agora, usando a fórmula LCOO = |P| – |Q|, se |P| > |Q| ou LCOO = 0, tem-se que, para a classe *Point*, LCOO = 30 – 15, ou seja, LCOO = 15. Isso significa que na classe *Point* há 15 pares de métodos que não acessam atributo em comum a mais do que os pares que acessam pelo menos um atributo em comum.

5.4. Métricas de Tamanho

As métricas de tamanho tratam de diferentes aspectos do tamanho do sistema. O conjunto de métricas do framework de avaliação é composto das seguintes métricas de tamanho: Tamanho do Vocabulário (VS[10]), Linhas de Código (LOC[11]), Número de Atributos (NOA[12]) e Peso de Operações por Componente (WOC[13]).

5.4.1. Tamanho do Vocabulário (VS)

Definição: VS conta o número de componentes do sistema, isto é, o número de classes, interfaces e aspectos que constituem o sistema. Essa métrica mede tamanho o vocabulário do sistema. Cada nome de componente é contado como parte do vocabulário do sistema. As instâncias não são contadas.

Relevância: Quanto maior for o tamanho do vocabulário, mais difícil será para entender o sistema. Quanto mais difícil for para entender o sistema, mais difícil será para encontrar os componentes que precisam ser modificados durante

[10] Do inglês *Vocabulary Size*.

[11] Do inglês *Lines of Code*.

[12] Do inglês *Number of Attributes*.

[13] Do inglês *Weighted Operations per Components*.

atividades de evolução, e mais difícil será para encontrar os componentes que implementam as funcionalidades que se deseja reutilizar.

Exemplo: A implementação do padrão *State* em AspectJ proposta por Hannemann & Kiczales [5] é composta pelas classes *Main*, *Queue*, *QueueEmpty*, *QueueFull*, *QueueNormal*, *QueueState* e pelo aspecto *QueueStateAspect*. Portanto o valor da métrica VS para essa implementação é sete.

5.4.2. Linhas de Código (LOC)

Definição: Essa métrica conta o número de linhas de código. É uma medida de tamanho tradicional. Comentários relativos à documentação ou implementação não são interpretados como código. Linhas em branco também não são contadas. Estilos diferentes de programação podem influenciar os resultados obtidos pela aplicação dessa métrica. Para evitar esse problema, nos estudos empíricos apresentados nesta dissertação (Capítulo 6), o mesmo estilo de programação foi assegurado em todos os sistemas avaliados.

Relevância: Quanto maior for o número de linhas de código, maior será a dificuldade para entender o sistema. Quanto maior for o número de linhas de código, mais difícil será para encontrar as linhas que precisam ser alteradas durante atividades de evolução, e mais difícil será para entender a implementação das funcionalidades que se deseja reutilizar.

Exemplo: O código, em AspectJ, do aspecto *AbstractFactoryEnhancement* é apresentado na Figura 12. A contagem da métrica LOC deve desprezar comentários e linhas em branco, portanto o valor de LOC para esse aspecto é oito.

5.4.3. Número de Atributos (NOA)

Definição: Essa métrica mede o vocabulário interno de cada componente, isto é, o número de atributos de cada classe ou aspecto. Atributos herdados de superclasses ou superaspectos não são contados por essa métrica. Num aspecto, além dos atributos internos ao aspecto, são contados também os atributos que o aspecto introduz nas classes que ele afeta por meio de *inter-type declarations*.

```
01  public aspect AbstractFactoryEnhancement {
02
03     /**
04       * Represents a default implementation to all Abstract Factories
05       * for the <code>createLabel()</code> method.
06       */
07
08      public JLabel AbstractFactory.createLabel() {
09             return new JLabel("This Label was created by " +getName());
10      }
11
12      /**
13       * Represents a default implementation to all Abstract Factories
14       * for the <code>createButton()</code> method.
15       */
16
17      public JButton AbstractFactory.createButton(String label) {
18             return new JButton(label);
19      }
20  }
```

Figura 12 – Código para exemplificar o uso da métrica de tamanho LOC

Relevância: Quanto maior for o número de atributos por componente, mais difícil será para entender o sistema. Quanto maior for o número de atributos, mais difícil será para encontrar os locais que precisam ser modificados durante atividades de evolução. Será mais difícil também para entender a implementação das funcionalidades que se deseja reutilizar durante atividades de reutilização.

Exemplo: A Figura 13 mostra trechos do código, em AspectJ, do aspecto *QueueStateAspect* , que tem três atributos: *empty*, *normal* e *full*. Portanto, o valor de NOA para esse aspecto é três. Esse aspecto faz parte da implementação em AspectJ do padrão de projeto *State* proposta por Hannemann & Kiczales [5].

```
01  public aspect QueueStateAspect {
02
03  protected QueueEmpty  empty  = new QueueEmpty();
04  protected QueueNormal normal = new QueueNormal();
05  protected QueueFull   full   = new QueueFull();
06
07    after(Queue queue): initialization(new()) && target(queue) {
08       queue.setState(empty);
09    }
10
11    after(Queue queue, QueueState qs, Object arg): call(boolean
       QueueState+.insert(Object)) && target(qs) && args(arg) && this(queue) {
        ...
        ...
22    }
23
24    after(Queue queue, QueueState qs): call(boolean QueueState+.removeFirst())
       && target(qs) && this(queue) {
        ...
        ...
35    }
36  }
```

Figura 13 – Código para exemplificar o uso das métricas de tamanho NOA e WOC

5.4.4.
Peso de Operações por Componente (WOC)

Definição: Essa métrica mede a complexidade de um componente em termos de suas operações (métodos e *advices*). Considere um componente C_1 com operações O_1,..., O_n. Sejam c_1,..., c_n as complexidades das operações. Então: WOC = c_1 + ... + c_n. Originalmente, essa métrica não especifica como deve ser medida a complexidade das operações, que deve ser adaptada para os contextos específicos. Neste trabalho especificamente, a medida de complexidade de cada operação é obtida contando o número de parâmetros da operação. A complexidade de uma operação sem parâmetros é igual a um, de uma operação com um parâmetro é dois, e assim por diante. Essa forma de medir a complexidade da operação assume que uma operação com mais parâmetros que uma outra é provavelmente mais complexa. Num aspecto, além dos *advices* e métodos internos ao aspecto, são considerados também os métodos que o aspecto introduz nas classes que ele afeta por meio de *inter-type declarations*. Essa métrica é uma extensão da métrica WMC de Chidamber & Kemerer [6]. Essa extensão é caracterizada pelo fato de que a métrica WOC leva em consideração os *advices* e métodos dos aspectos e os trata da mesma forma que a métrica WMC de Chidamber & Kemerer trata os métodos das classes.

Relevância: Quanto maior for o número e a complexidade das operações por componente, mais difícil será para entender o sistema. Da mesma forma que o número de atributos, quanto maior for o número e a complexidade das operações, mais difícil será para encontrar os locais que precisam ser modificados durante atividades de evolução. Será mais difícil também para entender a implementação das funcionalidades que se deseja reutilizar durante atividades de reutilização.

Exemplo: Ainda analisando o aspecto *QueueStateAspect* da Figura 13, pode-se notar que ele possui três *advices*, ou seja três operações. Mas além das operações, a métrica WOC proposta conta a quantidade de parâmetros de cada operação. O *advice* da linha sete tem um parâmetro, o da linha onze tem três e o da linha 24 tem dois. Portanto o valor da métrica WOC para esse aspecto é 3 + 1 + 3 + 2, que é igual a 9.