UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

LUIZA FIGUEIREDO PAGLIARI

Avaliação Quantitativa de Refatorações Orientadas a Aspectos

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Daltro Nunes
Orientador

Porto Alegre, outubro de 2007

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Pagliari, Luiza Figueiredo

Avaliação Quantitativa de Refatorações Orientadas a Aspectos / Luiza Figueiredo Pagliari. – Porto Alegre: PPGC da UFRGS, 2007.

191 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2007. Orientador: Daltro Nunes.

1. Avaliação de refatorações. 2. Programação orientada a aspectos. 3. Métricas. 4. Refatoração. I. Nunes, Daltro. II. Título.

AGRADECIMENTOS

Em primeiro lugar, agradeço ao meu orientador de Mestrado, professor Daltro Nunes, por me guiar neste difícil processo de grande aprendizado e por propiciar grandes oportunidades durante o período de sua orientação.

Aos meus pais, meus primeiros orientadores da vida e eternos exemplos, por me ensinarem a ser correta e a sempre buscar o melhor de mim. Espero que este trabalho tenha qualidade condizente com tudo o que vocês me ensinaram.

Não posso deixar de agradecer aos meus colegas do grupo Prosoft, pelos ótimos comentários, críticas e sugestões para tornarem este trabalho cada vez melhor. Agradeço especialmente ao Lincoln Rabelo, por estar sempre disposto a ajudar da melhor forma possível e por sempre ter bons conselhos.

Aos pesquisadores citados neste trabalho que se dispuseram a ajudar sempre que foi necessário: Miguel Monteiro, Cláudio Sant'Anna, Eduardo Figueiredo, Josh Kataoka, Jianjun Zhao e especialmente ao Eduardo Piveta, pela ajuda também nas etapas iniciais deste trabalho.

À Nara Figueiredo, por ter feito (bem) o seu trabalho.

E por último, porém absolutamente não menos importante, agradeço à Suzi Camey e à Luciana Nunes por estarem sempre presentes. Não existem palavras para agradecer por tudo o que fizeram para me ajudar a superar os obstáculos que surgiram ao longo deste trabalho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

CBC	Acoplamento Entre Componentes
CDC	Difusão do Interesse por Componentes
CDLOC	Difusão do Interesse por Linhas de Código
CDO	Difusão do Interesse por Operações
DIT	Profundidade da Árvore de Herança
LCOO	Falta de Coesão nas Operações
LOC	Linhas de Código
NOA	Número de Atributos
OA	Orientação/Orientado a Aspectos
OO	Orientação/Orientado a Objetos
POA	Programação Orientada a Aspectos
POO	Programação Orientada a Objetos
VS	Tamanho do Vocabulário
WOC	Peso de Operações por Componente

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Diversas refatorações têm sido propostas nos últimos anos para os mais variados paradigmas de programação, dentre eles o orientado a objetos e o orientado a aspecto. Seus impactos em atributos de qualidade são diversos, porém nem sempre a descrição original da refatoração apresenta todos os impactos que ela pode ter. Assim, é importante definir métodos de avaliação de refatorações para obter seus impactos em diferentes atributos de qualidade. A literatura apresenta trabalhos que utilizam métricas de software para fazer isso através de medições antes e depois de refatorar o código, porém este tipo de avaliação não permite obter conclusões válidas para todos os contextos em que a refatoração for aplicada. Outros trabalhos obtêm impactos abrangentes de refatorações orientadas a objetos, porém não foram encontrados métodos aplicáveis a refatorações orientadas a aspectos. Assim, este trabalho propõe uma forma de avaliar refatorações orientadas a aspectos para obter impactos abrangentes de sua aplicação, definindo um processo para avaliar refatorações orientadas a aspectos através do uso de métricas. Ele divide as etapas da refatoração em alterações pontuais e mede o impacto dessas alterações nos valores de um conjunto de métricas. O processo é usado para avaliar um conjunto de refatorações existentes na literatura definidas com o objetivo de extrair interesses transversais para aspectos. Para isso, são usados como critério de avaliação métricas para medir separação de interesses, tamanho, acoplamento e coesão do software. Como resultado, tem-se o impacto da refatoração em cada uma das métricas selecionadas, o que permite saber como o código será alterado antes mesmo de aplicar a refatoração.

Palavras-chave: Avaliação de refatorações, programação orientada a aspectos, métricas, refatoração.

Quantitative Assessment of Aspect-Oriented Refactorings

ABSTRACT

Several software refactorings have been proposed on the last years for different programming paradigms, like object-oriented and aspect-oriented. They have several impacts on quality attributes, but their descriptions don't describe all of these impacts, so it is important to have methods to assess refactorings to get their impacts on different quality attributes. Some papers apply software metrics on code before and after using the refactoring, but this kind os assessment avoids getting valid conclusions for all contexts where the refactoring can be used. Other papers propose assessment methods that get general conclusions for object-oriented refactorings, but no methods were found for assessing aspect-oriented refactorings. This work presents a process to assess aspect-oriented refactorings using software metrics to get their impacts on different quality attributes. It splits the refactoring steps into basic changes and measures the effects of these changes on some metrics. The process is used to assess some aspect-oriented refactorings for extracting crosscutting concerns into aspects, having as criteria software metrics to measure separation of concerns, size, coupling and cohesion. As result, we have the impact of the refactoring on each of the metric chosen, and know the consequences of the refactoring on the code before applying it.

Keywords: Refactoring assessment, aspect-oriented programming, metrics, refactoring.

TRADUÇÕES PARA O PORTUGUÊS

A orientação a aspectos é uma área relativamente recente na literatura, e mais ainda quando se trata de trabalhos escritos em português. Por esse motivo não se tem até o momento palavras em português consolidadas para traduzir os termos usados na literatura. Alguns trabalhos utilizam os termos originais em inglês, outros optam por traduções feitas pelo próprio autor. Existem iniciativas para tentar estabelecer um vocabulário comum em Português para a orientação a aspectos, dentre elas a do Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (WASP), cujos resultados estão disponíveis na internet (WASP, 2004).

Neste trabalho optou-se por evitar os termos em inglês e usar apenas a língua portuguesa, tanto nos termos relacionados à orientação a aspectos quanto nos nomes de refatorações. Seguem abaixo os termos da literatura ou nome da refatoração em inglês e a respectiva tradução adotada neste texto.

Advice: Adendo

Aspect: Aspecto

Change Abstract Class to Interface: Trocar Classe Abstrata por Interface

Code scattering: Espalhamento de código

Code tangling: Entrelaçamento de código

Concern: Interesse

Crosscut: Entrecortar

Crosscutting concern: Interesse transversal ou Interesse entrecortante

Encapsulate Field: Encapsular Atributo

Extract Feature into Aspect: Extrair Funcionalidade para Aspecto

Extract Fragment into Advice: Extrair Fragmento para Adendo

Extract Inner Class to Standalone: Extrair Classe Interna para Autônoma

Extract Method: Extrair Método

Generalise Target Type with Marker Interface: Generalizar Tipo-Alvo com Interface de Marcação

Inline Class within Aspect: Internalizar Classe em Aspecto

Inline Interface within Aspect: Internalizar Interface em Aspecto

Inline Method: Internalizar Método

Inter-type declaration: Declaração intertipo

Join point: Ponto de junção

Marker interface: Interface marcadora

Move Field from Class to Inter-type: Mover Atributo de Classe para Inter-tipo

Move Method from Class to Inter-type: Mover Método de Classe para Inter-tipo

Partition Constructor Signature: Particionar Assinatura de Construtor

Pointcut: Conjunto (de pontos) de junção

Pull Up Field: Subir Campo na Hierarquia

Push Down Field: Descer Campo na Hierarquia

Remove Parameter: Remover Parâmetro

Replace Implements with Declare Parents: Trocar `implements` por `declare parents`

Role: Papel

Self Encapsulate Field: Auto Encapsular Atributo

Separation of concerns: Separação de interesses

Split Abstract Class into Aspect and Interface: Dividir Classe Abstrata em Aspecto e Interface

1 INTRODUÇÃO

Durante o ciclo de vida de um software, é inevitável que ele evolua. No entanto, à medida que o software é estendido, modificado e adaptado aos novos requisitos, seu código se torna mais complexo e se afasta de seu projeto original, o que diminui a qualidade do programa (MENS; TOURWÉ, 2004). Uma das técnicas que reduz a complexidade do software e ao mesmo tempo melhora sua qualidade interna são as *refatorações de software*. Refatorações são transformações do software que melhoram a estrutura interna do programa sem que haja alteração do seu comportamento externo (FOWLER, 1999). Elas podem ser propostas para melhorar programas de diversas maneiras: algumas eliminam código repetido, outras elevam o nível de abstração, há ainda refatorações para aumentar o reúso ou reduzir o acoplamento entre módulos, etc.

Particularmente, as chamadas refatorações orientadas a aspectos (OA) são necessárias para extrair aspectos a partir de código onde eles ainda não existem ou para melhorar a organização interna de programas com a presença de aspectos. Aspectos são construções da Programação Orientada a Aspectos (POA) (KICZALES et al., 1997) usadas para reunir em um único módulo todo o código relacionado a um mesmo interesse transversal, evitando que esse interesse fique espalhado pelos componentes do software e que sua implementação fique entrelaçada à implementação de outros interesses.

Os impactos de uma refatoração podem ir além dos listados em sua descrição, isto é, também podem ser modificadas (positiva ou negativamente) outras características do software que não as identificadas como características a serem alteradas pela refatoração. Por exemplo, a refatoração *Extrair Método* (FOWLER, 1999) lista como objetivos (1) eliminar código duplicado e (2) tornar métodos mais fáceis de serem compreendidos, porém segundo Bois e Mens (2003) ela também pode aumentar a vulnerabilidade do software a propagações de alterações na classe, embora este impacto secundário não seja apontado na descrição original da refatoração.

Avaliar uma refatoração permite conhecer seus diversos impactos e, conseqüentemente, saber de antemão como os atributos de qualidade podem ser modificados ao aplicá-la em um programa, obtendo-se assim as vantagens e desvantagens de seu uso. Isso permite que o desenvolvedor tenha ciência das conseqüências da refatoração antes mesmo de aplicá-la, evitando que a mesma tenha que ser desfeita caso os atributos de qualidade deteriorados sejam mais relevantes do que os atributos melhorados pela refatoração. Outra vantagem de se conhecer previamente os impactos da refatoração é permitir a indicação de sua aplicação quando os atributos de qualidade que ela melhora forem identificados como deficientes em um software. Assim, é importante definir métodos de avaliação de refatorações para obter seus impactos em diferentes atributos de qualidade, uma vez que nem todos são apontados na descrição da refatoração.

Existem trabalhos na literatura que usam métricas de software para indicar possíveis

impactos de refatorações em atributos de qualidade. Alguns, como (BENN et al., 2005), fazem um estudo de caso onde as medições são feitas antes e depois da refatoração ser aplicada, obtendo-se os impactos subtraindo o valor final de cada métrica do valor inicial. Esses resultados, no entanto, são válidos apenas para o caso estudado e não abrangem outros casos em que a refatoração é aplicada. Outros estudos, como (TAHVILDARI; KONTOGIANNIS, 2004; BOIS; MENS, 2003), analisam as alterações estruturais que a refatoração faz no software e verificam como o valor das métricas é modificado por essas alterações, obtendo assim impactos abrangentes [1] da refatoração avaliada. No entanto, as propostas encontradas na literatura para obtenção de impactos abrangentes podem ser utilizadas apenas com refatorações orientadas a objetos (OO), não tendo sido encontrados métodos que permitam encontrar os impactos de refatorações OA.

Assim, o objetivo deste trabalho é propor uma forma de avaliar refatorações orientadas a aspectos para obter impactos abrangentes de sua aplicação. Para isso, é proposto um processo de avaliação baseado na divisão dos passos da refatoração em pequenas modificações, chamadas de *passos básicos*. Este processo, ao contrário das outras propostas de avaliação de refatorações existentes, não está restrito a um único paradigma de programação das refatorações avaliadas. Além disso, ele é dividido em processos menores que podem ser reusados em avaliações futuras, diminuindo o esforço necessário para completar a avaliação de uma refatoração à medida que novas refatorações são avaliadas.

O processo proposto é aplicado em algumas das refatorações orientadas a aspectos existentes na literatura, tendo como critério de avaliação métricas para avaliar *tamanho*, *acoplamento*, *coesão* e *separação de interesses* do software. Para avaliar as refatorações de acordo com as etapas do processo, é necessário decompor cada refatoração em passos mais simples, por isso este trabalho também apresenta um catálogo com todas as pequenas alterações que as refatorações avaliadas fazem no software.

Este trabalho está organizado da seguinte forma: no capítulo 2 são descritos trabalhos relacionados e conceitos fundamentais para o entendimento do trabalho, ligados a temas como programação orientada a aspectos, refatoração de software e medições de atributos de qualidade; no capítulo 3 detalha-se o processo de avaliação; o capítulo 4 contém o catálogo com a definição e medição dos impactos dos passos básicos; o capítulo 5 descreve a aplicação do processo nas avaliações das refatorações OA; e por fim, no capítulo 6 são apresentadas as conclusões deste trabalho.

[1] o termo "abrangente" é usado neste trabalho para caracterizar as formas de avaliação de refatorações que apresentam impactos válidos para diferentes casos em que a refatoração é usada, ou seja, não se restringem a um único código no qual a refatoração foi aplicada, como ocorre com os estudos de caso.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo explica os principais conceitos necessários para a compreensão deste trabalho. Além de descrever os conceitos presentes nos trabalhos fundamentais de áreas como *Programação Orientada a Aspectos* e *Refatoração de Software*, este capítulo também mostra como o presente trabalho está contextualizado nos trabalhos existentes na literatura.

2.1 Programação Orientada a Aspectos

Projetos de software são mais facilmente planejados e mantidos quando separados em partes menores, com funcionalidades bem definidas e separação de código de acordo com as propriedades que elas manipulam. As linguagens mais comumente utilizadas, sejam elas orientadas a objetos, procedurais ou funcionais, provêm mecanismos que permitem ao programador efetuar essa divisão através da definição de abstrações de sub-unidades do sistema, e da composição dessas abstrações de diferentes formas para produzir o sistema como um todo. Dentre esses mecanismos, pode-se citar sub-rotinas, procedimentos, funções, objetos, classes e APIs, todos eles formas de efetuar a decomposição funcional do sistema, isto é, dividir o sistema em unidades de comportamento (KICZALES et al., 1997).

Entretanto, existem propriedades que não se enquadram em componentes de decomposição funcional, tais como padrões de acesso de memória, tratamento de erros e sincronização de objetos concorrentes. São propriedades transversais, com algo em comum, e que normalmente se encontram espalhadas em diversos componentes do sistema (KICZALES et al., 1997). Kiczales et al. (1997) chamam essas propriedades de *aspectos*.

Apesar de aspectos poderem ser visualizados e analisados isoladamente (do resto do sistema), sua implementação usando linguagens orientadas a objetos ou estruturadas não permite esse isolamento. Como resultado, obtém-se um programa onde um mesmo interesse está disperso em várias partes do sistema (espalhamento de código), e ao mesmo tempo uma unidade modular fica responsável por mais de um interesse (entrelaçamento de código). Estas duas características prejudicam a modularidade do programa e, portanto, são indesejadas.

Assim, Kiczales et al. (1997) propõem um novo paradigma de programação chamado Orientação a Aspectos (OA), o qual tem como objetivo "auxiliar o programador na separação clara de componentes e aspectos, provendo mecanismos que tornam possível abstraí-los e compô-los para produzir o sistema global" (KICZALES et al., 1997). Ele provê mecanismos de linguagem que capturam explicitamente estrutura entrecortante, tornando possível programar interesses transversais de forma modular, e atingir os benefícios usuais de melhorias de modularidade: código mais simples, fácil de desenvolver e manter, e que

possui grande potencial de reúso. Isso contrasta com programação funcional, orientada a objetos ou procedural, que auxiliam programadores na separação apenas de componentes uns dos outros.

Existem diversas linguagens para desenvolvimento de software orientado a aspectos, tanto para programas cujos componentes são implementados em linguagens orientadas a objetos (Java, C++, Smalltalk) quanto para componentes implementados em linguagens de outros paradigmas (C, Haskell, Lisp). As mais conhecidas e utilizadas nos exemplos da literatura são AspectJ (KICZALES et al., 2001; ASPECTJ, 2007), HyperJ (OSSHER; TARR, 2001; HYPERJ, 2007) e AspectWerkz (BONéR, 2004; ASPECTWERKZ, 2007)[1], em especial a primeira. Uma vez que todas as refatorações OA avaliadas neste trabalho são definidas para código AspectJ, torna-se necessário descrever os conceitos fundamentais desta linguagem para auxiliar na compreensão das avaliações, o que é feito na seção a seguir.

2.1.1 Captura de estrutura entrecortante em AspectJ

A linguagem AspectJ é implementada como uma extensão da linguagem Java, adicionando construções que dão suporte à implementação modular de interesses transversais. Essas construções, chamadas de *conjuntos de pontos de junção*, *adendos*, *declarações inter-tipo* e *aspectos*, permitem definir novas operações em tipos existentes, alterar a hierarquia das classes, modificar o comportamento do programa em pontos específicos, dentre outros (ASPECTJ, 2007).

Conjunto de pontos de junção

Um *ponto de junção* é um ponto bem-definido na execução de um programa (KICZALES et al., 2001), e pode ser uma chamada de método, acesso a atributo, lançamento de exceção, inicialização de um objeto, dentre outros (ASPECTJ, 2007). *Conjuntos de pontos de junção* são construções sintáticas para agrupar uma coleção de pontos de junção. Eles não têm comportamento, apenas especificam quais pontos do programa são entrecortados pelo comportamento transversal do aspecto.

O exemplo abaixo mostra a declaração de um conjunto de pontos de junção em AspectJ, o qual contém as chamadas aos métodos `setXY(int,int)` do componente `ElementoDeFigura`, `setX(int)` e `setY(int)` de `Ponto`, e `setP1(int)` e `setP2(int)` de `Linha`:

```
pointcut mover():
    call(void ElementoDeFigura.setXY(int,int)) ||
    call(void Ponto.setX(int))                 ||
    call(void Ponto.setY(int))                 ||
    call(void Linha.setP1(Ponto))              ||
    call(void Linha.setP2(Ponto));
```

A descrição dos pontos que compõem o conjunto é feita de forma textual, por meio de expressões regulares que podem usar ou não os caracteres-coringa definidos em AspectJ (‘`*`’, ‘`+`’ e ‘`..`’) para flexibilizar a definição do conjunto. O exemplo acima não usa nenhum caracter-coringa e especifica os pontos capturados de forma direta, porém no próximo exemplo esses caracteres especiais são usados para agrupar diversos métodos em uma única condição do conjunto de pontos de junção. No exemplo seguinte, o conjunto de pontos de junção `criar()` reúne as chamadas a todos os métodos da classe `Figura`

[1] todas usadas com componentes Java (JAVA, 2007)

sem parâmetros que retornam um objeto do tipo `ElementoDeFigura` e com nomes que iniciam com “criar”:

```
pointcut criar():
    call(ElementoDeFigura Figura.criar*());
```

Adendos

Adendos são construções similares a métodos de classes, e são usados para definir o código a ser executado em cada um dos pontos de junção do conjunto relacionado ao adendo (KICZALES et al., 2001). Esses códigos podem ser executados antes, depois ou mesmo no lugar do ponto de junção. No exemplo abaixo, o adendo define que será impressa uma mensagem antes de cada ponto de junção do conjunto `criar()`:

```
before(): criar() {
    System.out.println("Vai criar uma nova figura...");
```

Declarações inter-tipos

Além de alterar o comportamento de uma classe por meio de conjuntos de pontos de junção e adendos, também é possível modificar sua estrutura estática. Através de construções chamadas *declarações inter-tipo*, é possível modificar a herança das classes e adicionar a elas atributos e métodos. No exemplo abaixo, o atributo `texto` e o método `display()` são adicionados a `Ponto`:

```
private JTextField Ponto.texto = new JTextField(20);

private void Ponto.display() {
    texto.setText(toString());
}
```

Já neste próximo exemplo a herança de `Ponto` e `Tela` é modificada: a classe `Ponto` torna-se herdeira de `Observable` e `Tela` passa a implementar a interface `Observer`:

```
declare parents: Ponto extends Observable;

declare parents: Tela implements Observer;
```

Aspecto

Aspecto é a unidade modular de AspectJ que contém a implementação de um interesse transversal (KICZALES et al., 2001), e sua declaração é bastante similar à de uma classe em Java. Aspectos contêm declarações de conjuntos de pontos de junção, de adendos e declarações inter-tipos, além de todas as declarações permitidas em classes [2].

O exemplo a seguir, explicado na seção 5.2.11 deste trabalho, mostra um aspecto com algumas das construções introduzidas por AspectJ descritas anteriormente:

[2] A única exceção são construtores com parâmetros ou que lancem exceções. Esta restrição existe porque aspectos são implementados em AspectJ de forma a serem instanciados automaticamente (ASPECTJ, 2007)

```
public aspect WindowView {
    private JLabel TangledStack._label = new JLabel("Stack ");
    private JTextField TangledStack._text = new JTextField(20);

    public TangledStack.new(JFrame frame) {
        this();
        frame.getContentPane().add(_label);
        _text.setText("[]");
        frame.getContentPane().add(_text);
    }
    private void TangledStack.display() {
        _text.setText(toString());
    }

    pointcut stateChange(TangledStack stack):
        (execution(public void stack.TangledStack.push(Object))
        || execution(public void stack.TangledStack.pop()))
        && this(stack);

    after(TangledStack _this) returning :
        stateChange(_this) {
        _this.display();
    }
}
```

2.2 Refatoração de Software

Durante o ciclo de vida de um software, é inevitável que ele evolua. No entanto, à medida que o software é estendido, modificado e adaptado aos novos requisitos, seu código se torna mais complexo e se afasta de seu projeto original, o que diminui a qualidade do programa (MENS; TOURWÉ, 2004). Periodicamente, portanto, devem ser feitas reformulações no software que melhorem sua qualidade interna sem que com isso sejam introduzidos erros no programa. Uma maneira de fazer isso é através das *refatorações de software*. Fowler (1999) define este conceito como "o processo de alteração de um sistema de software, de modo que o comportamento externo do código não mude, mas que sua estrutura interna seja melhorada[3]". Como conseqüência da aplicação de refatorações, obtém-se um código mais fácil de entender, melhora-se o projeto do software e encontra-se falhas mais facilmente, o que resulta em uma maior velocidade de programação (FOWLER, 1999).

Além do uso para aumentar a qualidade do software durante sua evolução, refatorações também podem ser usadas no contexto da reengenharia, que é a "avaliação e alteração de um sistema para reconstituí-lo em uma nova forma e na subseqüente implementação desta nova forma" (CHIKOFSKY; CROSS, 1990). Segundo Mens e Tourwé (2004), "neste contexto a refatoração pode ser usada para converter código legado ou deteriorado em uma forma mais modular ou estruturada, ou mesmo migrar código para uma linguagem diferente ou até outro paradigma".

Existem pares de refatorações que executam modificações inversas no código, como *Extrair Método* e *Internalizar Método* ou *Subir Campo na Hierarquia* e *Descer Campo na Hierarquia* (FOWLER, 1999). Elas são chamadas de "refatorações opostas", e são muito importantes por permitirem ao desenvolvedor desfazer mais facilmente uma refatoração quando os resultados não forem adequados (MONTEIRO, 2005). Caso uma refatoração não tenha uma oposta definida, ela até poderá ser desfeita, porém será um processo muito

[3] O termo *melhorada* é usado no sentido de tornar o software mais fácil de entender e menos custoso de ser modificado, e não para casos de melhorias no desempenho do software. Fowler explica que, apesar de alterações que visem otimizações de desempenho geralmente não alterarem o comportamento observável de um componente, muitas vezes elas tornam o código mais difícil de ser compreendido.

mais complexo e propenso à inserção de erros no programa.

Formato das Refatorações

Para descrever uma refatoração, diversos trabalhos da literatura (FOWLER, 1999; GARCIA et al., 2004; MONTEIRO, 2005; REFACTORING, 2007) usam o formato definido por Fowler (1999), o qual possui as seguintes partes:

Nome: Identificador da refatoração. O nome é importante para construir um vocabulário comum, e é uma forma de referenciar posteriormente a refatoração descrita.

Resumo: Descrição sucinta da situação na qual a refatoração é necessária e do que ela faz. Pode incluir um esboço para mostrar um exemplo simples de antes e depois, em código ou diagrama UML.

Motivação: Deve mostrar porque a refatoração deve ser feita e as circunstâncias nas quais ela não deve ser aplicada.

Mecânica: Descrição passo-a-passo de como executar a refatoração. Cada um desses passos deve ser tão pequeno quanto possível, a fim de minimizar a introdução de erros. Passos de refatorações podem usar outras refatorações ou mesmo a própria refatoração recursivamente.

Exemplos: Mostram um uso bem simples da refatoração para ilustrar como ela funciona e contêm explicações do porquê da execução dos passos. Podem ser omitidos no caso de refatorações muito simples, onde exemplos não acrescentem muito à descrição.

No presente trabalho, uma descrição detalhada da mecânica da refatoração é extremamente importante para sua avaliação, pois o processo aqui proposto utiliza as informações contidas nesta parte para estimar com maior precisão o impacto que a refatoração tem nas métricas selecionadas. Por outro lado, outras informações presentes na descrição original da refatoração podem não ser tão relevantes para sua avaliação. Assim, neste trabalho optou-se por reproduzir na descrição das refatorações (capítulo 5) apenas as informações consideradas relevantes.

2.2.1 Refatorações Orientadas a Aspectos

Apesar de refatorações de software serem freqüentemente citadas no contexto da orientação a objetos, também pode-se refatorar software em outros contextos, como na orientação a aspectos. Assim como os programas orientados a objetos, programas orientados a aspectos evoluem e sofrem manutenção, por isso eles devem ser periodicamente refatorados para melhorar sua qualidade interna e reduzir sua complexidade. Refatorações OA também são necessárias quando se deseja reestruturar código legado não-OA para que o programa passe a ser implementado usando o paradigma OA (processo também referenciado na literatura como *extração de aspectos*).

Diversos trabalhos na literatura apresentam diferentes refatorações para lidar com código orientado a aspectos. Iwamoto e Zhao (2003) identificam 24 potenciais refatorações particulares a programas OA, sem no entanto detalhar sua mecânica ou em que situações elas devem ser aplicadas. Os autores afirmam que um trabalho futuro conteria essas descrições, porém o referido trabalho nunca foi publicado.

Hanenberg, Oberschulte e Unland (2003) propõem adaptações para 3 refatorações OO quando estas forem aplicadas em programas com a presença de aspectos, bem como descreve 3 pequenas refatorações OA para extrair algumas das construções propostas por AspectJ ou melhorar sua compreensão.

Garcia et al. (2004) descrevem 10 refatorações simples para serem usadas tanto na extração de aspectos quando para melhoria de código OA. Algumas dessas refatorações modificam as novas construções de AspectJ de forma similar às construções da orientação a objetos modificadas pelas refatorações de (FOWLER, 1999): Garcia et al. definem *Renomear Conjunto de Pontos de Junção* e *Renomear Aspecto*, enquanto que Fowler (1999) apresenta *Renomear Método*; a refatoração OA *Subir Adendo/Conjunto de Ponto de Junção na Hierarquia* corresponde à refatoração OO *Subir Método na Hierarquia*; e *Internalizar Definição de Conjunto de Ponto de Junção* equivale a *Internalizar Método*.

Hanneman, Murphy e Kiczales (2005) apresentam refatorações que modularizam alguns padrões de projeto em aspectos, através do mapeamento dos papéis dos padrões de projeto em componentes do sistema. Além delas, outras refatorações mais simples são citadas no trabalho, porém os autores apenas descrevem resumidamente seus objetivos, sem dar maiores detalhes de sua mecânica.

Monteiro e Fernandez (2005a; 2006) apresentam o conjunto mais completo de refatorações OA, descritas com maiores detalhes em (MONTEIRO, 2005). São 28 refatorações propostas para código Java e AspectJ, divididas em quatro grupos: 10 refatorações para extração de aspectos de código legado, 6 refatorações para organização interna de aspectos, 11 refatorações para extração e transferência de códigos repetidos na hierarquia de aspectos, e uma refatoração para lidar com limitações de código legado. Em relação às outras propostas, as refatorações apresentadas por Monteiro e Fernandez são mais detalhadas e abrangem um conjunto maior de situações em que podem ser aplicadas. Por exemplo, suas refatorações para extração de aspectos não são específicas para modularização somente de padrões de projeto, assim como não se resumem a mover para aspectos apenas atributos, métodos ou trechos de código, mas também prevêem a movimentação de classes internas e interfaces.

Como pode ser visto pelas descrições acima, o conjunto de refatorações OA existentes na literatura é bastante extenso. Para selecionar as refatorações a serem avaliadas neste trabalho são usados os seguintes critérios: (1) maior detalhamento na descrição da refatoração, especialmente de sua mecânica; (2) maior abrangência de situações em que a refatoração pode ser aplicada; e (3) refatorações usadas com maior freqüência. Os dois primeiros critérios restringem o conjunto inicial às 28 refatorações propostas por Monteiro e Fernandez (2005a; 2006). Desse subconjunto, as mais freqüentemente usadas são, segundo os próprios autores, as refatorações para extração de aspectos. Dessa forma, pelo terceiro critério, o conjunto inicial de refatorações OA escolhido para a avaliação neste trabalho fica reduzido às refatorações para extração de aspectos propostas por Monteiro e Fernandez. Como algumas das refatorações selecionadas usam refatorações menores em sua mecânica e uma vez que para avaliar uma refatoração é necessário avaliar cada um de seus passos (conforme descrito no capítulo 3), a esse conjunto inicial são adicionadas ainda outras cinco refatorações: uma OA definida por Monteiro (2005) e quatro OO definidas por Fowler (1999).

2.3 Avaliação de Refatorações

Na literatura, o termo "refatorar" muitas vezes é usado apenas para referenciar a aplicação de uma seqüência de passos que melhoram a estrutura interna do software. No entanto, o processo de refatoração é muito mais abrangente do que isso. Mens e Tourwé (2004) listam outras cinco atividades deste processo, no qual a aplicação dos passos é apenas uma etapa intermediária:

1. Identificar uma oportunidade de refatoração;
2. Determinar quais refatorações devem ser aplicadas;
3. Garantir que as refatorações não alterarão o comportamento;
4. Aplicar as refatorações;
5. Avaliar os efeitos das refatorações em características de qualidade do software (ex.: complexidade, manutenibilidade) ou do processo (ex.: produtividade, custos);
6. Manter a consistência entre o código refatorado e outros artefatos de software (documentação, especificação de requisitos, testes, etc.).

Os trabalhos citados anteriormente com propostas de novas refatorações têm como preocupação principal descrever como aplicar as refatorações. Alguns deles também descrevem formas de identificar oportunidades de aplicação das refatorações propostas (FOWLER, 1999; MONTEIRO, 2005; MONTEIRO; FERNANDES, 2005a, 2006) e de garantir a preservação do comportamento ao aplicar as refatorações (FOWLER, 1999; HANENBERG; OBERSCHULTE; UNLAND, 2003), porém em nenhum desses trabalhos há uma avaliação do impacto que as refatorações têm em características de qualidade do software ou do processo.

Para medir ou estimar o impacto de refatorações em atributos de qualidade do software, uma das técnicas que podem ser usadas são as métricas (MENS; TOURWÉ, 2004). Existem diversos trabalhos na literatura com medições ou estimativas de impactos de refatorações através do uso desta técnica. Stroggylos e Spinellis (2007) e Moser et al. (2006) medem os impactos que etapas de refatorações têm ao longo do desenvolvimento de alguns programas. Estes dois trabalhos têm como objetivo avaliar apenas se etapas de reorganização interna do software melhoram a qualidade do mesmo, por isso nenhuma refatoração específica é avaliada, mas sim os conjuntos de alterações dos softwares classificados pelos desenvolvedores como "etapas de refatoração".

Benn et al. (2005) medem os impactos que algumas refatorações descritas por Garcia et al. (2004) têm ao extrair aspectos de código OO. O objetivo principal do trabalho é verificar se a orientação a aspectos melhora a qualidade do software, por isso os autores fazem um estudo de caso no qual as refatorações são aplicadas em código originalmente OO para gerar código OA. A partir da comparação dos valores de métricas OO e OA antes e depois da refatoração, os autores concluem que de fato o código OA é superior ao OO em relação a atributos de software como acoplamento, coesão, complexidade e tamanho. No entanto, eles ressaltam que esta conclusão se aplica apenas ao estudo de caso do trabalho, sendo necessário um estudo mais geral para determinar se esta é a conclusão para todos os casos.

Além de trabalhos que medem o impacto de refatorações em casos específicos, como os citados acima, outros trabalhos propõem formas de obter estimativas abrangentes do

quanto determinadas métricas são alteradas ao efetuar mudanças no código. Esses trabalhos obtêm conclusões válidas para diferentes casos em que as refatorações podem ser usadas, e portanto não têm a limitação que os trabalhos citados anteriormente possuem.

Um desses trabalhos é (TAHVILDARI; KONTOGIANNIS, 2004), cujo objetivo é prover indicações de que tipo de transformação deve ser feita em um software para melhorar determinados atributos de qualidade. Para isso, os autores agrupam refatorações OO em meta-padrões de transformação do código, como transformações de *Abstração*, de *Extensão*, de *Movimentação* e de *Encapsulamento*, e avaliam se cada meta-padrão melhora ou piora algumas métricas OO relacionadas aos atributos de qualidade selecionados. Esta abordagem, no entanto, só pode ser aplicada nas refatorações que se enquadram nos meta-padrões definidos pelos autores, o que limita sua abrangência a um subconjunto das refatorações OO existentes na literatura.

Outra proposta é apresentada por Bois e Mens (2003), que definem um formalismo para descrever o impacto de refatorações na estrutura de programas OO. Os autores representam programas através de árvores sintáticas abstratas estendidas e descrevem as alterações que refatorações fazem no software através de modificações nessas árvores. No entanto, a forma escolhida para representar os programas limita as avaliações às métricas que afetam apenas as estruturas representadas na árvore – métricas como a Complexidade Ciclomática (COPPICK; CHEATHAM, 1992) não podem ser usadas na avaliação por falta de informação no modelo sobre o fluxo do programa (BOIS; MENS, 2003). Este modelo também limita a aplicabilidade da abordagem a refatorações OO porque contempla apenas as estruturas presentes na orientação a objetos: para avaliar refatorações OA, é necessário estender o modelo, o que os próprios autores afirmam que "torna o modelo ainda mais complexo".

Assim, como todas as propostas de avaliação de refatorações encontradas na literatura possuem limitações quando adaptadas para avaliar refatorações OA, optou-se por propor uma nova maneira de avaliação de refatorações. O processo apresentado neste trabalho (capítulo 3) é definido para que (1) refatorações OA possam ter seus impactos obtidos através de avaliações quantitativas e (2) os resultados obtidos sejam abrangentes, isto é, os impactos nos valores das métricas medidas não sejam restritos a um único caso em que a refatoração for aplicada.

2.3.1 Métricas para Programas Orientados a Aspectos

Para avaliar refatorações OA, é necessário selecionar um conjunto de métricas adequadas para medir atributos de qualidade desejáveis em programas com a presença de aspectos. Sant'Anna et al. (2003) propõem um conjunto de métricas aplicáveis em software OO e OA que adapta e estende algumas métricas definidas para programas procedurais ou OO, tais como Complexidade Ciclomática (COPPICK; CHEATHAM, 1992), as métricas CK (CHIDAMBER; KEMERER, 1994) e Número de Linhas de Código (FENTON; PFLEEGER, 1997), de forma a permitir que programas OO e OA sejam avaliados de acordo com quatro atributos de qualidade baseados em princípios bem-estabelecidos da engenharia de software: tamanho, acoplamento, coesão e separação de interesses (FIGUEIREDO; STAA, 2005). *Tamanho* é uma medida física da extensão do projeto e do código do sistema de software (LI; HENRY, 1993). *Acoplamento* indica a força das interconexões entre os componentes do sistema – quanto mais alto o acoplamento, mais fortes as interconexões e, portanto, maior a dependência das unidades do programa (SOMMERVILLE, 2001). A *coesão* de um componente é a medida da proximidade do relacionamento entre seus componentes internos (SOMMERVILLE, 2001). E a *separa-*

ção de interesses se refere à habilidade para identificar, encapsular e manipular as partes do software que são relevantes para determinado interesse (TARR et al., 1999).

Para possibilitar a aplicação das mesmas métricas em sistemas OO e OA, é necessário homogeneizar o modo de contagem das novas formas de abstração presentes na OA (seção 2.1.1). Assim, as métricas de Sant'Anna et al. tratam aspectos, classes e interfaces pelo termo genérico **componentes**, enquanto que métodos e adendos são chamados de **operações**.

Cada métrica é descrita brevemente a seguir, porém informações mais completas podem ser obtidas em (SANT'ANNA et al., 2003; SANT'ANNA, 2004; FIGUEIREDO; STAA, 2005; GARCIA et al., 2005).

2.3.1.1 Métricas de Separação de Interesses

Sant'Anna et al. (2003) definem três métricas de separação de interesses: Difusão do Interesse por Componentes (CDC[4]), Difusão do Interesse por Operações (CDO[5]) e Difusão do Interesse por Linhas de Código (CDLOC[6]). Essas métricas geram valores por interesse, diferentemente da maioria das métricas de acoplamento, coesão e tamanho, que geram valores por componente (seções 2.3.1.2, 2.3.1.3, 2.3.1.4).

Difusão do Interesse por Componentes (CDC) – possui duas parcelas: (i) o número de componentes cujo propósito principal é contribuir para a implementação do interesse avaliado; e (ii) o número de componentes que fazem referência aos componentes principais do interesse.

Difusão do Interesse por Operações (CDO) – possui duas parcelas: (i) o número de operações cujo propósito principal é contribuir para a implementação do interesse avaliado; e (ii) o número de operações que acessam alguma das operações principais do interesse.

Difusão do Interesse por Linhas de Código (CDLOC) – conta o número de pontos de transição entre o interesse avaliado e os outros interesses do sistema através das linhas de código. Para fazer essa contagem é necessário que as linhas de código que implementam o interesse sejam sombreadas, formando áreas sombreadas e não-sombreadas no código. Os pontos de transição são os pontos em que há uma mudança de um tipo de área para outro.

2.3.1.2 Métricas de Acoplamento

Duas métricas de acoplamento são definidas por Sant'Anna et al.: Acoplamento Entre Componentes (CBC[7]) e Profundidade da Árvore de Herança (DIT[8]).

Acoplamento Entre Componentes (CBC) – estendendo a métrica CBO definida por Chidamber e Kemerer (1994), esta métrica mede quantos componentes estão acoplados a cada componente do sistema, isto é, conta o número de componentes usados em declarações de atributos, parâmetros de operações, tipos de retorno e variáveis locais; conta também, para o caso dos aspectos, os componentes que são interceptados pelo aspecto ou que tenham atributos e métodos introduzidos ou acessados pelo aspecto. Se um componente é definido dentro de outro, o acoplamento entre eles não é considerado no cálculo de CBC, assim como se um componente estiver acoplado a outro de diversas

[4]sigla para o nome em inglês *Concern Diffusion over Components*
[5]sigla para o nome em inglês *Concern Diffusion over Operations*
[6]sigla para o nome em inglês *Concern Diffusion over Lines of Code*
[7]sigla para o nome em inglês *Coupling Between Components*
[8]sigla para o nome em inglês *Depth of Inheritance Tree*

formas, o acoplamento entre eles será contado apenas uma vez.

Profundidade da Árvore de Herança (DIT) – estende a métrica de mesmo nome definida por Chidamber e Kemerer (1994). Mede a profundidade na árvore de herança de classes e aspectos, isto é, a maior distância entre a raiz da árvore e as classes e aspectos do software.

2.3.1.3 Métricas de Coesão

Sant'Anna et al. define apenas uma métrica para obter informações sobre a coesão do sistema: Falta de Coesão nas Operações (LCOO[9]).

Falta de Coesão nas Operações (LCOO) – também uma extensão de uma métrica definida em (CHIDAMBER; KEMERER, 1994), esta métrica mede a quantidade de pares de métodos/adendos que não acessam pelo menos um atributo em comum. Seu valor é obtido pela contagem do número de pares de operações que não compartilham nenhum atributo, menos o número de pares de operações que compartilham atributos. Caso o resultado seja um número negativo, assume-se LCOO como zero.

2.3.1.4 Métricas de Tamanho

As métricas de tamanho tratam do tamanho do software sob diferentes perspectivas (SANT'ANNA et al., 2003). Fazem parte do conjunto de métricas de tamanho quatro métricas derivadas de outros trabalhos: Tamanho do Vocabulário (VS[10]), Linhas de Código (LOC[11]), Número de Atributos (NOA[12]) e Peso de Operações por Componente (WOC[13]).

Tamanho do Vocabulário (VS) – conta o número de componentes do sistema, isto é, quantas classes, aspectos e interfaces existem no sistema.

Linhas de Código (LOC) – conta o número de linhas de código, descontadas linhas em branco e comentários. Em (SANT'ANNA et al., 2003) não há menção ao caso das linhas com importações de outros componentes. Neste trabalho optou-se por não considerá-las na contagem por dois motivos:

- Um componente não será mais difícil de ser compreendido do que outro por ter mais linhas de importação, uma vez que essas linhas não são analisadas quando se deseja entender o que o componente faz;
- Atualmente existem diversos ambientes de programação que gerenciam as importações dos componentes, não sendo necessário ao desenvolvedor nenhuma interação nesta área dos arquivos do software.

Esta métrica pode ser bastante influenciada pelo estilo de programação. Para minimizar este viés, o mesmo estilo de programação deve ser usado antes e depois de refatorar o código.

Número de Atributos (NOA) – conta o número de atributos internos de cada componente, desconsiderando-se atributos herdados. Esta métrica mede o vocabulário interno de cada componente, enquanto que VS mede o tamanho do vocabulário do sistema como um todo.

[9] sigla para o nome em inglês *Lack of Cohesion in Operations*
[10] sigla para o nome em inglês *Vocabulary Size*
[11] sigla para o nome em inglês *Lines of Code*
[12] sigla para o nome em inglês *Number of Attributes*
[13] sigla para o nome em inglês *Weighted Operations per Components*

Peso de Operações por Componente (WOC) – extensão da métrica WMC de Chidamber e Kemerer (1994), esta métrica mede a complexidade de um componente em termos de suas operações. Este valor é obtido atribuindo um valor para a complexidade de cada operação do componente e depois somando esses valores. Apesar de não ser especificado em (CHIDAMBER; KEMERER, 1994) como a complexidade deve ser medida, Sant'Anna et al. definem que a complexidade de cada operação é obtida contando-se o número de parâmetros da operação. Assim, operações sem parâmetros têm complexidade um, com um parâmetro têm complexidade dois, e assim por diante.

3 O PROCESSO DE AVALIAÇÃO

Neste capítulo é descrito o processo proposto para avaliar refatorações OA. Ele é necessário para sistematizar e simplificar a avaliação de refatorações, pois uma refatoração muitas vezes possui diversos passos simples que juntos compõem uma modificação complexa do software, modificação esta que é de difícil avaliação porque diversas partes do programa podem ser alteradas de uma só vez, e de maneiras diferentes. O processo está baseado na divisão da refatoração em pequenas alterações mais simples do que a refatoração original. Como é mais fácil obter os impactos na qualidade de uma pequena modificação do software do que os de uma refatoração inteira, optou-se por avaliar separadamente as modificações menores que compõem uma mesma refatoração, e em uma etapa posterior juntar os resultados parciais dessas modificações, obtendo assim o impacto total na qualidade da refatoração avaliada. Essas pequenas modificações menores do software são denominadas **passos básicos**.

3.1 Etapas do processo

A figura 3.1 apresenta uma representação gráfica do processo adotado. Cada uma de suas partes será descrita com maiores detalhes nas próximas seções, porém ressalta-se de antemão a divisão do processo em quatro grandes etapas: (1) a etapa inicial, onde é definida a refatoração avaliada e as métricas que serão usadas nesta avaliação[1]; (2) a divisão da refatoração em modificações menores, representada pelo processo "Decomposição da refatoração"; (3) a estimativa dos impactos de cada passo básico na qualidade do software, mostrada no processo "Avaliação dos passos básicos"; e (4) a etapa final da avaliação, onde os impactos de cada passo básico na qualidade do software são reunidos para obter os impactos totais da refatoração e permitir sua análise.

3.1.1 Escolha da refatoração e das métricas

Para iniciar a execução do processo de avaliação, devem ser definidas primeiramente a refatoração que será avaliada e as métricas que serão usadas para fazer isso. A refatoração fornece os dados a serem usados no processo de decomposição da refatoração em passos básicos, enquanto que as métricas são os critérios pelos quais os passos básicos usados serão avaliados no processo de "Avaliação dos passos básicos". Para dar suporte à tarefa de seleção das métricas, existem alguns métodos na literatura, como o GQM (BASILI; CALDIERA; ROMBACH, 1994), o QFD (KOGURE; AKAO, 1983; QFD, 2007) e a Árvore de Características de Qualidade de Boehm, Brown e Lipow (1976).

[1]Na verdade esta é uma etapa que ocorre antes do início do processo de avaliação, porém ela também será explicada neste capítulo por ser de grande importância para o processo de avaliação em si

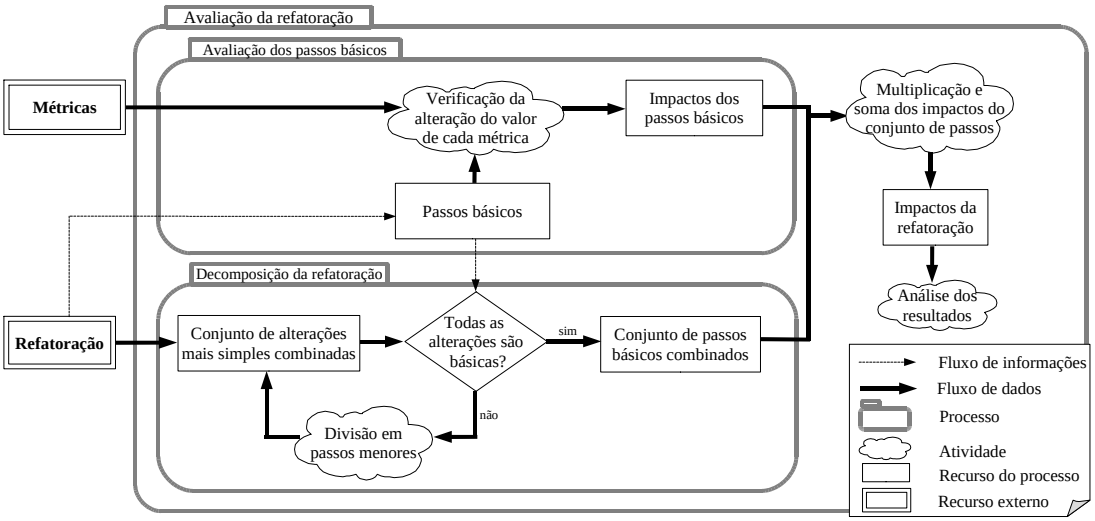


Figura 3.1: Processo usado para avaliar uma refatoração

Neste trabalho, a descrição das refatorações avaliadas é dada no capítulo 5, no início de cada seção que apresenta as etapas de avaliações das mesmas. Já a seleção das métricas é feita a partir do modelo GQM apresentado no apêndice B, o qual utiliza o conjunto de métricas proposto por Sant'Anna et al. (2003) descrito na seção 2.3.1.

Uma vez definida a refatoração a ser avaliada e as métricas que servirão como critério, é possível iniciar o processo de avaliação da refatoração, começando por medir os impactos de cada passo básico usado pela mesma.

3.1.2 Processo de avaliação dos passos básicos

A primeira coisa a ser feita no processo de avaliação de uma refatoração é medir os impactos que as pequenas alterações feitas por ela no software têm no valor das métricas escolhidas. Essas pequenas alterações são chamadas de "passos básicos" porque devem servir de base para a construção das mais variadas modificações de código. A divisão da refatoração em passos menores tem como objetivo facilitar a medição das variações das métricas, por isso os passos básicos devem ser simples o bastante para serem fáceis de avaliar. Eles também podem ser usados na decomposição de diferentes refatorações, razão pela qual devem ser genéricos o suficiente para permitir sua composição com outros passos na formação de alterações mais complexas. Exemplos de passos básicos são: "Criar aspecto vazio", "Adicionar implementação de interface a componente", "Remover atributo de componente" e "Mover código entre operações do mesmo componente". Os passos básicos usados na decomposição das refatorações avaliadas neste trabalho são apresentados no capítulo 4, onde também são descritos os seus respectivos impactos em cada uma das métricas escolhidas.

O processo "Avaliação dos passos básicos" começa com a definição do conjunto de passos básicos que será avaliado. Para fazer isso deve-se levar em conta quais pequenas alterações são feitas durante a execução da refatoração que está sendo avaliada, pois este mesmo conjunto de passos básicos será usado na decomposição da refatoração (seção 3.1.3).

Com os passos básicos definidos, passa-se à medição dos impactos de cada um desses passos no valor das métricas adotadas. Por exemplo, para um conjunto de métricas que

meça quantas classes e quantas linhas de código existem no sistema, o passo básico "Criar classe vazia" aumenta o número de classes em uma unidade e o número de linhas de código em duas unidades – uma linha com a declaração da classe e outra para o fechamento de seu escopo (representado em linguagens como Java e AspectJ por "}"). Algumas métricas como "número de linhas de código" podem ter resultados diferentes de acordo com a linguagem ou estilo de programação adotados, por isso o avaliador deve levar em conta essas possíveis variações ao fazer a avaliação dos passos básicos.

Após medir os impactos de todos os passos básicos em cada uma das métricas, obtém-se um conjunto de avaliações básicas que serão usadas na etapa final do processo de avaliação da refatoração (seção 3.1.4).

3.1.3 Processo de decomposição da refatoração

Após medir o impacto dos passos básicos necessários nas métricas escolhidas, deve-se iniciar o processo de decomposição da refatoração. Este processo busca dividir continuamente a refatoração original em passos cada vez menores, até obter apenas passos básicos. Sua primeira etapa, representada na figura 3.1 pela seta entre a "Refatoração" e o "Conjunto de alterações mais simples combinadas", corresponde à divisão da refatoração em uma seqüência de passos, o que já é feito previamente pelos autores que adotam o formato de descrição de refatorações apresentado por Fowler (FOWLER, 1999), como é o caso dos autores das refatorações avaliadas neste trabalho. Apesar de refatorações que seguem este formato serem um pouco mais simples de avaliar – por não ser necessário realizar a divisão da refatoração em uma seqüência de passos –, refatorações descritas em outros formatos também podem ser avaliadas, bastando para isso que o avaliador fique encarregado da divisão da refatoração original em um conjunto de alterações mais simples.

Depois desta primeira divisão da refatoração em uma seqüência de passos, inicia-se o ciclo de decomposição desses passos. Este ciclo começa verificando se todos os passos são básicos, tendo como critério o conjunto de passos básicos definidos no processo descrito na seção 3.1.2. Em caso positivo, o ciclo se encerra e com ele o processo de decomposição da refatoração. Em caso negativo, é preciso continuar a divisão dos passos em alterações cada vez mais simples: para cada passo não-básico, deve-se dividi-lo em passos menores que quando combinados façam as mesmas alterações descritas no passo original. Feito isso, o ciclo reinicia. Quando não houver mais nenhum passo não-básico no conjunto, o ciclo se encerra e a decomposição da refatoração terá terminado. O resultado será um conjunto de passo básicos que quando combinados fazem as mesmas alterações da refatoração que está sendo avaliada.

Um caso da evolução deste ciclo pode ser observado através da decomposição do primeiro passo da refatoração *Encapsular Atributo* (seção 5.1.3), que possui a seguinte descrição: "criar métodos de escrita e leitura para o atributo". Suponha que tenham sido definidos e avaliados previamente os seguintes passos básicos:

- Criar operação vazia;
- Adicionar parâmetro a operação;
- Adicionar linha de leitura/escrita de valor de atributo a operação.

O primeiro passo da refatoração diz que devem ser criados dois métodos de acesso ao atributo, o que não é um dos passos básicos definidos previamente. Dessa forma, este passo deve ser dividido em passos menores:

1. Criar métodos de escrita e leitura para o atributo =
    (a) Criar método de escrita de atributo;
    (b) Criar método de leitura de atributo.

Esses dois passos fazem o mesmo que o passo original, porém ainda não são passos básicos. Isso torna necessário prosseguir a divisão deles em alterações ainda mais simples:

1. Criar métodos de escrita e leitura para o atributo =
    (a) Criar método de escrita de atributo =
        i. Criar operação vazia;
        ii. Adicionar parâmetro a operação;
        iii. Adicionar linha de leitura/escrita de valor de atributo a operação.
    (b) Criar método de leitura de atributo =
        i. Criar operação vazia;
        ii. Adicionar linha de leitura/escrita de valor de atributo a operação.

Após esta divisão, todos os passos combinados pertencem ao conjunto de passos básicos definido anteriormente, portanto o ciclo de decomposição se encerra.

A decomposição das refatorações avaliadas neste trabalho são apresentadas no capítulo 5, juntamente com a descrição e a análise dos impactos da refatoração.

3.1.4 Obtenção dos impactos totais e análise dos resultados

Após avaliar os passos básicos e transformar a refatoração original em um conjunto de passos básicos combinados, o processo principal entra em sua etapa final. É nesta etapa que as informações obtidas nos dois processos anteriores são combinadas para produzir uma descrição dos impactos da refatoração nos valores das métricas escolhidas, e é feita a análise dos resultados alcançados.

A primeira fase desta etapa é a multiplicação dos impactos de cada passo básico de acordo com o número de vezes que o passo for usado na refatoração. No exemplo acima, para criar os métodos de escrita e leitura, os passos básicos “Criar operação vazia” e “Adicionar linha de leitura/escrita de valor de atributo a operação” são usados duas vezes, enquanto que “Adicionar parâmetro a operação” é usado apenas uma vez. Dessa forma, os impactos deste passo nos valores das métricas devem ser multiplicados por um e os impactos daqueles por dois.

Em seguida, os impactos multiplicados devem ser somados para obter as variações totais dos valores de cada uma das métricas, ou seja, descobrem-se quais são os impactos da refatoração quando todos os seus passos forem executados seqüencialmente. Maiores explicações sobre a aritmética das operações de multiplicação e soma de impactos são dadas no capítulo 5. De posse dos impactos totais, é possível analisar quais as conseqüências da aplicação da refatoração em um programa, como por exemplo quais atributos de qualidade serão melhorados, quais os impactos negativos nas métricas adotadas, se os objetivos da refatoração são alcançados, etc.

A soma dos impactos e a análise dos resultados das refatorações avaliadas neste trabalho são mostradas no capítulo 5, na parte final de cada seção que descreve as etapas da avaliação das refatorações. Como este trabalho utiliza um modelo de medição GQM, a interpretação dos dados coletados leva em consideração também o objetivo e as questões definidas neste modelo.

3.2 Reaproveitamento de resultados do processo

A fragmentação do processo em etapas permite dividir o trabalho de avaliar uma refatoração em blocos de atividades. Alguns desses blocos podem não ter de ser re-executados ao aplicar o processo de avaliação mais de uma vez, diminuindo-se assim o esforço necessário para completar um conjunto de avaliações.

A primeira forma de não-repetição de blocos de atividade se refere ao reúso da avaliação de um passo básico. Ela ocorre quando um mesmo passo básico é usado na decomposição de duas refatorações que são avaliadas por um único conjunto de métricas. Ora, um passo básico tem sempre os mesmos impactos em um conjunto de métricas, independente de com quais outros passos ele é combinado na decomposição de uma refatoração. Assim, uma vez avaliado um passo básico por um conjunto de métricas, sempre que este passo for usado na avaliação de uma refatoração pelo mesmo conjunto de métricas não será necessário repetir o processo de avaliação deste passo porque seus impactos já serão conhecidos.

Este trabalho aproveita esta oportunidade de reúso definindo um catálogo avaliações de passos básicos no capítulo 4. Neste capítulo os passos básicos necessários para avaliar as refatorações escolhidas são definidos e avaliados pelo conjunto de métricas descrito na seção 2.3.1. Posteriormente, nas avaliações das refatorações (capítulo 5), sempre que um passo básico for usado na decomposição de uma refatoração seus impactos serão recuperados do catálogo, evitando assim a repetição de um trabalho já executado anteriormente.

Outra forma de não repetir blocos de atividades diz respeito ao reaproveitamento completo da avaliação de uma refatoração. Ela ocorre quando uma refatoração menor é usada nos passos de uma refatoração maior e ambas são avaliadas pelo mesmo conjunto de métricas. Ao terminar a avaliação de uma refatoração, passa-se a conhecer seus impactos no conjunto de métricas, e esses impactos serão sempre os mesmos independente de quais outras refatorações ou passos básicos esta refatoração for combinada para formar refatorações maiores. Assim, uma vez avaliada uma refatoração, ela pode ser considerada um tipo especial de passo básico, pois ela possui as características necessárias para ser usada como um: seus impactos são fáceis de serem obtidos – basta consultar a avaliação da refatoração – e é genérica o suficiente para ser composta com outros passos básicos para formar alterações maiores do software – pela própria natureza das refatorações, que são pouco específicas para permitir sua aplicação em diferentes situações

Neste trabalho as avaliações de refatorações menores são reaproveitadas em diversas partes do capítulo 5, sempre que uma refatoração é usada nos passos de outra. A figura 5.1 mostra que refatorações avaliadas neste trabalho usam refatorações menores em seus passos, ilustrando o quanto são reaproveitadas as avaliações de refatorações.

A terceira forma de não repetir blocos de atividades é através do reaproveitamento da decomposição de uma refatoração. Esta situação ocorre quando uma refatoração avaliada anteriormente por um determinado conjunto de métricas deve ser reavaliada usando-se outro conjunto de métricas. Neste caso, não será necessário repetir a decomposição da refatoração porque as pequenas mudanças que esta refatoração faz no software independem de quais métricas são usadas em sua avaliação, portanto os passos básicos que combinados formam a refatoração avaliada serão os mesmos. Assim, ao reavaliar a refatoração trocando o conjunto de métricas usado, é necessário apenas reavaliar os passos básicos pelos novos critérios e executar a etapa final do processo de avaliação.

Este tipo de reaproveitamento de resultados não ocorre neste trabalho porque todas as aplicações do processo utilizam o mesmo conjunto de métricas, porém em trabalhos futuros ele poderá ocorrer caso esses trabalhos reavaliem as refatorações usando como

critério um conjunto diferente de métricas.

4 AVALIAÇÃO DOS PASSOS BÁSICOS

Este capítulo apresenta um catálogo com 82 passos básicos que são usados na decomposição das refatorações avaliadas no capítulo 5. Esses passos básicos são definidos conforme as orientações do processo de avaliação descrito no capítulo 3, isto é, são genéricos o bastante para serem compostos com diferentes outros passos na formação de alterações mais complexas do software, e simples o suficiente para ser fácil obter seus impactos nos valores das métricas escolhidas. Em futuras avaliações de refatorações que usarem o processo proposto no capítulo 3, os passos básicos aqui definidos poderão ser reaproveitados, assim como o catálogo de passos poderá ser estendido se o mesmo não contemplar todas as pequenas alterações que as refatorações futuramente avaliadas fizerem no software.

Além de descrever a definição de cada passo básico, este capítulo também apresenta as avaliações dos mesmos tendo como critério as métricas de Sant'Anna et al. (2003) descritas na seção 2.3.1, ou seja, são listados os impactos dos passos básicos nos valores das métricas. Assim, a descrição de cada passo básico segue a seguinte estrutura:

- Título do passo básico;
- Descrição da modificação que o passo básico faz no software;
- Exemplo de aplicação no código (quando aplicável);
- Impacto nos valores das métricas e respectivas justificativas.

Não foi incluída nesta descrição a análise das vantagens e desvantagens do passo básico porque o objetivo deste capítulo é apenas descrever os passos básicos e listar seus impactos nos valores das métricas. É importante ressaltar que passos básicos muitas vezes não preservam o comportamento do software, podendo até inserir erros de compilação no mesmo – erros que serão corrigidos com a combinação de outros passos básicos para formar uma refatoração. Assim, não faz sentido analisar possíveis melhorias que um único passo básico faz no software porque o comportamento do programa não seria o mesmo, sendo mais adequado fazê-lo apenas para as refatorações avaliadas.

Notação para representar resumidamente os impactos

A fim de representar os impactos de forma resumida nas tabelas deste trabalho, foi adotada uma notação que mostra (1) um comparativo entre o valor da métrica antes e depois da alteração do software (se seu valor aumenta ou diminui) e (2) a magnitude do impacto no valor da métrica (o quanto este valor aumenta ou diminui). Exemplos de impactos de passos básicos são "$\leq_{(1)}$", "$>_{(X)}$" e "$\gtrless_{(-1aX)}$".

Tabela 4.1: Notação usada para representar o impacto de uma modificação

Símbolo	Significado
$<$	Após o passo básico o valor da métrica será certamente **menor** que seu valor inicial.
$\leq$	Após o passo básico o valor da métrica poderá ser **menor** que seu valor inicial ou não ser modificado por ele.
=	O valor da métrica não é alterado pelo passo básico.
$\geq$	Após o passo básico o valor da métrica poderá ser **maior** que seu valor inicial ou não ser modificado por ele.
$>$	Após o passo básico o valor da métrica será certamente **maior** que seu valor inicial.
$\gtrless$	Após o passo básico o valor da métrica poderá ser **maior**, **menor** ou manter-se **igual** a seu valor inicial.

Para representar a comparação entre os valores inicial e final da métrica, é adotada a notação da tabela 4.1. Esta notação é usada para deixar mais evidente quais métricas foram melhoradas e quais foram pioradas pelo passo básico: o ideal de todas as métricas usadas é ter seu valor igual a zero, portanto quanto mais métricas diminuírem, melhor será o passo básico para os atributos de software medidos pelas métricas. Em outras palavras, os símbolos '$<$' e '$\leq$' significam melhorias do software, enquanto '$>$' e '$\geq$' têm significado contrário.

Já a representação da magnitude do impacto usa números inteiros, ou o símbolo 'X'. O número inteiro é usado quando a magnitude do impacto é igual para qualquer contexto em que o passo básico for aplicado (por exemplo: sempre que um aspecto for criado, o número de componentes do software aumentará uma unidade); o símbolo 'X' é usado quando a magnitude varia de acordo com as características do software em que o passo básico for aplicado (por exemplo: ao criar uma cópia de uma classe autônoma dentro de um aspecto, o aumento do número de linhas de código do software depende de quantas linhas a classe copiada tem).

É importante ressaltar que a magnitude de um impacto pode ser tanto um valor único quanto um intervalo de valores. No primeiro caso, o uso do passo básico sempre modifica o valor de métrica da mesma forma; no segundo, o valor da métrica poderá variar dentro do intervalo do impacto dependendo do contexto que o passo básico é usado. Por exemplo, "Adicionar atributo a componente" (seção 4.4.1) sempre aumenta o número de atributos em uma unidade, portanto seu impacto em NOA é +1. Por outro lado, este mesmo passo básico pode aumentar o número de acoplamentos de um componente em uma unidade ou não alterá-lo, por isso seu impacto em CBC varia entre 0 e +1. Assim, usando a notação descrita anteriormente, a magnitude deve representar seu valor mínimo e seu valor máximo através de um número inteiro ou do símbolo 'X'. Abaixo são mostrados alguns exemplos de representações resumidas de impactos e a forma como essas representações devem ser lidas:

$\leq_{(1)}$	"Após o passo básico o valor da métrica poderá ser uma unidade menor ou não ser modificado por ele"
$>_{(X)}$	"Após o passo básico o valor da métrica será certamente maior, porém a magnitude deste aumento dependerá do código em que este passo básico for usado"
$\gtrless_{(-1aX)}$	"Após o passo básico o valor da métrica poderá diminuir até uma unidade, aumentar um número variável de unidades ou não ser modificado"
$\gtrless$	"Após o passo básico o valor da métrica poderá diminuir ou aumentar um número variável de unidades, ou mesmo não ser modificado"

A existência de magnitudes variáveis (em um intervalo de valores) se deve à baixa especificidade dos passos básicos: como os passos são definidos de forma genérica, eles podem ser aplicados em diferentes situações, e em cada situação as conseqüências nos valores das métricas podem ser distintas. Para evitar magnitudes variáveis seria necessário definir passos básicos mais específicos, porém isso diminuiria seu reaproveitamento na decomposição das refatorações. Além disso, como as refatorações descrevem modificações genéricas no software para permitir seu uso em diferentes situações, a opção por passos básicos mais específicos tornaria necessário definir um passo básico para cada diferente situação que a refatoração pudesse ser aplicada, tornando muito mais complexo o processo de decomposição das refatorações e aumentando o número de passos básicos necessários para avaliar uma mesma refatoração.

Organização do catálogo de passos básicos

Para melhor organizar a apresentação deste capítulo, os passos básicos propostos foram divididos nas seguintes categorias, de acordo com a estrutura modificada pelo passo e do tipo de modificação que ele faz no software:

- Passos básicos que alteram componentes;
- Passos básicos que alteram conjuntos de pontos de junção;
- Passos básicos que alteram operações;
- Passos básicos que alteram atributos e declarações inter-tipos;
- Passos básicos que modificam a visibilidade de elementos;
- Outros passos básicos.

Nas próximas seções cada uma dessas categorias é descrita e seus passos básicos apresentados conforme a estrutura descrita anteriormente.

4.1 Passos básicos que alteram componentes

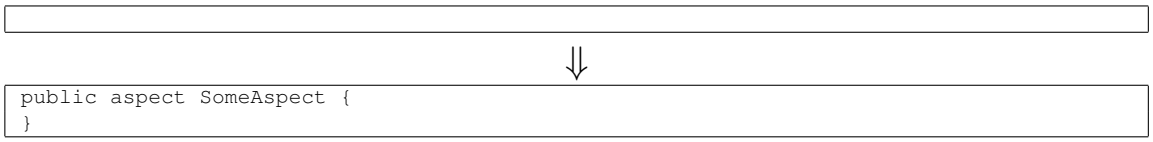
Nesta categoria de passos básicos estão as modificações que criam ou removem componentes do sistema, que modificam sua cadeia de herança, e que adicionam ou removem implementações de interfaces. A tabela 4.2 resume como cada métrica avaliada é alterada pelos passos básicos desta categoria.

Tabela 4.2: Variação das métricas ao usar os passos básicos que alteram componentes

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Criar aspecto vazio	$>_{(1)}$	=	=	$>_{(1)}$	$>_{(2)}$	=	=	=	=	=
Remover aspecto vazio	$<_{(1)}$	=	=	$<_{(1)}$	$<_{(2)}$	=	=	=	=	=
Criar classe ou interface vazia	$\geq_{(1)}$	=	=	$>_{(1)}$	$>_{(2)}$	=	=	=	=	=
Remover classe ou interface vazia	$\leq_{(1)}$	=	=	$<_{(1)}$	$<_{(2)}$	=	=	=	=	=
Adicionar implementação de interface a componente	$\geq_{(1)}$	=	$\geq_{(2)}$	=	=	=	=	$\geq_{(1)}$	=	=
Remover implementação de interface de componente	$\leq_{(1)}$	=	$\leq_{(2)}$	=	=	=	=	$\leq_{(1)}$	=	=
Adicionar herança a interface	$\geq_{(1)}$	=	$\geq_{(2)}$	=	=	=	=	$\geq_{(1)}$	=	=
Remover herança de interface	$\leq_{(1)}$	=	$\leq_{(2)}$	=	=	=	=	$\leq_{(1)}$	=	=
Adicionar herança a classe ou aspecto	$\geq_{(1)}$	=	$\geq_{(2)}$	=	=	=	=	$\geq_{(1)}$	$\geq_{(X)}$	=
Remover herança de classe ou aspecto	$\leq_{(1)}$	=	$\leq_{(2)}$	=	=	=	=	$\leq_{(1)}$	$\leq_{(X)}$	=
Criar cópia interna de componente autônomo	$\geq_{(1)}$	$\geq_{(X)}$	$\geq_{(X)}$	$>_{(1)}$	$>_{(X)}$	$\geq_{(X)}$	$\geq_{(X)}$	$\geq_{(X)}$	=	$\geq_{(X)}$
Remover cópia interna de componente autônomo	$\leq_{(1)}$	$\leq_{(X)}$	$\leq_{(X)}$	$<_{(1)}$	$<_{(X)}$	$\leq_{(X)}$	$\leq_{(X)}$	$\leq_{(X)}$	=	$\leq_{(X)}$
Criar cópia autônoma de componente interno	$\geq_{(1)}$	$\geq_{(X)}$	$\geq_{(X)}$	$>_{(1)}$	$>_{(X)}$	$\geq_{(X)}$	$\geq_{(X)}$	$\geq_{(X)}$	=	$\geq_{(X)}$
Remover cópia autônoma de componente interno	$\leq_{(1)}$	$\leq_{(X)}$	$\leq_{(X)}$	$<_{(1)}$	$<_{(X)}$	$\leq_{(X)}$	$\leq_{(X)}$	$\leq_{(X)}$	=	$\leq_{(X)}$

4.1.1 Criar aspecto vazio

Modificação: adicionar ao sistema um aspecto de corpo vazio, que não herde de nenhum aspecto e que não implemente nenhuma interface.



```
public aspect SomeAspect {
}
```

Métricas alteradas:

CDC: $>_{(1)}$

O interesse relacionado ao aspecto criado terá mais um componente responsável por sua implementação – o próprio aspecto vazio.

VS: $>_{(1)}$

O número de componentes do programa aumenta em uma unidade por causa do aspecto criado.

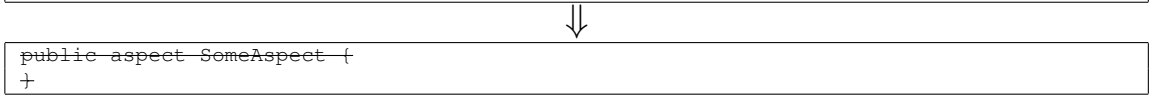
LOC: $>_{(2)}$

Com a criação do aspecto vazio são adicionadas duas linhas ao sistema: a de declaração do aspecto e a de fechamento de seu escopo (em AspectJ representada por "}").

4.1.2 Remover aspecto vazio

Modificação: remover do sistema um aspecto de corpo vazio, que não herde de nenhum aspecto e que não implemente nenhuma interface.

```
public aspect SomeAspect {
}
```



~~public aspect SomeAspect {~~
~~}~~

Métricas alteradas:

CDC: $<_{(1)}$

O interesse relacionado ao aspecto removido terá um componente responsável por sua implementação a menos – o próprio aspecto vazio.

VS: $<_{(1)}$

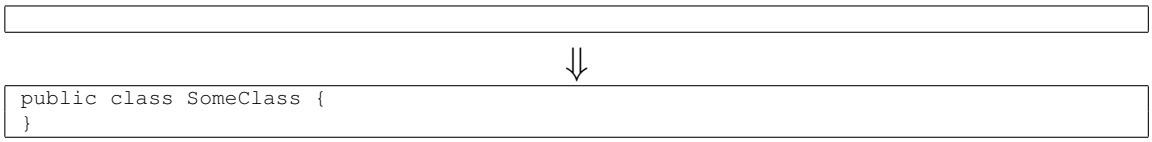
O número de componentes do programa diminui em uma unidade por causa da remoção do aspecto.

LOC: $<_{(2)}$

Todas as linhas de código do aspecto vazio serão eliminadas, o que muitas vezes significa apenas duas linhas: a linha de declaração do aspecto e a linha de encerramento de seu escopo (em AspectJ representada por "}").

4.1.3 Criar classe ou interface vazia

Modificação: adicionar ao sistema uma classe/interface de corpo vazio, que não herde de nenhuma classe/interface. No caso da criação de uma classe, a mesma também não deve implementar nenhuma interface.



```
public class SomeClass {
}
```

Métricas alteradas:

CDC: $\geq_{(1)}$

Se a classe/interface tiver como objetivo auxiliar na implementação de um interesse entrecortante, o valor de CDC desse interesse aumentará em uma unidade com a criação do componente.

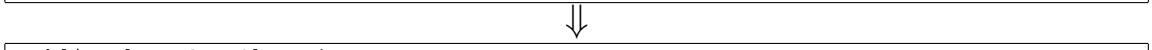
VS ($>_{(1)}$) e **LOC** ($>_{(2)}$)

Mesmas justificativas de 4.1.1.

4.1.4 Remover classe ou interface vazia

Modificação: remover do sistema uma classe ou interface de corpo vazio, que não herde de nenhum componente e que não implemente nenhuma interface (no caso da remoção de uma classe).

```
public class SomeClass {
}
```



```
~~public class SomeClass {~~
~~}~~
```

Métricas alteradas:

CDC: $\leq_{(1)}$

Se a classe/interface for originalmente usada para auxiliar na implementação de um interesse entrecortante, ao removê-la haverá um componente a menos no sistema relacionado a este interesse, diminuindo assim o CDC do mesmo.

VS ($<_{(1)}$) e **LOC** ($<_{(2)}$)

Mesmas justificativas de 4.1.2.

4.1.5 Adicionar implementação de interface a componente

Modificação: fazer com que um componente passe a implementar uma interface.

```
public class SomeClass implements TargetInterface1, ... {
    (...)
}
```

⇓

```
public class SomeClass implements TargetInterface0, TargetInterface1, ...  {
    (...)
}
```

Métricas alteradas:

CDC: $\geq_{(1)}$

Se a interface estiver relacionada a um interesse transversal diferente dos interesses relacionados ao componente, quando este passar a implementá-la o interesse da interface estará disperso em mais um componente, aumentando assim o valor de CDC deste interesse em uma unidade.

CDLOC: $\geq_{(2)}$

Para haver alteração no valor de CDLOC é necessário que as linhas modificadas pelo passo básico se tornem sombreadas ou deixem de ser, o que neste caso pode ocorrer apenas na linha de declaração do componente. Este passo altera o sombreamento desta linha apenas se a interface adicionada estiver relacionada a algum interesse transversal e não houver mais nenhuma interface implementada ou componente herdado na mesma situação. Neste caso, a adição da interface na linha de declaração do componente fará com que esta linha se torne sombreada.

Para modificar o valor de CDLOC, no entanto, é necessário mais uma condição: que na linha imediatamente abaixo da linha de declaração do componente (no início de seu corpo) não haja uma linha sombreada. Neste caso, a adição da interface criará dois novos pontos de transição e aumentará o valor de CDLOC em duas unidades, como pode ser visto no exemplo abaixo:

```
public class X {
    // primeira linha da classe
    (...)
}
```

⇒

```
public class X implements Y {
    // primeira linha da classe
    (...)
}
```

Se a interface adicionada estiver relacionada a um interesse transversal mas a primeira linha do corpo do componente também for sombreada para este interesse, a modificação do passo básico apenas deslocará o início do bloco de linhas sombreadas e não alterará o número de pontos de transição do código, como ocorre no exemplo abaixo:

```
public class X {
    // primeira linha da classe
    (...)
}
```

⇒

```
public class X implements Y {
    // primeira linha da classe
    (...)
}
```

CBC: $\geq_{(1)}$

Caso o componente ainda não seja acoplado à interface, ao adicionar esta interface à lista de interfaces implementadas pelo componente ocorrerá um aumento de uma unidade na quantidade de componentes acoplados a ele.

4.1.6 Remover implementação de interface de componente

Modificação: retirar uma interface da lista de interfaces implementadas de um componente.

```
public class SomeClass implements TargetInterface0, TargetInterface1, ... {
    (...)
}
```

⇓

```
public class SomeClass implements ~~TargetInterface0,~~ TargetInterface1, ...  {
    (...)
}
```

Métricas alteradas:

CDC: $\leq_{(1)}$

Se a interface estiver relacionada a um interesse transversal e se originalmente a única referência a esse interesse no componente fosse pela implementação da mesma, então após remover a interface essa referência será eliminada e o componente deixará de estar relacionado ao interesse, diminuindo assim o valor de CDC.

CDLOC: $\leq_{(2)}$

Para haver alteração no valor de CDLOC é necessário que as linhas modificadas pelo passo básico se tornem sombreadas ou deixem de ser, o que neste caso pode ocorrer apenas na linha de declaração do componente. Este passo altera o sombreamento desta linha apenas se a interface removida estiver relacionada a algum interesse transversal e não houver mais nenhuma interface implementada ou componente herdado na mesma situação. Neste caso, a remoção da interface da linha de declaração do componente fará com que a linha de declaração deixe de ser sombreada.

Para modificar o valor de CDLOC, no entanto, é necessário mais uma condição: que na linha imediatamente abaixo da linha de declaração do componente (no início de seu corpo) não haja uma linha sombreada. Neste caso, a remoção da interface eliminará dois pontos de transição e reduzirá o valor de CDLOC em duas unidades, como pode ser visto no exemplo abaixo:

```
public class X implements Y {
    // primeira linha da classe
    (...)
}
```

⇒

```
public class X {
    // primeira linha da classe
    (...)
}
```

Se a interface removida estiver relacionada a um interesse transversal mas a primeira linha do corpo do componente também for sombreada para este interesse, a modificação do passo básico apenas deslocará o início do bloco de linhas sombreadas e não alterará o número de pontos de transição do código, como ocorre no exemplo abaixo:

```
public class X {
    // primeira linha da classe
    (...)
}
```

⇒

```
public class X implements Y {
    // primeira linha da classe
    (...)
}
```

CBC: $\leq_{(1)}$

Se o componente só estiver acoplado à interface através da implementação da mesma (isto é, se no corpo do componente não houver referência à interface), então a remoção desta interface da lista de interfaces implementadas pelo componente eliminará o acoplamento existente entre eles, o que diminui o valor de CBC do componente em uma unidade.

4.1.7 Adicionar herança a interface

Modificação: uma interface passa a ser filha de outra, independentemente de já herdar de outras interfaces previamente.

```
public interface SomeInterface extends SuperInterface1, ... {
    (...)
}
```

⇓

```
public interface SomeInterface extends SuperInterface0,  SuperInterface1, ...  {
    (...)
}
```

Métricas alteradas:

As modificações deste passo básico são muito similares às de "Adicionar herança a classe ou aspecto", como será visto na seção 4.1.9. No entanto, existem diferenças no que diz respeito à profundidade da árvore de herança dos componentes, pois a métrica que mede esta característica (DIT) leva em consideração apenas a profundidade de classes e aspectos, descartando a profundidade das interfaces. Dessa forma, apesar da árvore de herança sofrer alterações, o valor de **DIT** não será modificado por este passo básico.

CDC: $\geq_{(1)}$

Se a interface-mãe estiver relacionada a um interesse diferente dos interesses relacionados à interface-filha, quando esta passar a estendê-la o interesse da interface-mãe estará disperso em mais um componente, aumentando assim o valor de seu CDC em uma unidade.

CDLOC: $\geq_{(2)}$

Se a interface-mãe estiver relacionada a um interesse transversal diferente dos interesses relacionados à interface-filha, e se originalmente a linha de declaração da interface-filha não for sombreada para este interesse, então este passo básico sombreará a linha da declaração da interface-filha. Esta alteração apenas modificará o valor de CDLOC se não houver imediatamente abaixo desta linha (no início do corpo da interface-filha) outra linha sombreada para o mesmo interesse, pois neste caso o sombreamento adicionará dois novos pontos de transição no código da interface-filha, como mostra o exemplo abaixo:

```
public interface X {
    // primeira linha da interface
    (...)
}
```

⇒

```
public interface X extends Y {
    // primeira linha da interface
    (...)
}
```

Caso a linha de declaração da interface-filha se torne sombreada mas sua primeira linha já for originalmente sombreada para o mesmo interesse, o passo não altera o número de pontos de transição, apenas modifica a linha do início do bloco de linhas sombreadas, de forma similar ao exemplo abaixo:

```
public interface X {
    // primeira linha da interface
    (...)
}
```

⇒

```
public interface X extends Y {
    // primeira linha da interface
    (...)
}
```

CBC: $\geq_{(1)}$

Se a futura interface-filha não estiver acoplada à interface-mãe antes da criação da relação de herança entre elas, então ao tornar uma filha da outra será criado o acoplamento entre ambas, aumentando assim o valor de CBC da interface-filha em uma unidade.

4.1.8 Remover herança de interface

Modificação: uma interface que inicialmente é filha de uma interface-mãe passa a não herdar de nenhuma.

```
public class SomeInterface extends SuperInterface0, SuperInterface1, ... {
    (...)
}
```

⇓

```
public class SomeInterface extends ~~SuperInterface0~~, SuperInterface1, ...  {
    (...)
}
```

Métricas alteradas:

As modificações deste passo básico são muito similares às de "Remover herança de classe ou aspecto" (seção 4.1.10). No entanto, existem diferenças no que diz respeito à profundidade da árvore de herança dos componentes, pois a métrica que mede esta característica (DIT) leva em consideração apenas a profundidade de classes e aspectos, descartando a profundidade das interfaces. Dessa forma, apesar da árvore de herança sofrer alterações, o valor de **DIT** não será modificado por este passo básico.

CDC: $\leq_{(1)}$

Se a interface-mãe estiver relacionada a um interesse transversal e se originalmente a única referência a esse interesse na interface-filha fosse pela relação de herança, então após remover esta relação a referência será eliminada e a interface-filha deixará de estar relacionada ao interesse, diminuindo assim o valor de CDC.

CDLOC: $\leq_{(2)}$

Se ao remover a relação de herança a linha de declaração da interface-filha deixar de ser sombreada para algum interesse transversal, e se imediatamente abaixo dela (no início do corpo da interface) não houver uma linha sombreada para o mesmo interesse, então a remoção da interface-mãe eliminará dois pontos de transição do código da interface-filha, como ocorre no exemplo abaixo:

```
public interface X extends Y {
    // primeira linha da interface
    (...)
}
```

⇒

```
public interface X {
    // primeira linha da interface
    (...)
}
```

Se no entanto a primeira linha da interface for sombreada, então este passo básico apenas alterará a linha em que começa o bloco sombreado e não modificará o número de pontos de transição, como ocorre neste exemplo:

```
public interface X extends Y {
    // primeira linha da interface
    (...)
}
```

⇒

```
public interface X {
    // primeira linha da interface
    (...)
}
```

CBC: $\leq_{(1)}$

Se a interface-filha só estiver acoplada à interface-mãe através da relação de herança entre elas (isto é, se não houver referência à interface-mãe dentro do corpo da interface-filha), a remoção da herança eliminará o acoplamento entre as duas interfaces, diminuindo assim o valor de CBC da interface-filha em uma unidade.

4.1.9 Adicionar herança a classe ou aspecto

Modificação: uma classe ou aspecto que inicialmente não possui um componente-pai passa a herdar de um.

```
public class SomeClass {
    (...)
}
```

⇓

```
public class SomeClass extends SuperClass {
    (...)
}
```

Métricas alteradas:

CDC ($\geq_{(1)}$), **CDLOC** ($\geq_{(2)}$) e **CBC** ($\geq_{(1)}$)

Mesmas justificativas de 4.1.7.

DIT: $\geq_{(X)}$

Ao criar a relação de herança entre o componente-filho e o componente-pai, a profundidade da árvore de herança poderá aumentar. Considere (1) d_{pai} o comprimento do caminho da raiz da árvore até o componente-pai e (2) d_{filho} a profundidade máxima da sub-árvore de herança cuja raiz é o componente-filho (figura 4.1). Após transformar o componente-filho em herdeiro do componente-pai, serão criados novos caminhos na árvore de herança do sistema, os quais terão profundidade máxima $d_{pai} + 1 + d_{filho}$. Se este valor for maior que a profundidade original da árvore de herança, então o valor de DIT terá aumentado com esta modificação.

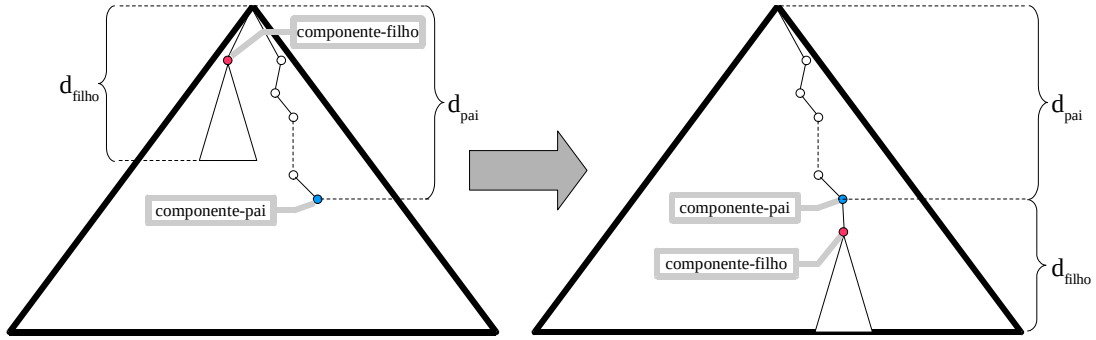


Figura 4.1: Alterações da árvore de herança ao tornar um componente filho do outro

4.1.10 Remover herança de classe ou aspecto

Modificação: uma classe ou aspecto que inicialmente é filho de um componente-pai passa a não herdar de nenhum.

```
public class SomeClass extends SuperClass {
    (...)
}
```

⇓

```
public class SomeClass ~~extends SuperClass~~ {
    (...)
}
```

Métricas alteradas:

CDC ($\leq_{(1)}$), **CDLOC** ($\leq_{(2)}$) e **CBC** ($\leq_{(1)}$)
Mesmas justificativas de 4.1.8.

DIT: $\leq_{(X)}$
Com a remoção da herança, o componente-filho será ligado à raiz da árvore de herança (figura 4.2), o que diminui sua distância até a raiz e também as distâncias de todos os componentes de sua sub-árvore (seus filhos diretos ou indiretos). Na figura 4.2, a profundidade original do componente-filho era $d_{pai} + 1$ e passou a ser 1 após a remoção da herança, assim como todos os componentes de sua sub-árvore tiveram suas respectivas profundidades também reduzidas em d_{pai} unidades.

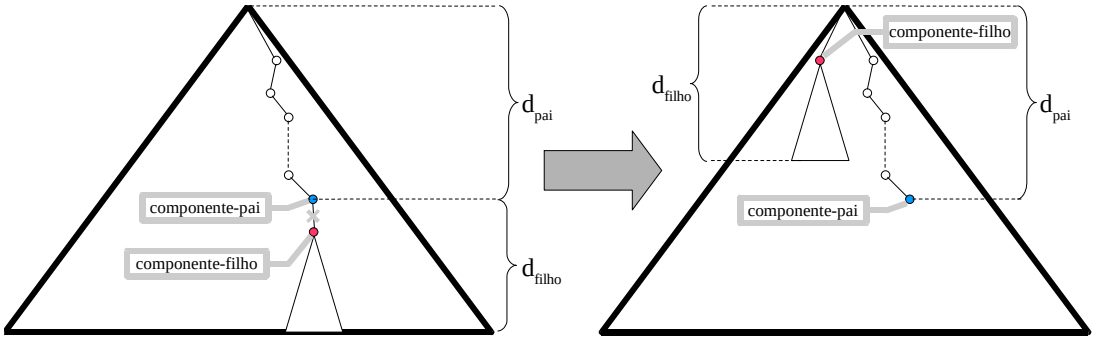


Figura 4.2: Alterações da árvore de herança ao remover relação de herança entre componentes

Dessa forma, se todos os caminhos de profundidade máxima na árvore de herança (os quais definem o valor de DIT) passarem pelo componente, a remoção de sua herança diminuirá a profundidade da árvore. No entanto, se existir algum caminho de profundidade máxima que não passe por este componente, a remoção da herança não alterará o valor de DIT porque após esta modificação da árvore ainda existirá um caminho com a profundidade original da árvore, a qual é superior a todas as outras.

Apesar de ser possível estabelecer em quais situações o valor de DIT é alterado por este passo básico, a magnitude desta variação só pode ser conhecida analisando o código do programa onde o passo é aplicado, pois depende da profundidade de todos os componentes do programa.

4.1.11 Criar cópia interna de componente autônomo

Modificação: criar uma cópia de um componente autônomo (definido em arquivo separado dos demais componentes do sistema) dentro do corpo de outro componente.

```
public class SomeClass {
    (...)
}
```

```
public class TargetComponent {
    (...)
}
```

⇓

```
public class SomeClass {
    (...)
}
```

```
public class TargetComponent {
    (...)
    public class SomeClass {
        (...)
    }
}
```

Métricas alteradas:

Quase todas as métricas são alteradas por este passo básico. A exceção é a métrica **DIT**, que mantém seu valor original porque a profundidade do componente copiado será igual à profundidade do componente autônomo, não tendo portanto nenhum componente novo com profundidade superior às existentes anteriormente.

CDC: $\geq_{(1)}$

Se o componente copiado estiver relacionado a algum interesse transversal, então sua clonagem fará com que este interesse esteja disperso em mais um componente, aumentando assim o valor de CDC deste interesse.

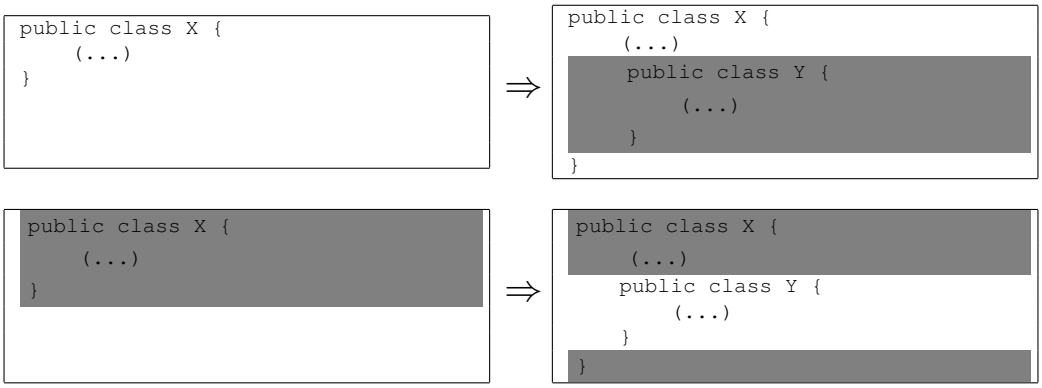
CDO: $\geq_{(X)}$

Se alguma operação do componente copiado estiver relacionada a um interesse transversal, então a clonagem do componente fará com que este interesse fique disperso em mais operações do que originalmente. O aumento do valor de CDO do interesse será igual ao número de operações do componente clonado relacionadas a este interesse.

CDLOC: $\geq_{(X)}$

O valor de CDLOC pode aumentar em duas frentes: (1) pelos pontos de transição dentro do corpo do componente copiado; e (2) por pontos de transição criados entre o componente clonado e o componente que recebe a cópia. No primeiro caso, todos os pontos de transição existentes no componente autônomo serão duplicados na cópia interna – se não houver nenhum ponto de transição, a clonagem não duplica nenhum ponto.

No segundo caso, ao adicionar o componente clonado dentro de outro, é possível que sejam criados novos pontos de transição (um no início e outro no fim do corpo da cópia interna), caso os interesses transversais dos dois componentes não sejam iguais. Os exemplos abaixo mostram duas situações em que novos pontos de transição são criados entre os componentes:



VS: $>_{(1)}$

A cópia do componente autônomo cria um novo componente no sistema, o que aumenta o valor de VS em uma unidade.

LOC: $>_{(X)}$

Cada linha do componente autônomo será copiada para o componente interno, com exceção das linhas com seus comandos de importação (estas serão copiadas apenas se o

componente-alvo não tiver as importações necessárias). Isso aumenta o número de linhas de código do sistema tantas unidades quanto forem as linhas de código existentes no componente autônomo.

NOA: $\geq_{(X)}$
Todos os atributos do componente autônomo serão copiados para o componente clonado, aumentando assim o valor de NOA. Se o componente copiado não tiver atributos, o valor de NOA não se altera.

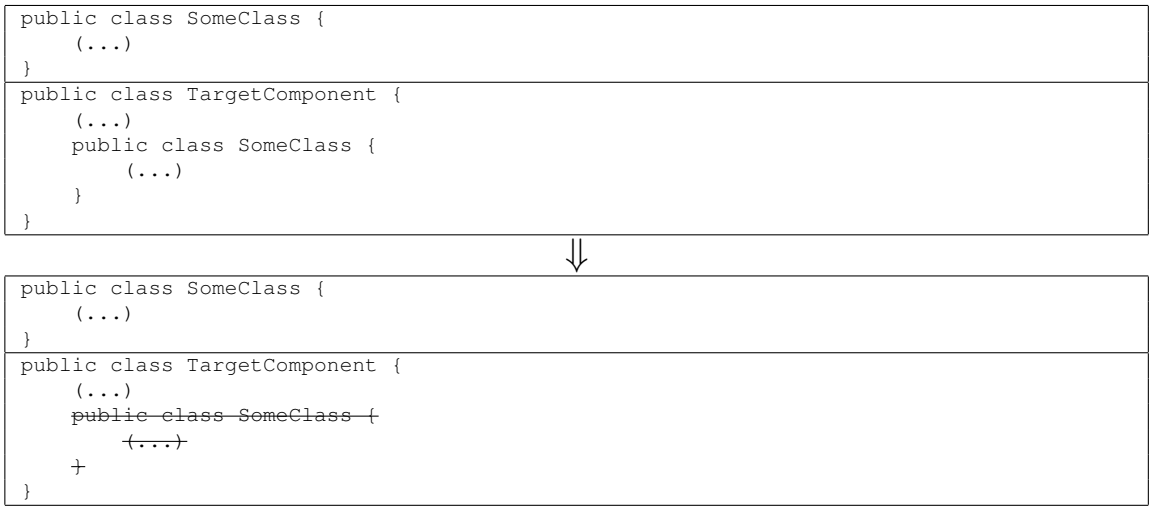
WOC: $\geq_{(X)}$
Todos as operações e respectivas listas de parâmetros do componente autônomo serão copiadas para o componente clonado, aumentando assim o valor de WOC. Se o componente copiado não tiver operações, o valor de WOC não se altera.

CBC: $\geq_{(X)}$
Todos os acoplamentos do componente autônomo serão criados no componente clonado, aumentando assim o valor de CBC. Se o componente copiado não for acoplado a nenhum componente (situação incomum, mas possível de acontecer), o valor de CBC não se altera.

LCOO: $\geq_{(X)}$
Por serem cópias idênticas, o componente autônomo e o componente interno terão o mesmo valor de LCOO. Assim, adicionar a cópia dentro do componente-alvo faz com que o valor total de LCOO do sistema aumente tanto quanto for o valor de LCOO do componente copiado.

4.1.12 Remover cópia interna de componente autônomo

Modificação: remove uma cópia de um componente autônomo (definido em arquivo separado dos demais componentes do sistema) de dentro do corpo de outro componente.



Métricas alteradas:

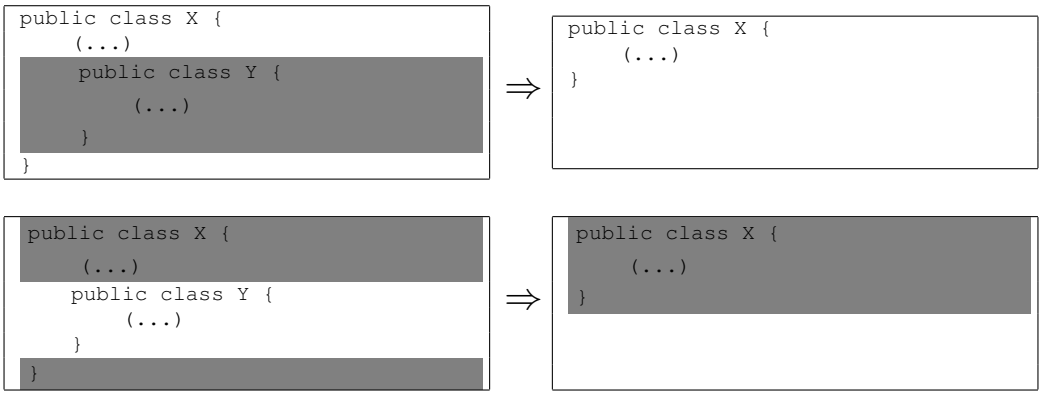
Quase todas as métricas são alteradas por este passo básico. A exceção é a métrica **DIT**, que mantém seu valor original porque a profundidade do componente removido é igual à profundidade do componente autônomo, portanto não será eliminado nenhum componente com profundidade superior às existentes anteriormente.

CDC: $\leq_{(1)}$
Se o componente interno estiver relacionado a algum interesse transversal, então sua remoção fará com que este interesse esteja disperso em um componente a menos, diminuindo assim o valor de CDC deste interesse.

CDO: $\leq_{(X)}$
Se alguma operação do componente interno estiver relacionada a um interesse transversal, então a remoção do componente fará com que este interesse fique disperso em menos operações do que originalmente. A diminuição do valor de CDO do interesse será igual ao número de operações do componente removido relacionadas a este interesse.

CDLOC: $\leq_{(X)}$
O valor de CDLOC pode diminuir em duas frentes: (1) pelos pontos de transição dentro do corpo do componente removido; e (2) por pontos de transição eliminados entre o componente removido e o componente onde estava a cópia. No primeiro caso, todos os pontos de transição existentes no componente interno serão removidos – se não houver nenhum ponto de transição, a remoção não elimina nenhum ponto.

No segundo caso, ao remover o componente de dentro do outro, é possível que sejam eliminados pontos de transição (um no início e outro no fim do corpo da cópia interna), caso os interesses transversais dos dois componentes não sejam iguais. Os exemplos abaixo mostram duas situações em que pontos de transição são eliminados do componente-alvo:



VS: $<_{(1)}$
A remoção do componente interno elimina um componente no sistema, o que diminui o valor de VS em uma unidade.

LOC: $<_{(X)}$
Todas as linhas do componente interno serão eliminadas do sistema. Isso diminui o valor de LOC em tantas unidades quanto forem as linhas de código existentes originalmente no componente interno.

NOA: $\leq_{(X)}$
Todos os atributos do componente interno serão removidos, diminuindo assim o valor de NOA. Se o componente removido não tiver atributos, o valor de NOA não se altera.

WOC: $\leq_{(X)}$
Todos as operações e respectivas listas de parâmetros do componente interno serão removidas, diminuindo assim o valor de WOC. Se o componente removido não tiver operações, o valor de WOC não se altera.

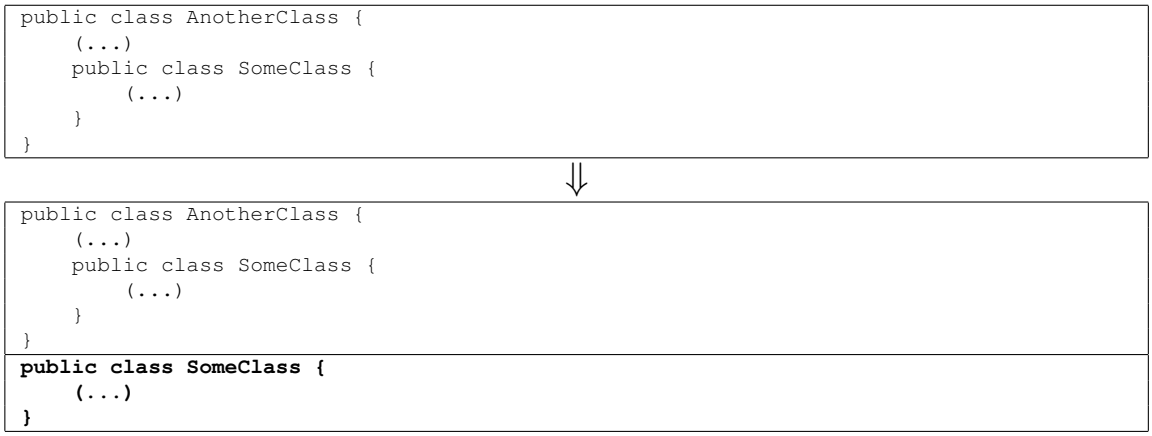
CBC: $\leq_{(X)}$
Todos os acoplamentos do componente interno com outros componentes do sistema serão removidos, diminuindo assim o valor de CBC. Se o componente interno não for acoplado a nenhum componente (situação incomum, mas possível de acontecer), o valor de CBC não se altera.

LCOO: $\leq_{(X)}$
Por serem cópias idênticas, o componente autônomo e o componente interno têm o mesmo

valor de LCOO. Assim, remover a cópia dentro do componente-alvo faz com que o valor total de LCOO do sistema diminua tanto quanto for o valor de LCOO do componente autônomo.

4.1.13 Criar cópia autônoma de componente interno

Modificação: criar uma cópia autônoma (em arquivo separado dos demais componentes do sistema) de um componente definido dentro do corpo de outro componente.



Métricas alteradas:

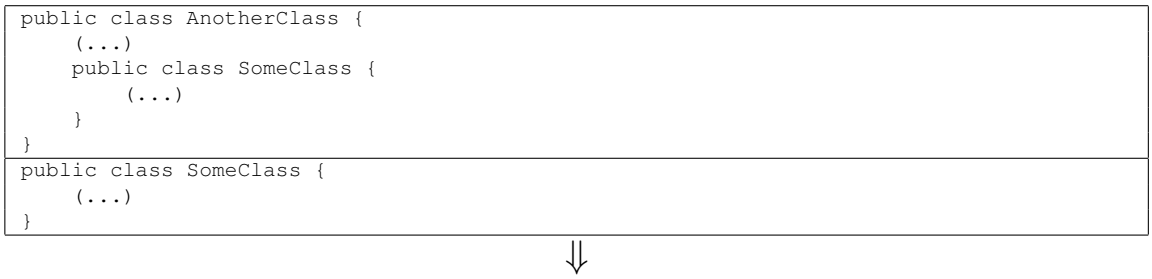
Este passo básico efetua modificações muito parecidas com "Criar cópia interna de componente autônomo" (4.1.11), por isso as justificativas para a alteração (ou manutenção) dos valores das métricas são quase todas iguais entre os dois passos.

CDC ($\geq_{(1)}$), **CDO** ($\geq_{(X)}$), **VS** ($>_{(1)}$), **LOC** ($>_{(X)}$), **NOA** ($\geq_{(X)}$), **WOC** ($\geq_{(X)}$), **CBC** ($\geq_{(X)}$) e **LCOO** ($\geq_{(X)}$)
Mesmas justificativas de 4.1.11.

CDLOC: $\geq_{(X)}$
Da mesma forma que ocorre no passo básico "Criar cópia interna de componente autônomo", os pontos de transição existentes no corpo do componente copiado são duplicados em seu clone. No entanto, a criação de pontos de transição entre a cópia adicionada e o componente que a abriga (prevista em "Criar cópia interna de componente autônomo") não ocorre neste passo básico porque se trata de uma cópia autônoma, em arquivo separado.

4.1.14 Remover cópia autônoma de componente interno

Modificação: remover uma cópia autônoma (em arquivo separado dos demais componentes do sistema) de um componente definido dentro do corpo de outro componente.



```
public class AnotherClass {
    (...)
    public class SomeClass {
        (...)
    }
}
```

```
~~public class SomeClass {~~
    ~~(...)~~
~~}~~
```

Métricas alteradas:

Este passo básico efetua modificações muito parecidas com "Remover cópia interna de componente autônomo" (4.1.12), por isso as justificativas para a alteração (ou manutenção) dos valores das métricas são quase todas iguais entre os dois passos.

CDC ($\leq_{(1)}$), **CDO** ($\leq_{(X)}$), **VS** ($<_{(1)}$), **LOC** ($<_{(X)}$), **NOA** ($\leq_{(X)}$), **WOC** ($\leq_{(X)}$), **CBC** ($\leq_{(X)}$) e **LCOO** ($\leq_{(X)}$)
Mesmas justificativas de 4.1.12.

CDLOC: $\leq_{(X)}$
Da mesma forma que ocorre no passo básico "Remover cópia interna de componente autônomo", os pontos de transição existentes no corpo do componente interno são eliminados. No entanto, a remoção de pontos de transição entre a cópia interna e o componente que a abriga (prevista em "remover cópia interna de componente autônomo") não ocorre neste passo básico porque se trata de uma cópia autônoma, em arquivo separado.

4.2 Passos básicos que alteram conjuntos de pontos de junção

Esta categoria de passos básicos possui as modificações que criam ou removem conjuntos de pontos de junção de aspecto e que modificam sua estrutura interna – conjunto de parâmetros, pontos de junção capturados, etc. A tabela 4.3 resume como cada métrica avaliada é alterada pelos passos básicos desta categoria.

Tabela 4.3: Variação das métricas ao usar os passos básicos que alteram conjuntos de pontos de junção

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Criar conjunto de pontos de junção	=	=	=	=	$>_{(X)}$	=	=	$\geq_{(X)}$	=	=
Remover conjunto de pontos de junção	=	=	=	=	$<_{(X)}$	=	=	$\leq_{(X)}$	=	=
Adicionar parâmetro a conjunto de pontos de junção	=	=	=	=	=	=	=	$\geq_{(1)}$	=	=
Remover parâmetro de conjunto de pontos de junção	=	=	=	=	=	=	=	$\leq_{(1)}$	=	=
Adicionar ponto de junção a conjunto de pontos de junção	=	=	=	=	$\gtrless$	=	=	$\geq_{(1)}$	=	=
Remover ponto de junção de conjunto de pontos de junção	=	=	=	=	$\gtrless$	=	=	$\leq_{(1)}$	=	=
Capturar contexto de ponto de junção	=	=	=	=	$\geq_{(X)}$	=	=	=	=	=
Alterar referência a componente em `within` de conjunto de pontos de junção	=	=	=	=	=	=	=	$\gtrless_{(-1a1)}$	=	=

4.2.1 Criar conjunto de pontos de junção

Modificação: adicionar um conjunto de pontos de junção em um aspecto.

```
public aspect SomeAspect {
    (...)
}
```

```
public aspect SomeAspect {
    pointcut pointcutName(<parâmetros>):  <condições>;
    (...)
}
```

Métricas alteradas:

LOC: $>_{(X)}$

A criação de um novo conjunto de pontos de junção adicionará algumas linhas de código ao aspecto, sendo que a quantidade de linhas adicionadas dependerá do estilo de programação adotado e da quantidade de parâmetros e condições que estabelecem o conjunto.

CBC: $\geq_{(X)}$

Componentes são acoplados ao conjunto de pontos de junção de duas formas: por seu uso na lista de parâmetros e por sua interceptação nas condições que estabelecem o alcance do conjunto. A adição de um novo conjunto de pontos de junção no aspecto, portanto, traz novos acoplamentos ao aspecto, o que aumenta o valor de CBC caso algum desses acoplamentos não exista anteriormente (lembrando que acoplamentos ao mesmo componente são contados apenas uma vez por CBC, conforme visto em 2.3.1.2).

4.2.2 Remover conjunto de pontos de junção

Modificação: remover um conjunto de pontos de junção de um aspecto.

```
public aspect SomeAspect {
    pointcut pointcutName(<parâmetros>): <condições>;
    (...)
}
```

```
public aspect SomeAspect {
    ~~pointcut pointcutName(<parâmetros>):  <condições>;~~
    (...)
}
```

Métricas alteradas:

LOC: $<_{(X)}$

A remoção do conjunto de pontos de junção removerá do aspecto todas as linhas de código de sua definição, sendo que a quantidade de linhas removidas dependerá do estilo de programação adotado e da quantidade de parâmetros e condições que estabelecem o conjunto.

CBC: $\leq_{(X)}$

Se algum componente estiver acoplado ao aspecto apenas pelo conjunto de pontos de junção, então a remoção do conjunto eliminará este acoplamento, o que diminui o valor de CBC. Esta métrica diminuirá tantas unidades quanto forem os componentes nesta situação – que podem ser vários, pois conjunto de pontos de junção possuem acoplamentos a componentes de duas formas: ao usá-los na lista de parâmetros e ao interceptá-los nas condições que estabelecem o alcance do conjunto.

4.2.3 Adicionar parâmetro a conjunto de pontos de junção

Modificação: adicionar um parâmetro à lista de parâmetros de um conjunto de pontos de junção.

```
public aspect SomeAspect {
    pointcut pointcutName(Param1 p1, Param2 p2, ...): <condições>;
    (...)
}
```

```
public aspect SomeAspect {
    pointcut pointcutName(Param0 p0, Param1 p1, Param2 p2, ...):  <condições>;
    (...)
}
```

Métricas alteradas:

Apesar de este passo básico não adicionar nenhum comando novo ao aspecto, é possível que LOC sofra um aumento dependendo do estilo de programação adotado. Isso porque ao aumentar o tamanho da lista de parâmetros pode ser necessário usar mais uma linha para definir o conjunto de pontos de junção, o que modifica o valor de LOC. No entanto, por se tratar de uma modificação dependente de estilo de programação, ela não será considerada como alteração relevante no valor desta métrica.

CBC: $\geq_{(1)}$

Se o aspecto não fosse originalmente acoplado ao componente adicionado como novo parâmetro, então este passo básico aumentará o número de componentes acoplados a este aspecto, incrementando o valor de CBC em uma unidade.

4.2.4 Remover parâmetro de conjunto de pontos de junção

Modificação: remover um parâmetro da lista de parâmetros de um conjunto de pontos de junção.

```
public aspect SomeAspect {
    pointcut pointcutName(Param0 p0, Param1 p1, Param2 p2, ...): <condições>;
    (...)
}
```

```
public aspect SomeAspect {
    pointcut pointcutName(~~Param0 p0,~~ Param1 p1, Param2 p2, ...):  <condições>;
    (...)
}
```

Métricas alteradas:

Apesar de este passo básico não remover nenhum comando do aspecto, é possível que LOC sofra um aumento dependendo do estilo de programação adotado. Isso porque ao diminuir o tamanho da lista de parâmetros pode ser reduzido o número de linhas de código necessárias para definir o conjunto de pontos de junção, o que modifica o valor de LOC. No entanto, por se tratar de uma modificação dependente de estilo de programação, ela não será considerada como alteração relevante no valor desta métrica.

CBC: $\leq_{(1)}$

Se não houver no aspecto nenhuma outra referência ao componente a ser removido da lista de parâmetros, então sua remoção fará com que o aspecto deixe de estar acoplado a este componente, diminuindo assim o valor de CBC em uma unidade.

4.2.5 Adicionar ponto de junção a conjunto de pontos de junção

Modificação: alterar as condições de um conjunto de pontos de junção de forma que ele passe a capturar mais um ponto de junção.

```
public aspect SomeAspect {
    pointcut pointcutName(<parâmetros>): <condições>;
    (...)
}
```

```
public aspect SomeAspect {
    pointcut pointcutName(<parâmetros>): <condições alteradas>;
    (...)
}
```

Métricas alteradas:
LOC: $\geqslant$

A adição de um ponto de junção em um conjunto deve ser feita alterando as condições que descrevem esse conjunto. Uma forma de fazer isso é adicionar uma condição que explicite exatamente o ponto a ser adicionado ao conjunto, conforme o exemplo abaixo:

```
public aspect SomeAspect {
    pointcut pointcutName(<parâmetros>):
        <condições iniciais>;
    (...)
}
```

```
public aspect SomeAspect {
    pointcut pointcutName(<parâmetros>):
        <condições iniciais>
        && execution(public void SomeComponent.someOperation());
    (...)
}
```

Adicionando o ponto de junção desta maneira, o número de linhas de código do aspecto aumenta uma unidade.

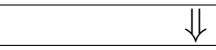
No entanto, como as condições que definem o conjunto de pontos de junção são estabelecidas por descrições textuais – onde caracteres-coringa como '`*`' e '`..`' podem ser usados para agrupar diversas condições em uma só –, é possível também adicionar um ponto de junção sem alterar ou até mesmo diminuindo o número de linhas de código do aspecto. Se for possível agrupar a nova condição às condições existentes, o valor de LOC não aumentará com a execução deste passo básico.

Os exemplos abaixo mostram duas maneiras de adicionar o mesmo ponto de junção `execution(public void SomeComponent.someOperation())` do exemplo anterior, de forma a respectivamente manter e diminuir o número de linhas de código:

```
public aspect SomeAspect {
    pointcut pointcutName(<parâmetros>):
        execution(public void SomeComponent.oneOperation() &&
        execution(public void AnotherComponent.anotherOperation() &&
        (...)
        execution(public void AThirdComponent.aThirdOperation();
    (...)
}
```

```
public aspect SomeAspect {
    pointcut pointcutName(<parâmetros>):
        ~~execution(public void SomeComponent.oneOperation() &&~~
        execution(public void SomeComponent.*Operation() &&
        execution(public void AnotherComponent.anotherOperation() &&
        (...)
        execution(public void AThirdComponent.aThirdOperation();
    (...)
}
```

```
public aspect SomeAspect {
    pointcut pointcutName(<parâmetros>):
        execution(public void SomeComponent.oneOperation() &&
        execution(public void SomeComponent.anotherOperation() &&
        (...)
        execution(public void SomeComponent.aThirdOperation();
    (...)
}
```



```
public aspect SomeAspect {
    pointcut pointcutName(<parâmetros>):
        ~~execution(public void SomeComponent.oneOperation() &&~~
        ~~execution(public void SomeComponent.anotherOperation() &&~~
        ~~(...)~~
        ~~execution(public void SomeComponent.aThirdOperation();~~
        execution(public void SomeComponent.*Operation();
    (...)
}
```

Assim, é possível adicionar o ponto de junção de formas diferentes, que aumentam ou diminuem o número de linhas de código do aspecto. Para saber como LOC será afetado por este passo básico, é preciso analisar o código em que o passo será aplicado.

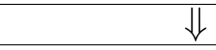
CBC: $\geq_{(1)}$

Após executar este passo básico, o componente interceptado pelo ponto de junção adicionado estará acoplado ao aspecto. Se originalmente o acoplamento do aspecto a esse componente não existisse, então o passo terá aumentado o valor de CBC em uma unidade, caso contrário CBC não se altera.

4.2.6 Remover ponto de junção de conjunto de pontos de junção

Modificação: alterar as condições de um conjunto de pontos de junção de forma que ele passe a capturar um ponto de junção a menos.

```
public aspect SomeAspect {
    pointcut pointcutName(<parâmetros>): <condições>;
    (...)
}
```



```
public aspect SomeAspect {
    pointcut pointcutName(<parâmetros>): <condições alteradas>;
    (...)
}
```

Métricas alteradas:

LOC: $\geqslant$

A remoção de um ponto de junção em um conjunto deve ser feita alterando as condições que descrevem esse conjunto. Caso a condição que explicite exatamente o ponto a ser removido não agrupe outros pontos [1], basta removê-la para eliminar o ponto de junção, como mostra o exemplo abaixo:

```
pointcut pointcutName(<parâmetros>):
    <condição 1> &&
    <condição 2> &&
    (...)
    <condição a ser removida> &&
    (...)
    <condição N>;
```

[1] O agrupamento é feito com o uso de caracteres-coringa, como '*' e '..'. No entanto, é possível usar estes caracteres e ainda assim a condição capturar apenas um ponto de junção. Isto ocorre caso apenas um local do código satisfaça as condições estabelecidas pela condição.

⇓

```
pointcut pointcutName(<parâmetros>):
    <condição 1> &&
    <condição 2> &&
    (...)
    <condição a ser removida> &&
    (...)
    <condição N>;
```

Removendo o ponto de junção desta maneira, o número de linhas de código do aspecto diminui uma unidade.

No entanto, como é possível capturar diversos pontos de junção em uma mesma condição, a remoção de um ponto de junção também pode ser feita sem alterar ou até mesmo aumentando o número de linhas de código do aspecto. Se o ponto de junção a ser removido for capturado por uma condição que agrupa outros pontos, o valor de LOC não diminuirá com a execução deste passo básico.

Os exemplos abaixo mostram duas maneiras de remover o mesmo ponto de junção `execution(public void SomeComponent.someOperation())`, de forma a respectivamente manter e aumentar o número de linhas de código:

```
pointcut pointcutName(<parâmetros>):
    execution(public void SomeComponent.*Operation() &&
    execution(public void AnotherComponent.anotherOperation() &&
    execution(public void AThirdComponent.aThirdOperation();
```

⇓

```
pointcut pointcutName(<parâmetros>):
    execution(public void SomeComponent.*Operation() &&
    execution(public void SomeComponent.oneOperation() &&
    execution(public void AnotherComponent.anotherOperation() &&
    execution(public void AThirdComponent.aThirdOperation();
```

```
pointcut pointcutName(<parâmetros>):
    execution(public void SomeComponent.*Operation();
```

⇓

```
pointcut pointcutName(<parâmetros>):
    execution(public void SomeComponent.*Operation();
    execution(public void SomeComponent.oneOperation() &&
    execution(public void SomeComponent.anotherOperation() &&
    execution(public void SomeComponent.aThirdOperation();
```

Assim, é possível remover o ponto de junção de formas diferentes, que aumentam ou diminuem o número de linhas de código do aspecto. Para saber como LOC será afetado por este passo básico, é preciso analisar o código em que o passo será aplicado.

CBC: $\leq_{(1)}$

Se o componente interceptado pelo ponto de junção removido não for mais referenciado ou interceptado pelo aspecto em outras partes do corpo do mesmo, então este passo básico terá diminuído o número de componentes acoplados ao aspecto. Neste caso, o valor de CBC terá sido decrementado em uma unidade.

4.2.7 Capturar contexto de ponto de junção

Modificação: adicionar a um conjunto de pontos de junção condições que capturem o contexto necessário a seus pontos de junção, a partir de parâmetros existentes no conjunto.

```
public aspect SomeAspect {
    pointcut pointcutName(Param1 p1, Param2 p2, ...):
        <condições iniciais>;
    (...)
}
```

⇓

```
public aspect SomeAspect {
    pointcut pointcutName(Param1 p1, Param2 p2, ...):
        <condições iniciais>
        && this(p1)
        && target(p2);
    (...)
}
```

Métricas alteradas:

LOC: $\geq_{(X)}$

Cada nova condição será adicionada em uma nova linha, o que aumenta o número de linhas de código do conjunto de pontos de junção.

4.2.8 Alterar referência a componente em `within` de conjunto de pontos de junção

Modificação: alterar o componente dentro do delimitador de escopo `within` de um conjunto de pontos de junção.

```
public aspect SomeAspect {
    pointcut pointcutName(<parâmetros>):
        <condições>
        && within (SomeComponent);
    (...)
}
```

⇓

```
public aspect SomeAspect {
    pointcut pointcutName(<parâmetros>):
        <condições>
        && within (AnotherComponent);
    (...)
}
```

Métricas alteradas:

CBC: $\gtrless_{(-1a1)}$

O delimitador de escopo `within` define qual componente é interceptado pelo conjunto de pontos de junção. Ao trocar o componente dentro deste delimitador é alterado o componente interceptado. Portanto, o componente originalmente interceptado deixará de sê-lo e o componente trocado passará a ser interceptado. Dessa forma, o valor de CBC pode:

Aumentar uma unidade: se o componente original for acoplado ao aspecto em outro local que não o conjunto de pontos de junção, e o componente trocado não for originalmente acoplado ao aspecto;

Diminuir uma unidade: se o componente original não for acoplado ao aspecto em nenhum outro local que não o conjunto de pontos de junção alterado, e o componente trocado for acoplado ao aspecto em outro local do aspecto;

Não se alterar: caso contrário.

4.3 Passos básicos que alteram operações

Nesta categoria de passos básicos estão as modificações que criam ou removem operações de componentes, que modificam sua estrutura interna – conjunto de parâmetros, variáveis locais e linhas de código –, e que alteram sua classificação (concreta/abstrata e obsoleta). A tabela 4.4 resume como cada métrica avaliada é alterada pelos passos básicos desta categoria.

Tabela 4.4: Variação das métricas ao usar os passos básicos que alteram operações

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Criar operação vazia	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(2)}$	=	$>_{(1)}$	$\geq_{(1)}$	=	$\geq_{(X)}$
Remover operação vazia	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(2)}$	=	$<_{(2)}$	=	$<_{(1)}$	$\leq_{(1)}$	=	$\leq_{(X)}$
Adicionar parâmetro a operação	=	=	=	=	=	=	$>_{(1)}$	$\geq_{(1)}$	=	=
Remover parâmetro de operação	=	=	=	=	=	=	$<_{(1)}$	$\leq_{(1)}$	=	=
Adicionar variável local a operação	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(1)}$	=	=	$\geq_{(1)}$	=	=
Remover variável local de operação	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(2)}$	=	$<_{(1)}$	=	=	$\leq_{(1)}$	=	=
Mover código entre operações de componentes diferentes	$\gtrless_{(-1a1)}$	$\gtrless_{(-1a1)}$	$\gtrless_{(-2a2)}$	=	=	=	=	$\gtrless$	=	$\gtrless$
Mover código entre operações do mesmo componente	=	$\gtrless_{(-1a1)}$	$\gtrless_{(-2a2)}$	=	=	=	=	=	=	$\gtrless$
Trocar referência a `this` por parâmetro de operação	=	=	=	=	=	=	=	=	=	=
Trocar chamada a operação sobrecarregada por versão com mais parâmetros	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$\geq_{(X)}$	=	=	$\geq_{(X)}$	=	=
Trocar chamada a operação sobrecarregada por versão com menos parâmetros	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(X)}$	=	$\leq_{(X)}$	=	=	$\leq_{(X)}$	=	=
Trocar passagem de parâmetro em construtor por chamada de método de escrita	=	=	=	=	$>_{(X)}$	=	=	=	=	=
Adicionar linha de leitura/escrita de valor de atributo a operação	=	=	=	=	$>_{(1)}$	=	=	=	=	$\leq_{(X)}$
Remover linha de leitura/escrita de valor de atributo a operação	=	=	=	=	$<_{(1)}$	=	=	=	=	$\geq_{(X)}$
Substituir referência direta a atributo por chamada a operação de leitura ou escrita	=	$\geq_{(1)}$	=	=	=	=	=	=	=	$\geq_{(X)}$
Substituir chamada a operação de leitura ou escrita por referência direta a atributo	=	$\leq_{(1)}$	=	=	=	=	=	=	=	$\leq_{(X)}$
Adicionar linhas de código a operação	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(X)}$	=	$>_{(X)}$	=	=	$\geq_{(X)}$	=	$\leq_{(X)}$
Remover linhas de código de operação	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(X)}$	=	$<_{(X)}$	=	=	$\leq_{(X)}$	=	$\geq_{(X)}$
Adicionar chamada a operação	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(1)}$	=	=	$\geq_{(1)}$	=	=
Marcar operação como obsoleta	=	=	=	=	=	=	=	=	=	=
Remover marcação de operação obsoleta	=	=	=	=	=	=	=	=	=	=
Tornar operação abstrata	=	=	=	=	$<_{(1)}$	=	=	=	=	$\gtrless$
Tornar operação concreta	=	=	=	=	$>_{(1)}$	=	=	=	=	$\gtrless$

4.3.1 Criar operação vazia

Modificação: adicionar uma operação sem parâmetros e de corpo vazio a um componente.

```
public class SomeClass {
    (...)
}
```

```
public class SomeClass {
    public ReturnComponent newOperation() {
    }
    (...)
}
```

Métricas alteradas:

CDC: $\geq_{(1)}$

Se a operação criada tiver como objetivo contribuir com a implementação de um interesse transversal, e se originalmente este interesse não estiver relacionado ao componente, então a adição da operação fará com que o interesse esteja disperso em mais um componente, o que aumenta o valor de CDC em uma unidade.

CDO: $\geq_{(1)}$

Se a operação criada tiver como objetivo contribuir com a implementação de um interesse transversal, então este interesse estará disperso em mais uma operação, aumentando assim o valor de CDO em uma unidade.

CDLOC: $\geq_{(2)}$

Se a operação criada tiver como objetivo contribuir com a implementação de um interesse transversal, e se ela não puder ser posicionada logo acima ou logo abaixo de uma linha sombreada, então sua adição criará dois novos pontos de transição no componente – um no início da operação e um no fim.

LOC: $>_{(2)}$

Com a criação da operação são adicionadas duas linhas ao componente: a de declaração da operação e a de fechamento de seu escopo (em AspectJ e Java representada por "}").

WOC: $>_{(1)}$

A criação da operação adiciona uma nova complexidade à soma que calcula WOC, que neste caso vale 1 porque a operação não possui parâmetros.

CBC: $\geq_{(1)}$

Se a operação criada não tiver retorno vazio (`void`) e se o tipo de retorno não estiver acoplado ao componente antes do passo básico, então a criação da operação criará um novo acoplamento entre os dois componentes, aumentando assim o valor de CBC em uma unidade.

LCOO: $\geq_{(X)}$

Caso haja outras operações no componente, a operação adicionada por este passo básico formará novos pares de operação com elas. Todos esses pares não acessarão um atributo em comum porque o corpo da operação adicionada é vazio, então obviamente esta operação não pode acessar nenhum atributo do componente. Dessa forma, o valor de LCOO aumentará tanto quanto forem as operações existentes no componente antes da execução do passo básico.

4.3.2 Remover operação vazia

Modificação: remover de um componente uma operação sem parâmetros e de corpo vazio.

```
public class SomeClass {
    public ReturnComponent operationToRemove() {
    }
    (...)
}
```

```
public class SomeClass {
    public ReturnComponent operationToRemove() {
    }
    (...)
}
```

Métricas alteradas:

CDC: $\leq_{(1)}$

Se a operação removida tiver como objetivo contribuir com a implementação de um interesse transversal, e se originalmente este interesse estiver relacionado ao componente apenas por esta operação, então a remoção da mesma fará com que o componente deixe de estar relacionado ao interesse, o que diminui o valor de CDC em uma unidade.

CDO: $\leq_{(1)}$

Se a operação removida tiver como objetivo contribuir com a implementação de um interesse transversal, então sua eliminação fará com que este interesse passe a estar disperso em uma operação a menos, diminuindo assim o valor de CDO em uma unidade.

CDLOC: $\leq_{(2)}$

Se a operação criada tiver como objetivo contribuir com a implementação de um interesse transversal, e se ela não estiver posicionada logo acima ou logo abaixo de uma linha sombreada, então sua remoção eliminará dois pontos de transição no componente – um no início da operação e um no fim.

LOC: $<_{(2)}$

Todas as linhas da operação serão eliminadas por este passo básico, o que normalmente significa duas linhas de código: a de declaração da operação e a de fechamento de seu escopo (em AspectJ e Java representada por "`}`").

WOC: $<_{(1)}$

A remoção da operação elimina uma das complexidades da soma que calcula WOC, complexidade que neste caso vale 1 porque a operação removida não possui parâmetros.

CBC: $\leq_{(1)}$

Se a operação removida não tiver retorno vazio (`void`) e se o tipo de retorno não estiver acoplado ao componente em nenhum outro local deste componente, então a remoção da operação eliminará o acoplamento existente entre os dois componentes, diminuindo assim o valor de CBC em uma unidade.

LCOO: $\leq_{(X)}$

Caso haja outras operações no componente, os pares de operações formados pela operação removida com as demais operações do componente serão eliminados por este passo básico. Todos esses pares não acessam um atributo em comum porque o corpo da operação removida é vazio, então obviamente esta operação não pode acessar nenhum atributo do componente. Dessa forma, o valor de LCOO diminuirá tanto quanto forem as operações existentes no componente depois da execução do passo básico.

4.3.3 Adicionar parâmetro a operação

Modificação: adicionar um parâmetro à lista de parâmetros de uma operação de um componente.

```
public class SomeClass {
    public ReturnType operationName(Param1 p1, Param2 p2, ...) {
        (...)
    }
    (...)
}
```

```
public class SomeClass {
    public ReturnType operationName(Param0 p0, Param1 p1, Param2 p2, ...)  {
        (...)
    }
    (...)
}
```

Métricas alteradas:

Apesar de este passo básico não adicionar nenhum comando novo ao componente, é possível que LOC sofra um aumento dependendo do estilo de programação adotado. Isso porque ao aumentar o tamanho da lista de parâmetros pode ser necessário usar mais uma linha para definir a operação, o que modifica o valor de LOC. No entanto, por se tratar de uma modificação dependente de estilo de programação, ela não será considerada como alteração relevante no valor desta métrica.

WOC: $>_{(1)}$

A adição do parâmetro à operação faz com que a complexidade da mesma aumente seu valor (calculado em função do número de parâmetros) em uma unidade. Conseqüentemente, a soma das complexidades das operações também aumentará da mesma forma, o que torna WOC uma unidade maior que seu valor original.

CBC: $\geq_{(1)}$

Se o componente não fosse originalmente acoplado ao tipo do novo parâmetro, então este passo básico criará o acoplamento entre ambos, aumentando assim o valor de CBC em uma unidade.

4.3.4 Remover parâmetro de operação

Modificação: remover um parâmetro da lista de parâmetros de uma operação de um componente.

```
public class SomeClass {
    public ReturnType operationName(Param0 p0, Param1 p1, Param2 p2, ...) {
        (...)
    }
    (...)
}
```

⇓

```
public class SomeClass {
    public ReturnType operationName(~~Param0 p0,~~ Param1 p1, Param2 p2, ...)  {
        (...)
    }
    (...)
}
```

Métricas alteradas:

Apesar de este passo básico não remover nenhum comando novo ao componente, é possível que LOC sofra uma diminuição dependendo do estilo de programação adotado. Isso porque ao diminuir o tamanho da lista de parâmetros pode ser possível usar uma linha a menos para definir a operação, o que modifica o valor de LOC. No entanto, por se tratar de uma modificação dependente de estilo de programação, ela não será considerada como alteração relevante no valor desta métrica.

WOC: $<_{(1)}$

A remoção do parâmetro da operação faz com que a complexidade da mesma diminua seu valor (calculado em função do número de parâmetros) em uma unidade. Conseqüentemente, a soma das complexidades das operações também diminuirá da mesma forma, o que torna WOC uma unidade menor que seu valor original.

CBC: $\leq_{(1)}$
Se o componente não for acoplado ao tipo do parâmetro removido em nenhuma outra parte do código do componente, então ao final do passo básico não haverá mais acoplamento entre ambos, o que diminui o valor de CBC em uma unidade.

4.3.5 Adicionar variável local a operação

Modificação: adicionar uma variável local a uma operação de um componente.

```
public class SomeClass {
    public ReturnType operationName(<parâmetros>) {
        (...)
    }
    (...)
}
```

```
public class SomeClass {
    public ReturnType operationName(<parâmetros>) {
        VariableType variableName;
        (...)
    }
    (...)
}
```

Métricas alteradas:

CDC: $\geq_{(1)}$
Se a variável for usada para implementar um interesse transversal e se originalmente o componente não estiver relacionado a este interesse, então o passo básico aumentará o número de componentes por onde o interesse transversal está disperso, o que incrementa o valor de CDC em uma unidade.

CDO: $\geq_{(1)}$
Se a variável for usada para implementar um interesse transversal e se originalmente a operação não estiver relacionada a este interesse, então o passo básico aumentará o número de operações por onde o interesse transversal está disperso, o que incrementa o valor de CDO em uma unidade.

CDLOC: $\geq_{(2)}$
Se a variável for usada para implementar um interesse transversal, sua linha de declaração será sombreada. Se logo acima ou logo abaixo dela não houver uma linha sombreada, então sua adição criará dois novos pontos de transição no componente, aumentando assim o valor de CDLOC em duas unidades.

LOC: $>_{(1)}$
A criação da variável local é feita adicionando uma linha de código à operação, o que aumenta o valor de LOC em uma unidade.

CBC: $\geq_{(1)}$
Se o componente não for originalmente acoplado ao tipo da nova variável, então este passo básico criará o acoplamento entre ambos, aumentando assim o valor de CBC em uma unidade.

4.3.6 Remover variável local de operação

Modificação: remover uma variável local de uma operação de um componente.

```
public class SomeClass {
    public ReturnType operationName(<parâmetros>) {
        VariableType variableName;
        (...)
    }
    (...)
}
```

```
public class SomeClass {
    public ReturnType operationName(<parâmetros>) {
        ~~VariableType variableName;~~
        (...)
    }
    (...)
}
```

Métricas alteradas:

CDC: $\leq_{(1)}$
Se a variável fosse usada para implementar um interesse transversal e se o componente não estiver relacionado a este interesse em nenhuma outra parte de seu código, então o passo básico diminuirá o número de componentes por onde o interesse transversal está disperso, o que decrementa o valor de CDC em uma unidade.

CDO: $\leq_{(1)}$
Se a variável fosse usada para implementar um interesse transversal e se a operação não estiver relacionada a este interesse em nenhum outro trecho, então o passo básico diminuirá o número de operações por onde o interesse transversal está disperso, o que decrementa o valor de CDO em uma unidade.

CDLOC: $\leq_{(2)}$
Se a variável fosse usada para implementar um interesse transversal, sua linha de declaração seria sombreada. Se logo acima ou logo abaixo dela não houver uma linha sombreada, então sua remoção eliminará dois pontos de transição do componente, diminuindo assim o valor de CDLOC em duas unidades.

LOC: $<_{(1)}$
A remoção da variável local é feita eliminando uma linha de código da operação, o que diminui o valor de LOC em uma unidade.

CBC: $\leq_{(1)}$
Se o componente não for acoplado ao tipo da variável em nenhuma outra parte de seu código, então este passo básico eliminará o acoplamento entre ambos, diminuindo assim o valor de CBC em uma unidade.

4.3.7 Mover código entre operações de componentes diferentes

Modificação: transferir algumas linhas de código em seqüência de uma operação de um componente para uma operação de outro. Para mover um bloco contíguo de linhas de código, este passo básico deve ser utilizado uma única vez para mover o bloco todo, e não diversas vezes para mover cada uma das partes do bloco, pois a avaliação deste passo foi feita levando-se esta premissa em consideração.

```
public class SomeClass {
    public ReturnType1 operation1(<parâmetros1>) {
        <trecho inicial da operação 1>
        <trecho a ser movido>
        <trecho final da operação 1>
    }
    (...)
}
```

```
public class AnotherClass {
    public ReturnType2 operation2(<parâmetros2>) {
        <trecho inicial da operação 2>
        <trecho final da operação 2>
    }
    (...)
}
```

```
public class SomeClass {
    public ReturnType1 operation1(<parâmetros1>) {
        <trecho inicial da operação 1>
        <trecho a ser movido>
        <trecho final da operação 1>
    }
    (...)
}
```

```
public class AnotherClass {
    public ReturnType2 operation2(<parâmetros2>) {
        <trecho inicial da operação 2>
        <trecho a ser movido>
        <trecho final da operação 2>
    }
    (...)
}
```

Métricas alteradas:

CDC: $\gtrless_{(-1a1)}$

Se o trecho movido estiver relacionado a algum interesse transversal, então sua movimentação poderá mudar o valor de CDC. Para entender como isso ocorre, serão avaliados separadamente o componente em que o trecho estava (componente-fonte) e o componente para onde o trecho foi movido (componente-destino).

No componente-fonte, a retirada do trecho fará com que este componente deixe de estar relacionado ao interesse se não houver no restante de seu corpo nenhuma linha relacionada ao interesse. Isto faz com que o interesse transversal esteja disperso em um componente a menos do que antes do passo básico.

No componente-alvo, a adição do trecho fará com que este componente passe a estar relacionado ao interesse se não houver no restante de seu corpo nenhuma linha relacionada ao interesse. Isto faz com que o interesse transversal esteja disperso em um componente a mais do que antes do passo básico.

Juntando as avaliações acima, temos a variação total de CDC de acordo com as condições abaixo:

CDC diminui 1 unidade Se o componente-fonte deixar de estar relacionado ao interesse e o componente-alvo já for previamente relacionado;

CDC se mantém Se o componente-fonte deixar de estar relacionado ao interesse e o componente-alvo não for previamente relacionado, ou se ambos os componentes continuarem relacionados antes e depois da alteração;

CDC aumenta 1 unidade Se o componente-fonte continuar relacionado ao interesse e o componente-alvo não for previamente relacionado.

CDO: $\gtrless_{(-1a1)}$

Assim como ocorre no nível dos componentes, também as operações podem passar a estar ou deixar de ser relacionadas ao interesse do trecho (se houver algum). Usando um raciocínio semelhante ao da explicação de CDC, o valor de CDO será alterado de acordo com as condições abaixo:

CDO diminui 1 unidade Se a operação-fonte deixar de estar relacionada ao interesse e a operação-alvo já for previamente relacionada;

CDO se mantém Se a operação-fonte deixar de estar relacionada ao interesse e a operação-alvo não for previamente relacionada, ou se ambas as operações continuarem relacionadas antes e depois da alteração;

CDO aumenta 1 unidade Se a operação-fonte continuar relacionado ao interesse e a operação-alvo não for previamente relacionada.

CDLOC: $\gtrless_{(-2a2)}$

O número de pontos de transição existentes **dentro** do trecho movido não sofrerão alterações (figura 4.3), sendo apenas transferidos de um componente para outro (o que não afeta o valor de CDLOC porque esta métrica mede o valor em todo o código).

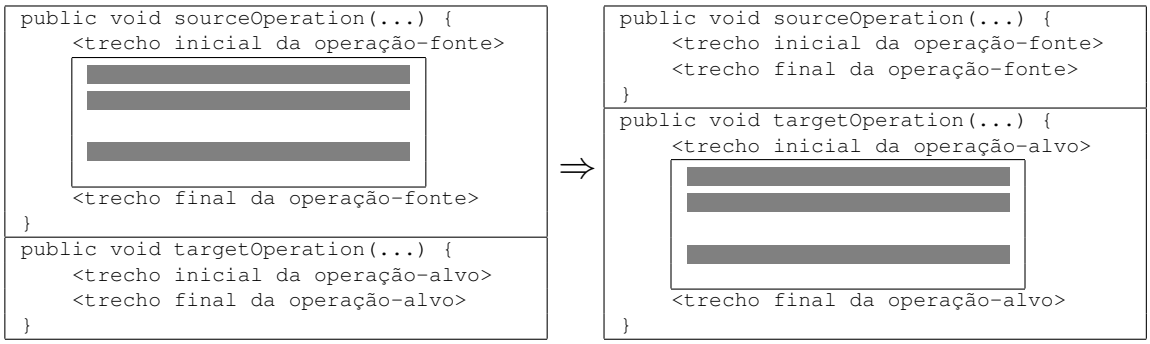
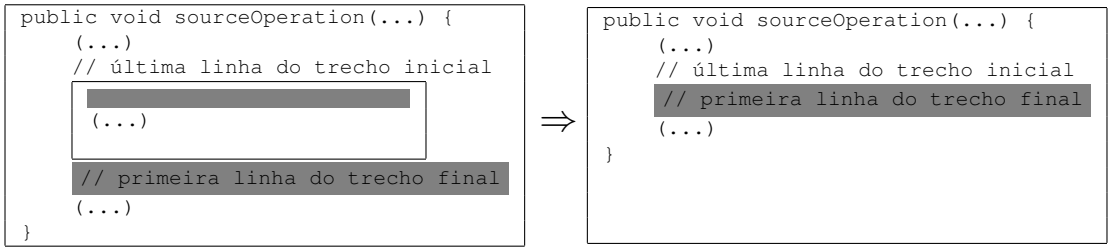


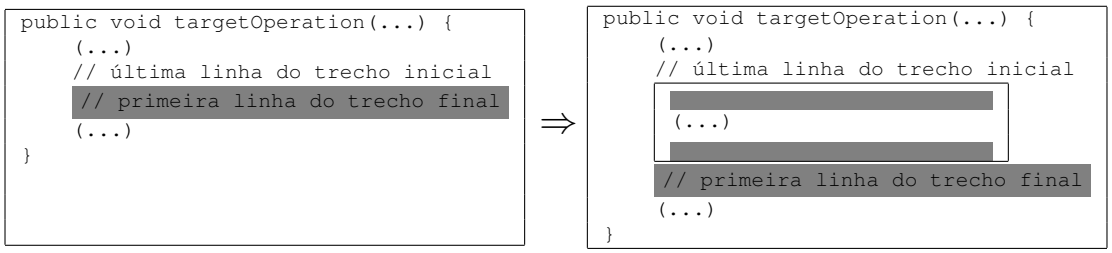
Figura 4.3: Manutenção do número de pontos de transição dentro do trecho movido

São as linhas das fronteiras do trecho movido (isto é, antes de seu início e após seu término) que podem alterar o valor de CDLOC, criando ou eliminando pontos de transição no código das operações envolvidas. Na operação-fonte, podem ser eliminados os dois pontos de transição nas fronteiras do trecho removido e criado um ponto entre os trechos inicial e final da operação. O exemplo abaixo mostra uma situação em que todas essas situações ocorrem, porém qualquer combinação delas pode acontecer, dependendo do código em que este passo básico for aplicado.



Neste exemplo, o ponto de transição entre a última linha do trecho inicial (não-sombreada) e a primeira linha do trecho movido (sombreada) é removido, bem como o ponto de transição entre a última linha do trecho movido (não-sombreada) e a primeira linha do trecho final (sombreada). Com a remoção do trecho, é criado um novo ponto de transição entre a última linha do trecho inicial e a primeira do trecho final da operação-fonte.

Já na operação-alvo ocorre o contrário: dois novos pontos de transição podem ser criados nas fronteiras do trecho recebido e um ponto pode ser eliminado entre os trechos inicial e final da operação. Todas essas situações podem acontecer ou qualquer combinação delas, dependendo do código em que o passo básico for aplicado. Assim, um caso em que apenas um ponto é criado e um é removido é mostrado abaixo:



Neste exemplo, o ponto de transição entre a última linha do trecho inicial (não-sombreada) e a primeira do trecho final (sombreada) é removido. Em relação às fronteiras no trecho movido, um novo ponto de transição é criado entre a última linha do trecho inicial (não-sombreada) e a primeira linha do trecho movido (sombreada), enquanto que a última linha do trecho movido e a primeira linha do trecho final (ambas sombreadas) não formam um novo ponto de transição.

Para saber quais os valores máximos e mínimos que CDLOC pode ser alterado, usa-se as avaliações acima para analisar o comportamento da movimentação de um **mesmo** trecho entre as operações fonte e alvo. A maior alteração negativa que este passo básico pode fazer em CDLOC teoricamente seria -3, resultado da eliminação de dois pontos de transição na operação-fonte e de um ponto na operação-alvo. No entanto, por se tratar do mesmo trecho movido, este valor nunca será alcançado. Vejamos como: para alcançar o valor mínimo na remoção do trecho da operação-fonte, devem ser eliminados dois pontos e nenhum pode ser criado na operação. Como nenhum ponto pode ser criado, a última linha do trecho inicial e a primeira linha do trecho final da operação têm que ter a mesma classificação (são ambas sombreadas ou ambas não-sombreadas). E como dois pontos devem ser eliminados, então o início e o fim do trecho movido devem ter classificações opostas às suas fronteiras, portanto a primeira e a última linha do trecho são, respectivamente, ambas não-sombreadas ou sombreadas. Para simplificar a explicação, adotaremos um dos casos acima sem perda de generalidade [2]: o trecho movido tem sua primeira e última linhas sombreadas. No caso da operação-alvo, para alcanças o valor mínimo na adição do trecho, deve ser eliminado um ponto de transição e não pode ser criado nenhum ponto na operação. Como deve ser eliminado um ponto, a última linha do trecho inicial e a primeira linha do trecho final da operação têm que ter classificações opostas (uma deve ser sombreada e a outra não). No entanto, ao mover um trecho onde a primeira e a última linha são sombreadas, um ponto de transição é criado, contrariando a condição inicial que nenhum ponto poderia ser criado. Assim, o mínimo que o valor de CDLOC pode alcançar é duas unidades a menos que seu valor original.

[2] o caso oposto funciona da mesma maneira, apenas invertendo linhas sombreadas por não-sombreadas e vice-versa

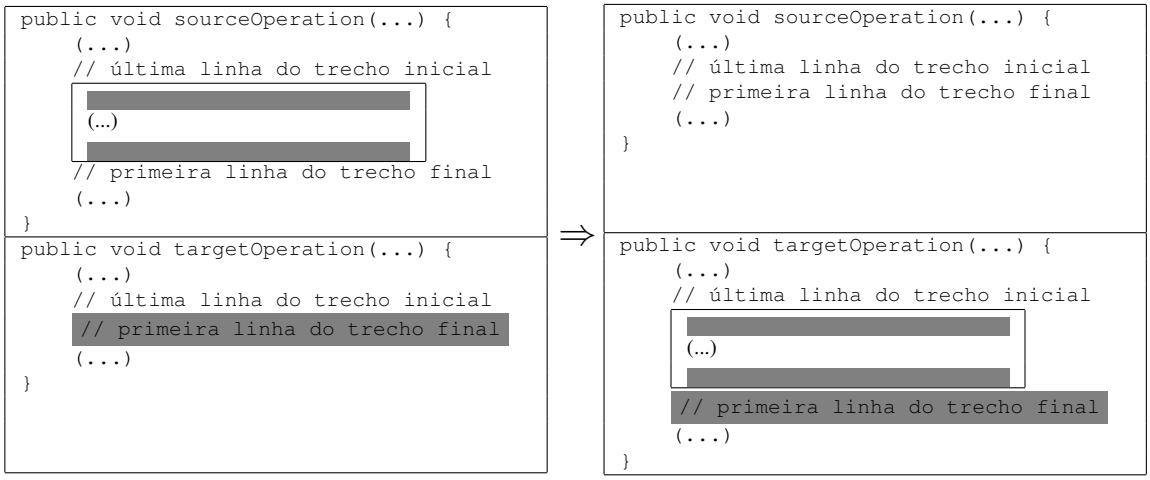


Figura 4.4: Exemplo de maior variação negativa que CDLOC pode ter em *Mover código entre operações de componentes diferentes*: 3 pontos de transição eliminados e 1 criado

A obtenção da maior variação positiva de CDLOC é feita de forma análoga, apenas buscando criar mais pontos de transição de que removê-los. O resultado obtido é que não é possível aumentar CDLOC em três unidades, sendo possível apenas criar dois pontos de transição a mais do que o número de pontos eliminados. Dessa forma, a variação total de CDLOC varia de -2 a +2.

CBC: $\geqslant$

O trecho movido está acoplado a alguns componentes, acoplamentos estes que serão movidos junto com ele para o componente-alvo. O componente-fonte perderá acoplamentos se algum dos componentes acoplados ao trecho não estiver relacionado ao componente-fonte no restante de seu corpo, enquanto que o componente-alvo ganhará acoplamentos se os componentes acoplados ao trecho não forem previamente acoplados ao componente-alvo. O saldo deste ganho e perda depende do código em que este passo básico é aplicado e só pode ser definido analisando o caso específico da aplicação do passo.

LCOO: $\geqslant$

Como este passo básico trata da movimentação de um trecho de código genérico, não está definida nenhuma informação sobre o acesso a atributos dos componentes envolvidos. Assim, é possível que dentre as linhas movidas entre os componentes haja comandos de acesso a atributos do componente-fonte ou do componente-alvo. Este tipo de comando pode alterar a classificação dos pares de operações dos componentes, fazendo com que um par originalmente considerado na parcela negativa de LCOO (operações que não compartilham atributos) passe a ser considerado na parcela positiva (operações com atributos compartilhados), e vice-versa. O quanto LCOO varia depende dos comandos movidos entre os componentes e das demais operações dos mesmos.

4.3.8 Mover código entre operações do mesmo componente

Modificação: transferir algumas linhas de código em seqüência entre operações de um componente. Para mover um bloco contíguo de linhas de código, este passo básico deve ser utilizado uma única vez para mover o bloco todo, e não diversas vezes para mover cada uma das partes do bloco, pois a avaliação deste passo foi feita levando-se esta premissa em consideração.

```
public class SomeClass {
    public ReturnType1 operation1(<parâmetros1>) {
        <trecho inicial da operação 1>
        <trecho a ser movido>
        <trecho final da operação 1>
    }
    public ReturnType2 operation2(<parâmetros2>) {
        <trecho inicial da operação 2>
        <trecho final da operação 2>
    }
    (...)
}
```

```
public class SomeClass {
    public ReturnType1 operation1(<parâmetros1>) {
        <trecho inicial da operação 1>
        <trecho a ser movido>
        <trecho final da operação 1>
    }
    public ReturnType2 operation2(<parâmetros2>) {
        <trecho inicial da operação 2>
        <trecho a ser movido>
        <trecho final da operação 2>
    }
    (...)
}
```

Métricas alteradas:

Esta refatoração é muito semelhante a *Mover código entre operações de componentes diferentes*, por isso suas métricas alteradas também são. As únicas diferenças estão nas métricas **CDC** e **CBC**, que não se alteram neste passo básico porque a movimentação das linhas de código é feita dentro do mesmo componente.

CDO ($\gtrless_{(-1a1)}$), **CDLOC** ($\gtrless_{(-2a2)}$) e **LCOO** ($\gtrless$):
Mesmas justificativas de 4.3.7, porém com o componente-fonte e componente-alvo sendo o mesmo componente.

4.3.9 Trocar referência a `this` por parâmetro de operação

Modificação: trocar uma referência à variável `this` por um parâmetro da operação. A variável `this` e o parâmetro devem ser do mesmo tipo.

```
public class SomeClass {
    public void someOperation(SomeClass sc) {
        (...)
        SomeClass temp = this;
        (...)
        this.anotherOperation();
        (...)
    }
    (...)
}
```

```
public class SomeClass {
    public void someOperation(SomeClass sc) {
        (...)
        SomeClass temp = sc;
        (...)
        sc.anotherOperation();
        (...)
    }
    (...)
}
```

Este passo básico também pode ser aplicado em um componente que usa `this` sem ter a intenção de referenciar a ele mesmo. Por exemplo, em uma etapa intermediária de

uma refatoração, um trecho de código com referências a `this` pode ser movido de um componente para o outro, o que torna esta variável temporariamente uma referência ao componente errado, como mostra o exemplo abaixo. Usar este passo básico corrige esse suposto "erro".

```
public class SomeClass {
    public void someOperation() {
        this.anotherOperation();
        (...)
    }
    (...)
}
```

```
public aspect SomeAspect {
    before(SomeClass sc): somePointcut(sc) {
        (...)
    }
    (...)
}
```

```
public class SomeClass {
    public void someOperation() {
        ~~this.anotherOperation();~~
        (...)
    }
    (...)
}
```

```
public aspect SomeAspect {
    before(SomeClass sc): somePointcut(sc) {
        (...)
        this.anotherOperation();
        (...)
    }
    (...)
}
```

```
public class SomeClass {
    public void someOperation() {
        (...)
    }
    (...)
}
```

```
public aspect SomeAspect {
    before(SomeClass sc): somePointcut(sc) {
        (...)
        sc.anotherOperation();
        (...)
    }
    (...)
}
```

Métricas alteradas:

Como se trata apenas de alterar o "nome" da variável (uma vez que `this` e o parâmetro devem ser do mesmo tipo), este passo básico não modifica o valor de nenhuma das métricas escolhidas. Não há alteração no tamanho do software, os interesses transversais continuarão entrelaçando e entrecortando os mesmos trechos de código, o acoplamento dos componentes será o mesmo e sua coesão também.

4.3.10 Trocar chamada a operação sobrecarregada por versão com mais parâmetros

Modificação: trocar uma chamada a uma operação sobrecarregada por sua versão com mais parâmetros.

```
public class SomeClass {
    public void someOperation(Param1 p1) {
        (...)
    }
    public void someOperation(Param1 p1, Param2 p2) {
        (...)
    }
    (...)
}
```

```
public class AnotherClass {
    public void operation(SomeClass sc) {
        Param1 p1 = (...);
        sc.someOperation(p1);
        (...)
    }
}
```

```
public class SomeClass {
    public void someOperation(Param1 p1) {
        (...)
    }
    public void someOperation(Param1 p1, Param2 p2) {
        (...)
    }
    (...)
}
```

```
public class AnotherClass {
    public void operation(SomeClass sc) {
        Param1 p1 = (...);
        Param2 p2 = (...);
        sc.someOperation(p1, p2);
        (...)
    }
}
```

Métricas alteradas:

A troca da chamada da operação sobrecarregada por sua versão com mais parâmetros torna necessário obter novos valores para serem passados no lugar dos parâmetros extras, o que faz com que alguma das métricas sejam alteradas por este passo básico.

CDC: $\geq_{(1)}$

Se um dos novos valores for obtido usando componentes relacionados a um interesse transversal diferente dos interesses relacionados ao componente onde a chamada é trocada, então este componente passará a estar relacionado ao interesse, o que aumenta em uma unidade o número de componentes pelos quais o interesse transversal está disperso.

CDO: $\geq_{(1)}$

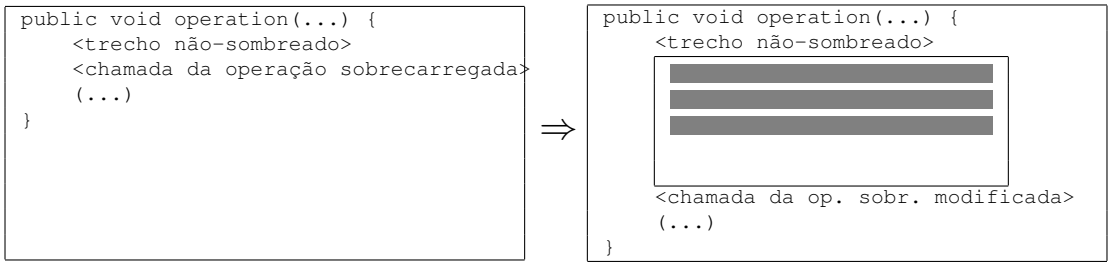
Se um dos novos valores for obtido usando operações relacionadas a um interesse transversal diferente dos interesses relacionados à operação onde a chamada é trocada, então esta operação passará a estar relacionada ao interesse, o que aumenta em uma unidade o número de operações pelas quais o interesse transversal está disperso.

CDLOC: $\geq_{(2)}$

Caso os novos valores não estejam disponíveis originalmente na operação (isto é, se for necessário adicionar comandos para obtê-los), novas linhas serão incorporadas a ela. Essas linhas poderão ser sombreadas ou não, dependendo dos componentes e operações usados nestas novas linhas. Por se tratar de comandos apenas para a obtenção dos novos valores, essas linhas podem ser agrupadas de acordo com seu sombreamento, formando um bloco sombreado e outro não-sombreado, o que minimiza ao máximo o aumento do número de pontos de transição.

O valor de CDLOC será alterado apenas se todas as linhas até a chamada da operação sobrecarregada (inclusive a linha da chamada) tiverem um tipo de sombreamento – to-

das sombreadas ou todas não-sombreadas – e for adicionado um bloco de sombreamento oposto a elas na obtenção dos novos valores:



LOC: $\geq_{(X)}$

Caso os novos valores não estejam disponíveis originalmente na operação, novas linhas serão incorporadas a ela, aumentando assim o valor de LOC. Esta métrica aumenta tanto quanto forem os comandos necessários para obter os novos valores.

CBC: $\geq_{(X)}$

Caso os novos valores não estejam disponíveis originalmente na operação, novos comandos serão necessários para obtê-los. Esses comandos podem fazer referência a componentes que não estavam acoplados originalmente ao componente onde a chamada será trocada, situação em que o valor de CBC aumentaria. Esta métrica é modificada tantas unidades quanto forem os acoplamentos criados pelos novos comandos.

4.3.11 Trocar chamada a operação sobrecarregada por versão com menos parâmetros

Modificação: trocar uma chamada a uma operação sobrecarregada por sua versão com menos parâmetros.

```
public class SomeClass {
    public void someOperation(Param1 p1) {
        (...)
    }
    public void someOperation(Param1 p1, Param2 p2) {
        (...)
    }
    (...)
}
```

```
public class AnotherClass {
    public void operation(SomeClass sc) {
        Param1 p1 = (...);
        Param2 p2 = (...);
        sc.someOperation(p1, p2);
        (...)
    }
}
```

⇓

```
public class SomeClass {
    public void someOperation(Param1 p1) {
        (...)
    }
    public void someOperation(Param1 p1, Param2 p2) {
        (...)
    }
    (...)
}
```

```
public class AnotherClass {
    public void operation(SomeClass sc) {
        Param1 p1 = (...);
        ~~Param2 p2 = (...);~~
        sc.someOperation(p1);
        (...)
    }
}
```

Métricas alteradas:

A troca da chamada da operação sobrecarregada por sua versão com menos parâmetros alguns valores passados originalmente se tornam desnecessários, o que faz com que alguma das métricas sejam alteradas por este passo básico.

CDC: $\leq_{(1)}$
Se algum interesse transversal só estiver presente nos comandos de obtenção dos valores desnecessários e se for possível remover esses comandos (caso os valores não sejam usados em outras partes da operação), então o componente onde a chamada é trocada deixará de estar relacionado a esse interesse após a remoção desses comandos, o que diminui em uma unidade o número de componentes pelos quais o interesse transversal está disperso.

CDO: $\leq_{(1)}$
Se algum interesse transversal só estiver presente nos comandos de obtenção dos valores desnecessários e se for possível remover esses comandos (caso os valores não sejam usados em outras partes da operação), então a operação onde a chamada é trocada deixará de estar relacionada a esse interesse após a remoção desses comandos, o que diminui em uma unidade o número de operações pelas quais o interesse transversal está disperso.

CDLOC: $\leq_{(X)}$
Caso os comandos para obtenção dos valores desnecessários possam ser removidos, linhas de código serão eliminadas da operação. Essas linhas podem ser sombreadas ou não, dependendo dos componentes e operações usados nestas linhas. O valor de CDLOC será alterado se for possível remover linhas de código e se entre elas houver pontos de transição entre linhas sombreadas e não-sombreadas (ou vice-versa).

O passo básico inverso deste passo (*Trocar chamada a operação sobrecarregada por versão com mais parâmetros*, seção 4.3.10) pode aumentar o valor de CDLOC de 0 a 2 unidades, o que não é o oposto deste passo, que pode diminuir mais que 2 unidades. Este fato ocorre porque os comandos eliminados por *Trocar chamada (...) com menos parâmetros* podem estar dispersos pelo código de uma forma muito entrelaçada, aumentando muito o número de pontos de transição entre eles, enquanto que o passo básico inverso cria os comandos necessários de maneira a minimizar o aumento do número de pontos de transição.

LOC: $\leq_{(X)}$
Caso os comandos para obtenção dos valores desnecessários possam ser removidos, linhas de código serão eliminadas da operação, diminuindo assim o valor de LOC. Esta métrica diminui tanto quanto forem os comandos usados para obter os valores desnecessários.

CBC: $\leq_{(X)}$
Caso os comandos para obtenção dos valores desnecessários possam ser removidos, é possível que alguns componentes deixem de ser usados pela operação. Se no restante do componente onde a chamada é trocada não houver mais nenhuma referência aos componentes que deixaram de ser usados na operação, então ele deixará de estar acoplado a esses componentes, diminuindo assim o valor de CBC.

4.3.12 Trocar passagem de parâmetro em construtor por chamada de método de escrita

Modificação: trocar uma instanciação com passagem de parâmetros por uma instanciação sem parâmetros e chamadas a métodos de escrita (*setters*).

```
public class SomeClass {
    public void operation() {
        AnotherClass ac = new AnotherClass(p1, p2);
        (...)
    }
}
```

```
public class SomeClass {
    public void operation() {
        AnotherClass ac = new AnotherClass(p1, p2);"
        ac.setP1(p1);
        ac.setP2(p2);
        (...)
    }
}
```

Métricas alteradas:

LOC: $>_{(X)}$

Este passo básico torna necessário usar mais comandos para instanciar e definir os valores da instâncias – antes era usado apenas o construtor com parâmetros, depois usou-se um construtor sem parâmetros e métodos de escrita. Assim, cada comando extra precisará de uma nova linha de código, o que aumenta o valor de LOC tantas unidades quanto forem os parâmetros do construtor original.

4.3.13 Adicionar linha de leitura/escrita de valor de atributo a operação

Modificação: adicionar, em uma operação, uma linha com um comando de leitura ou escrita do valor de um atributo do componente.

```
public class SomeClass {
    private SomeComponent attribute;
    public void operation() {
        (...)
    }
}
```

```
public class SomeClass {
    private SomeComponent attribute;
    public void operation() {
        SomeComponent sc = this.attribute;
        (...)
    }
}
```

Métricas alteradas:

LOC: $>_{(1)}$

Por se tratar de um passo básico que acrescenta uma linha de código, é imediato constatar que o valor de LOC aumentará em uma unidade após aplicá-lo em um componente.

LCOO: $\leq_{(X)}$

Este passo básico faz com que a operação que recebe a linha de leitura/escrita passe a acessar um atributo do componente a mais. Caso haja alguma outra operação do componente que acesse esse mesmo atributo, e se as duas operações só acessarem este atributo em comum, então o passo básico diminuirá o valor de LCOO. Isso ocorre porque originalmente o par formado pelas duas operações não compartilhava atributos do componente, portanto este par era contado na parcela positiva de LCOO. A partir do momento que a operação recebeu a linha de leitura/escrita do atributo, o par passou a compartilhar um atributo e, portanto, passou a ser contado na parcela negativa de LCOO, o que diminui o valor desta métrica em duas unidades.

A diminuição total de LCOO será igual a duas vezes o número de operações que passarem a compartilhar um atributo com a operação que recebe a linha de leitura/escrita (salvo no caso de LCOO se tornar um valor negativo por este cálculo, caso em que LCOO passa a ser zero e a diminuição desta métrica é igual ao seu valor original).

4.3.14 Remover linha de leitura/escrita de valor de atributo a operação

Modificação: remover de uma operação uma linha com um comando de leitura ou escrita do valor de um atributo do componente.

```
public class SomeClass {
    private SomeComponent attribute;
    public void operation() {
        SomeComponent sc = this.attribute;
        (...)
    }
}
```

```
public class SomeClass {
    private SomeComponent attribute;
    public void operation() {
        SomeComponent sc = this.attribute;
        (...)
    }
}
```

Métricas alteradas:

LOC: $<_{(1)}$

Por se tratar de um passo básico que remove uma linha de código, é imediato constatar que o valor de LOC diminuirá em uma unidade após aplicá-lo em um componente.

LCOO: $\geq_{(X)}$

Este passo básico faz com que a operação de onde a linha de leitura/escrita é removida deixe de acessar um atributo do componente. Caso haja alguma outra operação do componente que acesse esse mesmo atributo, e se as duas operações só acessarem este atributo em comum, então o passo básico aumentará o valor de LCOO. Isso ocorre porque originalmente o par formado pelas duas operações compartilhava atributos do componente, portanto este par era contado na parcela negativa de LCOO. A partir do momento que a linha de leitura/escrita do atributo foi removida da operação, o par deixou de compartilhar um atributo e, portanto, passou a ser contado na parcela positiva de LCOO, o que aumenta o valor desta métrica em duas unidades.

O aumento total de LCOO será igual a duas vezes o número de operações que deixarem de compartilhar um atributo com a operação de onde foi removida a linha de leitura/escrita (salvo no caso de LCOO ser inicialmente zero, caso em que LCOO pode ter um aumento menor por sua parcela negativa não ser totalmente válida no valor original da métrica).

4.3.15 Substituir referência direta a atributo por chamada a operação de leitura ou escrita

Modificação: substituir, em uma operação, o acesso direto a um atributo por seu método de leitura ou escrita. O atributo pode ser definido no próprio componente ou em outro local.

```
public class SomeClass {
    private SomeComponent attribute;
    public void operation() {
        SomeComponent sc = this.attribute;
        (...)
    }
}
```

```
public class SomeClass {
    private SomeComponent attribute;
    public void operation() {
        SomeComponent sc = this.getAttribute();
        (...)
    }
}
```

Métricas alteradas:

CDO: $\geq_{(1)}$

Se o método de leitura/escrita for definido para contribuir com a implementação de um interesse transversal, e a operação onde o acesso direto é substituído ainda não for relacionada a esse interesse, então o passo básico fará com que esta operação passe a ser considerada no cálculo de CDO, aumentando seu valor em uma unidade.

LCOO: $\geq_{(X)}$

Caso o atributo cujo acesso direto é substituído seja do próprio componente onde ocorre a substituição, este passo básico fará com que a operação passe a acessar um atributo do componente a menos. Caso haja alguma outra operação do componente que originalmente acessasse esse mesmo atributo, e se as duas operações só acessassem este atributo em comum, então o passo básico aumentará o valor de LCOO. Isso ocorre porque originalmente o par formado pelas duas operações compartilhava um atributo do componente, portanto este par era contado na parcela negativa de LCOO. A partir do momento que a operação perdeu o acesso direto ao atributo, o par deixou de compartilhar um atributo e, portanto, passou a ser contado na parcela positiva de LCOO, o que aumenta o valor desta métrica em duas unidades.

O aumento total de LCOO será igual a duas vezes o número de operações que deixarem de compartilhar um atributo com a operação onde o acesso direto foi substituído (salvo no caso de LCOO for inicialmente zero, caso em que LCOO pode ter um aumento menor por sua parcela negativa não ser totalmente válida no valor original da métrica).

4.3.16 Substituir chamada a operação de leitura ou escrita por referência direta a atributo

Modificação: substituir, em uma operação, um método de leitura ou escrita pelo acesso direto ao atributo. O atributo pode ser definido no próprio componente ou em outro local.

```
public class SomeClass {
    private SomeComponent attribute;
    public void operation() {
        SomeComponent sc = this.getAttribute();
        (...)
    }
}
```

```
public class SomeClass {
    private SomeComponent attribute;
    public void operation() {
        SomeComponent sc = this.attribute;"
        (...)
    }
}
```

Métricas alteradas:

CDO: $\leq_{(1)}$

Se o método de leitura/escrita for definido para contribuir com a implementação de um interesse transversal, e a operação onde a chamada ao método de leitura/escrita é substituída pelo acesso direto não for relacionada a esse interesse em outro trecho de código, então o passo básico fará com que esta operação deixe de ser considerada no cálculo de CDO, diminuindo seu valor em uma unidade.

LCOO: $\leq_{(X)}$

Caso o atributo cujo método de acesso é substituído seja do próprio componente onde ocorre a substituição, este passo básico fará com que a operação passe a acessar um atributo do componente a mais. Caso haja alguma outra operação do componente que originalmente acessasse esse mesmo atributo, e se as duas operações só acessassem este atributo em comum, então o passo básico aumentará o valor de LCOO. Isso ocorre porque originalmente o par formado pelas duas operações não compartilhava nenhum atributo do componente, portanto este par era contado na parcela positiva de LCOO. A partir do momento que foi adicionado à operação o acesso direto ao atributo, o par passou a compartilhar um atributo e, portanto, passou a ser contado na parcela negativa de LCOO, o que diminui o valor desta métrica em duas unidades.

A diminuição total de LCOO será igual a duas vezes o número de operações que passarem a compartilhar um atributo com a operação onde o método de acesso foi substituído (salvo no caso de LCOO se tornar um valor negativo por este cálculo, caso em que LCOO passa a ser zero e a diminuição desta métrica é igual ao seu valor original).

4.3.17 Adicionar linhas de código a operação

Modificação: adicionar alguns comandos em uma operação. Para adicionar um bloco contíguo de linhas de código, este passo básico deve ser utilizado uma única vez para adicionar o bloco todo, e não diversas vezes para adicionar cada um dos comandos do bloco, pois a avaliação deste passo foi feita levando-se esta premissa em consideração.

```
public class SomeClass {
    public void operation() {
        <trecho inicial da operação>
        <trecho final da operação>
    }
}
```

```
public class SomeClass {
    public void operation() {
        <trecho inicial da operação>
        <trecho adicionado>
        <trecho final da operação>
    }
}
```

Métricas alteradas:

CDC: $\geq_{(1)}$

Se o trecho adicionado contiver algum componente definido com objetivo de contribuir

para a implementação um interesse transversal, e se originalmente o componente onde o trecho foi adicionado não fosse relacionado a esse interesse, então a adição das linhas da operação fará com que o interesse passe a estar presente no componente, aumentando assim o valor de CDC em uma unidade.

CDO: $\geq_{(1)}$

Se o trecho adicionado usar alguma operação definida com objetivo de contribuir para a implementação um interesse transversal, e se originalmente a operação onde o trecho foi adicionado não fosse relacionada a esse interesse, então a adição das linhas da operação fará com que o interesse passe a estar presente na operação, aumentando assim o valor de CDO em uma unidade.

CDLOC: $\geq_{(X)}$

A adição das linhas de código pode criar pontos de transição tanto dentro do trecho adicionado quanto entre suas linhas de fronteira com o conteúdo original da operação alterada, como mostrado no exemplo da figura 4.5.

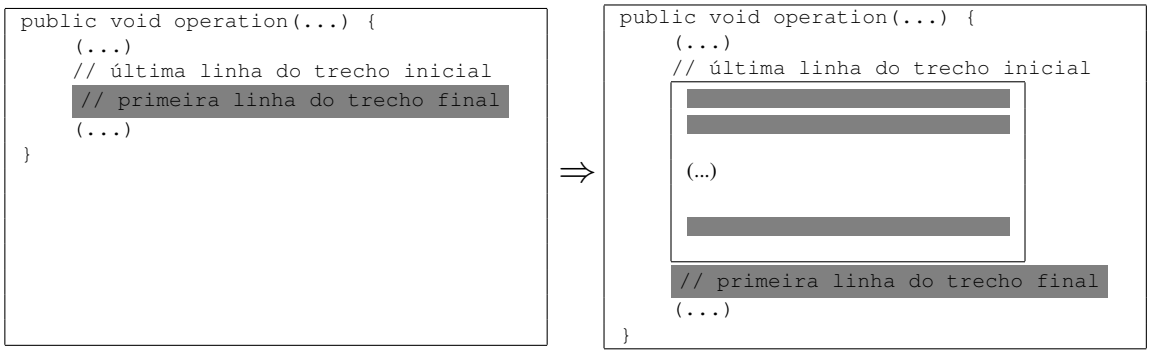


Figura 4.5: Criação de pontos de transição dentro do trecho adicionado e entre o trecho e as linhas originais da operação alterada

LOC: $>_{(X)}$

Com a adição dos comandos, linhas de código serão criadas na operação, aumentando assim o valor de LOC do componente.

CBC: $\geq_{(X)}$

O trecho adicionado pode fazer referências a componentes que não fossem originalmente usados no componente onde ocorre a adição. Neste caso, a adição do trecho fará com que esses componentes passem a estar acoplados ao componente onde ocorre a adição, aumentando assim o valor de CBC. O aumento do valor da métrica será igual ao número de componentes que só forem referenciados no trecho adicionado.

LCOO: $\leq_{(X)}$

Caso o trecho adicionado acesse um atributo do componente que não seja acessado no restante da operação, este passo básico fará com que a operação passe a acessar um atributo do componente a mais. Caso haja alguma outra operação do componente que acesse esse mesmo atributo, e se as duas operações originalmente não acessassem nenhum atributo em comum, então o passo básico diminuirá o valor de LCOO. Isso ocorre porque originalmente o par formado pelas duas operações não compartilhava um atributo do componente, portanto este par era contado na parcela positiva de LCOO. A partir do momento que a operação passou a acessar o atributo (pela adição do trecho onde há este

acesso), o par passou a compartilhar um atributo e, portanto, passou a ser contado na parcela negativa de LCOO, o que diminui o valor desta métrica em duas unidades.

A diminuição total de LCOO será igual a duas vezes o número de operações que passarem a compartilhar um atributo com a operação onde o trecho foi adicionado (salvo no caso de LCOO se tornar um valor negativo por este cálculo, caso em que LCOO passa a ser zero e a diminuição desta métrica é igual ao seu valor original).

4.3.18 Remover linhas de código de operação

Modificação: eliminar alguns comandos de uma operação. Para remover um bloco contíguo de linhas de código, este passo básico deve ser utilizado uma única vez para eliminar o bloco todo, e não diversas vezes para eliminar cada um dos comandos do bloco, pois a avaliação deste passo foi feita levando-se esta premissa em consideração.

```
public class SomeClass {
    public void operation() {
        <trecho inicial da operação>
        <trecho a ser removido>
        <trecho final da operação>
    }
}
```

```
public class SomeClass {
    public void operation() {
        <trecho inicial da operação>
        ~~<trecho a ser removido>~~
        <trecho final da operação>
    }
}
```

Métricas alteradas:

CDC: $\leq_{(1)}$

Se o trecho removido contiver algum componente definido com objetivo de contribuir para a implementação um interesse transversal, e se no restante do componente onde o trecho foi removido não houver mais referências a esse interesse, então a remoção das linhas da operação fará com que o interesse deixe de estar presente no componente, diminuindo assim o valor de CDC em uma unidade.

CDO: $\leq_{(1)}$

Se o trecho removido usar alguma operação definida com objetivo de contribuir para a implementação um interesse transversal, e se no restante da operação onde o trecho foi removido não houver mais referências a esse interesse, então a remoção das linhas da operação fará com que o interesse deixe de estar presente na operação, diminuindo assim o valor de CDO em uma unidade.

CDLOC: $\leq_{(X)}$

O trecho removido pode ter alguns pontos de transição, os quais serão eliminados do componente e diminuirão o valor de CDLOC após este passo básico ser executado.

LOC: $<_{(X)}$

Com a remoção dos comandos, linhas de código serão eliminadas da operação, diminuindo assim o valor de LOC do componente.

CBC: $\leq_{(X)}$

O trecho removido pode fazer referências a componentes que não são usados no restante do componente onde ocorre a remoção. Neste caso, a eliminação do trecho fará com que esses componentes deixem de estar acoplados ao componente onde ocorre a remoção,

diminuindo assim o valor de CBC. A diminuição do valor da métrica será igual ao número de componentes que só forem referenciados no trecho removido.

LCOO: $\geq_{(X)}$

Caso o trecho removido acesse um atributo do componente que não é mais acessado no restante da operação, este passo básico fará com que a operação passe a acessar um atributo do componente a menos. Caso haja alguma outra operação do componente que originalmente acessasse esse mesmo atributo, e se as duas operações só acessassem este atributo em comum, então o passo básico aumentará o valor de LCOO. Isso ocorre porque originalmente o par formado pelas duas operações compartilhava um atributo do componente, portanto este par era contado na parcela negativa de LCOO. A partir do momento que a operação perdeu o acesso ao atributo (pela remoção do trecho onde havia este acesso), o par deixou de compartilhar um atributo e, portanto, passou a ser contado na parcela positiva de LCOO, o que aumenta o valor desta métrica em duas unidades.

O aumento total de LCOO será igual a duas vezes o número de operações que deixarem de compartilhar um atributo com a operação onde o trecho foi removido (salvo no caso de LCOO for inicialmente zero, caso em que LCOO pode ter um aumento menor por sua parcela negativa não ser totalmente válida no valor original da métrica).

4.3.19 Adicionar chamada a operação

Modificação: adicionar, em uma operação, uma linha com a chamada a uma segunda operação (do mesmo componente ou de outro).

```
public class SomeClass {
    public void operation() {
        (...)
    }
}
```

```
public class SomeClass {
    public void operation() {
        this.anotherOperation();
        (...)
    }
}
```

Métricas alteradas:

CDC: $\geq_{(1)}$

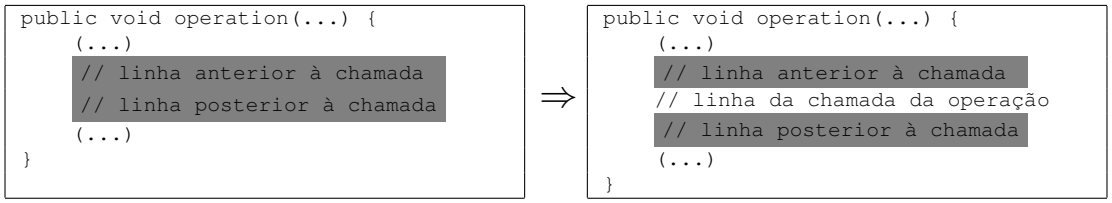
Se a operação chamada for de um componente definido para contribuir com a implementação de um interesse transversal, e se o componente onde a chamada à operação foi adicionada não for relacionado a esse interesse antes do passo básico, então a adição da chamada fará com que o interesse passe a estar disperso em mais um componente, o que aumenta o valor de CDC em uma unidade.

CDO: $\geq_{(1)}$

Se a operação chamada for definida para contribuir com a implementação de um interesse transversal, e se a operação onde a chamada foi adicionada não for relacionada a esse interesse antes do passo básico, então a adição da chamada fará com que o interesse passe a estar disperso em mais uma operação, o que aumenta o valor de CDO em uma unidade.

CDLOC: $\geq_{(2)}$

Se a linha da chamada adicionada tiver um sombreamento oposto à linha imediatamente antes e à linha imediatamente depois dela, então o passo básico criará dois novos pontos de transição no código do componente, como mostra o exemplo abaixo:



LOC: $>_{(1)}$
A adição da nova linha com a chamada da operação faz com que o número de linhas de código do componente aumente em uma unidade.

CBC: $\geq_{(1)}$
Se a operação chamada pertencer a outro componente e se não houver originalmente nenhuma referência a esse componente no componente em que a chamada é adicionada, então a adição da linha fará com que os dois componentes passem a estar acoplados, aumentando em uma unidade o valor de CBC.

4.3.20 Marcar operação como obsoleta

Modificação: adicionar a uma operação o marcador `@deprecated` para sinalizar que a mesma está obsoleta e seu uso deve ser evitado.

```
public class SomeClass {
    public void operation() {(...)}
    (...)
}
```

⇓

```
public class SomeClass {
    /** @deprecated <explicações> */
    public void operation() {(...)}
    (...)
}
```

Métricas alteradas:

Ao tornar uma operação obsoleta, adiciona-se uma recomendação de uso do software – o marcador `@deprecated` mostra que esta operação não deve ser usada (pelas razões descritas nas `<explicações>` do marcador). No entanto, nenhuma das métricas escolhidas para fazer a avaliação deste passo básico sofre alteração, pois não se trata de uma alteração que modifica a separação dos assuntos, o tamanho do software ou o acoplamento e coesão de seus componentes. Nem mesmo LOC é alterada por este passo básico, pois a linha adicionada é um comentário e faz parte dos tipos de linha de código descontadas no cálculo desta métrica.

4.3.21 Remover marcação de operação obsoleta

Modificação: remover de uma operação o marcador `@deprecated` para sinalizar que a mesma não é mais obsoleta e seu uso deve não precisa ser evitado.

```
public class SomeClass {
    /** @deprecated <explicações> */
    public void operation() {(...)}
    (...)
}
```

⇓

```
public class SomeClass {
    /** @deprecated <explicações> */
    public void operation() {(...)}
    (...)
}
```

Métricas alteradas:

Ao remover a marcação de uma operação obsoleta, elimina-se uma recomendação de uso do software – o marcador `@deprecated` mostra que esta operação não deve ser usada (pelas razões descritas nas `<explicações>` do marcador). No entanto, nenhuma das métricas escolhidas para fazer a avaliação deste passo básico sofre alteração, pois não se trata de uma alteração que modifica a separação dos assuntos, o tamanho do software ou o acoplamento e coesão de seus componentes. Nem mesmo LOC é alterada por este passo básico, pois a linha removida é um comentário e faz parte dos tipos de linha de código descontadas no cálculo desta métrica.

4.3.22 Tornar operação abstrata

Modificação: transformar uma operação concreta de corpo vazio em uma operação abstrata.

```
public class SomeClass {
    public void operation() {
    }
    (...)
}
```

```
public class SomeClass {
    public abstract void operation();
    (...)
}
```

Métricas alteradas:

LOC: $<_{(1)}$

Para transformar uma operação concreta de corpo vazio em uma operação abstrata, basta acrescentar a palavra-chave `abstract` e substituir o corpo da operação por '`;`'. Esta última modificação diminui o número de linhas de código do componente porque a linha de fechamento do escopo da operação é removida, enquanto que a linha de declaração da operação tem o caractere de sinalização de início de escopo (em AspectJ e Java representada por '`{`') substituído por '`;`'.

LCOO: $\gtrless$

Ao transformar uma operação concreta em abstrata, os pares de operações formado por ela com as demais operações do componente serão eliminados. O valor de LCOO aumentará caso sejam removidos mais pares de operações que compartilham atributos do que pares de operações sem atributos em comum, e diminuirá caso contrário.

4.3.23 Tornar operação concreta

Modificação: transformar uma operação abstrata em uma operação concreta de corpo vazio.

```
public class SomeClass {
    public abstract void operation();
    (...)
}
```

```
public class SomeClass {
    public ~~abstract~~ void operation() {
    }
    (...)
}
```

Métricas alteradas:

LOC: $>_{(1)}$

Para transformar uma operação abstrata em uma operação concreta de corpo vazio, basta remover a palavra-chave `abstract` e substituir o '`;`' após a declaração da operação pelos indicadores de início e fim de escopo de operação (em AspectJ e Java representados por '`{`' e '`}`'). Esta última modificação aumenta o número de linhas de código do componente porque o fechamento do escopo da operação é declarado em uma linha separada, conforme o exemplo acima.

LCOO: $\gtrless$

Ao transformar uma operação abstrata em uma operação concreta, serão criados novos os pares de operações entre ela e as demais operações do componente. O valor de LCOO aumentará caso sejam criados mais pares de operações que não compartilham atributos do que pares de operações com atributos em comum, e diminuirá caso contrário.

4.4 Passos básicos que alteram atributos e declarações inter-tipos

Nesta categoria de passos básicos estão as modificações que criam ou removem atributos e os diferentes tipos de declarações inter-tipo de aspectos – de implementação, herança, atributo e operação. A tabela 4.5 resume como cada métrica avaliada é alterada pelos passos básicos desta categoria.

Tabela 4.5: Variação das métricas ao usar os passos básicos que alteram atributos e declarações inter-tipos

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Adicionar atributo a componente	$\geq_{(1)}$	=	$\geq_{(2)}$	=	$>_{(1)}$	$>_{(1)}$	=	$\geq_{(1)}$	=	=
Remover atributo de componente	$\leq_{(1)}$	=	$\leq_{(2)}$	=	$<_{(1)}$	$<_{(1)}$	=	$\leq_{(1)}$	=	=
Adicionar declaração inter-tipo de implementação	=	=	=	=	$>_{(1)}$	=	=	$\geq_{(1a2)}$	=	=
Remover declaração inter-tipo de implementação	=	=	=	=	$<_{(1)}$	=	=	$\leq_{(1a2)}$	=	=
Adicionar declaração inter-tipo de herança	=	=	=	=	$>_{(1)}$	=	=	$\geq_{(1a2)}$	$\geq_{(X)}$	=
Remover declaração inter-tipo de herança	=	=	=	=	$<_{(1)}$	=	=	$\leq_{(1a2)}$	$\leq_{(X)}$	=
Adicionar declaração inter-tipo de atributo	=	=	=	=	$>_{(1)}$	$>_{(1)}$	=	$\geq_{(1a2)}$	=	=
Remover declaração inter-tipo de atributo	=	=	=	=	$<_{(1)}$	$<_{(1)}$	=	$\leq_{(1a2)}$	=	=
Adicionar declaração inter-tipo de operação	=	$>_{(1)}$	=	=	$>_{(2)}$	=	$>_{(1)}$	$\geq_{(1)}$	=	$\geq_{(X)}$
Remover declaração inter-tipo de operação	=	=	=	=	$<_{(2)}$	=	$<_{(1)}$	$\leq_{(1)}$	=	$\leq_{(X)}$

4.4.1 Adicionar atributo a componente

Modificação: adicionar um atributo em um componente.

```
public class SomeClass {
    (...)
}
```

```
public class SomeClass {
    private AttributeType attributeName;
    (...)
}
```

Métricas alteradas:

CDC: $\geq_{(1)}$

Se o tipo do atributo adicionado for um componente definido com o objetivo de contribuir para a implementação de um interesse transversal, e se originalmente o componente que recebe o atributo não for relacionado a esse interesse, então a adição do atributo fará com que o interesse esteja disperso em mais um componente, o que aumenta o valor de CDC em uma unidade.

CDLOC: $\geq_{(2)}$

Se a linha do atributo for sombreada para algum interesse transversal, e se ela não puder ser colocada logo abaixo ou logo acima de outra linha sombreada para o mesmo interesse no componente, então a adição do atributo criará dois novos pontos de transição no código do componente.

LOC: $>_{(1)}$

Para adicionar o atributo é necessário usar uma nova linha de código, o que aumenta o valor de LOC em uma unidade.

NOA: $>_{(1)}$

Uma vez adicionado o novo atributo, o componente passará a ter um atributo a mais do que antes do passo básico, aumentando assim o valor de NOA.

CBC: $\geq_{(1)}$

Se o tipo do atributo não for originalmente acoplado ao componente, então a adição do atributo criará o acoplamento entre o componente e o tipo do atributo, aumentando assim o valor de CBC do componente em uma unidade.

4.4.2 Remover atributo de componente

Modificação: remover um atributo de um componente.

```
public class SomeClass {
    private AttributeType attributeName;
    (...)
}
```

⇓

```
public class SomeClass {
    ~~private AttributeType attributeName;~~
    (...)
}
```

Métricas alteradas:

CDC: $\leq_{(1)}$

Se o tipo do atributo removido for um componente definido com o objetivo de contribuir para a implementação de um interesse transversal, e se o componente de onde o atributo é removido só estiver relacionado a esse interesse pela declaração do atributo, então sua remoção fará com que o interesse passe a estar disperso em um componente a menos, o que diminui o valor de CDC em uma unidade.

CDLOC: $\leq_{(2)}$

Se a linha do atributo for sombreada para algum interesse transversal, e se ela não estiver logo abaixo ou logo acima de outra linha sombreada para o mesmo interesse no componente, então a remoção do atributo eliminará dois pontos de transição no código do componente.

LOC: $<_{(1)}$

Ao remover o atributo a linha de código que contém a declaração é eliminada do componente, diminuindo o número de linhas de código do sistema em uma unidade.

NOA: $<_{(1)}$
Uma vez removido o atributo, o componente passará a ter um atributo a menos do que antes do passo básico, diminuindo assim o valor de NOA.

CBC: $\leq_{(1)}$
Se o tipo do atributo só for acoplado ao componente através da declaração do atributo, então sua remoção eliminará o acoplamento entre o componente e o tipo do atributo, diminuindo assim o valor de CBC do componente em uma unidade.

4.4.3 Adicionar declaração inter-tipo de implementação

Modificação: adicionar a um aspecto uma declaração inter-tipo da espécie `declare parents` que faça com que um componente implemente uma interface do sistema.

```
public aspect SomeAspect {
    (...)
}
```

```
public aspect SomeAspect {
    declare parents:  TargetComponent implements SomeInterface;
    (...)
}
```

Métricas alteradas:

LOC: $>_{(1)}$
Para adicionar a declaração inter-tipo é necessário usar uma nova linha de código, o que aumenta o valor de LOC em uma unidade.

CBC: $\geq_{(1a2)}$
A declaração inter-tipo possui acoplamento a dois componentes: a interface implementada (no exemplo acima chamada de `SomeInterface`) e o componente-alvo que deve implementá-la (`TargetComponent`). Após adicionar a declaração inter-tipo no aspecto, o número de acoplamentos do aspecto com outros componentes terá:

- aumentado duas unidades, se tanto a interface quanto o componente-alvo não fossem previamente acoplados ao aspecto;
- aumentado uma unidade, se apenas a interface **ou** o componente-alvo não fosse previamente acoplado ao aspecto;
- se mantido igual, se tanto a interface quanto o componente-alvo já fossem previamente acoplados ao aspecto.

4.4.4 Remover declaração inter-tipo de implementação

Modificação: eliminar de um aspecto uma declaração inter-tipo da espécie `declare parents` que originalmente fizesse com que um componente implementasse uma interface do sistema.

```
public aspect SomeAspect {
    declare parents: TargetComponent implements SomeInterface;
    (...)
}
```

```
public aspect SomeAspect {
    ~~declare parents:  TargetComponent implements SomeInterface;~~
    (...)
}
```

Métricas alteradas:

LOC: $<_{(1)}$

Ao remover a declaração inter-tipo, todas as linhas necessárias para sua definição serão excluídas do aspecto. Normalmente o número de linhas necessárias para esta declaração é 1, portanto esta modificação reduziria o número de linhas de código do aspecto em uma unidade.

CBC: $\leq_{(1a2)}$

A declaração inter-tipo possui acoplamento a dois componentes: a interface implementada (no exemplo acima chamada de `SomeInterface`) e o componente-alvo que deve implementá-la (`TargetComponent`). Após remover a declaração inter-tipo do aspecto, o número de acoplamentos do aspecto com outros componentes terá:

- diminuído duas unidades, se tanto a interface quanto o componente-alvo não estiverem mais acoplados ao aspecto em nenhum outro trecho do mesmo;
- diminuído uma unidade, se apenas a interface **ou** o componente-alvo não estiver mais acoplados ao aspecto em nenhum outro trecho do mesmo;
- se mantido igual, se tanto a interface quanto o componente-alvo ainda estiverem acoplados ao aspecto em alguma parte do mesmo.

4.4.5 Adicionar declaração inter-tipo de herança

Modificação: adicionar a um aspecto uma declaração inter-tipo da espécie `declare parents` que faça com que um componente se torne filho de outro componente do sistema.

```
public aspect SomeAspect {
    (...)
}
```

⇓

```
public aspect SomeAspect {
    declare parents:  SubComponent extends SuperComponent;
    (...)
}
```

Métricas alteradas:

LOC ($>_{(1)}$) e **CBC** ($\geq_{(1a2)}$)

Mesmas justificativas de 4.4.3.

DIT ($\geq_{(X)}$)

Mesmas justificativas de 4.1.9.

4.4.6 Remover declaração inter-tipo de herança

Modificação: eliminar de um aspecto uma declaração inter-tipo da espécie `declare parents` que originalmente fizesse com que um componente fosse filho de outro componente do sistema.

```
public aspect SomeAspect {
    declare parents: SubComponent extends SuperComponent;
    (...)
}
```

⇓

```
public aspect SomeAspect {
    ~~declare parents:  SubComponent extends SuperComponent;~~
    (...)
}
```

Métricas alteradas:

LOC ($<_{(1)}$) e **CBC** ($\leq_{(1a2)}$)

Mesmas justificativas de 4.4.4.

DIT ($\leq_{(X)}$)

Mesmas justificativas de 4.1.10.

4.4.7 Adicionar declaração inter-tipo de atributo

Modificação: adicionar a um aspecto uma declaração inter-tipo que crie um atributo em um componente do sistema.

```
public aspect SomeAspect {
    (...)
}
```

⇓

```
public aspect SomeAspect {
    public AttributeType HostComponent.attributeName;
    (...)
}
```

Métricas alteradas:

LOC ($>_{(1)}$)

Mesmas justificativas de 4.4.3.

NOA ($>_{(1)}$)

A declaração inter-tipo faz com que o aspecto tenha um atributo a mais do que antes do passo básico[3], aumentando assim o valor de NOA em uma unidade.

CBC: $\geq_{(1a2)}$

A declaração inter-tipo possui acoplamento a dois componentes: o tipo do atributo (no exemplo acima chamada de `AttributeType`) e o componente-alvo que deve receber o atributo (`HostComponent`). Após adicionar a declaração inter-tipo no aspecto, o número de acoplamentos do aspecto com outros componentes terá:

- aumentado duas unidades, se tanto o tipo do atributo quanto o componente-alvo não fossem previamente acoplados ao aspecto;
- aumentado uma unidade, se apenas o tipo do atributo **ou** o componente-alvo não fosse previamente acoplado ao aspecto;
- se mantido igual, se tanto o tipo do atributo quanto o componente-alvo já fossem previamente acoplados ao aspecto.

4.4.8 Remover declaração inter-tipo de atributo

Modificação: remover de um aspecto uma declaração inter-tipo que coloca um atributo em um componente do sistema.

```
public aspect SomeAspect {
    public AttributeType HostComponent.attributeName;
    (...)
}
```

⇓

```
public aspect SomeAspect {
    ~~public AttributeType HostComponent.attributeName;~~
    (...)
}
```

[3] declarações inter-tipo adicionam elementos (atributos, operações, etc.) em classes e interfaces, porém sua definições está dentro de um aspecto e, portanto, é este componente que possui o elemento adicionado

Métricas alteradas:

LOC ($<_{(1)}$)

Mesmas justificativas de 4.4.4.

NOA ($<_{(1)}$)

A remoção da declaração inter-tipo faz com que o aspecto tenha um atributo a menos do que antes do passo básico, diminuindo assim o valor de NOA em uma unidade.

CBC: $\leq_{(1a2)}$

A declaração inter-tipo possui acoplamento a dois componentes: o tipo do atributo (no exemplo acima chamada de `AttributeType`) e o componente-alvo onde é colocado o atributo (`HostComponent`). Após remover a declaração inter-tipo no aspecto, o número de acoplamentos do aspecto com outros componentes terá:

- diminuído duas unidades, se tanto o tipo do atributo quanto o componente-alvo não estiverem mais acoplados ao aspecto em nenhum outro trecho do mesmo;
- diminuído uma unidade, se apenas o tipo do atributo **ou** o componente-alvo não estiver mais acoplados ao aspecto em nenhum outro trecho do mesmo;
- se mantido igual, se tanto o tipo do atributo quanto o componente-alvo ainda estiverem acoplados ao aspecto em alguma parte do mesmo.

4.4.9 Adicionar declaração inter-tipo de operação

Modificação: adicionar a um componente uma operação sem parâmetros e de corpo vazio, por meio de uma declaração inter-tipo.

```
public class SomeAspect {
    (...)
}
```

⇓

```
public class SomeAspect {
    public ReturnComponent HostClass.newOperation() {
    }
    (...)
}
```

Métricas alteradas:

Este passo básico é bastante similar a *Criar operação vazia* (4.3.1), pois ambos criam uma operação em um componente. No entanto, o componente afetado por cada passo básico muda – em *Criar operação vazia* é o componente que recebe a operação, enquanto que neste passo básico é o aspecto onde é definida a declaração inter-tipo. Por causa das semelhanças entre esses passos, as alterações das métricas e as respectivas justificativas também são muito parecidas.

CDO: $>_{(1)}$

A declaração inter-tipo faz com que o aspecto tenha uma operação a mais implementando seu interesse transversal, o que aumenta o valor de CDO em uma unidade.

LOC: $>_{(2)}$

Com a criação da operação são adicionadas duas linhas ao aspecto: a de declaração da operação e a de fechamento de seu escopo (em AspectJ e Java representada por "}").

WOC: $>_{(1)}$

A criação da operação adiciona uma nova complexidade à soma que calcula WOC, que neste caso vale 1 porque a operação não possui parâmetros.

CBC: $\geq_{(1)}$
Se a operação criada não tiver retorno vazio (`void`) e se o tipo de retorno não estiver acoplado ao aspecto antes do passo básico, então a criação da operação criará um novo acoplamento entre os dois componentes, aumentando assim o valor de CBC em uma unidade.

LCOO: $\geq_{(X)}$
Caso haja outras operações no aspecto, a operação adicionada por este passo básico formará novos pares de operação com elas. Todos esses pares não acessarão um atributo em comum porque o corpo da operação adicionada é vazio, então obviamente esta operação não pode acessar nenhum atributo do aspecto. Dessa forma, o valor de LCOO aumentará tanto quanto forem as operações existentes no aspecto antes da execução do passo básico.

4.4.10 Remover declaração inter-tipo de operação

Modificação: remover de um aspecto uma declaração inter-tipo que adiciona uma operação sem parâmetros e de corpo vazio em outro componente.

```
public class SomeAspect {
    public ReturnComponent HostClass.newOperation() {
    }
    (...)
}
```

```
public class SomeAspect {
    ~~public ReturnComponent HostClass.newOperation() {~~
    ~~}~~
    (...)
}
```

Métricas alteradas:

Este passo básico é bastante similar a *Remover operação vazia* (4.3.2), pois ambos removem uma operação em um componente. No entanto, o componente afetado por cada passo básico muda – em *Remover operação vazia* é o componente que possui a operação, enquanto que neste passo básico é o aspecto onde é definida a declaração inter-tipo. Por causa das semelhanças entre esses passos, as alterações das métricas e as respectivas justificativas também são muito parecidas.

LOC: $<_{(2)}$
Todas as linhas da operação serão eliminadas por este passo básico, o que normalmente significa duas linhas de código: a de declaração da operação e a de fechamento de seu escopo (em AspectJ e Java representada por "`}`").

WOC: $<_{(1)}$
A remoção da operação elimina uma das complexidades da soma que calcula WOC, complexidade que neste caso vale 1 porque a operação removida não possui parâmetros.

CBC: $\leq_{(1)}$
Se a operação removida não tiver retorno vazio (`void`) e se o tipo de retorno não estiver acoplado ao aspecto em nenhum outro local deste componente, então a remoção da operação eliminará o acoplamento existente entre os dois componentes, diminuindo assim o valor de CBC em uma unidade.

LCOO: $\leq_{(X)}$
Caso haja outras operações no aspecto, os pares de operações formados pela operação removida com as demais operações do aspecto serão eliminados por este passo básico. Todos esses pares não acessam um atributo em comum porque o corpo da operação removida é vazio, então obviamente esta operação não pode acessar nenhum atributo do

aspecto. Dessa forma, o valor de LCOO diminuirá tanto quanto forem as operações existentes no aspecto depois da execução do passo básico.

4.5 Passos básicos que modificam a visibilidade de elementos

Esta categoria de passos básicos possui as modificações que alteram a visibilidade de elementos do software – componentes, operações, atributos, etc. – seja pela troca de palavras-chave, seja por meio de novas declarações que a OA provê. A tabela 4.6 resume como cada métrica avaliada é alterada pelos passos básicos desta categoria.

Tabela 4.6: Variação das métricas ao usar os passos básicos que modificam a visibilidade de elementos

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Tornar componente público	=	=	=	=	=	=	=	=	=	=
Tornar componente privado	=	=	=	=	=	=	=	=	=	=
Tornar componente protegido	=	=	=	=	=	=	=	=	=	=
Tornar operação pública	=	=	=	=	=	=	=	=	=	=
Tornar operação privada	=	=	=	=	=	=	=	=	=	=
Tornar operação protegida	=	=	=	=	=	=	=	=	=	=
Tornar atributo público	=	=	=	=	=	=	=	=	=	=
Tornar atributo privado	=	=	=	=	=	=	=	=	=	=
Tornar atributo protegido	=	=	=	=	=	=	=	=	=	=
Tornar aspecto privilegiado	=	=	=	=	=	=	=	=	=	=
Tornar aspecto não-privilegiado	=	=	=	=	=	=	=	=	=	=
Criar `declare warning` de sinalização de acesso a atributo	=	=	=	=	$>_{(5)}$	=	=	=	=	=
Remover `declare warning` de sinalização de acesso a atributo	=	=	=	=	$<_{(5)}$	=	=	=	=	=
Criar `declare error` de proteção de acesso a atributo	=	=	=	=	$>_{(6)}$	=	=	=	=	=

4.5.1 Tornar componente público

Modificação: um componente originalmente privado ou protegido passa a ser público.

```
protected class SomeClass {
    (...)
}
```

```
public class SomeClass {
    (...)
}
```

Métricas alteradas:

Ao tornar um componente público, aumenta-se o alcance deste componente – todo o sistema passa a poder usá-lo e estendê-lo. No entanto, nenhuma das métricas escolhidas para fazer a avaliação deste passo básico sofre alteração, pois não se trata de uma alteração que modifica a separação dos assuntos, o tamanho do software ou o acoplamento e coesão de seus componentes.

4.5.2 Tornar componente privado

Modificação: um componente originalmente público ou protegido passa a ser privado.

```
protected class SomeClass {
    (...)
}
```

```
private class SomeClass {
    (...)
}
```

Métricas alteradas:

Ao tornar um componente privado, diminui-se o alcance deste componente – nenhuma parte do sistema passa a poder usá-lo e estendê-lo, salvo o componente em que o componente alterado está definido [4]. No entanto, nenhuma das métricas escolhidas para fazer a avaliação deste passo básico sofre alteração, pois não se trata de uma alteração que modifica a separação dos assuntos, o tamanho do software ou o acoplamento e coesão de seus componentes.

4.5.3 Tornar componente protegido

Modificação: um componente originalmente público ou privado passa a ser protegido.

```
public class SomeClass {
    (...)
}
```

```
protected class SomeClass {
    (...)
}
```

Métricas alteradas:

Ao tornar um componente protegido, permite-se que uma parte do sistema possa usá-lo e estendê-lo (o que significa uma restrição de alcance caso originalmente o componente fosse público e uma expansão de alcance caso o componente fosse originalmente privado). No entanto, nenhuma das métricas escolhidas para fazer a avaliação deste passo básico sofre alteração, pois não se trata de uma alteração que modifica a separação dos assuntos, o tamanho do software ou o acoplamento e coesão de seus componentes.

4.5.4 Tornar operação pública

Modificação: uma operação originalmente protegida ou privada passa a ser pública.

```
public class SomeClass {
    private void operation() {
        (...)
    }
    (...)
}
```

```
public class SomeClass {
    public void operation() {
        (...)
    }
    (...)
}
```

[4] Componentes privados não são visíveis para outros componentes fora do arquivo de sua definição. Por causa disso, só faz sentido tornar um componente privado se ele estiver definido junto a outro que tenha acesso a ele, ou seja, dentro de outro componente

Métricas alteradas:

Ao tornar uma operação pública, aumenta-se o alcance desta operação – todo o sistema passa a poder usá-la. No entanto, nenhuma das métricas escolhidas para fazer a avaliação deste passo básico sofre alteração, pois não se trata de uma alteração que modifica a separação dos assuntos, o tamanho do software ou o acoplamento e coesão de seus componentes.

4.5.5 Tornar operação privada

Modificação: uma operação originalmente pública ou protegida passa a ser privada.

```
public class SomeClass {
    protected void operation() {
        (...)
    }
    (...)
}
```

```
public class SomeClass {
    private void operation() {
        (...)
    }
    (...)
}
```

Métricas alteradas:

Ao tornar uma operação privada, diminui-se o alcance desta operação – apenas o componente que a possui poderá usá-la. No entanto, nenhuma das métricas escolhidas para fazer a avaliação deste passo básico sofre alteração, pois não se trata de uma alteração que modifica a separação dos assuntos, o tamanho do software ou o acoplamento e coesão de seus componentes.

4.5.6 Tornar operação protegida

Modificação: uma operação originalmente pública ou privada passa a ser protegida.

```
public class SomeClass {
    private void operation() {
        (...)
    }
    (...)
}
```

```
public class SomeClass {
    protected void operation() {
        (...)
    }
    (...)
}
```

Métricas alteradas:

Ao tornar uma operação protegida, permite-se que uma parte do sistema possa usá-la (o que significa uma restrição de alcance caso originalmente a operação fosse pública e uma expansão de alcance caso a operação fosse originalmente privada). No entanto, nenhuma das métricas escolhidas para fazer a avaliação deste passo básico sofre alteração, pois não se trata de uma alteração que modifica a separação dos assuntos, o tamanho do software ou o acoplamento e coesão de seus componentes.

4.5.7 Tornar atributo público

Modificação: um atributo originalmente privado ou protegido passa a ser público.

```
public class SomeClass {
    protected AttributeType attribute;
    (...)
}
```

```
public class SomeClass {
    public AttributeType attribute;
    (...)
}
```

Métricas alteradas:

Ao tornar um atributo público, aumenta-se o alcance deste atributo – todo o sistema passa a poder usá-lo e alterá-lo diretamente. No entanto, nenhuma das métricas escolhidas para fazer a avaliação deste passo básico sofre alteração, pois não se trata de uma alteração que modifica a separação dos assuntos, o tamanho do software ou o acoplamento e coesão de seus componentes.

4.5.8 Tornar atributo privado

Modificação: um atributo originalmente público ou protegido passa a ser privado.

```
public class SomeClass {
    protected AttributeType attribute;
    (...)
}
```

```
public class SomeClass {
    private AttributeType attribute;
    (...)
}
```

Métricas alteradas:

Ao tornar um atributo privado, diminui-se o alcance deste atributo – nenhuma parte do sistema passa a poder usá-lo ou alterá-lo diretamente, salvo o componente em que o atributo está definido. No entanto, nenhuma das métricas escolhidas para fazer a avaliação deste passo básico sofre alteração, pois não se trata de uma alteração que modifica a separação dos assuntos, o tamanho do software ou o acoplamento e coesão de seus componentes.

4.5.9 Tornar atributo protegido

Modificação: um atributo originalmente privado ou público passa a ser protegido.

```
public class SomeClass {
    private AttributeType attribute;
    (...)
}
```



```
public class SomeClass {
    protecteds AttributeType attribute;
    (...)
}
```

Métricas alteradas:

Ao tornar um atributo protegido, permite-se que uma parte do sistema possa usá-lo e alterá-lo diretamente (o que significa uma restrição de alcance, caso originalmente o atributo fosse público, e uma expansão de alcance, caso o atributo fosse originalmente privado). No entanto, nenhuma das métricas escolhidas para fazer a avaliação deste passo básico sofre alteração, pois não se trata de uma alteração que modifica a separação dos assuntos, o tamanho do software ou o acoplamento e coesão de seus componentes.

4.5.10 Tornar aspecto privilegiado

Modificação: um aspecto originalmente sem privilégios passa a ser privilegiado.

```
public aspect SomeAspect {
    (...)
}
```

```
privileged public aspect SomeAspect {
    (...)
}
```

Métricas alteradas:

Ao tornar um aspecto privilegiado, todo o sistema passa a ser acessível a ele, como se todos os componentes, atributos e operações fossem públicos. No entanto, nenhuma das métricas escolhidas para fazer a avaliação deste passo básico sofre alteração, pois não se trata de uma alteração que modifica a separação dos assuntos, o tamanho do software ou o acoplamento e coesão de seus componentes.

4.5.11 Tornar aspecto não-privilegiado

Modificação: um aspecto originalmente privilegiado passa a não sê-lo.

```
privileged public aspect SomeAspect {
    (...)
}
```

```
~~privileged~~ public aspect SomeAspect {
    (...)
}
```

Métricas alteradas:

Ao tornar um aspecto não-privilegiado, ele não será mais capaz de acessar todo o sistema, apenas os elementos protegidos de seu pacote ou públicos. No entanto, nenhuma das métricas escolhidas para fazer a avaliação deste passo básico sofre alteração, pois não se trata de uma alteração que modifica a separação dos assuntos, o tamanho do software ou o acoplamento e coesão de seus componentes.

4.5.12 Criar `declare warning` de sinalização de acesso a atributo

Modificação: adicionar em um aspecto um `declare warning` que sinaliza quando um atributo de um componente é acessado fora do aspecto. A declaração do atributo deve ser feita em uma declaração inter-tipo no aspecto.

```
public aspect SomeAspect {
    public AttributeType SomeClass.attributeName;
    (...)
}
```

⇓

```
public aspect SomeAspect {
    public AttributeType SomeClass.attributeName;
    declare warning:
        (get (AttributeType SomeClass.attributeName) ||
         set (AttributeType SomeClass.attributeName))
        && !within(SomeAspect):
        "Field attributeName is accessed outside aspect.";
    (...)
}
```

Métricas alteradas:

LOC: $>_{(5)}$

A adição do `declare warning` criará novas linhas de código no aspecto, o que aumenta o valor de LOC. O número de linhas adicionadas pode variar dependendo do estilo de programação adotado, porém nesta avaliação será usado o estilo mostrado no exemplo acima, que resulta em um acréscimo de 5 linhas de código ao aspecto.

4.5.13 Remover `declare warning` de sinalização de acesso a atributo

Modificação: remover de um aspecto um `declare warning` que sinaliza quando um atributo de um componente é acessado fora do aspecto. A declaração do atributo deve ser feita em uma declaração inter-tipo no aspecto.

```
public aspect SomeAspect {
    public AttributeType SomeClass.attributeName;
    declare warning:
        (get (AttributeType SomeClass.attributeName) ||
         set (AttributeType SomeClass.attributeName))
        && !within(SomeAspect):
        "Field attributeName is accessed outside aspect.";
    (...)
}
```

⇓

```
public aspect SomeAspect {
    public AttributeType SomeClass.attributeName;
    declare warning:
        (get (AttributeType SomeClass.attributeName) ||
         set (AttributeType SomeClass.attributeName))
        && !within(SomeAspect):
        "Field attributeName is accessed outside aspect.";
    (...)
}
```

Métricas alteradas:

LOC: $<_{(5)}$

A remoção do `declare warning` eliminará todas as suas linhas do aspecto, o que diminui o valor de LOC. O número de linhas eliminadas pode variar dependendo do estilo de programação adotado, porém nesta avaliação será usado o estilo mostrado no exemplo acima, que resulta em um decréscimo de 5 linhas de código ao aspecto.

4.5.14 Criar `declare error` de proteção de acesso a atributo

Modificação: adicionar em um aspecto um `declare error` que evita que um atributo de um componente seja acessado fora do aspecto ou das sub-classes da classe que o possui. A declaração do atributo deve ser feita em uma declaração inter-tipo no aspecto.

```
public aspect SomeAspect {
    public AttributeType SomeClass.attributeName;
    (...)
}
```

⇓

```
public aspect SomeAspect {
    public AttributeType SomeClass.attributeName;
    declare error:
        (get (AttributeType SomeClass.attributeName) ||
         set (AttributeType SomeClass.attributeName))
        && !within(SomeAspect)
        && !within(SomeClass+):
        "Field attributeName is accessed outside SomeClass chain and SomeAspect.";
    (...)
}
```

Métricas alteradas:

LOC: $>_{(6)}$

A adição do `declare error` criará novas linhas de código no aspecto, o que aumenta o valor de LOC. O número de linhas adicionadas pode variar dependendo do estilo de programação adotado, porém nesta avaliação será usado o estilo mostrado no exemplo acima, que resulta em um acréscimo de 6 linhas de código ao aspecto.

4.6 Outros passos básicos

Nesta categoria de passos básicos estão as modificações não enquadradas nas demais categorias do catálogo, incluindo as que envolvem trocas ou remoções de palavras-chave. A tabela 4.7 resume como cada métrica avaliada é alterada pelos passos básicos desta categoria.

Tabela 4.7: Variação das métricas ao usar os passos básicos não-enquadrados nas demais categorias do catálogo

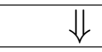
	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Adicionar importações	=	=	=	=	=	=	=	=	=	=
Remover importações	=	=	=	=	=	=	=	=	=	=
Trocar referência a componente externo por referência a componente interno	=	=	=	=	=	=	=	$<_{(1)}$	=	=
Trocar referência a componente interno por referência a componente externo	=	=	=	=	=	=	=	$>_{(1)}$	=	=
Substituir `public` por `static` em classe	=	=	=	=	=	=	=	=	=	=
Substituir `static` por `public` em classe	=	=	=	=	=	=	=	=	=	=
Mudar palavra-chave `class` para `interface`	=	=	=	=	=	=	=	=	=	=
Mudar palavra-chave `interface` para `class`	=	=	=	=	=	=	=	=	=	=
Remover palavra-chave `abstract` de componente	=	=	=	=	=	=	=	=	=	=
Adicionar palavra-chave `abstract` a componente	=	=	=	=	=	=	=	=	=	=
Remover palavra-chave `abstract` de operação de interface	=	=	=	=	=	=	=	=	=	=
Adicionar palavra-chave `abstract` a operação de interface	=	=	=	=	=	=	=	=	=	=
Compilar e testar	=	=	=	=	=	=	=	=	=	=

4.6.1 Adicionar importações

Modificação: adicionar ao arquivo do componente as importações necessárias para que o mesmo seja compilado sem erros. Para adicionar um bloco de importações, este

passo básico deve ser utilizado uma única vez para adicionar o bloco todo, e não diversas vezes para adicionar cada uma das importações do bloco, pois a avaliação deste passo foi feita levando-se esta premissa em consideração.

```
import somePackage.Component0;
(...)
public class SomeClass {
    (...)
}
```



```
import somePackage.Component0;
import somePackage.Component1;
import somePackage.Component2;
(...)
public class SomeClass {
    (...)
}
```

Métricas alteradas:

A adição de importações ao arquivo do componente evita que o mesmo gere erros durante a compilação do software. No entanto, essa alteração não tem efeito sobre nenhuma das métricas selecionadas, nem mesmo LOC – linhas com comandos de importação são descontadas na soma das linhas de código do software (ver definição de LOC em 2.3.1.4), por isso a adição de importações não é considerada uma modificação do tamanho do software.

4.6.2 Remover importações

Modificação: remover do arquivo do componente as importações não usadas no corpo do mesmo. Para remover um bloco de importações, este passo básico deve ser utilizado uma única vez para eliminar o bloco todo, e não diversas vezes para eliminar cada uma das importações do bloco, pois a avaliação deste passo foi feita levando-se esta premissa em consideração.

```
import somePackage.Component0;
import somePackage.Component1;
import somePackage.Component2;
(...)
public class SomeClass {
    (...)
}
```

```
import somePackage.Component0;
~~import somePackage.Component1;~~
~~import somePackage.Component2;~~
(...)
public class SomeClass {
    (...)
}
```

Métricas alteradas:

A remoção de importações do arquivo do componente evita que o mesmo carregue componentes desnecessários para sua compilação e execução. No entanto, essa alteração não tem efeito sobre nenhuma das métricas selecionadas, nem mesmo LOC – linhas com comandos de importação são descontadas na soma das linhas de código do software (ver definição de LOC em 2.3.1.4), por isso a remoção de importações não é considerada uma modificação do tamanho do software.

4.6.3 Trocar referência a componente externo por referência a componente interno

Modificação: trocar as referências a um componente autônomo por referências a um componente interno de mesmo nome e corpo idêntico.

```
package somePackage;
public class SomeClass {
    (...)
}
```

```
package anotherPackage;
public class AnotherClass {
    (...)
    // referência a somePackage.SomeClass
    (...)
    public class SomeClass {
        (...)
    }
}
```

```
package somePackage;
public class SomeClass {
    (...)
}
```

```
package anotherPackage;
public class AnotherClass {
    (...)
    // referência a SomeClass
    (...)
    public class SomeClass {
        (...)
    }
}
```

Métricas alteradas:

CBC: $<_{(1)}$

Como os acoplamentos a componentes internos não são considerados no cálculo de CBC, ao trocar as referências a um componente definido externamente por referências a um componente interno, o valor desta métrica deixará de contar o acoplamento a este componente, diminuindo assim o valor de CBC em uma unidade.

4.6.4 Trocar referência a componente interno por referência a componente externo

Modificação: trocar as referências a um componente interno por referências a um componente autônomo (definido externamente) de mesmo nome e corpo idêntico.

```
package somePackage;
public class SomeClass {
    (...)
}
```

```
package anotherPackage;
public class AnotherClass {
    (...)
    // referência a SomeClass
    (...)
    public class SomeClass {
        (...)
    }
}
```

```
package somePackage;
public class SomeClass {
    (...)
}
```

```
package anotherPackage;
public class AnotherClass {
    (...)
    // referência a somePackage.SomeClass
    (...)
    public class SomeClass {
        (...)
    }
}
```

Métricas alteradas:

CBC: $>_{(1)}$

Como os acoplamentos a componentes internos não são considerados no cálculo de CBC, ao trocar as referências a um componente definido internamente por referências a um componente autônomo, o valor desta métrica passará a contar o acoplamento a este componente, aumentando assim o valor de CBC em uma unidade.

4.6.5 Substituir `public` por `static` em classe

Modificação: uma classe interna originalmente pública passa a ser estática. Esta modificação só pode ser feita em classes definidas dentro de outros componentes[5].

```
public class SomeClass {
    public class InnerClass {
        (...)
    }
    (...)
}
```

```
public class SomeClass {
    static class InnerClass {
        (...)
    }
    (...)
}
```

Métricas alteradas:

Assim como ocorre nos passos básicos "Tornar componente público" (4.5.1), "Tornar componente protegido" (4.5.3) e "Tornar componente privado" (4.5.2), este passo básico altera a visibilidade da classe para o resto do sistema. E, da mesma forma que nos passos básicos citados, este passo não altera nenhuma das métricas escolhidas.

4.6.6 Substituir `static` por `public` em classe

Modificação: uma classe interna originalmente estática passa a ser pública. Esta modificação só pode ser feita em classes definidas dentro de outros componentes[6].

```
public class SomeClass {
    static class InnerClass {
        (...)
    }
    (...)
}
```

[5]Pela definição da linguagem Java

[6]Pela definição da linguagem Java

```
public class SomeClass {
    public class InnerClass {
        (...)
    }
    (...)
}
```

Métricas alteradas:

Assim como ocorre nos passos básicos "Tornar componente público" (4.5.1), "Tornar componente protegido" (4.5.3) e "Tornar componente privado" (4.5.2), este passo básico altera a visibilidade da classe para o resto do sistema. E, da mesma forma que nos passos básicos citados, este passo não altera nenhuma das métricas escolhidas.

4.6.7 Mudar palavra-chave `class` para `interface`

Modificação: trocar a palavra-chave `class` por `interface`.

```
public class SomeComponent {
    (...)
}
```

```
public interface SomeComponent {
    (...)
}
```

Métricas alteradas:

A troca da palavra-chave `class` por `interface` transforma uma classe em interface, porém esta alteração não é suficiente para finalizar a transformação, já que outros elementos da classe original precisam ser modificados para o componente de fato se tornar uma interface (suas interfaces implementadas, uma possível herança de outra classe, operações não-abstratas, variáveis de instância, etc.).

Esta modificação parcial, no entanto, não altera nenhuma das métricas escolhidas para a avaliação, pois classes e interfaces são tipos diferentes da mesma categoria "componente" e são consideradas da mesma forma pelas métricas.

4.6.8 Mudar palavra-chave `interface` para `class`

Modificação: trocar a palavra-chave `interface` por `class`.

```
public interface SomeComponent {
    (...)
}
```

```
public class SomeComponent {
    (...)
}
```

Métricas alteradas:

A troca da palavra-chave `interface` por `class` transforma uma interface em classe, porém esta alteração não é suficiente para finalizar a transformação, já que outros elementos da interface original precisam ser modificados para o componente de fato se tornar uma classe (suas interfaces implementadas, uma possível herança de outra classe, operações não-abstratas, variáveis de instância, etc.).

Esta modificação parcial, no entanto, não altera nenhuma das métricas escolhidas para a avaliação, pois classes e interfaces são tipos diferentes da mesma categoria "componente" e são consideradas da mesma forma pelas métricas.

4.6.9 Remover palavra-chave `abstract` de componente

Modificação: remover a palavra-chave `abstract` de uma classe ou aspecto originalmente abstrato.

```
public abstract class SomeClass {
    (...)
}
```

```
public ~~abstract~~ class SomeClass {
    (...)
}
```

Métricas alteradas:

A retirada da palavra-chave `abstract` transforma um componente abstrato em um componente concreto, porém podem ser necessárias outras alterações do código para finalizar essa transformação. Por exemplo, uma classe originalmente abstrata pode ter métodos abstratos que devem se tornar concretos (isto é, devem ter uma implementação definida) para que a classe seja considerada concreta e não ocorram erros de compilação ou execução.

Esta transformação, no entanto, não afeta nenhuma das métricas escolhidas para a avaliação, pois não há diferença nas medições entre componentes abstratos e concretos.

4.6.10 Adicionar palavra-chave `abstract` a componente

Modificação: adicionar a palavra-chave `abstract` a uma classe ou aspecto originalmente concreto.

```
public class SomeClass {
    (...)
}
```

```
public abstract class SomeClass {
    (...)
}
```

Métricas alteradas:

A adição da palavra-chave `abstract` transforma um componente concreto em um componente abstrato, porém esta transformação não afeta nenhuma das métricas escolhidas para a avaliação, pois não há diferença nas medições entre componentes abstratos e concretos.

4.6.11 Remover palavra-chave `abstract` de operação de interface

Modificação: remover a palavra-chave `abstract` de uma operação de uma interface.

```
public interface SomeInterface {
    public abstract void getResult();
    (...)
}
```

```
public interface SomeInterface {
    public ~~abstract~~ void getResult();
    (...)
}
```

Métricas alteradas:

Interfaces podem ter suas operações definidas com ou sem o uso da palavra-chave `abstract`, sendo equivalentes as duas formas de declaração. Desta forma, adicionar ou remover esta palavra-chave representa apenas uma alteração textual da interface, não tendo conseqüências em nenhuma das métricas usadas na avaliação.

4.6.12 Adicionar palavra-chave `abstract` a operação de interface

Modificação: adicionar a palavra-chave `abstract` a uma operação de uma interface.

```
public interface SomeInterface {
    public void getResult();
    (...)
}
```

```
public interface SomeInterface {
    public abstract void getResult();
    (...)
}
```

Métricas alteradas:

Interfaces podem ter suas operações definidas com ou sem o uso da palavra-chave `abstract`, sendo equivalentes as duas formas de declaração. Desta forma, adicionar ou remover esta palavra-chave representa apenas uma alteração textual da interface, não tendo conseqüências em nenhuma das métricas usadas na avaliação.

4.6.13 Compilar e testar

Modificação: nenhuma. Apenas indica o momento de compilar o código e aplicar os testes.

Métricas alteradas:

Este passo, presente em praticamente todas as descrições de refatoração, apenas indica o momento que as alterações feitas pela refatoração devem ser testadas para verificar a não-alteração do comportamento do software. Ele não modifica o código de nenhuma maneira, portanto seu efeito sobre o conjunto de métricas é nulo.

5 AVALIAÇÃO DAS REFATORAÇÕES

Neste capítulo são descritas as avaliações das refatorações escolhidas para serem analisadas (seção 2.2.1), conforme o processo definido no capítulo 3. A etapa do processo que avalia os passos básicos usados nas refatorações já tem sua execução descrita no capítulo 4, por isso este capítulo se concentra no restante do processo de avaliação, ou seja, na decomposição das refatorações, na obtenção dos impactos totais das mesmas nos valores das métricas, e nas análises dos resultados obtidos.

Para avaliar as refatorações assume-se que elas estejam corretas, isto é, que seus passos preservem o comportamento do software e nenhum erro seja inserido no código. Caso isso não fosse verdade, a refatoração não deveria ser utilizada porque processos de refatoração exigem a preservação do comportamento do software para serem realizados (FOWLER, 1999; MENS; TOURWÉ, 2004), portanto conhecer os impactos dos passos da refatoração não seria relevante já que esses impactos nunca seriam necessários. Dessa forma, não há neste capítulo qualquer preocupação em garantir a preservação do comportamento das refatorações avaliadas.

Outro pressuposto é que as refatorações avaliadas são aplicadas apenas nos contextos recomendados para sua utilização, e nunca nas situações em que seu uso deve ser evitado. Esse pressuposto é feito porque algumas refatorações utilizam essas informações para delimitar o contexto em que os passos básicos são executados, o que influencia nos impactos no valor das métricas.

Cada seção deste capítulo possui a avaliação de uma única refatoração, e é iniciada com a descrição da motivação e dos passos da refatoração – tendo como base o trabalho em que a refatoração é apresentada. Em seguida descreve-se como cada passo da refatoração é decomposto em um subconjunto dos passos básicos definidos no capítulo 4, ressaltando a quantidade que cada passo básico é necessário. Depois o impacto total da refatoração em cada uma das métricas é mostrado e justificado, mesmo que ele seja nulo. Por fim, é feita a análise dos resultados encontrados, verificando se os objetivos descritos pelo autor da refatoração foram alcançados, mostrando os efeitos colaterais da aplicação da refatoração e avaliando em que casos a refatoração é indicada ou deve ser evitada.

Ordem das avaliações

Como visto na seção 2.2, uma refatoração pode usar em seus passos outras refatorações menores. Na avaliação desse tipo de refatoração, as refatorações menores que já tenham sido avaliadas se comportam como passos básicos, isto é, não precisam ser decompostas em alterações mais simples e seus impactos devem ser conhecidos antes de iniciar a etapa final do processo de avaliação. Isso facilita a avaliação de refatorações mais complexas e aumenta o reaproveitamento de resultados obtidos em avaliações anteriores

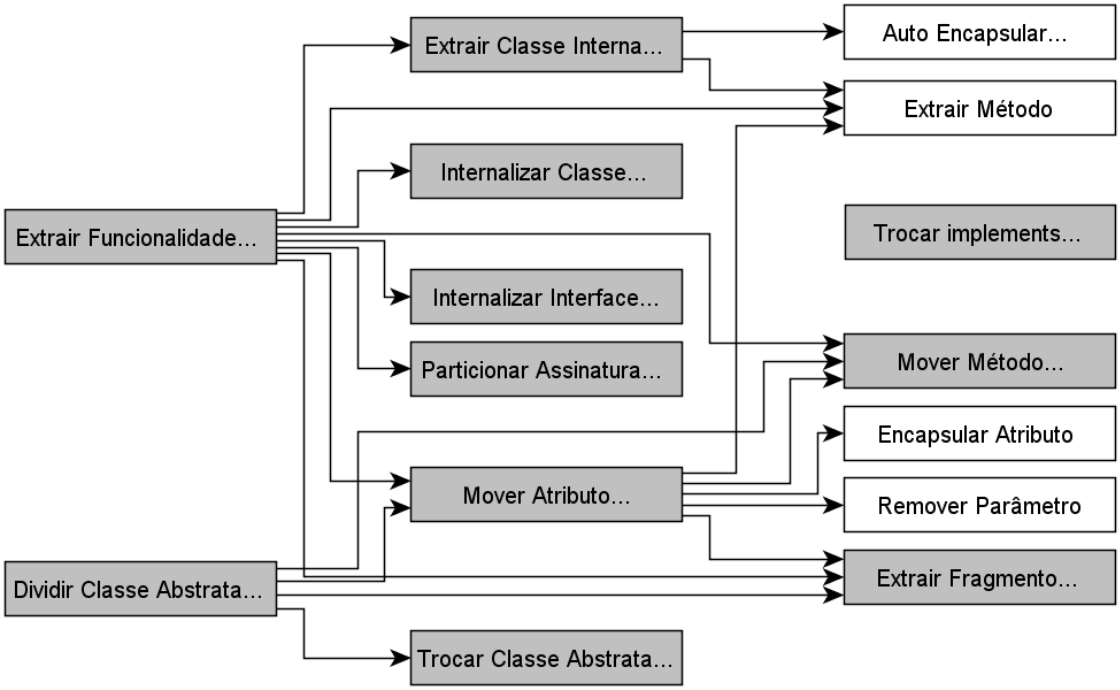


Figura 5.1: Relações de uso entre as refatorações avaliadas

pelo processo definido no capítulo 3. Por essa razão, as avaliações das refatorações OA de Monteiro (2005) não foram feitas na mesma ordem que elas são descritas no trabalho que as define[1], mas sim em uma ordem em que as refatorações menores são avaliadas antes das refatorações que as usam.

As refatorações OA avaliadas também usam quatro refatorações OO definidas por Fowler et al. (1999): *Extrair Método*, *Remover Parâmetro*, *Encapsular Atributo* e *Auto Encapsular Atributo*. Essas quatro refatorações são avaliadas da mesma maneira que as refatorações OA – usando o mesmo processo de avaliação e as mesmas métricas – e cada uma dessas avaliações é descrita em uma seção deste capítulo.

A figura 5.1 mostra um grafo que representa quais refatorações são usadas pelas refatorações avaliadas neste trabalho. Cada nodo representa uma refatoração – nodos cinzas são refatorações OA e nodos brancos refatorações OO –, e cada arco representa a relação de uso de uma refatoração por outra, onde a refatoração de origem do arco **usa** a refatoração de destino do mesmo. Dessa forma, a ordem das avaliações deve iniciar nas refatorações que não são origem de nenhum arco (isto é, que não usam nenhuma refatoração) e terminar nas refatorações que não são destino de nenhum arco (ou seja, que não são usadas por nenhuma refatoração).

Aritmética de impactos

Como visto no capítulo 4, a magnitude de um impacto pode ser um valor único ou um intervalo de valores, e esses valores podem ser um número inteiro ou um número variável (representado pelo símbolo ‘X’). Como devem ser feitas, então, operações aritméticas

[1] Aparentemente Monteiro optou por apresentar as refatorações em ordem alfabética

como soma e multiplicação entre magnitudes de uma mesma métrica? Qual o impacto total, por exemplo, quando um passo básico aumenta uma métrica entre uma e três unidades ('$>_{(1a3)}$') e outro aumenta sempre duas unidades ('$>_{(2)}$')? Antes de responder essas perguntas é necessário fazer algumas considerações sobre os impactos.

Em primeiro lugar, intervalos representam os extremos que o valor de uma métrica pode ser alterado após um passo básico, isto é, o mínimo e o máximo que o valor da métrica será mudado. No caso de impactos com valor único, eles também podem ser representados por um intervalo de valores, onde o mínimo e o máximo são iguais. Outra observação é sobre impactos representados por '$\leq$' e '$\geq$': eles omitem, respectivamente, um máximo e mínimo igual a zero, sendo sempre impactos com magnitude em um intervalo de valores. Assim, um impacto '$\leq_{(1)}$' também pode ser representado por '$<_{(0a1)}$', e isso quer dizer que a variação do valor da métrica está entre -1 e 0 – a inversão dos sinais ocorre por se tratar de um impacto que diminui ($<$) o valor da métrica.

Feitas essas observações, pode-se explicar agora como somar dois impactos da mesma métrica utilizando aritmética intervalar (OLIVEIRA; DIVERIO; CLAUDIO, 2005). Representando todos os impactos como intervalos de valores e lembrando que intervalos são as variações mínima e máxima do valor de uma métrica, o impacto total quando dois passos básicos p_1 e p_2 com impactos (min_1 a max_1) e (min_2 a max_2) forem executados em seqüência é ($min_1 + min_2$ a $max_1 + max_2$).

A representação dos impactos totais usa a mesma notação dos impactos dos passos básicos. Assim, no exemplo do início desta seção – soma dos impactos (1 a 3) e (2 a 2) – o impacto total é (3 a 5) e é representado por '$>_{(3a5)}$'. Como representar, no entanto, a soma de um inteiro z com 'X'? Ora, o símbolo 'X' é usado para representar um valor variável, que depende da situação em que o passo básico for aplicado. Assim, a soma "z + X" também terá um valor variável e, portanto, ela pode ser representada pelo símbolo 'X'. Por exemplo, $>_{(1aX)}$ + $>_{(2)}$ = $>_{(3aX)}$.

Para saber como multiplicar um impacto por um número natural, usa-se o comportamento da soma de impactos. A multiplicação de um impacto (min, max) por um número natural n é feita da seguinte maneira:

$$\begin{aligned} n * (min, max) &= \underbrace{(min, max) + (min, max) + ... + (min, max)}_{n\,vezes} \\ &= (\underbrace{min + min + ... + min}_{n\,vezes}, \underbrace{max + max + ... + max}_{n\,vezes}) \\ &= (n * min, n * max) \end{aligned}$$

Assim, para multiplicar um impacto por um número natural basta multiplicar seus extremos.

5.1 Refatorações Orientadas a Objetos

5.1.1 Extrair Método

A refatoração *Extrair Método* (FOWLER, 1999, p. 110) é uma das mais usadas durante o desenvolvimento de software, segundo Fowler et al. (1999) . Ela busca reunir porções de código relacionadas em um mesmo método, dando a ele um nome adequado para explicar sua função. Isso torna o código mais claro, aumenta as chances do método ser usado em outras partes do código e facilita a movimentação desta operação para ou-

tros componentes do código – através da transformação do método em uma declaração inter-tipo, por exemplo.

Para extrair o método da classe, devem ser executados os seguinte passos:

1. Criar um novo método e dar a ele um nome adequado;
2. Copiar o código a ser extraído do método de origem para o novo método;
3. Buscar no código extraído referências a variáveis que sejam locais no escopo do método de origem (parâmetros e variáveis declaradas localmente);
4. Caso haja variáveis locais usadas apenas dentro do código extraído, declará-las dentro do novo método como variáveis temporárias;
5. Caso haja uma variável local modificada pelo código extraído, verificar se é possível tratar o código extraído como uma consulta e atribuir o resultado da consulta a esta variável. Se isso for muito complicado ou se houver mais de uma variável nessa situação, não será possível extrair o método usando esta refatoração – Fowler et al. (1999, p. 111) apresentam sugestões para resolver esse impasse usando outras refatorações;
6. Se houver variáveis locais lidas pelo código extraído, elas devem ser passadas para o novo método como parâmetros.
7. Compilar após tratar todas as variáveis locais;
8. Substituir o código extraído no método de origem por uma chamada ao novo método e remover as declarações de variáveis temporárias movidas para dentro do método no passo 4;
9. Compilar e testar;

Exemplo:

Para melhor ilustrar as diferentes ações a serem executadas para cada tipo de acesso a variáveis locais, será mostrado um exemplo a seguir baseado no código apresentado por Fowler et al. (1999) na descrição desta refatoração.

Inicialmente o método `printOwing()` contém todos os comandos para imprimir a dívida do cliente e usa comentários para separas os grupos de código:

```
void printOwing() {
    Enumeration e = orders.elements();
    double outstanding = 0.0;
    //print banner
    System.out.println("*****************");
    System.out.println("* Customer Owes *");
    System.out.println("*****************");

    //get outstanding
    while (e.hasMoreElements()) {
        Order each = (Order)e.nextElement();
        outstanding += each.getAmount();
    }

    //print details
    System.out.println("name:  "+ name);
    System.out.println("amount "+ outstanding);
}
```

A seguir extrai-se um método apenas para imprimir o cabeçalho da mensagem, o qual é chamado de `printBanner()`. Ele não usa nenhuma variável local e é apenas uma cópia das linhas do método original:

```
void printOwing() {
    Enumeration e = orders.elements();
    double outstanding = 0.0;
    printBanner();

    //get outstanding
    while (e.hasMoreElements()) {
        Order each = (Order)e.nextElement();
        outstanding += each.getAmount();
    }

    //print details
    System.out.println("name:  "+ name);
    System.out.println("amount "+ outstanding);
}

void printBanner() {
    System.out.println("*****************");
    System.out.println("* Customer Owes *");
    System.out.println("*****************");
}
```

No próximo passo, extrai-se o método que imprime detalhes da mensagem, chamado de `printDetails()`. Este método lê o valor da variável local `outstanding` mas não o altera, por isso a operação deve ter também apenas um parâmetro:

```
void printOwing() {
    Enumeration e = orders.elements();
    double outstanding = 0.0;
    printBanner();

    //get outstanding
    while (e.hasMoreElements()) {
        Order each = (Order)e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}

void printBanner() {(...)}

void printDetails(double outstanding) {
    System.out.println("name:  "+ name);
    System.out.println("amount "+ outstanding);
}
```

Por fim, ainda é possível extrair outro método, desta vez o que calcula o valor da dívida. Esta extração é um pouco mais complicada que as anteriores, pois usa uma variável local que deve ser declarada como temporária (a enumeração `e`) e também altera o valor de uma variável necessária no método de origem (`outstanding`). Após a extração deste método, o qual recebe o nome de `getOutstanding()`), os códigos resultantes são os seguintes:

```
void printOwing() {
    printBanner();
    double outstanding = getOutstanding();
    printDetails(outstanding);
}

void printBanner() {(...)}

void printDetails(double outstanding) {(...)}

double getOutstanding() {
    Enumeration e = orders.elements();
    double outstanding = 0.0;
    while (e.hasMoreElements()) {
        Order each = (Order)e.nextElement();
        outstanding += each.getAmount();
    }
    return outstanding;
}
```

Decomposição da Refatoração em Passos Básicos

Para executar o primeiro passo da refatoração, usa-se o passo básico "Criar operação vazia" (58) para adicionar um método sem parâmetros. Se no passo 5 houver uma variável local modificada pelo código extraído, o método criado deve ter o retorno do mesmo tipo desta variável, caso contrário o método deve ter retorno vazio (`void`).

No passo 2, a cópia do código para o novo método é feita por "Adicionar linhas de código a operação" (76).

O 3o. passo trata apenas da localização de variáveis que poderão ser movidas para o método extraído pelos passos 4, 5 e 6. Dessa forma, nenhuma alteração é executada no software, portanto o passo não precisa ser decomposto em passos básicos.

No passo 4, para cada uma das $n_{vartemp}$ variáveis locais usadas apenas dentro do código extraído, deve ser declarada uma variável dentro do novo método pelo passo básico "Adicionar variável local a operação" (62).

Caso o passo 5 tenha que ser executado (se houver uma variável modificada pelo código extraído), a transformação do método em consulta é feita por duas ações: adicionar um tipo de retorno ao método e adicionar uma linha de retorno ao final de seu corpo. A primeira ação já é executada pelo passo básico "Criar operação vazia" no passo 1, enquanto que a segunda deve ser feita executando mais uma vez o passo básico "Adicionar linhas de código a operação" (76).

O passo 6 adiciona as n_{param} variáveis lidas como parâmetros ao novo método, ação que é feita pelo passo básico "Adicionar parâmetro a operação" (60).

Os passos 7 e 9 são executados pelo passo básico "Compilar e testar" (101).

E, por fim, o passo 8 é decomposto em três passos básicos: "Adicionar chamada a operação" (79) e "Remover linhas de código de operação" fazem a substituição do código no método de origem pela chamada ao novo método; e "Remover variável local de operação" (62) é usado $n_{vartemp}$ vezes para eliminar as variáveis temporárias movidas para o novo método no passo 4.

A tabela 5.1 mostra quantas vezes e onde (em que passo) cada passo básico é utilizado por esta refatoração.

Cálculo da Variação Total das Métricas

Uma vez decompostos os passos da refatoração, deve-se calcular qual o impacto total das alterações nas métricas escolhidas para avaliar a refatoração. A tabela 5.2 mostra o impacto de cada passo básico usado pela refatoração. Nesta tabela é possível observar que as métricas **VS**, **NOA** e **DIT** não são modificadas por nenhum dos passos, portanto

Tabela 5.1: Quantidade de vezes que cada passo básico é usado em *Extrair Método*

Passo da refatoração	Quantidade	Passo básico
1	1	Criar operação vazia
2	1	Adicionar linhas de código a operação (n linhas)
3	-	-
4	$n_{vartemp}$	Adicionar variável local a operação
5	1	Adicionar linhas de código a operação (1 linha)
6	n_{param}	Adicionar parâmetro a operação
7	1	Compilar e testar
8	1	Adicionar chamada a operação
	1	Remover linhas de código de operação (n linhas)
	$n_{vartemp}$	Remover variável local de operação
9	1	Compilar e testar

conclui-se que essas métricas não terão seu valor alterado quando *Extrair Método* for aplicada. A variação total de cada uma das demais métricas é explicada a seguir.

Tabela 5.2: Impacto nas métricas de cada passo básico de *Extrair Método*

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Criar operação vazia	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(2)}$	=	$>_{(1)}$	$\geq_{(1)}$	=	$\geq_{(X)}$
Adicionar linhas de código a operação	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(X)}$	=	$>_{(X)}$	=	=	$\geq_{(X)}$	=	$\leq_{(X)}$
Adicionar variável local a operação	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(1)}$	=	=	$\geq_{(1)}$	=	=
Adicionar parâmetro a operação	=	=	=	=	=	=	$>_{(1)}$	$\geq_{(1)}$	=	=
Adicionar chamada a operação	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(1)}$	=	=	$\geq_{(1)}$	=	=
Remover linhas de código de operação	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(X)}$	=	$<_{(X)}$	=	=	$\leq_{(X)}$	=	$\geq_{(X)}$
Remover variável local de operação	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(2)}$	=	$<_{(1)}$	=	=	$\leq_{(1)}$	=	=
Compilar e testar	=	=	=	=	=	=	=	=	=	=

CDC: =

Todas as alterações feitas por esta refatoração ocorrem no mesmo componente. Assim, a criação da operação vazia, a adição e remoção das linhas de código, a adição e remoção das variáveis locais e a adição de parâmetros são todas aplicadas no mesmo componente. Dessa forma, a presença ou não de interesses transversais neste componente após a refatoração dependerá da sua existência no mesmo antes da refatoração, pois esta refatoração trata apenas da alteração do local de algumas linhas de código, e não da adição ou remoção de interesses do componente. Por causa disso, só haverá um determinado interesse transversal entrecortando o código do componente se ele já estivesse presente antes e, portanto, o valor de CDC não é modificado por esta refatoração.

CDO: $\geq_{(1)}$

Ao final da extração do método haverá uma operação a mais no componente do que originalmente. Se o método extraído e a operação de onde foram tiradas as linhas do novo método forem ambos relacionados a um interesse transversal, então esta refatoração terá aumentado o número de operações por onde este interesse está disperso, incrementando assim seu CDO em uma unidade. Se no entanto não houver interesse transversal presente no trecho extraído, o valor de CDO não é modificado.

CDLOC: $\leq_{(X)}$

Segundo a motivação para o uso de *Extrair Método*, esta refatoração deve ser usada

para “reunir porções de código relacionadas em um mesmo método” (FOWLER, 1999). Dessa forma, ao aplicar esta refatoração, espera-se que linhas de código sombreadas que originalmente estivessem separadas passem a estar juntas numa mesma operação (e linhas sombreadas adjacentes não passem a estar separadas). Com isso, pode haver uma diminuição do número de pontos de transição no componente, diminuindo assim o valor de CDLOC. O quanto esses pontos diminuirão depende do código em que a refatoração é aplicada e dos interesses transversais que entrecortam esse código.

LOC: $>_{(3a4)}$

Considerando que foram extraídas n linhas de código da operação original, e ponderando os impactos dos passos básicos em LOC de acordo com a tabela 5.1, a variação total do valor de LOC será: $2 + (n + 1) + n_{vartemp} + 1 - n - n_{vartemp} = 4$. Estas 4 linhas correspondem às duas linhas de declaração do novo método, à linha de chamada do novo método e à linha com o comando de retorno do novo método, caso o método possa ser tratado como uma consulta (passo 5). Se o método não alterar nenhuma variável em seu trecho extraído, o passo 5 não é necessário e esta última linha não será criada, resultando em um aumento de LOC de apenas três unidades.

Algumas outras linhas são adicionadas ou removidas do componente, porém as variações de LOC resultantes dessas alterações se anulam quando analisadas conjuntamente. É o caso das n linhas adicionadas ao novo método por *Adicionar linhas de código a operação*, as quais são anuladas pela remoção das n linhas da operação original por *Remover linhas de código de operação*; também as linhas das $n_{vartemp}$ variáveis adicionadas por *Adicionar variável local a operação* serão anuladas pela remoção das $n_{vartemp}$ variáveis da operação original por *Remover variável local de operação*.

WOC: $>_{(n_{param}+1)}$

Dois passos básicos aumentam o valor de WOC: *Criar operação vazia* e *Adicionar parâmetro a operação*. Ambos aumentam o valor desta métrica em uma unidade, portanto ao multiplicar este valor pelo número de vezes que cada passo é executado (tabela 5.1), tem-se que WOC aumenta $1 + n_{param}$ unidades.

CBC: =

Como esta refatoração trata apenas da mudança da localização de algumas linhas de código dentro do mesmo componente, não é necessário criar ou remover acoplamentos dele com os demais componentes do sistema. Todas as linhas, os tipos de retorno, os parâmetros e as variáveis adicionadas (ou removidas) tem acoplamentos que já existiam previamente no componente, por isso esta refatoração não altera o valor de CBC.

LCOO: -X a +X

Com a extração das linhas em um novo método, serão criados novos pares de operações entre o novo método e as demais operações do componente, os quais poderão fazer parte da parcela positiva (operações que não compartilham atributos) ou da parcela negativa (operações que compartilham atributos) do cálculo de LCOO. Além disso, alguns pares contendo a operação original podem ter sua classificação alterada, pois dentre as linhas extraídas pode haver alguma que acesse algum atributo do componente não mais acessado no resto da operação após a refatoração.

Assim, dependendo das linhas extraídas e das demais operações do componente, o valor de LCOO poderá aumentar, diminuir ou mesmo não sofrer alteração.

Análise dos Resultados

Analisando os resultados mostrados na tabela 5.3, é possível observar que a extração do método pode aumentar ou diminuir a separação dos interesses (ou mesmo fazer os

Tabela 5.3: Variação das métricas em *Extrair Método*

Separação de Interesses			Tamanho				Acoplamento		Coesão
CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
=	$\geq_{(1)}$	$\leq_{(X)}$	=	$>_{(3a4)}$	=	$>_{(X)}$	=	=	$\gtrless$

dois) em níveis diferentes – operações e linhas de código. Já o tamanho do software será obrigatoriamente maior, enquanto que o acoplamento não é alterado, uma vez que esta refatoração trata apenas de uma melhor organização interna de um componente e não modifica sua relação com os outros componentes do sistema. Por fim, a coesão pode ser modificada positiva ou negativamente dependendo das características do componente em que o método for extraído.

Estes resultados mostram que a refatoração pode ser vantajosa ou não dependendo do caso em que é aplicada. É possível diminuir muito o entrelaçamento de código (medido por CDLOC) mesmo com um pequeno aumento de seu espalhamento (incremento de uma unidade de CDO), assim como o aumento do tamanho do componente pode ser compensado por uma melhor coesão do mesmo (diminuição de LCOO). Em outros casos as variações das métricas podem não ser satisfatórias e a refatoração resultar em um sistema pior do que antes das alterações feitas por ela.

5.1.2 Remover Parâmetro

Ao longo do tempo, métodos são modificados para se adaptar à evolução do programa, e em determinado momento pode ser que um parâmetro deixe de ser necessário para o método que o contém. Neste caso, a refatoração *Remover Parâmetro* (FOWLER, 1999, p. 277) recomenda eliminar este parâmetro para evitar preocupações desnecessárias dos componentes que utilizam este método. Caso o método faça parte da interface publicada da classe ou se ele for sobrecarregado, talvez não seja possível executar a remoção.

Os passos para remover o parâmetro do método são os seguintes:

1. Verificar se a assinatura do método é implementada por uma super-classe ou subclasse e se nestes casos o parâmetro é utilizado. Se for, não executar a refatoração;

2. Declarar um novo método sem o parâmetro e copiar o corpo do método original para o novo método;

3. Compilar;

4. Alterar o corpo do método antigo para que ele chame o método novo;

5. Compilar e testar;

6. Encontrar todas as referências ao método original e alterá-las para se referirem ao novo. Compilar e testar após cada alteração;

7. Remover o método original. Se ele fizer parte da interface publicada da classe e não puder ser removido, apenas marcá-lo como obsoleto;

8. Compilar e testar.

Ao trocar as chamadas ao método antigo por chamadas ao método novo, um dos valores passados originalmente em cada chamada deixará de ser necessário por causa da remoção do parâmetro do método. Com isso, todas os comandos usados apenas para a obtenção desse valor deixarão de ser necessários e poderão ser removidos do código. Esta alteração, juntamente com a criação do método sem o parâmetro no passo 2 e a remoção do método original no passo 7 alterarão algumas das métricas, conforme será explicado a seguir.

Decomposição da Refatoração em Passos Básicos

O primeiro passo não executa nenhuma ação no código e serve apenas para verificar se é possível aplicar a refatoração, por isso ele não precisa ser decomposto em nenhum passo básico.

O passo 2 executa duas ações: cria o novo método e preenche seu corpo com as linhas de código do método original. Para criar o novo método, é necessário usar os passos básicos "Criar operação vazia" (58) e "Adicionar parâmetro a operação" (60) – este último usado diversas vezes (n_{param}) para copiar a lista de parâmetros do método original, sem o parâmetro removido. Já para preencher o corpo do novo método usa-se o passo básico "Adicionar linhas de código a operação" (76).

Os passos 3, 5 e 8 são todos equivalentes ao passo básico "Compilar e testar" (101).

Para executar o passo 4 deve-se remover o corpo do método original e adicionar a ele uma chamada ao novo método. Isso é feito pelos passos básicos "Remover linhas de código de operação" (78) e "Adicionar chamada a operação" (79).

O 6o. passo troca as chamadas ao método original por chamadas ao novo método. Além da troca da operação referenciada, esta ação também deve remover os comandos usados apenas para obter o valor passado como parâmetro nas chamadas, conforme explicado anteriormente. Esses comandos serão desnecessários depois que o parâmetro não for mais usado na chamada da operação e, portanto, poderão ser removidos. Assim, este passo da refatoração é decomposto nos seguintes passos básicos: "Trocar chamada a operação sobrecarregada por versão com menos parâmetros" (71) – usado $n_{chamadas}$ vezes para substituir cada uma das chamadas ao método original que devem ser trocadas –, "Remover linhas de código de operação" (78) – necessário para remover os comandos que se tornam dispensáveis com a remoção do parâmetro nas diversas operações onde a chamada é substituída – e "Compilar e testar" (101) – também usado $n_{chamadas}$ vezes, pois após cada troca de chamada e remoção de linhas desnecessárias o software deve ser compilado.

Por fim, o passo 7 será executado de duas formas diferentes dependendo da interface publicada do componente. Se o método puder ser removido, são usados os passos básicos "Remover linhas de código de operação" (78) para remover os comandos de seu corpo, "Remover parâmetro de operação" (61) para remover seus $n_{param} + 1$ parâmetros e "Remover operação vazia" (59) para eliminar o próprio método. Se, no entanto, o método não puder ser removido por fazer parte da interface publicada do componente, então ele deve ser marcado como obsoleto através do uso do passo básico "Marcar operação como obsoleta" (80).

A tabela 5.4 mostra quantas vezes e onde (em que passo) cada passo básico é utilizado por esta refatoração.

Tabela 5.4: Quantidade de vezes que cada passo básico é usado em *Remover Parâmetro*

Passo da refatoração	Quantidade	Passo básico
1	-	-
2	1	Criar operação vazia
	n_{param}	Adicionar parâmetro a operação
	1	Adicionar linhas de código a operação
3	1	Compilar e testar
4	1	Remover linhas de código de operação
	1	Adicionar chamada a operação
5	1	Compilar e testar
6	$n_{chamadas}$	Trocar chamada a operação sobrecarregada por versão com menos parâmetros
	$\leq n_{chamadas}$	Remover linhas de código de operação
	$n_{chamadas}$	Compilar e testar
7	1	Remover linhas de código de operação
	$n_{param} + 1$	Remover parâmetro de operação
	1	Remover operação vazia
	1	Marcar operação como obsoleta
8	1	Compilar e testar

Cálculo da Variação Total das Métricas

Uma vez decompostos os passos da refatoração, deve-se calcular qual o impacto total das alterações nas métricas escolhidas para avaliar a refatoração. A tabela 5.5 mostra o impacto de cada passo básico usado pela refatoração. Nesta tabela é possível observar que as métricas **VS**, **NOA** e **DIT** não são modificadas por nenhum dos passos, portanto conclui-se que essas métricas não terão seu valor alterado quando *Remover Parâmetro* for aplicada. A variação total de cada uma das demais métricas é explicada a seguir.

Tabela 5.5: Impacto nas métricas de cada passo básico de *Remover Parâmetro*

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Criar operação vazia	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(2)}$	=	$>_{(1)}$	$\geq_{(1)}$	=	$\geq_{(X)}$
Adicionar parâmetro a operação	=	=	=	=	=	=	$>_{(1)}$	$\geq_{(1)}$	=	=
Adicionar linhas de código a operação	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(X)}$	=	$>_{(X)}$	=	=	$\geq_{(X)}$	=	$\leq_{(X)}$
Remover linhas de código de operação	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(X)}$	=	$<_{(X)}$	=	=	$\leq_{(X)}$	=	$\geq_{(X)}$
Adicionar chamada a operação	=	=	=	=	=	=	=	=	=	=
Trocar chamada a operação sobrecarregada por versão com menos parâmetros	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(X)}$	=	$\leq_{(X)}$	=	=	$\leq_{(X)}$	=	=
Remover parâmetro de operação	=	=	=	=	=	=	$<_{(1)}$	$\leq_{(1)}$	=	=
Remover operação vazia	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(2)}$	=	$<_{(2)}$	=	$<_{(1)}$	$\leq_{(1)}$	=	$\leq_{(X)}$
Marcar operação como obsoleta	=	=	=	=	=	=	=	=	=	=
Compilar e testar	=	=	=	=	=	=	=	=	=	=

CDC: $-n_{chamadas} - 1$ a 0

Se o método com mais parâmetros puder ser removido no passo 7 e o tipo do parâmetro removido for um dos componentes principais de um interesse transversal, então o valor de CDC deste interesse diminuirá uma unidade caso não haja outra referência a este interesse no restante do componente de onde o método foi removido.

Além disso, a refatoração também modifica outras partes do código que podem diminuir o valor de CDC: as operações onde estão as chamadas ao método original (passo 6). Nessas operações é possível que alguns comandos se tornem desnecessários após a refatoração porque não será mais preciso passar um dos valores como parâmetro, por isso algumas linhas de código poderão ser removidas dos componentes. Neste caso, se houver nessas linhas algum componente principal de um interesse transversal, e se o componente onde a linha estivesse originalmente não for relacionado a esse interesse em nenhum outro local, então a remoção da linha fará com que o componente deixe de estar relacionado ao interesse e, portanto, diminuirá o número de componentes pelos quais o interesse está espalhado, reduzindo o valor de CDC em uma unidade para cada caso em que isso acontecer. Esta situação poderá ocorrer tantas vezes quanto existirem operações onde sejam feitas substituições do método chamado, isto é, até $n_{chamadas}$ vezes.

CDO: $-n_{chamadas}$ a +1

Se a operação cujo parâmetro deve ser removido for uma das operações principais de um interesse transversal, e se a operação original não puder ser removida no passo 7, então ao final da refatoração haverá uma operação a mais por onde o interesse está espalhado.

No entanto, também é possível que o interesse deixe de estar espalhado em algumas operações caso alguns comandos se tornem desnecessários e sejam removidos no passo 6. Isso ocorre se dentre os comandos removidos houver uma chamada a uma das operações principais do interesse e se a operação que originalmente continha os comandos removidos não for relacionada ao interesse em nenhuma outra linha de código. Neste caso, ao final da refatoração a operação não estará mais relacionada ao interesse e o número de operações por onde o interesse está espalhado diminui. Esta situação poderá ocorrer tantas vezes quanto existirem operações onde ocorrem substituições do método chamado, isto é, até $n_{chamadas}$ vezes.

CDLOC: -X a 0

No trecho de código que compreende o método novo e o método original (de onde foi removido o parâmetro) não há mudança do número de pontos de transição após a refatoração. Isso ocorre porque o corpo do primeiro é igual ao corpo original do segundo, portanto todos os pontos de transição serão movidos do método com mais parâmetros para o método com menos parâmetros, sem haver inclusão ou exclusão de algum ponto de transição.

Já no caso das operações que puderem ter linhas removidas no passo 6, é possível que dentre os trechos excluídos existam pontos de transição, os quais deixarão de existir após a refatoração. O número de operações que podem ter trechos removidos é $n_{chamadas}$, porém em cada uma das operações podem ser eliminados diversos pontos de transição, portanto não é possível estabelecer um limite máximo para a diminuição de CDLOC.

LOC: -X a +3

O método novo terá o mesmo número de linhas de código que o método original, uma vez que o corpo de ambos é idêntico (passo 2). No entanto, se o método com mais parâmetros não puder ser removido no passo 7, então a refatoração adicionará três linhas de código ao componente – uma linha com a declaração do método não-removido, outra com a chamada do novo método (passo 4) e mais uma de encerramento do escopo do método.

A refatoração também pode diminuir o número de linhas de código, caso seja possível remover comandos ao trocar as chamadas do método antigo por chamadas do novo método. Neste caso, cada operação que tiver comandos removidos diminuirá seu número de linhas de código. Como o número de linhas removidas depende da operação onde a chamada é trocada – podendo ser apenas uma ou muito mais –, não é possível estabelecer

o número máximo de linhas excluídas.

WOC: -1 a $+n_{param}$

O número de parâmetros do novo método é uma unidade menor que o número de parâmetros do método original. Conseqüentemente, sua complexidade também é uma unidade menor que a complexidade do método original. Se for possível remover o método no passo 7, então a complexidade do método com mais parâmetros não será mais considerada no cálculo de WOC e seu valor terá diminuído uma unidade com a refatoração, caso contrário WOC contará as duas complexidades e seu valor será n_{param} unidades maior que originalmente.

CBC: -X a 0

Se o método com mais parâmetros puder ser removido no passo 7 e se o tipo do parâmetro removido for um componente que não é referenciado em nenhuma outra parte do componente que contém este método, então a remoção desta operação fará com que o componente que a continha perca um de seus acoplamentos.

No caso dos componentes cujas chamadas são trocadas no passo 6, o valor de CBC também poderá diminuir caso seja possível eliminar comandos após a troca da operação chamada. Se dentre os comandos houver um componente que não é referenciado em mais nenhum local do componente onde a chamada foi trocada, então este componente perderá acoplamentos, o que diminui o valor de CBC.

A diminuição máxima do valor desta métrica, no entanto, não pode ser definida porque é possível que em cada uma das $n_{chamadas}$ trocas de operações chamadas sejam eliminados diversos acoplamentos a outros componentes.

LCOO: 0 a $+n_{op}$

Os pares de operações que contenham o método original deixarão de existir caso o mesmo seja removido do código no passo 7. Em seu lugar entrarão os novos pares formados pelo método criado no passo 2 com as demais operações da classe, os quais terão a mesma classificação (se compartilham ou não atributos) dos pares do método original, uma vez que o novo método é apenas uma cópia do antigo sem um de seus parâmetros (passo 2). Assim, se o método com mais parâmetros puder ser removido, o valor de LCOO não é modificado pela refatoração.

No entanto, se este método não puder ser removido do componente no passo 7, novos pares de operações serão formados entre ele e as demais n_{op} operações do componente. Como este método possui apenas uma chamada a outra operação (passo 4) e, conseqüentemente, não acessa nenhum atributo do componente, todos esses pares serão classificados como não tendo atributos em comum. Assim, esses novos pares farão todos parte da parcela positiva de LCOO e, portanto, o valor desta métrica será maior do que seu valor original.

Tabela 5.6: Variação das métricas em *Remover Parâmetro*

Separação de Interesses			Tamanho				Acoplamento		Coesão
CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
$\leq_{(X)}$	$\gtrapprox_{(-Xa+1)}$	$\leq_{(X)}$	=	$\gtrapprox_{(-Xa+3)}$	=	$\gtrapprox_{(-1a+X)}$	$\leq_{(X)}$	=	$\geq_{(X)}$

Análise dos Resultados

Analisando os resultados mostrados na tabela 5.6, é possível observar que tanto a separação dos interesses quanto o tamanho do software poderão melhorar ou piorar dependendo do código em que esta refatoração for aplicada. Já o acoplamento poderá apenas

melhorar, enquanto que a coesão poderá apenas piorar.

Também é possível notar que a remoção dos comandos que se tornam desnecessários ao eliminar o parâmetro permite uma melhora da separação de interesses, do tamanho do software (no que diz respeito ao seu número de linhas de código), e do acoplamento entre os componentes. No entanto, caso o método original não possa ser removido por fazer parte da interface publicada do componente, ocorrerão pioras em alguns atributos internos, como tamanho do software e coesão.

Estes resultados mostram que a refatoração pode ser vantajosa ou não dependendo do caso em que é aplicada. É possível que quase todos os atributos internos diminuam ao usar esta refatoração – a exceção é a coesão dos componentes. Por outro lado, em alguns casos esta refatoração também pode ter o efeito inverso e aumentar quase todos os atributos internos (menos o acoplamento entre componentes). Assim, ao usar esta refatoração deve-se ter consciência que nem sempre as conseqüências serão positivas.

5.1.3 Encapsular Atributo

Algumas classes podem apresentar atributos públicos, os quais são acessíveis a qualquer componente do sistema. Apesar de ser simples obter os valores desses atributos e alterá-los, atributos públicos ferem um dos principais princípios da orientação a objetos: o encapsulamento, ou ocultação de dados (FOWLER, 1999). Conseqüentemente, a modularidade do programa será reduzida (FOWLER, 1999).

A longo prazo, essa "facilidade" pode se deteriorar, pois uma modificação no acesso aos dados terá de ser replicada ao longo do código, em todos os trechos onde os dados forem acessados. Caso os dados e o comportamento que os usa estiverem agrupados em um só local, será mais fácil modificar o código. Para transformar dados públicos em privados, é proposta a refatoração *Encapsular Atributo* (FOWLER, 1999, p. 206), a qual sugere tornar os dados privados e fornecer métodos de acesso a eles. Os passos para executar esta refatoração são os seguintes:

1. Criar métodos de gravação e leitura para o atributo;

2. Encontrar todos os clientes fora da classe que referenciem o atributo. Se o cliente usar o valor, substituir a referência por uma chamada ao método de leitura. Se o cliente alterar o valor, substituir a referência por uma chamada ao método de gravação;

3. Compilar e testar após cada alteração;

4. Assim que todos os clientes forem alterados, declarar o atributo como privado;

5. Compilar e testar.

Exemplo:
O código a seguir exemplifica como esta refatoração altera um programa:

```
public class Person {
    public String name;
    (...)
    public String toString() {
        StringBuffer result = new StringBuffer("Name:  ");
        result.append(name);
        (...)
        return result.toString();
    }
}
```

```
public class ReportManager {
    public Person[] clients;
    (...)
    public void printReports() {
        for(int i=0; i<=clients.length; i++) {
            System.out.println("Client "+ clients[i].name);
            (...)
        }
        (...)
    }
}
```

```
public class Person {
    private String name;
    (...)
    public String toString() {
        StringBuffer result = new StringBuffer("Name:  ");
        result.append(getName());
        (...)
        return result.toString();
    }
    public String getName() {
        return name;
    }
    public void setName(String newName) {
        name = newName;
    }
}
```

```
public class ReportManager {
    public Person[] clients;
    (...)
    public void printReports() {
        for(int i=0; i<=clients.length; i++) {
            System.out.println("Client "+ clients[i].getName());
            (...)
        }
        (...)
    }
}
```

Posteriormente, se fosse necessário alterar a forma de acesso ao atributo `name` da classe `Person`, bastaria modificar os métodos `getName()` e `setName(String)`, ao invés de modificar todos os acessos a esse atributo ao longo do código. Por exemplo, caso o nome da Pessoa devesse ter obrigatoriamente um valor padrão e nunca pudesse ser `null`, uma opção seria modificar o método `getName()` da seguinte forma:

```
public String getName() {
    if (name == null)
        return "[sem nome]";
    return name;
}
```

Se a refatoração *Encapsular Atributo* não tivesse sido aplicada, seria necessário modificar o código tanto no método `toString()` da classe `Person` quanto no método `printReports()` de `ReportManager`.

Decomposição da Refatoração em Passos Básicos

O primeiro passo da refatoração é responsável por criar as operações de leitura e escrita do atributo. Para elaborar a operação de leitura, é necessário usar "Criar operação vazia" (58) para criar a operação com retorno do mesmo tipo do atributo e "Adicionar linha de leitura/escrita de valor de atributo a operação" (73) para colocar a linha de leitura do atributo na nova operação. Já a operação de escrita é criada usando "Criar operação vazia" (58) para criar a operação sem valor de retorno, "Adicionar parâmetro a operação" (60) para colocar o parâmetro que possuirá o novo valor do atributo na operação criada, e "Adicionar linha de leitura/escrita de valor de atributo a operação" (73) para colocar a linha de escrita do atributo no novo método.

No 2o. passo são trocados os acessos diretos ao atributo por chamadas às operações de leitura e escrita criadas no passo anterior. Isto é feito executando "Substituir referência direta a atributo por chamada a operação de leitura ou escrita" (74) em cada um dos $n_{acessos}$ acessos diretos ao atributo.

Por fim, os passos 3 e 5 são equivalentes ao passo básico "Compilar e testar" (101), enquanto que o passo 4 é executado usando "Tornar atributo privado" (92).

A tabela 5.7 mostra quantas vezes e onde (em que passo) cada passo básico é utilizado por esta refatoração.

Tabela 5.7: Quantidade de vezes que cada passo básico é usado em *Encapsular Atributo*

Passo da refatoração	Quantidade	Passo básico
1	2	Criar operação vazia
	2	Adicionar linha de leitura/escrita de valor de atributo a operação
	1	Adicionar parâmetro a operação
2	$n_{acessos}$	Substituir referência direta a atributo por chamada a operação de leitura ou escrita
3	$n_{acessos}$	Compilar e testar
4	1	Tornar atributo privado
5	1	Compilar e testar

Cálculo da Variação Total das Métricas

Uma vez decompostos os passos da refatoração, deve-se calcular qual o impacto total das alterações nas métricas escolhidas para avaliar a refatoração. A tabela 5.8 mostra o impacto de cada passo básico usado pela refatoração. Nesta tabela é possível observar que as métricas **VS**, **NOA** e **DIT** não são modificadas por nenhum dos passos, portanto conclui-se que essas métricas não terão seu valor alterado quando *Encapsular Atributo* for aplicada. A variação total de cada uma das demais métricas é explicada a seguir.

CDC: =

As operações criadas por esta refatoração usam componentes que já estavam previamente presentes no componente que as contêm, seja em seus parâmetros, seja em seus tipos de retorno, seja em seus corpos. Isto significa que nenhum componente principal de interesses transversais que está no componente ao final da refatoração foi adicionado durante a refatoração. Portanto, o valor de CDC não é alterado pela refatoração.

CDO: 0 a $n_{acessos} + 2$

Se o atributo encapsulado for usado para implementar um interesse transversal, então

Tabela 5.8: Impacto nas métricas de cada passo básico de *Encapsular Atributo*

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Criar operação vazia	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(2)}$	=	$>_{(1)}$	$\geq_{(1)}$	=	$\geq_{(X)}$
Adicionar linha de leitura/escrita de valor de atributo a operação	=	=	=	=	$>_{(1)}$	=	=	=	=	$\leq_{(X)}$
Adicionar parâmetro a operação	=	=	=	=	=	=	$>_{(1)}$	$\geq_{(1)}$	=	=
Substituir referência direta a atributo por chamada a operação de leitura ou escrita	=	$\geq_{(1)}$	=	=	=	=	=	=	=	$\geq_{(X)}$
Tornar atributo privado	=	=	=	=	=	=	=	=	=	=
Compilar e testar	=	=	=	=	=	=	=	=	=	=

seus métodos de acesso serão operações principais desse interesse. Além disso, cada uma das operações em que o acesso direto ao atributo for substituído por uma chamada a um método de acesso se tornará relacionada ao interesse (se não o for originalmente) porque passará a acessar uma das operações principais do interesse. Assim, além do aumento do número de operações principais do interesse transversal em duas unidades, será incrementado também o número de operações que acessam uma das operações principais do interesse (em no máximo $n_{acessos}$ unidades), aumentando assim o valor de CDO em até $n_{acessos} + 2$ unidades.

CDLOC: =

Se o atributo encapsulado for usado para implementar um interesse transversal, então tanto a sua declaração quanto seus métodos de acesso terão suas linhas sombreadas. Neste caso, basta posicionar os métodos de acesso logo abaixo da declaração do atributo para não criar nenhum ponto de transição no código do componente. O mesmo vale para o caso em que o atributo não está relacionado a nenhum interesse transversal.

Em relação às linhas com acesso ao atributo – onde o acesso direto é trocado por uma chamada a um dos métodos de acesso criados (passo 2) –, também não há criação ou remoção de pontos de transição porque o sombreamento dessas linhas será o mesmo de antes: se o atributo for usado para implementar um interesse transversal, então serão sombreadas as linhas de declaração do atributo, as linhas dos métodos de acesso, as de acesso direto ao atributo e as que chamam seus métodos de acesso. Por isso, a troca do acesso direto pelo indireto não altera o sombreamento das linhas modificadas.

Portanto, nem os métodos criados nem as linhas alteradas alteram o número de pontos de transição do software.

LOC: +6

O aumento do número de linhas de código é dado pelos impactos de "Criar operação vazia" e "Adicionar linha de leitura/escrita de valor de atributo a operação" no valor de LOC: o primeiro passo básico aumenta LOC em duas unidades e o segundo em uma. Como cada um desses passos é usado pela refatoração duas vezes (tabela 5.7), então o aumento do valor de LOC será de 6 unidades.

WOC: +3

A variação do valor de WOC é dada pelos impactos de "Criar operação vazia" e "Adicionar parâmetro a operação". Cada um desses passos aumenta o valor desta métrica em uma unidade, e como o primeiro é usado duas vezes (tabela 5.7), então o aumento total é de 3 unidades.

CBC: =

Os dois passos básicos que prevêem uma possível variação no valor de CBC ("Criar operação vazia" e "Adicionar parâmetro a operação") aumentam o valor desta métrica caso o tipo de retorno da operação criada ou o tipo do parâmetro adicionado a ela não este-

jam previamente acoplados ao componente onde é colocada a operação. No caso desta refatoração, esses dois tipos serão o mesmo tipo do atributo cujos métodos de acesso são criados, o qual obviamente já está acoplado ao componente pelo próprio comando de declaração do atributo. Assim, o valor de CBC não sofre alterações ao executar esta refatoração.

LCOO: -X a +X

Os dois métodos adicionados à classe no passo 1 formarão novos pares com as outras operações pré-existentes neste componente. Como o único atributo acessado pelos métodos criados é o atributo que eles encapsulam, então os novos pares de operações diminuirão o valor de LCOO se a maior parte das demais operações também acessar este atributo, caso contrário o valor de LCOO aumentará. Vale ressaltar que o par composto pelos dois métodos de acesso criados é contado na parcela negativa de LCOO porque ambos acessam o mesmo atributo – o encapsulado.

Tabela 5.9: Variação das métricas em *Encapsular Atributo*

Separação de Interesses			Tamanho				Acoplamento		Coesão
CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
=	$\geq_{(X)}$	=	=	$>_{(6)}$	=	$>_{(3)}$	=	=	$\gtrless$

Análise dos Resultados

Analisando os resultados mostrados na tabela 5.9, é possível observar que o tamanho do software sempre irá aumentar, a coesão pode aumentar ou diminuir e a separação dos interesses (por operações) pode ficar maior. Na melhor das hipóteses o aumento do tamanho do software será compensado com uma maior coesão dos componentes, enquanto que o pior caso prevê maior separação de interesses, aumento de tamanho e menor coesão dos componentes do software. Isso mostra que esta refatoração pode ser vantajosa ou não dependendo do caso em que é aplicada.

É importante lembrar, no entanto, que o principal objetivo ao aplicar *Encapsular Atributo* é aumentar a modularidade do componente que contém o campo a ser encapsulado, e esta característica não é um dos atributos de qualidade avaliado pelas métricas usadas neste trabalho (capítulo 2.3.1). Como foi mostrado no exemplo, o encapsulamento do atributo evita que alguns trechos de código tenham que ser replicados em diversas partes do software e sejam concentrados apenas nos métodos de acesso, facilitando assim a manutenção do programa.

5.1.4 Auto Encapsular Atributo

Quando o acesso direto a um atributo da classe está se tornando muito complexo, pode ser mais simples lidar com métodos de acesso e acessar o atributo apenas através desses métodos. A refatoração *Auto Encapsular Atributo* (FOWLER, 1999, p. 171) propõe criar métodos de leitura e escrita do atributo e substituir os acessos diretos a ele por chamadas aos métodos criados. Para fazer essas alterações, devem ser executados os seguintes passos:

1. Criar métodos de gravação e leitura para o atributo;

2. Encontrar todas as referências ao atributo e substituir por um método de leitura ou escrita;

3. Tornar o atributo privado;

4. Compilar e testar.

Esta refatoração é bastante similar a *Encapsular Atributo* (seção 5.1.3), porém se restringe aos acessos diretos feitos dentro do próprio componente. Comparando os passos das duas refatorações, percebe-se que ambas são quase idênticas, diferindo apenas no local onde o acesso direto ao atributo é trocado por uma chamada a um método de gravação ou escrita – nesta refatoração são trocadas apenas as chamadas de dentro do componente que contém o atributo, enquanto que em *Encapsular Atributo* são trocadas apenas as chamadas que estão fora do componente – e em um passo de compilação e teste no meio da refatoração (passo 3 de 5.1.3).

Exemplo:

```
public class Person {
    public String name;
    (...)
    public String toString() {
        StringBuffer result = new StringBuffer("Name:  ");
        result.append(name);
        (...)
        return result.toString();
    }
}
```

```
public class Person {
    private String name;
    (...)
    public String toString() {
        StringBuffer result = new StringBuffer("Name:  ");
        result.append(getName());
        (...)
        return result.toString();
    }
    public String getName() {
        return name;
    }
    public void setName(String newName) {
        name = newName;
    }
}
```

Decomposição da Refatoração em Passos Básicos

Pelas semelhanças entre esta refatoração e *Encapsular Atributo*, a decomposição de seus passos também é bastante parecida. A tabela 5.10 mostra quantas vezes e onde (em que passo) cada passo básico é utilizado por esta refatoração. A diferença entre esta tabela e a tabela correspondente de *Encapsular Atributo* (tabela 5.7) são algumas execuções a menos de "Compilar e testar" e de "Substituir referência direta a atributo por chamada a operação de leitura ou escrita" – em *Encapsular Atributo* este passo básico é executado nos $n_{acessos}$ acessos diretos fora do componente onde está o atributo, enquanto que em *Auto Encapsular Atributo* ele é usado apenas nos $n_{ac.int}$ acessos internos.

Cálculo da Variação Total das Métricas

A tabela 5.11, idêntica à tabela correspondente de *Encapsular Atributo* (tabela 5.8) porque ambas as refatorações usam os mesmos passos básicos, mostra o impacto de cada passo básico usado pela refatoração.

Tabela 5.10: Quantidade de vezes que cada passo básico é usado em *Encapsular Atributo*

Passo da refatoração	Quantidade	Passo básico
1	2	Criar operação vazia
	2	Adicionar linha de leitura/escrita de valor de atributo a operação
	1	Adicionar parâmetro a operação
2	$n_{ac.int}$	Substituir referência direta a atributo por chamada a operação de leitura ou escrita
3	1	Tornar atributo privado
4	1	Compilar e testar

Tabela 5.11: Impacto nas métricas de cada passo básico de *Encapsular Atributo*

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Criar operação vazia	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(2)}$	=	$>_{(1)}$	$\geq_{(1)}$	=	$\geq_{(X)}$
Adicionar linha de leitura/escrita de valor de atributo a operação	=	=	=	=	$>_{(1)}$	=	=	=	=	$\leq_{(X)}$
Adicionar parâmetro a operação	=	=	=	=	=	=	$>_{(1)}$	$\geq_{(1)}$	=	=
Substituir referência direta a atributo por chamada a operação de leitura ou escrita	=	$\geq_{(1)}$	=	=	=	=	=	=	=	$\geq_{(X)}$
Tornar atributo privado	=	=	=	=	=	=	=	=	=	=
Compilar e testar	=	=	=	=	=	=	=	=	=	=

Como esta refatoração e *Encapsular Atributo* usam os mesmos passos básicos e só há diferença no número de vezes em que "Compilar e testar" e "Substituir referência direta a atributo por chamada a operação de leitura ou escrita" são necessários, a variação total das métricas dessas duas refatorações também serão muito semelhantes. Dentre as métricas alteradas por ambas as refatorações, apenas CDO e LCOO têm explicações distintas para esta refatoração.

CDO: 0 a $n_{ac.int} + 2$

A diferença entre a variação de CDO em *Encapsular Atributo* e em *Auto Encapsular Atributo* é a quantidade máxima que CDO pode aumentar: enquanto que no primeiro caso a variação máxima é igual ao número de acessos externos mais dois, nesta refatoração este valor é igual ao número de acessos internos mais dois.

LCOO: $(2 * n_{op}) - 1$ a $n_{op} * (n_{op} + 1) - 1$

Ao substituir todos os acessos diretos ao atributo por chamadas aos métodos de acesso, o número de operações do componente que acessam o atributo encapsulado será reduzido para dois – os próprios métodos de acesso. Se originalmente houver algum par de operações que compartilhasse apenas o atributo encapsulado, então após a refatorações este par deixará de ser contado na parcela negativa de LCOO (operações que compartilham atributos) porque não acessará mais o atributo, e passará a ser contado na parcela positiva (operações que não compartilham atributos), aumentando o valor de LCOO em duas unidades para cada par de operações em que essa situação ocorrer. O número máximo de vezes em que isso pode ocorrer é igual ao número de pares de operações que existiam originalmente no componente (assumindo que todos os pares acessavam o atributo). Supondo n_{op} o número de operações pré-existentes no componente, o número de pares de operações era $\frac{n_{op}!}{2!*(n_{op}-2)!}$, ou seja, $\frac{n_{op}*(n_{op}-1)}{2}$. A variação de LCOO causada pela mudança da classificação dos pares de operações pré-existentes no componente é, portanto, $n_{op} * (n_{op} - 1)$.

A criação dos métodos de acesso também alteram o valor de LCOO. Como os dois métodos acessam o atributo encapsulado, o par formado por essas duas novas operações diminui em uma unidade o valor de LCOO.

Por fim, os novos pares de operações formados entre os métodos de acesso e as n_{op} operações pré-existentes no componente serão todos classificados como não tendo atributos compartilhados, pois os métodos de acesso acessam apenas um atributo – o campo encapsulado – e apenas eles podem acessar este atributo. Assim, a refatoração criará $2 * n_{op}$ pares de operações que não compartilham atributos, aumentando assim mais ainda o valor de LCOO.

Juntando todos os pares de operações existentes após a refatoração, conclui-se que no pior caso (em que todos os pares pré-existentes acessavam o atributo e deixaram de acessá-lo) LCOO terá aumentado $n_{op} * (n_{op} - 1) - 1 + (2 * n_{op})$, ou seja, $n_{op} * (n_{op} + 1) - 1$ unidades, enquanto que no melhor caso (quando nenhum par pré-existente acessa o atributo) terá aumentado apenas $-1 + (2 * n_{op})$. Este melhor caso, no entanto, não é uma situação real para aplicar esta refatoração porque se nenhuma operação acessa o atributo não faz sentido encapsulá-lo porque o acesso direto a ele não está se tornando muito complexo – motivação para aplicar *Auto Encapsular Atributo*.

Tabela 5.12: Variação das métricas em *Auto Encapsular Atributo*

Separação de Interesses			Tamanho				Acoplamento		Coesão
CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
=	$\geq_{(X)}$	=	=	$>_{(6)}$	=	$>_{(3)}$	=	=	$>_{(X)}$

Análise dos Resultados

Analisando os resultados mostrados na tabela 5.12, é possível observar que o tamanho do software e a falta de coesão dos componentes sempre irão aumentar, enquanto que a separação dos interesses (por operações) pode ficar maior ou não se alterar. Isso mostra que esta refatoração sempre piorará alguns dos dos atributos internos do software sem ter como contrapartida uma melhora em outros atributos.

É importante lembrar, no entanto, que o principal objetivo ao aplicar *Auto Encapsular Atributo* é aumentar a modularidade do componente que contém o campo a ser encapsulado, e esta característica não é um dos atributos de qualidade avaliado pelas métricas usadas neste trabalho (capítulo 2.3.1). Assim como ocorre em *Encapsular Atributo*, o encapsulamento do atributo evita que alguns trechos de código tenham que ser replicados em diversas partes do software e sejam concentrados apenas nos métodos de acesso, facilitando assim a manutenção do programa.

5.2 Refatorações Orientadas a Aspectos

5.2.1 Trocar `implements` por `declare parents`

Caso uma classe implemente uma interface relacionada a um interesse transversal secundário e a implementação da interface seja usada apenas quando o interesse estiver presente no sistema, a refatoração *Trocar* `implements` *por* `declare parents` (MONTEIRO, 2005, seção 6.3.9) recomenda tirar essa atribuição de papel da classe e colocá-la no aspecto que implementa o interesse, trocando o "implements" da primeira por "declare parents" no segundo. Para fazer essa troca, são necessários 3 simples passos:

1. Criar o `declare parents` adequado no aspecto;
2. Remover o `implements` da classe;
3. Compilar e testar.

Decomposição da Refatoração em Passos Básicos

Para executar o primeiro passo da refatoração basta usar o passo básico "Adicionar declaração inter-tipo de implementação" (84) para criar uma declaração inter-tipo no aspecto que faça com que a classe implemente a interface.

O 2o. passo é executado pelo passo básico "Remover implementação de interface de componente" (42), enquanto que o último passo é equivalente ao passo básico "Compilar e testar" (101).

A tabela 5.13 mostra quantas vezes e onde (em que passo) cada passo básico é utilizado por esta refatoração.

Tabela 5.13: Quantidade de vezes que cada passo básico é usado em *Trocar* `implements` *por* `declare parents`

Passo da refatoração	Quantidade	Passo básico
1	1	Adicionar declaração inter-tipo de implementação
2	1	Remover implementação de interface de componente
3	1	Compilar e testar

Cálculo da Variação Total das Métricas

Uma vez decompostos os passos da refatoração, deve-se calcular qual o impacto total das alterações nas métricas escolhidas para avaliar a refatoração. A tabela 5.14 mostra o impacto de cada passo básico usado pela refatoração. Nesta tabela é possível observar que as métricas **CDO**, **VS**, **NOA**, **WOC**, **DIT** e **LCOO** não são modificadas por nenhum dos passos, portanto conclui-se que essas métricas não terão seu valor alterado quando *Trocar* `implements` *por* `declare parents` for aplicada. A variação total de cada uma das demais métricas é explicada a seguir.

Tabela 5.14: Impacto nas métricas de cada passo básico de *Trocar* `implements` *por* `declare parents`

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Adicionar declaração inter-tipo de implementação	=	=	=	=	$>_{(1)}$	=	=	$\geq_{(1a2)}$	=	=
Remover implementação de interface de componente	$\leq_{(1)}$	=	$\leq_{(2)}$	=	=	=	=	$\leq_{(1)}$	=	=
Compilar e testar	=	=	=	=	=	=	=	=	=	=

CDC: -1 a 0

Existe apenas um passo básico usado que altera o valor de CDC nesta refatoração: "Remover implementação de interface de componente". Baseado nas explicações mostradas na seção que o descreve (4.1.6), são deduzidas as as possíveis variações que esta métrica terá. Como foi dito anteriormente, a interface está relacionada a um interesse transversal

secundário, por isso antes da refatoração a classe que implementa essa interface está relacionada a este interesse. Se a classe não referenciar um componente principal do interesse em nenhum outro local, então após a refatoração esta classe deixará de estar relacionada ao interesse, o que diminui o número de componentes pelos quais o interesse transversal secundário está disperso. Neste caso, o valor de CDC diminuirá uma unidade.

CDLOC: -2 ou 0

Assim como ocorre com CDC, apenas "Remover implementação de interface de componente" pode alterar o valor de CDLOC nesta refatoração, por isso as justificativas aqui apresentadas são baseadas na seção que descreve esse passo básico.

Se a linha de declaração da classe for sombreada para o interesse transversal secundário apenas por causa da implementação da interface, e se a primeira linha do corpo da classe não for sombreada para esse interesse, então a remoção do `implements` no passo 2 fará com que a linha deixe de estar sombreada e, conseqüentemente, serão eliminados da classe dois pontos de transição.

LOC: +1

O único passo básico que altera o número de linhas de código é "Adicionar declaração inter-tipo de implementação", o qual adiciona uma nova linha ao aspecto com a declaração inter-tipo.

CBC: -1 a +2

Dois componentes são alterados por esta refatoração: o aspecto, onde é criada a declaração inter-tipo; e a classe, da qual é retirada a implementação da interface. No aspecto podem ser criados até dois acoplamentos, caso ele não seja previamente acoplado à classe ou à interface que são usadas na declaração inter-tipo. E na classe o acoplamento deste componente com a interface poderá ser eliminado se não houver nenhuma outra referência à interface no restante da classe. Assim, nos casos extremos, esta refatoração pode criar dois acoplamentos ou remover apenas um.

Tabela 5.15: Variação das métricas em *Trocar* `implements` *por* `declare parents`

Separação de Interesses			Tamanho				Acoplamento		Coesão
CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
$\leq_{(1)}$	=	$\leq_{(2)}$	=	$>_{(1)}$	=	=	$\gtrless_{(-1a+2)}$	=	=

Análise dos Resultados

Analisando os resultados mostrados na tabela 5.15, é possível observar que a separação dos interesses transversais poderá diminuir, o tamanho do software certamente aumentará (apesar de ser apenas um aumento de uma única linha de código) e o acoplamento entre os componentes poderá aumentar ou diminuir. No caso mais favorável, esta refatoração poderá separar melhor os interesses do sistema e reduzir o acoplamento entre eles, mantendo a coesão dos componentes e tendo apenas como custo negativo um pequeno aumento do número de linhas de código. No caso mais desfavorável, não haverá alteração da separação dos interesses nem da coesão dos componentes, mas o tamanho do software e o acoplamento entre componentes aumentarão um pouco. Estes resultados mostram que a refatoração pode ser vantajosa ou não dependendo do caso em que é aplicada.

5.2.2 Extrair Fragmento para Adendo

Durante a extração de um interesse para um aspecto, um método de outro componente pode ter trechos de código relacionados a este interesse que devem ser movidos

para o aspecto. Para fazer esta movimentação, a refatoração *Extrair Fragmento para Adendo* (MONTEIRO, 2005, seção 6.3.3) propõe criar um conjunto de pontos de junção que capture os pontos necessários e mover a porção de código para o adendo apropriado baseado nesse conjunto de pontos de junção. Esta refatoração difere de *Mover Método de Classe para Inter-tipo* (seção 5.2.3) porque não se trata de mover um método inteiro, mas sim apenas parte de um método. Em alguns casos é possível transformar o trecho a ser movido em um método separado (aplicando *Extrair Método* (FOWLER, 1999, p. 110)) e em seguida usar *Mover Método de Classe para Inter-tipo*, mas mesmo nestes casos será necessário criar um adendo que chame o método extraído.

Os passos necessários para efetuar essa refatoração são os seguintes:

1. Criar o conjunto de pontos de junção que capture os pontos desejados. Caso este conjunto já exista, alterá-lo de forma a incluir esses pontos;

2. Garantir que o conjunto de pontos de junção captura o contexto necessário pelo trecho do código. Em particular, verificar se o trecho extraído menciona `this` ou `super`, ou se inclui chamadas a operações do componente onde o trecho está. Nestes casos, uma referência ao objeto que estará executando o código deve ser capturada, conforme o exemplo abaixo, retirado de (MONTEIRO, 2005, p. 95):

```
pointcut stateChanged(TangledStack stack):
    execution(public void TangledStack.push(Object))
    && this(stack);

after(TangledStack _this) returning :  stateChanged(_this) {
    _this.display();
}
```

3. Adicionar as importações necessárias ao aspecto;

4. Criar o adendo para o conjunto de pontos de junção, com o corpo vazio – caso ele já não esteja em construção;

5. Mover os trechos a serem extraídos do método de origem para o corpo do adendo;

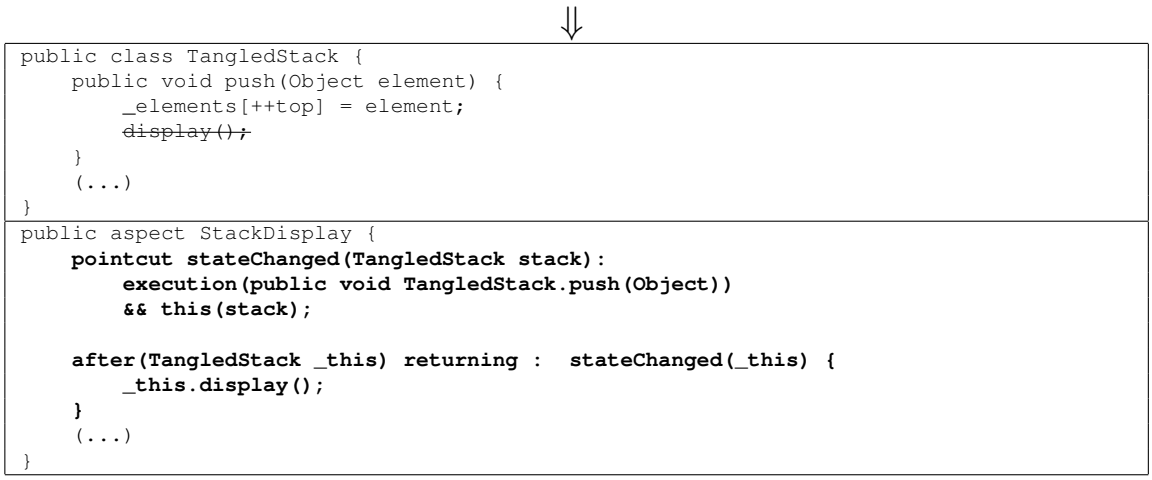
6. Trocar as referências ao `this` pela variável correspondente obtida no passo 2;

7. Verificar que variáveis usadas no código extraído são de escopo local no método de origem. Caso alguma variável temporária seja usada apenas nos trechos extraídos, sua declaração pode ser colocada no corpo do adendo;

Exemplo:

```
public class TangledStack {
    public void push(Object element) {
        _elements[++top] = element;
        display();
    }
    (...)
}
```

```
public aspect SomeAspect {
    (...)
}
```



```
public class TangledStack {
    public void push(Object element) {
        _elements[++top] = element;
        display();
    }
    (...)
}
```

```
public aspect StackDisplay {
    pointcut stateChanged(TangledStack stack):
        execution(public void TangledStack.push(Object))
        && this(stack);

    after(TangledStack _this) returning :  stateChanged(_this) {
        _this.display();
    }
    (...)
}
```

Decomposição da Refatoração em Passos Básicos

O primeiro passo da refatoração cria o conjunto de pontos de junção usando o passo básico "Criar conjunto de pontos de junção" (51). Se este conjunto já existir, os passos básicos usados devem ser "Adicionar ponto de junção a conjunto de pontos de junção" (53) – para capturar os $n_{pts.juncao}$ pontos de junção necessários – e "Adicionar parâmetro a conjunto de pontos de junção" (52) – usado n_{param} vezes caso seja necessário capturar o contexto no passo 2.

O 2o. passo da refatoração descreve como capturar o contexto necessário pelo trecho de código extraído. Se este passo tiver que ser executado, usa-se o passo básico "Capturar contexto de ponto de junção" (56).

O 3o. passo adiciona as importações necessárias ao aspecto, o que é feito usando o passo básico "Adicionar importações" (95).

No 4o. passo pode ser necessário criar um adendo de corpo vazio, o que é feito pelo passo básico "Criar operação vazia" (58). Se o conjunto de pontos de junção criado (ou adaptado) no passo 1 tiver parâmetros, então deve-se usar também o passo básico "Adicionar parâmetro a operação" n_{param} vezes para adicionar cada um dos parâmetros necessários ao adendo.

O passo 5 move os $n_{trechos}$ trechos a serem extraídos do componente de origem para o aspecto, o que é feito usando o passo básico "Mover código entre operações de componentes diferentes" (63) $n_{trechos}$ vezes. Já o passo 6 usa "Trocar referência a `this` por parâmetro de operação" (68) para substituir as ocorrências de `this`.

Por fim, o passo 7 move as variáveis usadas apenas pelo trecho extraído para dentro do adendo. Isto é feito removendo a declaração da variável na operação de origem e adicionando a mesma declaração no adendo de destino. Para cada uma das n_{var} variáveis que devem ser movidas, usa-se "Remover variável local de operação" (62) na operação de origem e "Adicionar variável local a operação" (62) no adendo de destino.

Apesar de não ser descrito explicitamente, ao final da refatoração as mudanças do código devem ser compiladas e testadas para garantir a preservação do comportamento da refatoração, o que é feito por "Compilar e testar" (101).

A tabela 5.16 mostra quantas vezes e onde (em que passo) cada passo básico é utilizado por esta refatoração.

Cálculo da Variação Total das Métricas

Uma vez decompostos os passos da refatoração, deve-se calcular qual o impacto total das alterações nas métricas escolhidas para avaliar a refatoração. A tabela 5.17 mostra

Tabela 5.16: Quantidade de vezes que cada passo básico é usado em *Extrair Fragmento para Adendo*

Passo da refatoração	Quantidade	Passo básico
1	1	Criar conjunto de pontos de junção
	$n_{pts.juncao}$	Adicionar ponto de junção a conjunto de pontos de junção
	n_{param}	Adicionar parâmetro a conjunto de pontos de junção
2	1	Capturar contexto de ponto de junção
3	1	Adicionar importações
4	1	Criar operação vazia
	n_{param}	Adicionar parâmetro a operação
5	$n_{trechos}$	Mover código entre operações de componentes diferentes
6	1	Trocar referência a `this` por parâmetro de operação
7	n_{var}	Remover variável local de operação
	n_{var}	Adicionar variável local de operação
	1	Compilar e testar

o impacto de cada passo básico usado pela refatoração. Nesta tabela é possível observar que as métricas **VS**, **NOA** e **DIT** não são modificadas por nenhum dos passos, portanto conclui-se que essas métricas não terão seu valor alterado quando *Extrair Fragmento para Adendo* for aplicada. A variação total de cada uma das demais métricas é explicada a seguir.

Tabela 5.17: Impacto nas métricas de cada passo básico de *Extrair Fragmento para Adendo*

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Criar conjunto de pontos de junção	=	=	=	=	$>_{(X)}$	=	=	$\geq_{(X)}$	=	=
Adicionar ponto de junção a conjunto de pontos de junção	=	=	=	=	$\gtrless$	=	=	$\geq_{(1)}$	=	=
Adicionar parâmetro a conjunto de pontos de junção	=	=	=	=	=	=	=	$\geq_{(1)}$	=	=
Capturar contexto de ponto de junção	=	=	=	=	$\geq_{(X)}$	=	=	=	=	=
Adicionar importações	=	=	=	=	=	=	=	=	=	=
Criar operação vazia	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(2)}$	=	$>_{(1)}$	$\geq_{(1)}$	=	$\geq_{(X)}$
Adicionar parâmetro a operação	=	=	=	=	=	=	$>_{(1)}$	$\geq_{(1)}$	=	=
Mover código entre operações de componentes diferentes	$\gtrless_{(-1a1)}$	$\gtrless_{(-1a1)}$	$\gtrless_{(-2a2)}$	=	=	=	=	$\gtrless$	=	$\gtrless$
Trocar referência a `this` por parâmetro de operação	=	=	=	=	=	=	=	=	=	=
Remover variável local de operação	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(2)}$	=	$<_{(1)}$	=	=	$\leq_{(1)}$	=	=
Adicionar variável local de operação	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(1)}$	=	=	$\geq_{(1)}$	=	=
Compilar e testar	=	=	=	=	=	=	=	=	=	=

CDC: $-n_{comp}$ a 0

A adição do adendo dentro do aspecto não aumenta o número de componentes pelos quais o interesse transversal está espalhado porque o aspecto está relacionado a este interesse desde a sua criação. Assim, nenhum dos passos básicos usados poderá aumentar o valor de CDC.

No entanto, é possível que a remoção do fragmento e das variáveis locais feitas nos passos 5 e 7 eliminem o relacionamento dos componentes de onde o fragmento foi extraído com o interesse transversal. Se após a extração do fragmento estes n_{comp} componentes deixarem de estar relacionados ao interesse, então o número de componentes pelos quais o interesse está disperso diminui n_{comp} unidades, reduzindo assim o valor de CDC.

CDO: $-n_{op} + 1$ a +1

O adendo criado será uma das operações que implementam o interesse transversal, por isso a sua criação aumenta em uma unidade o número de operações pelas quais o interesse está disperso.

No entanto, se a remoção do fragmento de suas n_{op} operações de origem fizer com que estas operações deixem de estar relacionadas ao interesse, então a extração do fragmento também terá diminuído o número de operações pelas quais o interesse está disperso.

CDLOC: $-2 * (n_{trechos} + n_{var})$ a 0

Como as linhas extraídas para o adendo são todas relacionadas ao interesse transversal do aspecto, sua adição ao adendo não cria nenhum ponto de transição e, portanto, não altera o valor de CDLOC.

Já nos componentes onde os $n_{trechos}$ trechos e n_{var} variáveis extraídos estavam originalmente, poderão ser eliminados alguns pontos de transição. Supondo o máximo entrelaçamento de código, em que cada trecho e cada declaração de variável cria 2 pontos de transição à operação que os contém. Neste caso, a extração de cada trecho e declaração de variável para dentro do aspecto eliminará os 2 pontos de transição, totalizando uma remoção de $2 * (n_{trechos} + n_{var})$ pontos.

LOC: -X a +X

Tanto o adendo quanto o conjunto de pontos de junção usados para extrair os trechos de código para o aspecto podem ter de ser criados ou reaproveitados. Se for necessário criá-los, certamente o número de linhas de código do software aumentará – a quantidade desse aumento depende do trecho extraído e não pode ser definida de forma geral. No entanto, se o conjunto de pontos de junção puder ser reaproveitado, a variação do número de linhas de código poderá ser tanto positiva quanto negativa, conforme explicado na descrição do passo básico "Adicionar ponto de junção a conjunto de pontos de junção" (seção 4.2.5). Portanto, não é possível estabelecer de antemão o quanto o valor de LOC variará ao usar esta refatoração porque, dependendo das características do código, essa variação poderá ter valores muito diferentes.

WOC: 0 a $n_{param} + 1$

Se for necessário criar o adendo ou adicionar parâmetros a ele, o valor de WOC aumentará, caso contrário esta métrica não se altera. Como um novo adendo vazio aumenta o valor de WOC em uma unidade e a adição de cada parâmetro a ele faz o mesmo, então o aumento máximo desta métrica será $n_{param} + 1$.

CBC: -X a +X

A variação do número de acoplamentos poderá ocorrer tanto no aspecto quanto nos componentes onde os trechos extraídos estavam originalmente. No aspecto serão criados acoplamentos aos componentes afetados pelo conjunto de pontos de junção, aos componentes usados nos trechos extraídos e aos parâmetros do conjunto de pontos de junção e do adendo – caso esses acoplamentos não existam originalmente no aspecto. Já nos componentes onde os trechos estavam originalmente, serão removidos acoplamentos aos componentes presentes nos trechos e nas declarações das variáveis locais extraídos – se as linhas extraídas forem os únicos locais em que esses componentes são referenciados.

Não é possível definir, no entanto, quantos acoplamentos serão criados e quantos serão

removidos porque esse número dependerá dos acoplamentos pré-existentes nos componentes afetados pela refatoração, ficando assim impossível estabelecer o quanto CBC variará.

LCOO: -X a +X

O valor de LCOO pode ser alterado por duas frentes ao aplicar esta refatoração:

Alteração do número de operações Se for necessário criar o adendo no aspecto (passo 4), novos pares serão formados entre este adendo e as operações pré-existentes no aspecto. Esses novos pares poderão compartilhar ou não atributos, alterando positiva ou negativamente o valor de LCOO. Vale ressaltar que esta alteração pode não ocorrer, caso o adendo já exista;

Alteração do corpo das operações Com a movimentação dos trechos de código, tanto a operação de origem quanto o adendo de destino terão seu corpo alterado, porém com conseqüências opostas sobre o valor de LCOO: na operação de origem, atributos podem deixar de ser acessados (se não houver nenhum outro trecho na operação em que estes atributos sejam usados), aumentando a parcela de pares de operações que não acessam atributos em comum. Conseqüentemente, o valor de LCOO diminuirá; no caso do adendo, atributos podem passar a ser acessados (caso estes atributos não fossem acessados no adendo antes da refatoração), aumentando a parcela de pares de operações com ao menos um atributo compartilhado. Como resultado, o valor de LCOO aumentará.

Em todos os casos em que o valor de LCOO pode ser alterado, não é possível saber se a variação será positiva ou negativa, pois dependerá das operações pré-existentes nos componentes e dos trechos a serem extraídos.

Tabela 5.18: Variação das métricas em *Extrair Fragmento para Adendo*

Separação de Interesses			Tamanho				Acoplamento		Coesão
CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
$\leq_{(X)}$	$\gtrless_{(-Xa+1)}$	$\leq_{(X)}$	=	$\gtrless$	=	$\geq_{(X)}$	$\gtrless$	=	$\gtrless$

Análise dos Resultados

Analisando os resultados mostrados na tabela 5.18, é possível observar que esta refatoração permite uma grande diminuição da separação dos interesses transversais, enquanto que os outros atributos internos (tamanho, acoplamento e coesão) podem aumentar ou diminuir, dependendo de condições pré-existentes nos componentes envolvidos nesta refatoração. No caso mais favorável, todos os atributos internos pode melhorar, enquanto que no pior caso todos podem piorar – até mesmo a separação dos interesses, pois na situação mais adversa CDC e CDLOC não seriam alterados e CDO aumentaria em uma unidade.

Isso mostra que a refatoração trará benefícios ao software apenas em alguns casos, especialmente nas situações em que se possa diminuir a separação dos interesses transversais. Esta conclusão apenas confirma a intenção desta refatoração, que é reunir trechos relacionados a um interesse transversal, e que estão espalhados pelo código do programa. É importante notar, no entanto, que a melhoria desse atributo interno pode ter como conseqüência a piora dos demais atributos.

5.2.3 Mover Método de Classe para Inter-tipo

Quando um método pertence a um interesse transversal diferente do interesse principal da classe que o contém, é recomendado mover esse método para o aspecto que encapsula o interesse secundário, transformando-o em uma declaração intertipo. A refatoração *Mover Método de Classe para Inter-tipo* (MONTEIRO, 2005, seção 6.3.8) apresenta os passos para efetuar essa movimentação, desde que as seguintes pré-condições sejam satisfeitas:

- exista apenas uma implementação da assinatura do método através da cadeia de herança;
- o método usa apenas seus parâmetros, membros públicos, variáveis locais e membros já movidos da classe para o aspecto.

Caso alguma dessas pré-condições não seja satisfeita, em (MONTEIRO, 2005) são descritas soluções alternativas que usam outras refatorações antes ou mesmo no lugar desta refatoração.

Os 5 passos que descrevem a movimentação do método da classe para o aspecto são os seguintes:

1. Copiar a definição do método para o aspecto, adicionando o nome da classe e '.' antes do nome do método;
2. Caso o método não seja público, talvez seja necessário alterar (temporariamente [2]) seu tipo de acesso para protegido – caso a classe esteja no mesmo pacote do aspecto – ou público;
3. Verificar que conjuntos de pontos de junção serão alterados com a mudança de lugar do método. Esses conjuntos são caracterizados por usarem `within` em sua definição, bem como obviamente capturarem o método a ser movido. Trocar a classe de origem dentro de `within` pelo aspecto de destino;
4. Remover o método da classe;
5. Compilar e testar.

Decomposição da Refatoração em Passos Básicos

O primeiro passo da refatoração usa uma vez o passo básico "Adicionar declaração inter-tipo de operação" (87) para copiar a definição do método dentro do aspecto, n_{param} vezes o passo básico "Adicionar parâmetro a operação" (60) para criar os parâmetros necessários à declaração inter-tipo, n_{var} vezes "Adicionar variável local a operação" (62) para incluir as variáveis usadas e, por fim, usa uma vez "Adicionar linhas de código a operação" (76) para preencher o corpo da declaração inter-tipo

O passo 2 relaxa o acesso do método usando o passo básico "Tornar operação pública" (90) ou "Tornar operação protegida" (91).

No passo 3, para cada um dos $n_{conj.pts}$ conjuntos de pontos de junção alterados com a mudança de lugar do método, deve ser usado o passo básico "Alterar referência a componente em `within` de conjunto de pontos de junção" (57) para trocar a classe de origem pelo aspecto de destino.

[2] A refatoração *Introduce Aspect Protection* (MONTEIRO, 2005, p. 116) deve ser executada após esta refatoração para tornar o método não-público novamente.

A remoção do método de dentro da classe é feita pelos seguintes passos básicos: “Remover linhas de código de operação” (78) para eliminar os comandos do método; “Remover variável local de operação” (62) para apagar cada uma das n_{var} variáveis do método; “Remover parâmetro de operação” (61) para eliminar os n_{param} parâmetros usados no método; e, finalmente, “Remover operação vazia” (59) para retirar o método da classe.

Por fim, o passo 5 executa “Compilar e testar” (101) para finalizar a refatoração.

A tabela 5.19 mostra quantas vezes e onde (em que passo) cada passo básico é utilizado por esta refatoração.

Tabela 5.19: Quantidade de vezes que cada passo básico é usado em *Mover Método de Classe para Inter-tipo*

Passo da refatoração	Quantidade	Passo básico
1	1	Adicionar declaração inter-tipo de operação
	n_{param}	Adicionar parâmetro a operação
	n_{var}	Adicionar variável local a operação
	1	Adicionar linhas de código a operação
2	1	Tornar operação pública
	1	Tornar operação protegida
3	$n_{conj.pts}$	Alterar referência a componente em `within` de conjunto de pontos de junção
4	1	Remover linhas de código de operação
	n_{var}	Remover variável local de operação
	n_{param}	Remover parâmetro de operação
	1	Remover operação vazia
5	1	Compilar e testar

Cálculo da Variação Total das Métricas

Uma vez decompostos os passos da refatoração, deve-se calcular qual o impacto total das alterações nas métricas escolhidas para avaliar a refatoração. A tabela 5.20 mostra o impacto de cada passo básico usado pela refatoração. Nesta tabela é possível observar que as métricas **VS**, **NOA** e **DIT** não são modificadas por nenhum dos passos, portanto conclui-se que essas métricas não terão seu valor alterado quando *Mover Método de Classe para Inter-tipo* for aplicada. A variação total de cada uma das demais métricas é explicada a seguir.

CDC: -1 a 0

Pela definição da refatoração, sabe-se que “o método pertence a um interesse transversal diferente do interesse principal da classe que o contém”. Por isso, se a classe só referenciar o interesse do método na declaração do próprio método – isto é, se não houver na classe uma referência ao interesse em nenhuma outra parte de seu corpo –, então a remoção do método no passo 4 fará com que a classe deixe de estar relacionada ao interesse e, portanto, o número de componentes pelos quais o interesse transversal está disperso diminuirá uma unidade.

CDO: =

O aumento de CDO que ocorre ao criar a declaração intertipo no aspecto é anulado quando o método é removido da classe de origem, pois ambas as operações servem para imple-

Tabela 5.20: Impacto nas métricas de cada passo básico de *Mover Método de Classe para Inter-tipo*

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Adicionar declaração inter-tipo de operação	=	$>_{(1)}$	=	=	$>_{(2)}$	=	$>_{(1)}$	$\geq_{(1)}$	=	$\geq_{(X)}$
Adicionar parâmetro a operação	=	=	=	=	=	=	$>_{(1)}$	$\geq_{(1)}$	=	=
Adicionar variável local a operação	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(1)}$	=	=	$\geq_{(1)}$	=	=
Adicionar linhas de código a operação	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(X)}$	=	$>_{(X)}$	=	=	$\geq_{(X)}$	=	$\leq_{(X)}$
Tornar operação pública	=	=	=	=	=	=	=	=	=	=
Tornar operação protegida	=	=	=	=	=	=	=	=	=	=
Alterar referência a componente em `within` de conjunto de pontos de junção	=	=	=	=	=	=	=	$\gtrless_{(-1a1)}$	=	=
Remover linhas de código de operação	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(X)}$	=	$<_{(X)}$	=	=	$\leq_{(X)}$	=	$\geq_{(X)}$
Remover variável local de operação	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(2)}$	=	$<_{(1)}$	=	=	$\leq_{(1)}$	=	=
Remover parâmetro de operação	=	=	=	=	=	=	$<_{(1)}$	$\leq_{(1)}$	=	=
Remover operação vazia	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(2)}$	=	$<_{(2)}$	=	$<_{(1)}$	$\leq_{(1)}$	=	$\leq_{(X)}$
Compilar e testar	=	=	=	=	=	=	=	=	=	=

mentar o mesmo interesse transversal e uma é substituída pela outra. Assim, o valor de CDO não é modificado por esta refatoração.

CDLOC: -2 a 0

A adição da declaração inter-tipo no aspecto no passo 1 não cria ou remove pontos de transição porque o aspecto já é todo sombreado para o interesse transversal da declaração inter-tipo. Já a remoção do método original no passo 4 pode eliminar pontos de transição porque o interesse principal da classe é diferente do interesse do método, e portanto a classe e o método têm sombreamentos diferentes. Caso não haja nenhuma linha sombreada adjacente ao método (imediatamente antes ou depois de sua definição), o número de pontos de transição removidos por esta refatoração será dois – um entre a classe e o início do método, e um entre o fim dessa operação e a classe.

LOC: =

Como a declaração inter-tipo é uma cópia do método original, o número de linhas dessas duas operações é o mesmo. Dessa forma, todas as linhas removidas da classe original serão adicionadas no aspecto, o que não altera ao final da refatoração o número de linhas de código que o software possui.

WOC: =

A complexidade da declaração inter-tipo e a do método original serão iguais, uma vez que ambas as operações possuem o mesmo número de parâmetros. Dessa forma, a parcela removida do WOC da classe será adicionada ao WOC do aspecto, o que não modifica o valor total de WOC do sistema.

CBC: $-n_{acopl}$ a $+n_{acopl}$

Os comandos do método a ser transformado em declaração inter-tipo possuem acoplamentos a diversos componentes do sistema. Supondo que esses comandos possuam n_{acopl} acoplamentos, então a diminuição máxima de CBC da classe será n_{acopl} e o aumento máximo do CBC do aspecto também será n_{acopl}. No entanto, é possível que os componentes usados nos comandos do método sejam referenciados previamente em outras partes da classe e do aspecto. Neste caso, a variação total do CBC desses componentes será menor que n_{acopl} porque esta métrica conta o acoplamento a um mesmo componente apenas uma vez.

O exemplo abaixo, apresentado em (MONTEIRO; FERNANDES, 2005b), mostra

uma situação na qual o método transformado em declaração inter-tipo (`opening()`) possui acoplamento a dois componentes: seu tipo de retorno (`Observable`) e o componente no qual ele é inserido através da declaração intertipo (`Flower`). No entanto, é criado apenas um acoplamento no aspecto `ObservingOpen` porque este já era previamente acoplado a `Flower` por outra declaração inter-tipo:

```
public aspect ObservingOpen {
   private OpenNotifier Flower.oNotify = new OpenNotifier(this);
   (...)
}
```

```
public aspect ObservingOpen {
   private OpenNotifier Flower.oNotify = new OpenNotifier(this);
   public Observable Flower.opening() {
       return oNotify;
   }
   (...)
}
```

Assim, a variação total do valor de CBC estará entre os extremos $-n_{acopl}$ (todos os acoplamentos removidos da classe e nenhum acoplamento criado no aspecto) e $+n_{acopl}$ (nenhum acoplamento removido da classe e todos os acoplamentos criados no aspecto).

LCOO: -X a +X

A remoção do método da classe de origem eliminará os pares de operações formados por esse método com as demais operações da classe. A variação do valor de LCOO da classe dependerá da classificação dos pares eliminados: se houver maior número de pares que compartilham atributos, LCOO aumentará, caso contrário, diminuirá. Da mesma forma, a adição de uma nova operação no aspecto (a declaração inter-tipo) criará novos pares de operações com as demais operações deste componente, os quais poderão compartilhar atributos ou não. A variação de LCOO do aspecto dependerá da classificação dos pares de operações criados: se forem adicionados mais pares de operações com atributos em comum, o valor de LCOO diminuirá, caso contrário, aumentará.

Portanto, a variação total de LCOO dependerá dos atributos acessados pelo método movido e das demais operações existentes na classe e no aspecto envolvidos nesta refatoração. É possível que o valor desta métrica aumente ou diminua, dependendo do código em que a refatoração for aplicada.

Tabela 5.21: Variação das métricas em *Mover Método de Classe para Inter-tipo*

Separação de Interesses			Tamanho				Acoplamento		Coesão
CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
$\leq_{(1)}$	=	$\leq_{(2)}$	=	=	=	=	$\gtrless$	=	$\gtrless$

Análise dos Resultados

Analisando os resultados mostrados na tabela 5.21, é possível observar que esta refatoração certamente diminuirá a separação dos interesses sem modificar o tamanho do software. Já a coesão e o acoplamento dos componentes podem variar positiva ou negativamente dependendo das características do código em que a refatoração é aplicada. No melhor caso, esta refatoração consegue melhorar a separação de interesses, o acoplamento e a coesão dos componentes, enquanto que no pior caso ocorre uma piora apenas nestes dois últimos atributos internos do software. Isso mostra que esta refatoração consegue de fato diminuir a separação dos interesses transversais, mesmo que para isso o acoplamento e a coesão dos componentes tenham que piorar.

5.2.4 Mover Atributo de Classe para Inter-tipo

Quando um atributo está relacionado a um interesse transversal diferente do interesse principal da classe que o contém, deve-se movê-lo da classe para o aspecto que implementa o interesse secundário através de uma declaração intertipo. A refatoração *Mover Atributo de Classe para Inter-tipo* (MONTEIRO, 2005, seção 6.3.7) apresenta 8 passos para efetuar essa movimentação:

1. Copiar a declaração do atributo da classe para o aspecto, incluindo a atribuição do valor inicial – se existir. Adicionar o nome da classe de origem e '.' antes do nome do atributo na declaração intertipo;

2. Caso o atributo seja privado, relaxar seu tipo de acesso. Se o aspecto e a classe estiverem no mesmo pacote, ele pode ser protegido, caso contrário ele deve ser público;

3. Remover a declaração do atributo na classe;

4. Verificar conjuntos de pontos de junção que usam `within` e se referem ao atributo movido, e alterá-los para que se adeqüem à mudança de localização do atributo (trocar a classe de origem dentro de `within` pelo aspecto);

5. Compilar e testar;

6. Criar um `declare warning` para expor todas as ocorrências do atributo movido no código, conforme o exemplo abaixo:

```
public aspect ThisAspect {
    declare warning:
        (get (AttributeType SomeClass.attribute) ||
         set (AttributeType SomeClass.attribute))
        && !within(SomeAspect):
        "Field attribute is accessed outside ThisAspect.";
    (...)
}
```

7. Para cada fragmento de código que usar o atributo, decidir se o método todo ou apenas parte dele deve ser movida para o aspecto: no primeiro caso, usar *Move Method from Class to Inter-type*; no segundo, aplicar *Extract Fragment into Advice*. Se um parâmetro de uma operação estiver relacionado ao atributo movido, usar *Extrair Método* para isolar a parte que usa o parâmetro, mover o método extraído com *Mover Método de Classe para Inter-tipo*, mover a chamada para um ponto mais adequado e então usar *Remover Parâmetro*. Compilar e testar após cada refatoração;

8. Assim que o último acesso ao atributo fora do escopo desejado for removido, remover o `declare warning` criado no passo 6 e restringir novamente o acesso do atributo. Normalmente isto quer dizer torná-lo privado, mas se por algum motivo for necessário manter na classe trechos de código relacionados ao atributo, usar *Encapsular Atributo*. Caso o atributo fosse originalmente protegido e subclasses da classe-alvo precisarem acessá-lo, criar um `declare error` conforme exemplo abaixo:

```
public aspect ThisAspect {
    declare error:
        (get (AttributeType SomeClass.attribute) ||
         set (AttributeType SomeClass.attribute))
        && !within(SomeAspect)
        && !within(SomeClass+):
        "Field attribute is accessed outside SomeClass chain and ThisAspect.";
    (...)
}
```

Decomposição da Refatoração em Passos Básicos

Para executar o primeiro passo da refatoração (declarar o atributo no aspecto), usa-se o passo básico "Adicionar declaração inter-tipo de atributo" (86).

O 2o. passo é feito usando o passo básico "Tornar atributo público" (92) ou "Tornar atributo protegido" (92), enquanto que o 3o. passo usa "Remover atributo de componente" (83) para eliminar o atributo da classe.

No 4o. passo, para cada um dos $n_{conj.pts}$ conjuntos de pontos de junção que tiver que ser alterado, será usado uma vez o passo básico "Alterar referência a componente em `within` de conjunto de pontos de junção" (57).

Para compilar e testar o software no passo 5, usa-se "Compilar e testar" (101). Já para criar o `declare warning` no passo 6 é usado o passo básico "Criar `declare warning` de sinalização de acesso a atributo" (93).

O passo 7, que move para o aspecto os trechos que acessam o atributo, possui três estratégias diferentes, de acordo com o trecho a ser movido. Para mover um método inteiro, é usada a refatoração *Mover Método de Classe para Inter-tipo* (130) uma vez para cada um dos $n_{metodos}$ métodos que devem ser movidos; para mover apenas um trecho, usa-se *Extrair Fragmento para Adendo* (124) uma vez para cada um dos $n_{trechos}$ trechos que devem ser movidos; e caso algum parâmetro esteja relacionado ao atributo, extrai-se um método para isolar a parte que usa o parâmetro usando *Extrair Método* (104), move-se esse método usando *Mover Método de Classe para Inter-tipo* (130), troca-se a chamada do método para outro local usando o passo básico "Mover código entre operações do mesmo componente" (67) e remove-se o parâmetro usando *Remover Parâmetro* (110). Para cada um dos n_{param} parâmetros relacionados ao atributo, executa-se uma vez cada uma dessas etapas. No fim, é usado também o passo básico "Compilar e testar" (101) para testar as alterações.

No último passo da refatoração o `declare warning` é removido pelo passo básico "Remover `declare warning` de sinalização de acesso a atributo" (94) e o acesso ao atributo é novamente restrito por "Tornar atributo privado" (92). Talvez seja necessário executar *Encapsular Atributo* (115) ou criar um `declare error` usando o passo básico "Criar `declare error` de proteção de acesso a atributo" (94) para aumentar a abrangência do acesso ao atributo.

Apesar de não ser descrito explicitamente, ao final da refatoração as mudanças do código devem ser compiladas e testadas para garantir a preservação do comportamento da refatoração, o que é feito por "Compilar e testar" (101).

A tabela 5.22 mostra quantas vezes e onde (em que passo) cada passo básico é utilizado por esta refatoração.

Cálculo da Variação Total das Métricas

Uma vez decompostos os passos da refatoração, deve-se calcular qual o impacto total das alterações nas métricas escolhidas para avaliar a refatoração. A tabela 5.23 mostra o

Tabela 5.22: Quantidade de vezes que cada passo básico é usado em *Mover Atributo de Classe para Inter-tipo*

Passo da refatoração	Quantidade	Passo básico
1	1	Adicionar declaração inter-tipo de atributo
2	1	Tornar atributo público
	1	Tornar atributo protegido
3	1	Remover atributo de componente
4	$n_{conj.pts}$	Alterar referência a componente em `within` de conjunto de pontos de junção
5	1	Compilar e testar
6	1	Criar `declare warning` de sinalização de acesso a atributo
7	$n_{metodos}$	*Mover Método de Classe para Inter-tipo*
	$n_{trechos}$	*Extrair Fragmento para Adendo*
	n_{param}	*Extrair Método*
	n_{param}	*Mover Método de Classe para Inter-tipo*
	n_{param}	Mover código entre operações do mesmo componente
	n_{param}	*Remover Parâmetro*
	1	Compilar e testar
8	1	Remover `declare warning` de sinalização de acesso a atributo
	1	Tornar atributo privado
	1	*Encapsular Atributo*
	1	Criar `declare error` de proteção de acesso a atributo
	1	Compilar e testar

impacto de cada passo básico usado pela refatoração. Nesta tabela é possível observar que as métricas **VS** e **DIT** não são modificadas por nenhum dos passos, portanto conclui-se que essas métricas não terão seu valor alterado quando *Mover Atributo de Classe para Inter-tipo* for aplicada. A variação total de cada uma das demais métricas é explicada a seguir.

CDC: -1 a 0

Todos os passos básicos que podem modificar o valor de CDC tratam da movimentação para o aspecto do atributo ou dos trechos que o acessam. Se a classe de origem do atributo (e dos trechos) for relacionada ao interesse transversal do aspecto apenas pela declaração do atributo e pelos acessos a eles, então após a refatoração a classe não será mais relacionada ao interesse porque todos os comandos com estes relacionamentos terão sido movidos para o aspecto. Dessa forma, o interesse transversal passará a estar disperso por um componente a menos do que antes da refatoração, logo o valor de CDC diminuirá uma unidade.

CDO: $(-n_{op.modificadas})$ a $(n_{trechos} + n_{metodos} + 2)$

A dispersão do interesse transversal do aspecto por operações do sistema pode mudar de diferentes formas ao usar esta refatoração. Em primeiro lugar, as $n_{op.modificadas}$ operações da classe nas quais o atributo é acessado podem deixar de estar relacionadas ao interesse caso todos os comandos que as relacionam a esse interesse sejam removidos no passo 7.

Tabela 5.23: Impacto nas métricas de cada passo básico de *Mover Atributo de Classe para Inter-tipo*

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Adicionar declaração inter-tipo de atributo	=	=	=	=	$>_{(1)}$	$>_{(1)}$	=	$\geq_{(1a2)}$	=	=
Tornar atributo público	=	=	=	=	=	=	=	=	=	=
Tornar atributo protegido	=	=	=	=	=	=	=	=	=	=
Remover atributo de componente	$\leq_{(1)}$	=	$\leq_{(2)}$	=	$<_{(1)}$	$<_{(1)}$	=	$\leq_{(1)}$	=	=
Alterar referência a componente em `within` de conjunto de pontos de junção	=	=	=	=	=	=	=	$\gtrless_{(-1a1)}$	=	=
Compilar e testar	=	=	=	=	=	=	=	=	=	=
Criar `declare warning` de sinalização de acesso a atributo	=	=	=	=	$>_{(5)}$	=	=	=	=	=
Mover Método de Classe para Inter-tipo	$\leq_{(1)}$	=	$\leq_{(2)}$	=	=	=	=	$\gtrless$	=	$\gtrless$
Extrair Fragmento para Adendo	$\leq_{(X)}$	$\gtrless_{(-Xa+1)}$	$\leq_{(X)}$	=	$\gtrless$	=	$\geq_{(X)}$	$\gtrless$	=	$\gtrless$
Extrair Método	=	$\geq_{(1)}$	$\leq_{(X)}$	=	$>_{(3a4)}$	=	$>_{(X)}$	=	=	$\gtrless$
Mover código entre operações do mesmo componente	=	$\gtrless_{(-1a1)}$	$\gtrless_{(-2a2)}$	=	=	=	=	=	=	$\gtrless$
Remover Parâmetro	$\leq_{(X)}$	$\gtrless_{(-Xa+1)}$	$\leq_{(X)}$	=	$\gtrless_{(-Xa+3)}$	=	$\gtrless_{(-1a+X)}$	$\leq_{(X)}$	=	$\geq_{(X)}$
Remover `declare warning` de sinalização de acesso a atributo	=	=	=	=	$<_{(5)}$	=	=	=	=	=
Tornar atributo privado	=	=	=	=	=	=	=	=	=	=
Encapsular Atributo	=	$\geq_{(X)}$	=	=	$>_{(6)}$	=	$>_{(3)}$	=	=	$\gtrless$
Criar `declare error` de proteção de acesso a atributo	=	=	=	=	$>_{(6)}$	=	=	=	=	=

Ao mesmo tempo, a refatoração cria novas operações no aspecto nas quais o interesse transversal estará disperso: os $n_{trechos}$ adendos extraídos por *Extrair Fragmento para Adendo*, os $n_{metodos}$ métodos gerados por *Extrair Método* e movidos para o aspecto por meio de *Mover Método de Classe para Inter-tipo*, e os dois métodos de acesso criados por *Encapsular Atributo*. Por fim, ao "mover a chamada para um ponto mais adequado" no passo 7, as $n_{metodos}$ operações que recebem a chamada ao método gerado se tornarão relacionadas ao interesse (caso não sejam previamente) porque a operação chamada é uma das operações principais do interesse.

Existe ainda outro passo básico que poderia alterar o valor de CDO, mas que nesta refatoração não o faz: a chamada de *Remover Parâmetro*. Ela prevê a diminuição de CDO caso não seja mais necessário obter o valor do parâmetro através de chamadas a operações principais do interesse, porém este valor continuará a ser necessário em outra operação – no método onde a operação cujo parâmetro foi removido é chamada – para ser passado ao método extraído por *Extrair Método* e movido para o aspecto. Por isso, não há

remoção de comandos que diminua o valor de CDO. Este valor também não aumenta uma unidade porque, mesmo que o método onde está o parâmetro a ser removido faça parte da interface publicada da classe e não possa ser removido, após a refatoração apenas o método sem o parâmetro estará relacionado ao interesse transversal – o método original, com mais parâmetros, terá apenas uma chamada ao outro método, não usando portanto nenhuma operação principal do interesse.

Assim, o número de operações pelas quais o interesse transversal do aspecto está disperso poderá diminuir até $n_{op.modificadas}$ e aumentar até $n_{trechos} + 2 * n_{metodos} + 2$ unidades.

CDLOC: -X a $n_{metodos}$

A linha de declaração do atributo é uma linha sombreada para o interesse transversal do aspecto, por isso sua remoção da classe no passo 3 eliminará dois pontos de transição caso essa linha não esteja em um bloco maior de linhas sombreadas. As linhas que acessam o atributo também são sombreadas para este mesmo interesse, por isso sua eliminação do código da classe também pode remover pontos de transição. O número dos pontos removidos depende do entrelaçamento das linhas sombreadas ao código não-sombreado da classe – diversos pontos de transição poderão ser removidos em uma só execução de *Extrair Fragmento para Adendo* ou *Extrair Método*.

No entanto, também é possível que sejam criados pontos de transição na classe ao mover as chamadas aos métodos extraídos por *Extrair Método* para outras operações. Se essas chamadas forem colocadas logo abaixo ou logo acima de outras linhas sombreadas, não há a criação de pontos de transição, caso contrário dois novos pontos serão criados para cada uma das $n_{metodos}$ chamadas movidas.

Por fim, *Remover Parâmetro* também poderia alterar o valor de CDLOC caso fosse possível remover os comandos usados apenas para obter o valor a ser passado à operação cujo parâmetro foi removido, porém estes comandos continuarão a ser usados após a refatoração para fornecer valores ao método extraído por *Extrair Método* e, portanto, não poderão ser removidos.

É importante notar que a adição no aspecto do atributo e dos trechos de código que acessam esse atributo não criam novos pontos de transição porque o aspecto é todo sombreado para o interesse transversal que ele implementa, por isso apenas as modificações na classe poderão resultar em uma modificação do valor de CDLOC.

Assim, o número de pontos de transição da classe poderá aumentar até $n_{metodos}$ unidades ou diminuir um número variável de unidades. O quanto esta refatoração diminui o valor de CDLOC não pode ser definido porque depende do sombreamento original das linhas adjacentes às linhas movidas para o aspecto – se houver linhas sombreadas logo acima ou logo abaixo das linhas movidas, sua remoção da classe para posterior adição ao aspecto não remove nenhum ponto de transição – e do entrelaçamento do código que acessa o atributo movido ao código não-sombreado da classe.

LOC: -X a +X

Esta refatoração possui diversas etapas em que o número de linhas de código aumenta ou diminui. Algumas delas se anulam, como a adição do atributo no aspecto e a remoção do atributo na classe, ou a criação do `declare warning` no passo 6 e a remoção desse mesmo `declare warning` no passo 8.

Dentre as demais etapas, o uso de *Encapsular Atributo*, *Extrair Método* e "Criar `declare error` de proteção de acesso a atributo" sempre aumentam o valor de LOC – respectivamente em até seis, $4 * n_{param}$ e seis unidades. Já *Extrair Fragmento para Adendo* e *Remover Parâmetro* podem aumentar ou diminuir o valor de LOC dependendo

dos fragmentos extraídos e das linhas que puderem ser removidas ao eliminar o parâmetro. Por causa dessas últimas duas refatorações usadas, o valor de LOC poderá aumentar ou diminuir uma quantidade variável de linhas, o que não permite estabelecer limites para a variação do valor desta métrica ao aplicar a refatoração.

NOA: =

O número de atributos existentes no software após esta refatoração é o mesmo de antes, pois o atributo criado no aspecto é removido da classe alguns passos depois. Assim, o valor de NOA não se altera com o uso da refatoração.

WOC: 0 a +X

Esta refatoração pode criar algumas operações no aspecto com os trechos que acessam o atributo movido – os adendos extraídos por *Extrair Fragmento para Adendo*, e os métodos extraídos por *Extrair Método* e movidos para o aspecto por meio de *Mover Método de Classe para Inter-tipo*. Essas novas operações terão um número variável de parâmetros e, conseqüentemente, suas complexidades serão variáveis. Além dessas operações, também podem ser criados métodos de acesso ao atributo pela refatoração *Encapsular Atributo*, o que aumenta o valor de WOC em 3 unidades.

No entanto, o uso de *Remover Parâmetro* para retirar o parâmetro relacionado ao atributo pode diminuir o valor de WOC uma unidade – caso o método não faça parte da interface publicada da classe e possa ser removido – ou aumentar este valor tantas unidades quanto forem os parâmetros da operação de onde o parâmetro deveria ter sido removido – já tendo descartado o parâmetro em questão –, como é explicado na avaliação de *Remover Parâmetro* na seção 5.1.2. É importante notar que para poder remover o parâmetro de uma operação deverá ter sido executada primeiro a refatoração *Extrair Método*, por isso a possível diminuição do valor de WOC pela remoção do parâmetro será sempre anulada pelo aumento de WOC via extração de método. Assim, não existe situação em que esta refatoração poderá diminuir o valor de WOC.

CBC: $-n_{acop.cl}$ a $n_{acop.asp}$

Esta refatoração pode remover acoplamentos da classe e adicionar acoplamentos ao aspecto por causa da movimentação, da classe para o aspecto, do atributo e dos trechos que o acessam. Essas linhas de código movidas fazem referência a diferentes componentes do sistema, componentes esses que poderiam ser acoplados à classe em outras partes do código ou serem referenciados no aspecto mesmo antes da refatoração.

Assim, sejam $n_{acop.cl}$ o número de acoplamentos removidos da classe e $n_{acop.asp}$ o número de acoplamentos criados no aspecto por essa refatoração. A variação de CBC é dada por $n_{acop.asp} - n_{acop.cl}$. Esta métrica irá diminuir caso sejam removidos mais acoplamentos da classe do que criados no aspecto, porém aumentará se o inverso ocorrer.

LCOO: -X a +X

Existem duas formas de alterar o valor de LCOO nesta refatoração. A primeira é pela remoção de um método usando *Mover Método de Classe para Inter-tipo*, situação na qual são eliminados os pares de operações formados por esse método e as demais operações da classe. Se forem eliminados mais pares de operações com atributos compartilhados do que pares sem atributos em comum, então o valor de LCOO aumentará após a remoção do método, caso contrário o valor desta métrica diminuirá.

A outra maneira é pela remoção dos acessos ao atributo usando *Extrair Fragmento para Adendo*. Neste caso, os pares de operações que originalmente compartilhavam apenas o atributo movido passarão a não ter nenhum atributo em comum, deixando de ser contado na parcela negativa de LCOO (pares com atributo em comum) e passando a ser contado na parcela positiva (operações que não compartilham atributos). Isso faz com que

o valor de LCOO da classe aumente.

Já no aspecto são criadas novas operações – os métodos e adendos extraídos da classe –, as quais formam novos pares de operações entre si e com as operações existentes originalmente no aspecto. Esses novos pares aumentarão o valor de LCOO se forem criados mais pares de operações que não compartilham atributos do que pares de operações com um atributo em comum, e diminuirão LCOO caso contrário.

Assim, tanto o valor de LCOO da classe quanto do aspecto poderão aumentar ou diminuir, dependendo dos trechos movidos entre esses componentes e das operações pré-existentes em ambos.

Tabela 5.24: Variação das métricas em *Mover Atributo de Classe para Inter-tipo*

Separação de Interesses			Tamanho				Acoplamento		Coesão
CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
$\leq_{(1)}$	$\gtrless$	$\gtrless$	=	$\gtrless$	=	$\geq_{(X)}$	$\gtrless$	=	$\gtrless$

Análise dos Resultados

Analisando os resultados mostrados na tabela 5.24, é possível observar que todos os atributos internos avaliados têm ao menos uma métrica que pode variar tanto positiva quanto negativamente. Isso faz com que esta refatoração melhore todos os atributos na situação mais favorável e piore todos os atributos na situação mais adversa, ou seja, seus impactos variam muito com as características dos componentes onde ela é aplicada.

Convém ressaltar, no entanto, que de fato é possível usar esta refatoração para diminuir a separação dos interesses transversais, mesmo que ocorram impactos negativos nos demais atributos internos do software.

5.2.5 Trocar Classe Abstrata por Interface

Como Java e AspectJ não permitem herança múltipla, uma classe abstrata evita que suas sub-classes possam herdar de outra classe. Se isto for uma limitação que precisa ser contornada, a refatoração *Trocar Classe Abstrata por Interface* (MONTEIRO, 2005, seção 6.3.1) sugere transformar essa classe abstrata em uma interface e trocar sua relação com as sub-classes de herança para implementação.

Sua aplicação, no entanto, possui algumas pré-condições, das quais serão citadas apenas as relevantes para a avaliação quantitativa desta refatoração. Primeiramente, é necessário que a super-classe seja uma "classe abstrata pura" (i.e., nas palavras de Monteiro (2005), "uma classe abstrata sem nenhum método concreto ou campos não-estáticos"). Em segundo lugar, se a classe abstrata herdar de outra classe, esta refatoração pode quebrar o código-fonte, por isso é necessário verificar potenciais problemas antes de aplicá-la. Caso não sejam encontrados problemas, a refatoração deve ser aplicada inicialmente na classe mais alta da hierarquia e depois ir descendo na árvore.

Os passos para completar a transformação de classe abstrata para interface são os seguintes:

1. Trocar as palavras-chave `abstract class` por `interface`;
2. Se a classe abstrata implementar interfaces, trocar `implements` por `extends`;
3. Opcionalmente pode-se remover a palavra-chave `abstract` das declarações dos métodos;

4. Atualizar todas as classes que herdam da classe abstrata. Para cada sub-classe, trocar `extends` por `implements`. Se a sub-classe implementar outras interfaces, apenas remova a palavra-chave `extends` e mova o nome da ex-classe abstrata para a lista de interfaces implementadas.

Decomposição da Refatoração em Passos Básicos

O primeiro passo da refatoração, apesar de simples, pode ser decomposto em dois passos básicos: "Mudar palavra-chave `class` para `interface`" (99) e "Remover palavra-chave `abstract` de componente" (100). Outra alternativa seria definir outro passo básico "Mudar palavras-chave `abstract class` por `interface`", porém este seria um passo muito mais específico e sem muitas possibilidades de reúso, por isso optou-se pela divisão em dois passos básicos mais simples e genéricos.

No passo 2, a troca da palavra-chave `implements` por `extends` equivale a retirar as implementações de interfaces e adicionar heranças, portanto são necessários dois passos básicos diferentes: "Remover implementação de interface de componente" (42) e "Adicionar herança a interface" (43). Para cada uma das n_{int} interfaces implementadas pela classe abstrata, deve ser retirada sua implementação pelo primeiro passo básico e adicionada sua extensão pelo segundo.

Para executar o 3o. passo da refatoração, deve ser usado o passo básico "Remover palavra-chave `abstract` de operação de interface" (100) tantas vezes quanto existirem operações abstratas na classe abstrata original (em um total de n_{op} aplicações).

Por fim, no último passo da refatoração a troca de `extends` por `implements` tem o mesmo efeito que remover a herança da sub-classe e adicionar uma implementação de interface na mesma – só que neste caso o componente removido e adicionado é o mesmo. Por isso, este passo pode ser decomposto usando dois passos básicos: "Remover herança de classe ou aspecto" (45) e "Adicionar implementação de interface a componente" (41). Em cada uma das n_{sub-cl} sub-classes, remove-se a sua herança usando o primeiro passo básico e adiciona-se a implementação de uma nova interface usando o segundo.

Apesar de não ser descrito explicitamente, ao final da refatoração as mudanças do código devem ser compiladas e testadas para garantir a preservação do comportamento da refatoração, o que é feito por "Compilar e testar" (101).

A tabela 5.25 mostra quantas vezes e onde (em que passo) cada passo básico é utilizado por esta refatoração.

Cálculo da Variação Total das Métricas

Uma vez decompostos os passos da refatoração, deve-se calcular qual o impacto total das alterações nas métricas escolhidas para avaliar a refatoração. A tabela 5.26 mostra o impacto de cada passo básico usado pela refatoração. Nesta tabela é possível observar que as métricas **CDO**, **VS**, **LOC**, **NOA**, **WOC** e **LCOO** não são modificadas por nenhum dos passos, portanto conclui-se que essas métricas não terão seu valor alterado quando *Trocar Classe Abstrata por Interface* for aplicada. A variação total de cada uma das demais métricas é explicada a seguir.

CDC: =

Alguns passos básicos são usados aos pares por esta refatoração: sempre que "Remover implementação de interface de componente" é necessário, usa-se também "Adicionar herança a interface"; e o passo básico "Remover herança de classe ou aspecto" deve ser sempre seguido de "Adicionar implementação de interface a componente". Como os membros de um mesmo par adicionam ou removem o mesmo componente – no primeiro

Tabela 5.25: Quantidade de vezes que cada passo básico é usado em *Trocar Classe Abstrata por Interface*

Passo da refatoração	Quantidade	Passo básico
1	1	Mudar palavra-chave `class` para `interface`
	1	Remover palavra-chave `abstract` de componente
2	n_{int}	Remover implementação de interface de componente
	n_{int}	Adicionar herança a interface
3	n_{op}	Remover palavra-chave `abstract` de operação de interface
4	n_{sub-cl}	Remover herança de classe ou aspecto
	n_{sub-cl}	Adicionar implementação de interface a componente

Tabela 5.26: Impacto nas métricas de cada passo básico de *Trocar Classe Abstrata por Interface*

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Mudar palavra-chave `class` para `interface`	=	=	=	=	=	=	=	=	=	=
Remover palavra-chave `abstract` de componente	=	=	=	=	=	=	=	=	=	=
Remover implementação de interface de componente	$\leq_{(1)}$	=	$\leq_{(2)}$	=	=	=	=	$\leq_{(1)}$	=	=
Adicionar herança a interface	$\geq_{(1)}$	=	$\geq_{(2)}$	=	=	=	=	$\geq_{(1)}$	=	=
Remover palavra-chave `abstract` de operação de interface	=	=	=	=	=	=	=	=	=	=
Remover herança de classe ou aspecto	$\leq_{(1)}$	=	$\leq_{(2)}$	=	=	=	=	$\leq_{(1)}$	$\leq_{(X)}$	=
Adicionar implementação de interface a componente	$\geq_{(1)}$	=	$\geq_{(2)}$	=	=	=	=	$\geq_{(1)}$	=	=
Compilar e testar	=	=	=	=	=	=	=	=	=	=

caso a interface removida é o mesmo componente a ser transformado em interface-mãe, enquanto que no segundo a classe-mãe original é o mesmo componente cuja implementação é adicionada –, sempre que o primeiro passo básico do par diminuir o valor de CDC o segundo passo aumentará esse valor. No caso do primeiro passo não modificar o valor da métrica, também o segundo não fará alterações. Dessa forma, o valor de CDC ao final da refatoração será o igual ao seu valor inicial.

CDLOC: =

Assim como ocorre com CDC, as modificações do valor de CDLOC serão anuladas entre os pares de passos básicos usados nesta refatoração. Conseqüentemente, esta métrica também não é modificada.

CBC: =

Assim como ocorre com CDC, as modificações do valor de CBC serão anuladas entre os pares de passos básicos usados nesta refatoração. Conseqüentemente, esta métrica também não é modificada.

DIT: -X a 0

Ao remover a herança das sub-classes, é possível que a profundidade máxima da árvore de herança seja reduzida. Isso ocorre porque cada sub-classe passará a ter profundidade 1, enquanto que as classes abaixo de suas árvores de herança terão suas profundidades

reduzidas em d_{pai} unidades (d_{pai} é a profundidade original das classe abstrata). Se todos os caminhos de maior comprimento da árvore de herança passarem pela classe transformada em interface, então o valor de DIT diminuirá com essa refatoração.

Tabela 5.27: Variação das métricas em *Trocar Classe Abstrata por Interface*

Separação de Interesses			Tamanho				Acoplamento		Coesão
CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
=	=	=	=	=	=	=	=	$\leq_{(X)}$	=

Análise dos Resultados

O objetivo principal desta refatoração é fazer com que uma "classe abstrata pura" deixe de ser um limitante para que suas sub-classes estendam outro componente. Como conseqüência, além de eliminar esta limitação, pode ocorrer também uma diminuição do acoplamento dos componentes do software por meio da redução da profundidade de sua árvore de herança (tabela 5.27). Futuramente esta redução pode ser neutralizada à medida que as ex-sub-classes passem a estender outra classe – que é, afinal, a motivação para aplicar esta refatoração.

5.2.6 Dividir Classe Abstrata em Aspecto e Interface

Como foi visto em 5.2.5, Java e AspectJ não permitem herança múltipla, o que evita que as sub-classes de uma classe abstrata possam herdar de uma segunda classe. Para lidar com esta limitação, a refatoração *Dividir Classe Abstrata em Aspecto e Interface* (MONTEIRO, 2005, seção 6.3.10) sugere mover todos os membros concretos da classe abstrata para um aspecto e transformar essa classe em interface.

Esta refatoração é bastante similar a *Mover Atributo de Classe para Inter-tipo* (seção 5.2.5), pois ambas têm a mesma motivação – permitir novas heranças para sub-classes de uma classe abstrata – e um resultado em comum – a transformação da classe abstrata em interface. No entanto, enquanto que *Mover Atributo de Classe para Inter-tipo* só pode ser aplicada em classes abstratas sem membros concretos, a refatoração a ser descrita a seguir não tem esta limitação e pode ser aplicada em qualquer tipo de classe abstrata. Além disso, as conseqüências de ambas são bastante distintas, como veremos a seguir.

Os passos desta refatoração são os seguintes:

1. Criar um aspecto para conter os membros concretos da classe abstrata;

2. Usar *Mover Atributo de Classe para Inter-tipo* para mover cada atributo da classe abstrata para o aspecto;

3. Caso a classe possua construtores, simplesmente movê-lo para o aspecto não será adequado porque a classe futuramente se transformará em interface e, portanto, não poderá ter esse tipo de operação. A solução neste caso depende da presença de parâmetros no construtor: se não forem necessários argumentos para a inicialização, usar *Extract Fragment indo Advice* para mover o código de inicialização do construtor da classe para o aspecto; caso contrário, criar métodos de escrita (*setters*) para os atributos que precisam de valores externos e refatorar o código para que esses métodos sejam usados, ao invés dos construtores com parâmetros;

4. Para cada método concreto, usar *Move Method from Class to Inter-type* para movê-lo da classe abstrata para o aspecto, deixando na classe uma declaração abstrata da assinatura do método;

5. Usar *Mover Atributo de Classe para Inter-tipo* para tornar a classe abstrata uma interface e atualizar as sub-classes.

Ao final destes passos, a classe abstrata original terá sido dividida em dois componentes: o aspecto, com os membros concretos da classe abstrata, e a interface, apenas com a assinatura das operações da classe original.

Exemplo:
O código abaixo resume o exemplo que Monteiro (2005, p. 110) apresenta para explicar o funcionamento desta refatoração:

```
public abstract class Creator {
    private static Point lastFrameLocation = new Point(0, 0);
    public abstract JComponent createComponent();
    public abstract String getTitle();
    public final void showFrame() {
        (...)
    }
}
```

```
public interface Creator {
    private static Point lastFrameLocation = new Point(0, 0);
    public abstract JComponent createComponent();
    public abstract String getTitle();
    public final void showFrame();
}
```

```
public aspect CreatorImplementation {
    private Point Creator.lastFrameLocation = new Point(0, 0);
    public final void Creator.showFrame() {
        (...)
    }
}
```

Decomposição da Refatoração em Passos Básicos

Para executar o primeiro passo da refatoração, basta usar o passo básico "Criar aspecto vazio" (39) para adicionar o aspecto que receberá os membros concretos da classe abstrata original.

No 2o. passo, a refatoração *Mover Atributo de Classe para Inter-tipo* (134) é usada para mover cada um dos n_{atrib} atributos da classe abstrata.

No passo 3, os códigos de inicialização dos construtores da classe abstrata são movidos para o aspecto. Se houver algum construtor c com parâmetros, (1) devem ser criados métodos de acesso para os n_{param_c} atributos modificados por esses parâmetros através de *Encapsular Atributo* (115), (2) as $n_{chamadas_c}$ chamadas ao construtor devem ser modificadas por "Trocar passagem de parâmetro em construtor por chamada de método de escrita" (72), e (3) se não existir um construtor sem parâmetros, ele deve ser gerado, o que é feito usando *Remover Parâmetro* (110) para remover os n_{param_c} parâmetros do construtor e "Remover linhas de código de operação" (78) para retirar do construtor os trechos que usam os parâmetros. Cada um dos n_{const} construtores com parâmetros da classe abstrata deve ter seu código de inicialização movido dessa forma. Para mover o código de inicialização de um construtor sem parâmetros é necessário usar *Extract Fragment indo Advice* (124).

O passo 4 pode ser decomposto em duas etapas: *Mover Método de Classe para Inter-tipo* (130) para mover o método concreto para o aspecto; e "Criar operação vazia" (58),

"Adicionar parâmetro a operação" (60) e "Tornar operação abstrata" (81) para criar a declaração abstrata da assinatura do método com $n_{param_{met}}$ parâmetros. Para cada um dos n_{met} métodos concretos, essa seqüência de passos deve ser executada.

Por fim, o passo 5 transforma a classe abstrata em interface através de *Mover Atributo de Classe para Inter-tipo* (140).

Apesar de não ser descrito explicitamente, ao final da refatoração as mudanças do código devem ser compiladas e testadas para garantir a preservação do comportamento da refatoração, o que é feito por "Compilar e testar" (101).

A tabela 5.28 mostra quantas vezes e onde (em que passo) cada passo básico é utilizado por esta refatoração.

Tabela 5.28: Quantidade de vezes que cada passo básico é usado em *Dividir Classe Abstrata em Aspecto e Interface*

Passo da refatoração	Quantidade	Passo básico
1	1	Criar aspecto vazio
2	n_{atrib}	*Mover Atributo de Classe para Inter-tipo*
3	$\sum_{c=1}^{n_{const}} n_{param_c}$	*Encapsular Atributo*
	$\sum_{c=1}^{n_{const}} n_{chamadas_c}$	Trocar passagem de parâmetro em construtor por chamada de método de escrita
	$\sum_{c=1}^{n_{const}} n_{param_c}$	*Remover Parâmetro*
	$\sum_{c=1}^{n_{const}} n_{param_c}$	Remover linhas de código de operação
	1	*Extract Fragment indo Advice*
4	n_{met}	*Mover Método de Classe para Inter-tipo*
	n_{met}	Criar operação vazia
	$n_{param_{met}}$	Adicionar parâmetro a operação
	n_{met}	Tornar operação abstrata
5	1	*Mover Atributo de Classe para Inter-tipo*
	1	Compilar e testar

Cálculo da Variação Total das Métricas

Uma vez decompostos os passos da refatoração, deve-se calcular qual o impacto total das alterações nas métricas escolhidas para avaliar a refatoração. A tabela 5.29 mostra o impacto de cada passo básico usado pela refatoração. Nesta tabela é possível observar que apenas **NOA** não é modificada por nenhum dos passos, portanto conclui-se que essa métrica não terá seu valor alterado quando *Dividir Classe Abstrata em Aspecto e Interface* for aplicada. A variação total de cada uma das demais métricas é explicada a seguir.

CDC: +1

Esta refatoração divide a classe abstrata original em uma interface e um aspecto, separando assim a parte abstrata da classe de sua parte concreta. Em outras palavras, após a refatoração as funcionalidades originalmente presentes apenas na classe abstrata estarão dispersas pela interface e pelo aspecto, o que aumenta em uma unidade o número de componentes pelos quais o interesse transversal da classe abstrata está disperso.

CDO: $(n_{met} + 2 * n_{atrib})$

Tanto o aspecto quanto a interface serão componentes principais de um mesmo interesse transversal após o término da refatoração, assim como todas as suas operações serão operações principais desse interesse. Para descobrir como CDO será alterado pela refatoração, basta conhecer a variação do número de operações desses dois componentes.

Tabela 5.29: Impacto nas métricas de cada passo básico de *Dividir Classe Abstrata em Aspecto e Interface*

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Criar aspecto vazio	$>_{(1)}$	=	=	$>_{(1)}$	$>_{(2)}$	=	=	=	=	=
Mover Atributo de Classe para Inter-tipo	$\leq_{(1)}$	$\gtrless$	$\gtrless$	=	$\gtrless$	=	$\geq_{(X)}$	$\gtrless$	=	$\gtrless$
Encapsular Atributo	=	$\geq_{(X)}$	=	=	$>_{(6)}$	=	$>_{(3)}$	=	=	$\gtrless$
Trocar passagem de parâmetro em construtor por chamada de método de escrita	=	=	=	=	$>_{(X)}$	=	=	=	=	=
Remover Parâmetro	$\leq_{(X)}$	$\gtrless_{(-Xa+1)}$	$\leq_{(X)}$	=	$\gtrless_{(-Xa+3)}$	=	$\gtrless_{(-1a+X)}$	$\leq_{(X)}$	=	$\geq_{(X)}$
Remover linhas de código de operação	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(X)}$	=	$<_{(X)}$	=	=	$\leq_{(X)}$	=	$\geq_{(X)}$
Extract Fragment indo Advice	$\leq_{(X)}$	$\gtrless_{(-Xa+1)}$	$\leq_{(X)}$	=	$\gtrless$	=	$\geq_{(X)}$	$\gtrless$	=	$\gtrless$
Mover Método de Classe para Inter-tipo	$\leq_{(1)}$	=	$\leq_{(2)}$	=	=	=	=	$\gtrless$	=	$\gtrless$
Criar operação vazia	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(2)}$	=	$>_{(1)}$	$\geq_{(1)}$	=	$\geq_{(X)}$
Adicionar parâmetro a operação	=	=	=	=	=	=	$>_{(1)}$	$\geq_{(1)}$	=	=
Tornar operação abstrata	=	=	=	=	$<_{(1)}$	=	=	=	=	$\gtrless$
Mover Atributo de Classe para Inter-tipo	=	=	=	=	=	=	=	=	$\leq_{(X)}$	=
Compilar e testar	=	=	=	=	=	=	=	=	=	=

Na classe, as operações abstratas são mantidas, os métodos concretos são transformado em abstratos e os construtores são removidos. Dessas modificações, apenas a última altera o número de operações da classe, portanto o número de operações deste componente diminuirá n_{const} unidades ao aplicar esta refatoração.

Já no aspecto são criadas diversas operações: as n_{met} declarações inter-tipo com os métodos concretos da classe abstrata original, os $2 * n_{atrib}$ métodos de acesso aos atributos encapsulados[3] no passo 3 e os n_{const} adendos criados a partir dos construtores da classe abstrata. Ao final da refatoração o aspecto terá um total de $n_{met} + 2 * n_{atrib} + n_{const}$ operações.

Assim, as modificações feitas por esta refatoração alteram o número de operações da interface e do aspecto em $(n_{met} + 2 * n_{atrib} + n_{const}) - (n_{const})$, ou seja, $(n_{met} + 2 * n_{atrib})$.

[3]O número de atributos encapsulados na verdade é $(\sum_{c=1}^{n_{const}} n_{param_c})$, mas como este valor pode ser maior que o número de atributos da classe (n_{atrib}) porque diferentes construtores podem alterar o mesmo atributo, optou-se por representar o número de atributos encapsulados através de seu valor máximo – n_{atrib}.

CDLOC: =

Tanto o aspecto quanto a interface implementam o mesmo interesse transversal da classe abstrata original, logo as linhas desses três componentes são todas sombreadas. Com isso, mesmo dividindo um componente em dois não há a criação ou remoção de pontos de transição entre interesses transversais, mantendo assim o valor de CDLOC inalterado.

VS: +1

Para dividir a classe abstrata original em uma interface e um aspecto, essa refatoração cria um aspecto em seu primeiro passo, aumentando assim o número de componentes do sistema em uma unidade.

LOC: $+2$ a $(2 + 6 * n_{atrib} + \sum_{c=1}^{n_{const}} n_{param_c} + n_{LOC_{conj.pts}} + n_{met})$

Esta refatoração cria linhas de código na interface ou no aspecto de diferentes maneiras: na adição do aspecto (passo 1) são criadas duas linhas; para cada atributo encapsulado são criadas 6 linhas, totalizando um aumento de até $6 * n_{atrib}$ unidades; cada chamada ao construtor c substituída por uma chamada ao construtor sem parâmetros cria até n_{param_c} linhas, em um total de $\sum_{c=1}^{n_{const}} n_{param_c}$ linhas adicionadas; a movimentação dos construtores através de *Extrair Fragmento para Adendo* adiciona outras linhas por causa da criação dos conjuntos de pontos de junção; e cada uma das n_{met} declarações abstratas dos métodos da interface cria uma linha de código.

Também podem ser removidas linhas de código caso um construtor com parâmetros tenha que ser transformado em um construtor sem parâmetros no passo 3. Neste caso, são eliminadas do construtor c no máximo n_{param_c} linhas – cada parâmetro usado para definir um único atributo. No entanto, caso essas linhas tenham que ser removidas, necessariamente o mesmo montante de linhas teve de ser adicionado na operação que chama este construtor porque cada atributo inicializado dentro do construtor passará a ter seu valor definido fora dele, conforme o exemplo abaixo.

```
public abstract class X {
    public X(Param p1, Param p2) {
        this.attribute1 = p1;
        this.attribute2 = p2;
        (...)
    }
    (...)
}
    public someOperation() {
        (...)
        X ex = new X(p1, p2);
        (...)
    }
```

⇒

```
public abstract class X {
    public X(~~Param p1, Param p2~~) {
        ~~this.attribute1 = p1;~~
        ~~this.attribute2 = p2;~~
        (...)
    }
    (...)
}
    public someOperation() {
        (...)
        X ex = new X(~~p1, p2~~);
        ex.setAttribute1(p1);
        ex.setAttribute2(p2);
        (...)
    }
```

Juntando o número de linhas removidas ao número de linhas criadas, tem-se uma variação de LOC entre $+2$ e $2+6*n_{atrib}+\sum_{c=1}^{n_{const}} n_{param_c}+n_{LOC_{conj.pts}}+n_{met}$ unidades, onde $n_{LOC_{conj.pts}}$ equivale ao número de linhas de código adicionadas com a criação de conjuntos de pontos de junção em *Extrair Fragmento para Adendo* – esta refatoração não define uma fórmula geral de prever este valor, por isto foi necessário definir esta nova variável.

WOC: 0 a $(3 * n_{atrib} + n_{param_c} + (\sum_{m=1}^{n_{met}} n_{param_m}))$

O valor de WOC pode ser alterado de três maneiras por esta refatoração: pelo encapsulamento de atributos, pela remoção de parâmetros do construtor e pela declaração das operações abstratas na interface. Nenhuma dessas etapas é obrigatória para toda aplicação da refatoração, portanto há casos em que o valor de WOC não é alterado.

O encapsulamento de atributos aumenta o valor de WOC em 3 unidades a cada atributo encapsulado, chegando ao máximo de $3 * n_{atrib}$. A remoção de parâmetros diminui a complexidade do construtor c em n_{param_c} unidades. E a declaração da operação abstrata m na interface adiciona uma complexidade de n_{param_m} unidades, totalizando um aumento de $(\sum_{m=1}^{n_{met}} n_{param_m})$.

Assim, a variação do valor de WOC será de até $3 * n_{atrib} + n_{param_c} + (\sum_{m=1}^{n_{met}} n_{param_m})$ unidades.

CBC: +1 a n_{acopl}

Inicialmente a classe abstrata possui uma quantidade determinada de acoplamentos com outros componentes do sistema, digamos n_{acopl}. Ao aplicar esta refatoração, esses acoplamentos serão divididos entre o aspecto e a interface. Para cada componente que estiver acoplado a ambos após a refatoração, o valor de CBC terá aumentado uma unidade, com um aumento máximo de n_{acopl} unidades. Isto ocorre porque inicialmente o acoplamento era contado apenas uma vez (só na classe abstrata), mas após a refatoração passou a ser contado duas vezes (na interface e no aspecto).

Existe também um acoplamento criado pela refatoração: o acoplamento do aspecto com a interface. Ele está presente nas declarações inter-tipo do aspecto responsáveis por adicionar os membros concretos aos componentes que implementarem a interface, pois é necessário explicitar em que componente esses métodos e atributos serão adicionados. No exemplo usado para ilustrar essa refatoração, as duas primeiras linhas do aspecto `CreatorImplementation` apresentam este acoplamento à interface `Creator`.

Assim, independentemente da maneira com que os acoplamentos originais da classe abstrata forem divididos entre o aspecto e a interface, o valor de CBC será obrigatoriamente maior ao final desta refatoração – entre uma e n_{acopl} unidades.

DIT: -X a 0

Ao transformar a classe abstrata em uma interface, suas sub-classes serão colocadas na raiz da árvore de herança, o que diminui o valor de DIT caso todos os caminhos de comprimento máximo da árvore passem pela classe transformada.

LCOO: -X a +X

Ao final da refatoração o aspecto terá quase todas as operações concretas da classe abstrata original, com exceção dos construtores, que terão apenas o seu representante sem parâmetros extraído para o aspecto (na forma de um adendo). Isso faz com que o aspecto tenha quase todos os pares de operações que a classe abstrata tinha originalmente, descontando-se os pares formados entre os construtores com parâmetros e as demais operações concretas da classe. Caso a maioria desses pares não-movidos para o aspecto compartilhassem atributos, o valor de LCOO terá aumentado com a refatoração, caso contrário este valor terá diminuído.

Outra forma de modificar o valor de LCOO é ao aplicar *Encapsular Atributo*. Essa refatoração cria dois métodos de acesso ao atributo, métodos esses que formarão novos pares com as demais operações do aspecto. Se a maior parte dos pares formados acessar algum atributo em comum, o valor de LCOO diminui, caso contrário aumenta.

Assim, essas duas modificações podem alterar o valor de LCOO tanto positiva quanto negativamente, dependendo dos atributos acessados pelos construtores e pelas demais operações concretas da classe abstrata.

Análise dos Resultados

Para liberar a herança das sub-classes de uma classe abstrata aplica-se esta refatoração, dividindo a classe abstrata em uma interface e um aspecto. Esse ganho de liberdade para

Tabela 5.30: Variação das métricas em *Dividir Classe Abstrata em Aspecto e Interface*

Separação de Interesses			Tamanho				Acoplamento		Coesão
CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
$>_{(1)}$	$>_{(X)}$	=	$>_{(1)}$	$>_{(2aX)}$	=	$\geq_{(X)}$	$>_{(1aX)}$	$\leq_{(X)}$	$\gtrless$

compor as sub-classes com outros componentes do sistema também possui conseqüências negativas (tabela 5.30): aumenta-se o tamanho do software e os interesses ficam mais separados – o que é até previsível, uma vez que o código do componente original (a classe abstrata) é dividido em outros dois (interface e aspecto). Além disso, observa-se que o acoplamento possui conseqüências opostas em suas duas métricas (CBC aumenta e DIT pode diminuir) e a coesão tem diferentes efeitos de acordo com o código em que *Dividir Classe Abstrata em Aspecto e Interface* for aplicada. Dessa forma, apesar da refatoração de fato permitir uma maior liberdade para criar relações de herança em Java e AspectJ, o custo dessa liberdade é bastante alto quando se leva em consideração a separação dos interesses transversais, o tamanho do software, o acoplamento e a coesão dos componentes.

5.2.7 Extrair Classe Interna para Autônoma

Classes internas são muito usadas em Java para separar funcionalidades não-relacionadas à funcionalidade principal da classe que as contém. No entanto, com as novas formas de composição da POA, essa abordagem deixa de ser a melhor opção: os tipos de funcionalidades secundárias encontrados em classes internas são melhor modularizados dentro de aspectos (MONTEIRO, 2005). Caso o interesse da classe interna já seja modularizado em um aspecto (ou vá ser), deve-se executar a refatoração *Extrair Classe Interna para Autônoma* (MONTEIRO, 2005, seção 6.3.4), que transforma a classe interna em uma classe separada, eliminando as dependências entre esta e a classe que a contém.

Esta refatoração é útil para reduzir a repetição de código quando usada para unificar diversas classes internas idênticas em uma única classe autônoma, porém pode não ser uma boa opção apenas para extrair uma única classe interna. Normalmente ela é usada em um processo de refatoração maior, que envolva a extração de um interesse em um aspecto, daí seu uso nos passos iniciais de *Extrair Funcionalidade para Aspecto* (seção 5.2.11) ou antes de executar refatorações como *Extract Fragment into Aspect* (seção 5.2.2) e *Mover Atributo de Classe para Inter-tipo* (seção 5.2.4).

Para extrair a classe interna, devem ser executados os seguintes passos:

1. Procurar por trechos da classe interna que devam permanecer na classe externa. Usar *Extrair Método* (seção 5.1.1) nessas partes;

2. Criar na classe interna um atributo privado do tipo da classe externa. Em alguns casos, são definidas diversas classes internas idênticas dentro de diferentes classes, as quais não possuem um tipo em comum. Nesses casos deve-se primeiro criar uma interface que exponha a interface comum usada nas classes internas e depois fazer com que as classes externas implementem essa interface. O atributo privado da classe interna deve ser do tipo desta interface, possibilitando assim a extração de uma classe interna comum a todas as classes externas;

3. Criar um construtor público na classe interna com os mesmos parâmetros de seu construtor já existente e incluir nele um parâmetro do mesmo tipo do atributo privado criado no passo anterior. Atualizar as criações de instâncias da classe interna

na classe externa com o novo parâmetro. Caso a classe interna não seja privada, fazer o mesmo fora da classe externa;

4. Compilar e testar;

5. Procurar por referências diretas a atributos da classe externa e usar *Auto Encapsular Atributo* (seção 5.1.4) nesses atributos;

6. Procurar chamadas a métodos privados feitas dentro da classe interna e relaxar as regras de acesso a esses métodos;

7. Compilar e testar;

8. Criar uma classe separada com o mesmo nome da classe interna. Copiar o código-fonte da classe interna para a nova classe e adicionar o modificador `public` antes do nome da classe. Adicionar as importações necessárias à nova classe externa;

9. Compilar e testar;

10. Remover a classe interna. Compilar e testar de novo.

Exemplo:
Os códigos aqui apresentados foram baseados no exemplo de (MONTEIRO, 2005, pp. 100-102) e mostram passo-a-passo o funcionamento da refatoração. Inicialmente, tem-se uma classe `Flower` com uma classe interna `OpenNotifier` que deve ser extraída:

```
public class Flower {
    private boolean _isOpen;
    private OpenNotifier _oNotify = new OpenNotifier();
    (...)
    private class OpenNotifier extends Observable {
        private boolean _alreadyOpen = false;
        public void notifyObservers() {
            if(_isOpen && !_alreadyOpen) {
                setChanged();
                super.notifyObservers();
                _alreadyOpen = true;
            }
        }
        public void close() {
            _alreadyOpen = false;
        }
    }
}
```

O primeiro passo da refatoração extrai trechos de código que devam permanecer na classe externa, o que não é necessário neste caso. Assim, este passo não é executado neste exemplo.

Os próximos dois passos criam o atributo privado `_enclosing` na classe interna e o construtor que recebe um valor para inicializar este atributo, além de atualizar as instanciações da classe interna:

```
public class Flower {
    (...)
    private OpenNotifier _oNotify = new OpenNotifier(this);
    (...)
    private class OpenNotifier extends Observable {
        private Flower _enclosing;
        private boolean _alreadyOpen = false;
        public OpenNotifier(Flower enclosing) {
            this._enclosing = enclosing;
        }
        (...)
    }
}
```

Em seguida, no passo 5, deve-se aplicar *Auto Encapsular Atributo* nos atributos da classe externa que são acessados diretamente na classe interna. Neste exemplo, isso é feito ao atributo `_isOpen`:

```
public class Flower {
    (...)
    public boolean isOpen() {
        return this._isOpen;
    }
    private class OpenNotifier extends Observable {
        (...)
        public void notifyObservers() {
            if(_enclosing.isOpen() && !_alreadyOpen) {
        (...)
    }
}
```

Como nenhum método privado da classe externa é chamado na classe interna, o passo 6 não precisa ser executado.

Por fim, é criada uma cópia da classe interna fora da classe que a contém, e remove-se a classe interna original:

```
public class Flower {
    (...)
    private class OpenNotifier extends Observable {
        (...)
    }
}
```

```
public class OpenNotifier extends Observable {
    (...)
}
```

Decomposição da Refatoração em Passos Básicos

O primeiro passo da refatoração usa *Extrair Método* (104) em cada um dos $n_{trechos}$ trechos que devam permanecer na classe externa. Se cada trecho for extraído para um método, *Extrair Método* será usado $n_{trechos}$ vezes; se trechos originalmente separados forem reunidos por alguma das extrações, *Extrair Método* será usado entre 1 e $(n_{trechos} - 1)$ vezes.

O 2o. passo cria o atributo com o tipo comum a todas as classes externas que contêm as classes a extraídas, o que é feito pelo passo básico "Adicionar atributo a componente" (82). Se não existir esse tipo em comum, uma interface será criada para expor a interface usada pelas classes internas e as classes externas terão que implementar esta nova interface. Para criar a interface, usa-se "Criar classe ou interface vazia" (40), "Criar operação vazia" (58) para cada uma das $n_{op.interface}$ operações que a nova interface deve ter, "Adicionar parâmetro a operação" (60) para adicionar às operações criadas os $n_{param.int}$

parâmetros necessários, e "Adicionar atributo a componente" (82) para criar os $n_{atrib.int}$ atributos na interface; e para fazer com que cada uma das $n_{cl.ext}$ classes externas implementem a interface usa-se "Adicionar implementação de interface a componente" (41).

No passo 3 é criado o construtor com um parâmetro a mais e são atualizadas as instanciações da classe interna. O construtor é criado usando os passos básicos "Criar operação vazia" (58) uma vez e "Adicionar parâmetro a operação" (60) $n_{param.const}$ vezes. Seu corpo é preenchido adicionando uma linha de chamada ao construtor pré-existente e outra linha com a inicialização do atributo criado no passo anterior, ações que são executadas respectivamente por "Adicionar chamada a operação" (79) e "Adicionar linha de leitura/escrita de valor de atributo a operação" (73). Por fim, as n_{inst} instanciações da classe interna são atualizadas aplicando "Trocar chamada a operação sobrecarregada por versão com mais parâmetros" (69) em cada uma delas.

Os passos 4, 7 e 9 compilam e testam o programa para fazer verificações parciais de correção do código, e são decompostos no passo básico "Compilar e testar" (101).

O 5o. passo encapsula os $n_{atrib.ext}$ atributos acessados diretamente pela classe interna usando *Auto Encapsular Atributo* (119). Como este passo executa apenas o encapsulamento desses atributos e a refatoração usada pra isso já foi avaliada, então o passo é decomposto somente por *Auto Encapsular Atributo*.

No passo 6 os $n_{met.ext}$ métodos da classe externa que são usados pela classe interna têm seus acessos relaxados, o que é feito usando apenas o passo básico "Tornar operação protegida" (91) – se a classe interna for extraída para o mesmo pacote da classe externa – ou "Tornar operação pública" (90) – caso contrário.

O 8o. passo cria uma cópia da classe interna, o que é feito pelo passo básico "Criar cópia autônoma de componente interno" (50). Para tornar pública essa cópia autônoma, usa-se "Tornar componente público" (89), enquanto que para adicionar as importações necessárias usa-se "Adicionar importações" (95).

Por fim, no último passo da refatoração, deve-se remover a classe interna usando o passo básico "Remover cópia interna de componente autônomo" (48) e eliminar as importações não mais necessárias ao arquivo da classe externa usando "Remover importações" (96). Em seguida compila-se e testa-se o programa usando "Compilar e testar" (101). No caso de múltiplas classes internas extraídas para a mesma classe autônoma, a remoção da classe interna e das importações deve ser executada $n_{cl.ext}$ vezes[4].

A tabela 5.31 mostra quantas vezes e onde (em que passo) cada passo básico é utilizado por esta refatoração.

Cálculo da Variação Total das Métricas

Esta refatoração é usada de formas um pouco diferentes quando se deseja extrair apenas uma classe interna e quando forem extraídas diversas classes ao mesmo tempo. Na extração de diversas classes, uma etapa a mais pode ser necessária (a criação da interface comum às classes externas no passo 2) e as classes internas são removidas e substituídas por uma única classe autônoma. Isso faz com que os impactos nas métricas sejam distintos para os dois casos, por isso a avaliação dos efeitos da refatoração nos valores das métricas será feita separadamente para essas duas situações.

Ressalta-se que, para efetuas as avaliações, foi considerado que tanto o atributo interno criado no passo 2 quanto o construtor com parâmetro do tipo deste atributo definido no

[4]o número de classes externas usadas nesta refatoração é igual ao número de classes internas extraídas ($n_{cl.ext}$), uma vez que cada classe externa considerada possui uma classe interna a ser extraída

Tabela 5.31: Quantidade de vezes que cada passo básico é usado em *Extrair Classe Interna para Autônoma*

Passo da refatoração	Quantidade	Passo básico
1	$\leq n_{trechos}$	*Extrair Método*
2	1	Adicionar atributo a componente
	1	Criar classe ou interface vazia
	$n_{op.interface}$	Criar operação vazia
	$n_{param.int}$	Adicionar parâmetro a operação
	$n_{atrib.int}$	Adicionar atributo a componente
	$n_{cl.ext}$	Adicionar implementação de interface a componente
3	1	Criar operação vazia
	$n_{param.const}$	Adicionar parâmetro a operação
	1	Adicionar chamada a operação
	1	Adicionar linha de leitura/escrita de valor de atributo a operação
	n_{inst}	Trocar chamada a operação sobrecarregada por versão com mais parâmetros
4	1	Compilar e testar
5	$n_{atrib.ext}$	*Auto Encapsular Atributo*
6	$n_{met.ext}$	Tornar operação pública
	$n_{met.ext}$	Tornar operação protegida
7	1	Compilar e testar
8	1	Criar cópia autônoma de componente interno
	1	Tornar componente público
	1	Adicionar importações
9	1	Compilar e testar
10	$n_{cl.ext}$	Remover cópia interna de componente autônomo
	$n_{cl.ext}$	Remover importações
	1	Compilar e testar

passo 3 não existiam anteriormente e foram de fato adicionados à classe interna durante a refatoração.

Extração de uma classe interna:
Alguns dos passos básicos citados na decomposição desta refatoração não serão necessários caso apenas uma classe interna seja extraída. É o caso dos passos básicos usados no passo 2 para construir a interface comum a todas as classes externas – como só será extraída uma classe, o tipo comum é a própria classe que contém a classe interna, portanto esta etapa não precisa ser executada. A tabela 5.32 mostra o impacto de cada passo básico usado pela refatoração neste caso.

Nesta tabela é possível observar que apenas a métrica **DIT** não pode ser modificada por nenhum dos passos básicos. No entanto, dois passos básicos fazem alterações inversas no código: "Criar cópia autônoma de componente interno" e "Remover cópia interna de componente autônomo". Tudo o que for criado por um dos passos será removido pelo outro na mesma medida, pois ambos são executados a mesma quantidade de vezes (uma) e a cópia criada deve ser idêntica à cópia removida. Assim, estes dois passos básicos têm seus impactos anulados um pelo outro quando analisados conjuntamente, o

que nos permite desconsiderar os efeitos desses passos quando avaliamos as métricas. Com isso conclui-se que além de **DIT**, também **VS** não terá seu valor alterado quando *Extrair Classe Interna para Autônoma* for aplicada para extrair uma única classe interna.

A variação total de cada uma das demais métricas é explicada a seguir, desconsiderando os impactos de "Criar cópia autônoma de componente interno" e "Remover cópia interna de componente autônomo" pelos motivos citados anteriormente.

Tabela 5.32: Impacto nas métricas de cada passo básico usado por *Extrair Classe Interna para Autônoma* para extrair apenas uma classe interna

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Extrair Método	=	$\geq_{(1)}$	$\leq_{(X)}$	=	$>_{(3a4)}$	=	$>_{(X)}$	=	=	$\gtrless$
Adicionar atributo a componente	$\geq_{(1)}$	=	$\geq_{(2)}$	=	$>_{(1)}$	$>_{(1)}$	=	$\geq_{(1)}$	=	=
Criar operação vazia	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(2)}$	=	$>_{(1)}$	$\geq_{(1)}$	=	$\geq_{(X)}$
Adicionar chamada a operação	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(1)}$	=	=	$\geq_{(1)}$	=	=
Adicionar linha de leitura/escrita de valor de atributo a operação	=	=	=	=	$>_{(1)}$	=	=	=	=	$\leq_{(X)}$
Adicionar parâmetro a operação	=	=	=	=	=	=	$>_{(1)}$	$\geq_{(1)}$	=	=
Trocar chamada a operação sobrecarregada por versão com mais parâmetros	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$\geq_{(X)}$	=	=	$\geq_{(X)}$	=	=
Compilar e testar	=	=	=	=	=	=	=	=	=	=
Auto Encapsular Atributo	=	$\geq_{(X)}$	=	=	$>_{(6)}$	=	$>_{(3)}$	=	=	$>_{(X)}$
Tornar operação pública	=	=	=	=	=	=	=	=	=	=
Tornar operação protegida	=	=	=	=	=	=	=	=	=	=
Criar cópia autônoma de componente interno	$\geq_{(1)}$	$\geq_{(X)}$	$\geq_{(X)}$	$>_{(1)}$	$>_{(X)}$	$\geq_{(X)}$	$\geq_{(X)}$	$\geq_{(X)}$	=	$\geq_{(X)}$
Tornar componente público	=	=	=	=	=	=	=	=	=	=
Adicionar importações	=	=	=	=	=	=	=	=	=	=
Remover cópia interna de componente autônomo	$\leq_{(1)}$	$\leq_{(X)}$	$\leq_{(X)}$	$<_{(1)}$	$<_{(X)}$	$\leq_{(X)}$	$\leq_{(X)}$	$\leq_{(X)}$	=	$\leq_{(X)}$
Remover importações	=	=	=	=	=	=	=	=	=	=

CDC: =

Dentre os passos básicos usados, quatro podem alterar o valor de CDC: (1) "Adicionar atributo a componente", (2) "Criar operação vazia", (3) "Adicionar chamada a operação" e (4) "Trocar chamada a operação sobrecarregada por versão com mais parâmetros". No entanto, esta métrica não terá seu valor alterado por nenhum deles, pelos respectivos motivos: (1) como a classe interna já era originalmente relacionada à classe externa, a adição do atributo do tipo da classe externa não modifica os interesses transversais presentes na classe interna; (2) a criação de uma nova operação só altera o valor de CDC se seu tipo de retorno não for vazio, o que não ocorre no caso da criação de um construtor; (3) por serem operações sobrecarregada, ambos os construtores estão relacionados aos mesmos interesses, por isso adicionar uma chamada de um dentro do corpo de outro não adiciona nenhum interesse transversal; e (4) a troca do construtor chamado por um construtor com um parâmetro a mais não aumenta o valor de CDC porque o parâmetro extra que poderia aumentar o valor dessa métrica é do tipo da classe externa, componente esse que já é usado antes da refatoração para referenciar a classe interna.

CDO: 1 a $(n_{trechos} + n_{ac.int} + 3)$

Tanto *Extrair Método* quanto *Auto Encapsular Atributo* criam novas operações na classe externa. Se estas novas operações forem operações principais de interesses transversais, então o número de operações pelas quais esses interesses estarão dispersos aumentará, respectivamente, até $n_{trechos}$ e $n_{ac.int} + 2$, onde $n_{ac.int}$ é o número de vezes que um dos métodos de acesso do atributo encapsulado é usado.

Outra modificação que aumenta o valor de CDO é a criação do novo construtor no

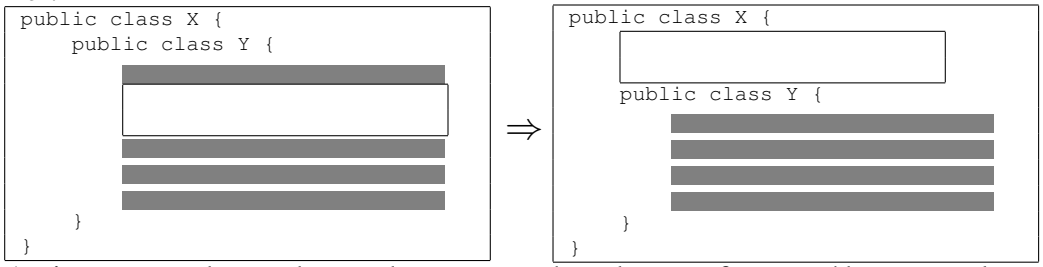
passo 3. Sabe-se que a classe interna é um dos componentes principais de um interesse transversal, pela própria descrição da refatoração – "caso o interesse da classe interna já seja modularizado em um aspecto (ou vá ser), deve-se executar a refatoração *Extrair Classe Interna para Autônoma*" –, por isso seus construtores são operações principais desse interesse. Dessa forma, a criação de um construtor aumenta o número de operações pelas quais o interesse está disperso, aumento esse que ocorre sempre que a refatoração for executada porque este passo não é opcional. Vale ressaltar que neste caso o interesse transversal não é o mesmo que é alterado por *Extrair Método* e *Auto Encapsular Atributo* – se fossem o mesmo, as classes externa e interna estariam relacionadas ao mesmo interesse e não faria sentido extrair a classe para separar esses componentes.

Os outros passos básicos que poderiam alterar o valor de CDO são "Adicionar chamada a operação" e "Trocar chamada a operação sobrecarregada por versão com mais parâmetros", mas nessa refatoração estes passos básicos não adicionam comandos que possam conter operações principais de outros interesses transversais. Isso ocorre, respectivamente, porque o construtor chamado está relacionado ao mesmo interesse do construtor onde a chamada é adicionada e porque o valor extra passado ao construtor com mais parâmetros é sempre `this`, atributo que obviamente está relacionado ao mesmo interesse do componente que o contém.

Portanto, juntando a variação de CDO de cada um dos interesses transversais, concluise que esta refatoração poderá alterar o valor dessa métrica em até $n_{trechos} + n_{ac.int} + 3$.

CDLOC: $(-2 * n_{trechos} - 2)$ a -2

Pela descrição da refatoração, sabe-se que a classe interna é usada para separar funcionalidades não-relacionadas à funcionalidade principal da classe externa, por isso seu sombreamento é diferente do sombreamento da classe externa. Se for necessário extrair trechos através de *Extrair Método*, isso quer dizer que originalmente existiam na classe interna comandos relacionados ao interesse da classe externa, ou seja, a classe interna tinha pontos de transição em seu corpo. A remoção dos trechos pela extração de métodos pode reunir linhas sombreadas que originalmente estavam separadas pelos trechos extraídos, eliminando assim pontos de transição. O exemplo abaixo mostra como esta situação pode ocorrer:



Assim, para cada um dos trechos não-sombreados que for extraído para a classe externa cuja remoção (da classe interna) juntar blocos sombreados originalmente separados por esse trecho, serão eliminados dois pontos de transição do software, gerando uma remoção máxima de $2 * n_{trechos}$ pontos.

Outra etapa que altera o valor de CDLOC é a remoção da classe interna de dentro da classe externa. Ela sempre eliminará dois pontos de transição, pois o sombreamento dos dois componentes não é o mesmo e, portanto, existe um ponto de transição entre o início da classe interna e o corpo da classe externa, e outro entre o fim da classe interna e o corpo da classe externa.

Existem outros passos da refatoração que poderiam alterar o valor de CDLOC: a adição do atributo no passo 2 e a criação do novo construtor no passo 3 adicionam novas linhas de código à classe interna, porém essas linhas serão sombreadas como as demais

linhas da classe porque são responsáveis por implementar o mesmo interesse transversal que a classe interna implementa. Também a troca das chamadas de um construtor por outro não cria ou remove pontos de transição porque ambos estão relacionados ao mesmo interesse transversal e, portanto, suas chamadas terão o mesmo sombreamento.

LOC: 5 a $(4 * n_{trechos} + 5 + 6 * n_{atrib.ext})$
Dos passos básicos que podem alterar o número de linhas de código do software, apenas "Trocar chamada a operação sobrecarregada por versão com mais parâmetros" não o faz, pois o valor extra a ser passado é a variável `this` e, por isso, não serão necessários novos comandos para obter este novo valor.

Os demais passos básicos sempre aumentam o número de linhas de código quando são usados: *Extrair Método* aumenta de 3 a 4 unidades cada vez que for aplicado, "Adicionar atributo a componente" aumenta uma unidade, "Criar operação vazia" aumenta duas unidades, "Adicionar chamada a operação" e "Adicionar linha de leitura/escrita de valor de atributo a operação" aumentam uma unidade cada, e *Auto Encapsular Atributo* aumenta 6 unidades a cada aplicação. Multiplicando os aumentos pelo número de vezes que cada passo básico é usado, e levando em consideração que *Extrair Método* e *Auto Encapsular Atributo* podem não ser necessários, tem-se um total de no mínimo 5 e no máximo $4 * n_{trechos} + 5 + 6 * n_{atrib.ext}$ linhas de código adicionadas.

NOA: +1
Apenas um passo básico pode modificar o número de atributos dos componentes do sistema ("Adicionar atributo a componente", executado no passo 2). Como este passo é obrigatório e é sempre executado quando esta refatoração é usada, o número de atributos dos componentes do software certamente aumenta uma unidade.

WOC: $(n_{param.const} + 1)$ a $(n_{param.const} + 1 + \sum_{i=1}^{n_{trechos}} p_i + 6 * n_{atrib.ext})$
Esta refatoração pode adicionar novas operações tanto na classe interna (o construtor) quanto na classe externa (métodos extraídos e métodos de acesso dos atributos encapsulados). Assim, ambos podem ter seus valores de WOC alterados.

Na classe interna, é criado um construtor com $n_{param.const}$ parâmetros, o que resulta em um aumento de $n_{param.const} + 1$ unidades. O construtor obrigatoriamente será construído, portanto o valor de WOC da classe interna certamente aumentará.

Na classe externa, dois passos básicos opcionais criam novas operações: *Extrair Método* e *Auto Encapsular Atributo*. O primeiro pode extrair até $n_{trechos}$ métodos, cada um com zero ou mais parâmetros. Supondo que cada método m_i possui p_i parâmetros, $1 \leq i \leq n_{trechos}$ e $p_i \geq 0$, então o aumento total máximo de WOC feito por este passo é de $\sum_{i=1}^{n_{trechos}} p_i$. Já *Auto Encapsular Atributo* aumenta o valor de WOC em seis unidades para cada um dos $n_{atrib.ext}$ atributos encapsulados, totalizando uma variação de $6 * n_{atrib.ext}$.

A variação total de WOC causada por esta refatoração, portanto, estará no intervalo de $(n_{param.const} + 1)$ a $(n_{param.const} + 1 + \sum_{i=1}^{n_{trechos}} p_i + 6 * n_{atrib.ext})$.

CBC: +2
Todos os componentes referenciados originalmente nas classes externa e interna continuarão sendo referenciados após a extração da classe, porém o acoplamento da classe externa com a classe interna (e vice-versa) não era considerado no cálculo de CBC porque um componente estava definido dentro do outro. Ao extrair a classe interna, o acoplamento de ambas passa a ser contado e o valor de CBC aumenta duas unidades (um acoplamento a mais na classe externa e outro na classe interna).

LCOO: -X a +X
Esta refatoração pode alterar o valor de LCOO tanto da classe interna quanto da classe externa. Na classe interna, uma nova operação é obrigatoriamente criada (o construtor

adicionado no passo 3), operação essa que acessa apenas um dos atributos da classe (o atributo criado no passo 2). Como nenhuma outra operação da classe acessa esse mesmo atributo, todos os pares de operações criados com a adição do construtor são contados na parcela positiva de LCOO (operações sem atributos compartilhados), aumentando assim o valor desta métrica.

Já na classe externa, se for necessário usar *Extrair Método* o valor de LCOO poderá aumentar ou diminuir, dependendo das operações pré-existentes na classe e dos atributos acessados nos trechos extraídos. No caso de ser usada *Auto Encapsular Atributo*, o valor da métrica pode apenas aumentar.

Assim, somando todos os impactos que as alterações desta refatoração podem ter, o valor de LCOO poderá variar tanto positiva quanto negativamente. O quanto esta métrica variará depende das alterações que tiverem que ser executadas pela refatoração e das características (atributos acessados nas operações criadas e pré-existentes) dos componentes envolvidos.

Extração de várias classes internas iguais:
Caso existam diversas classes internas iguais a serem extraídas, todas elas serão substituídas por uma única classe autônoma. Esta diferença para o caso da extração de apenas uma classe interna modifica as possíveis conseqüências na maioria das métricas. Não será mais possível afirmar, por exemplo, que os passos básicos "Criar cópia autônoma de componente interno" e "Remover cópia interna de componente autônomo" têm seus impactos anulados um pelo outro, já que o número de vezes que cada um é executado não é igual. A própria mecânica da refatoração possui uma etapa que não era executada no caso anterior: a extração da interface comum usada nas classes internas (passo 2).

Assim, para fazer a avaliação dos impactos desta refatoração para o caso em que diversas classes interna iguais são extraídas, é necessário levar em conta (1) a avaliação feita para o caso da extração de uma única classe, (2) as etapas que são necessárias apenas para extrair diversas classes – no caso, criar uma interface comum a todas as classes externas, tarefa que é feita pelos passos básicos "Criar classe ou interface vazia" (1 vez), "Criar operação vazia" ($n_{op.interface}$ vezes), "Adicionar parâmetro a operação" ($n_{param.int}$ vezes), "Adicionar atributo a componente" ($n_{atrib.int}$ vezes) e "Adicionar implementação de interface a componente" ($n_{cl.ext}$ vezes) –, e (3) que os impactos de "Criar cópia autônoma de componente interno" serão anulados por uma execução de "Remover cópia interna de componente autônomo", mas que ainda serão removidas outras $n_{cl.ext} - 1$ cópias internas.

A tabela 5.33 mostra o impacto de cada passo básico usado pela refatoração neste caso. Nesta tabela é possível observar que apenas **DIT** não é modificado por nenhum dos passos, portanto conclui-se que essa métrica não terá seu valor alterado quando *Extrair Classe Interna para Autônoma* for aplicada. A variação total de cada uma das demais métricas é explicada a seguir.

CDC: $n_{cl.ext} - 2$ a $n_{cl.ext} - 1$
A extração de uma única classe interna não modifica o valor de CDC, portanto só será possível modificar este valor se a criação da interface ou o diferente número de execuções de "Remover cópia interna de componente autônomo" tiverem algum impacto sobre essa métrica.

A interface criada no passo 2 e as classes internas serão componentes principais do mesmo interesse transversal, já que a primeira expõe a interface usada nas segundas – e ambas estão de fato relacionadas a um interesse pela própria motivação para aplicar a refatoração: "caso o interesse da classe interna já seja modularizado em um aspecto (ou vá ser), deve-se executar a refatoração *Extrair Classe Interna para Autônoma*". Por causa

Tabela 5.33: Impacto nas métricas de cada passo básico usado por *Extrair Classe Interna para Autônoma* para extrair várias classes internas iguais

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Extrair Método	=	$\geq_{(1)}$	$\leq_{(X)}$	=	$>_{(3a4)}$	=	$>_{(X)}$	=	=	$\gtrless$
Adicionar atributo a componente	$\geq_{(1)}$	=	$\geq_{(2)}$	=	$>_{(1)}$	$>_{(1)}$	=	$\geq_{(1)}$	=	=
Criar classe ou interface vazia	$\geq_{(1)}$	=	=	$>_{(1)}$	$>_{(2)}$	=	=	=	=	=
Criar operação vazia	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(2)}$	=	$>_{(1)}$	$\geq_{(1)}$	=	$\geq_{(X)}$
Adicionar parâmetro a operação	=	=	=	=	=	=	$>_{(1)}$	$\geq_{(1)}$	=	=
Adicionar implementação de interface a componente	$\geq_{(1)}$	=	$\geq_{(2)}$	=	=	=	=	$\geq_{(1)}$	=	=
Adicionar chamada a operação	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(1)}$	=	=	$\geq_{(1)}$	=	=
Adicionar linha de leitura/escrita de valor de atributo a operação	=	=	=	=	$>_{(1)}$	=	=	=	=	$\leq_{(X)}$
Trocar chamada a operação sobrecarregada por versão com mais parâmetros	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$\geq_{(X)}$	=	=	$\geq_{(X)}$	=	=
Compilar e testar	=	=	=	=	=	=	=	=	=	=
Auto Encapsular Atributo	=	$\geq_{(1)}$	$\leq_{(X)}$	=	$>_{(3a4)}$	=	$>_{(X)}$	=	=	$\gtrless$
Tornar operação pública	=	=	=	=	=	=	=	=	=	=
Tornar operação protegida	=	=	=	=	=	=	=	=	=	=
Criar cópia autônoma de componente interno	$\geq_{(1)}$	$\geq_{(X)}$	$\geq_{(X)}$	$>_{(1)}$	$>_{(X)}$	$\geq_{(X)}$	$\geq_{(X)}$	$\geq_{(X)}$	=	$\geq_{(X)}$
Tornar componente público	=	=	=	=	=	=	=	=	=	=
Adicionar importações	=	=	=	=	=	=	=	=	=	=
Remover cópia interna de componente autônomo	$\leq_{(1)}$	$\leq_{(X)}$	$\leq_{(X)}$	$<_{(1)}$	$<_{(X)}$	$\leq_{(X)}$	$\leq_{(X)}$	$\leq_{(X)}$	=	$\leq_{(X)}$
Remover importações	=	=	=	=	=	=	=	=	=	=

disso, a criação da interface aumentará em uma unidade o número de componentes pelos quais o interesse está disperso, o que modifica o valor de CDC desse interesse na mesma medida.

No entanto, todas as classes internas a serem extraídas serão unificadas em uma única classe autônoma, o que diminui o número dessas classes em $n_{cl.ext} - 1$ unidades. Como as classes internas estão relacionadas ao mesmo interesse transversal da interface criada no passo 2, essa diminuição do número das classes internas reduz o número de componentes pelos quais o interesse está disperso na mesma medida, resultando assim em uma diminuição do valor de CDC de $n_{cl.ext} - 2$ unidades. É importante notar que se forem extraídas duas classes internas iguais e for necessário criar a interface, o valor de CDC não será alterado, pois a redução do número de classes internas iguais será compensada com a criação da interface comum a elas.

CDO: $(-(n_{op.int} * (n_{cl.ext} - 1)) + 1)$ a $(-(n_{op.int} * (n_{cl.ext} - 1)) + (n_{trechos} + n_{ac.int} + n_{op.interface} + 3))$
Dois interesses transversais podem ter seu valor de CDO aumentado ao extrair apenas uma classe interna: o interesse da classe externa ($(n_{trechos} + n_{ac.int} + 2)$ unidades) – se esse interesse existir – e o da classe interna (1 unidade). Ao extrair diversas classes de uma vez, pode ser necessário criar a interface comum a elas e serão removidas diversas classes internas ao invés de apenas uma, modificações essas que lidam apenas com o interesse transversal da classe interna. Assim, o aumento do valor de CDO do interesse transversal da classe externa será o mesmo para a extração de múltiplas classes.

Em relação ao interesse transversal da classe interna, serão criadas $n_{op.interface}$ novas operações principais do mesmo se a interface tiver que ser criada no passo 2. Com isso, o aumento de CDO passará de 1 para $n_{op.interface} + 1$ unidades.

Já a remoção das classes internas no passo 10 eliminará as operações das $n_{cl.ext} - 1$ classes removidas – na verdade $n_{cl.ext}$ classes, porém com a criação da classe autônoma

no passo 8 o saldo de classes internas iguais removidas será $n_{cl.ext} - 1$. Supondo que cada classe interna tenha $n_{op.int}$ operações, a remoção total de operações principais do interesse transversal será de $n_{op.int} * (n_{cl.ext} - 1)$ unidades.

Para descobrir se o valor final de CDO será maior ou menor que seu valor inicial, é necessário saber se o número de operações principais removidas é superior ao número de operações principais criadas, por isso conclui-se que esta refatoração pode aumentar ou diminuir a dispersão dos interesses transversais pelas operações do software dependendo do código em que ela é aplicada.

CDLOC: $-(n_{cl.ext} * (2 * n_{trechos} + 2 + n_{pts}))$ a $(n_{cl.ext} * 2))$
Ao extrair uma única classe interna podem ser removidos de $(2 * n_{trechos} + 2)$ a 2 pontos de transição de dentro do corpo da classe interna. Na extração de várias classes, cada classe extraída terá o mesmo número de pontos de transição removidos, uma vez que o corpo de todas as classes internas são iguais. Assim, para $n_{cl.ext}$ extraídas, são removidos entre $(n_{cl.ext} * 2))$ e $(n_{cl.ext} * (2 * n_{trechos} + 2))$ pontos de transição.

Em relação às modificações específicas do caso em que várias classes são extraídas de uma vez, a criação da interface não cria ou remove pontos de transição do software porque o corpo todo dela é sombreado para o interesse transversal que ela e as classes internas implementam. Já a remoção de várias classes internas pelo passo básico "Remover cópia interna de componente autônomo" diminui ainda mais o valor de CDLOC caso existam pontos de transição nas classes internas removidas, pois com a unificação das $n_{cl.ext}$ classes em uma única classe autônoma os pontos de transição dentro de seus corpos serão eliminados do sistema. Supondo que cada classe interna tenha, antes de sua remoção, n_{pts} pontos de transição, após executar $n_{cl.ext}$ vezes "Remover cópia interna de componente autônomo" terão sido eliminados $n_{cl.ext} * n_{pts}$ pontos de transição.

Somando os impactos dos dois tipos de redução do valor de CDLOC, tem-se que a diminuição máxima desta métrica é de $(n_{cl.ext} * (2 * n_{trechos} + 2)) + (n_{cl.ext} * n_{pts})$ unidades, isto é, $(n_{cl.ext} * ((2 * n_{trechos} + 2) + n_{pts}))$.

VS: $n_{cl.ext} - 2$ a $n_{cl.ext} - 1$
As etapas que podem alterar o valor de VS na extração de várias classes internas são a criação da interface e a unificação das $n_{cl.ext}$ classes internas em uma class autônoma. A criação da interface nem sempre é necessária, porém quando feita aumenta o número de componentes do sistema em uma unidade. Já a unificação das classes internas é etapa obrigatória da refatoração e sempre diminui o número de componentes do sistema em $n_{cl.ext} - 1$ unidades. Sendo assim, a execução desta refatoração poderá diminuir $n_{cl.ext} - 2$ ou $n_{cl.ext} - 1$ o valor de VS. É importante notar que se forem extraídas duas classes internas iguais e for necessário criar a interface, o valor de VS não será alterado, pois a redução do número de classes internas iguais será compensada com a criação da interface comum a elas.

LOC: $(-(n_{cl.ext} - 1) * n_{loc.int} + 5)$ a $((-(n_{cl.ext} - 1) * n_{loc.int}) + n_{op.interface} + n_{atrib.int} + 4 * n_{trechos} + 6 * n_{atrib.ext} + 7)$
Ao extrair uma única classe interna, o valor de LOC aumenta de 5 a $4 * n_{trechos} + 5 + 6 * n_{atrib.ext}$ unidades, mas este valor ainda pode ser alterado pela criação da interface e pela unificação das classes internas. Se a interface tiver de ser criada, serão adicionadas ao sistema 2 linhas para a criação da interface, 1 linha para declarar cada uma das $n_{op.interface}$ operações da interface e 1 linha para cada um dos $n_{atrib.int}$ atributos desse componente, totalizando $2 + n_{op.interface} + n_{atrib.int}$ novas linhas de código. Já para unificar as classes internas, assumindo que cada classe interna tem $n_{loc.int}$ linhas, serão eliminadas do sistema $(n_{cl.ext} - 1) * n_{loc.int}$ linhas de código.

Assim, ao extrair diversas classes internas, o valor de LOC será modificado de $(-(n_{cl.ext}-1)*n_{loc.int}+5)$ a $(-(n_{cl.ext}-1)*n_{loc.int})+(2+n_{op.interface}+n_{atrib.int})+(4*n_{trechos}+5+6*n_{atrib.ext})$, ou seja, esta métrica poderá ter seu valor diminuído ou aumentado pela refatoração.

NOA: $(-(n_{cl.ext}-1)*n_{atrib.cl.int}+1)$ a $(-(n_{cl.ext}-1)*n_{atrib.cl.int}+n_{atrib.int}+1)$
A extração de uma única classe interna aumenta o valor de NOA por causa da adição de um novo atributo à classe extraída. Na extração de várias classes, além deste novo atributo, também podem ser criados $n_{atrib.int}$ atributos na interface a ser implementada pelas classes externas. No entanto, a unificação das classes internas em uma única classe remove os atributos pré-existentes nessas classes. Supondo que cada classe interna tivesse originalmente $n_{atrib.cl.int}$ atributos, a unificação das classes extraídas removerá $(n_{cl.ext}-1)*n_{atrib.cl.int}$ atributos do sistema.

Portanto, a execução desta refatoração pode alterar o número de atributos entre $(-(n_{cl.ext}-1)*n_{atrib.cl.int}+1)$ e $(-(n_{cl.ext}-1)*n_{atrib.cl.int}+n_{atrib.int}+1)$ unidades, ou seja, o valor de NOA pode aumentar ou diminuir dependendo do código em que a refatoração for aplicada.

WOC: $(-((n_{cl.ext}-1)*woc_{inicial})+(n_{param.const}+1))$ a $-(((n_{cl.ext}-1)*woc_{inicial})+(n_{op.interface}+n_{param.int})+(n_{param.const}+1+\sum_{i=1}^{n_{trechos}}p_i+6*n_{atrib.ext}))$
A extração de uma única classe interna aumenta o valor de WOC entre $(n_{param.const}+1)$ e $(n_{param.const}+1+\sum_{i=1}^{n_{trechos}}p_i+6*n_{atrib.ext})$ unidades por causa da criação de operações nas classes externa e interna. Na extração de várias classes, este valor poderá ser modificado de outras duas formas: pela criação da interface comum às classes internas e pela unificação dessas classes em uma classe autônoma. No primeiro caso, se for necessário criar a interface, serão adicionadas a ela $n_{op.interface}$ operações com um total de $n_{param.int}$ parâmetros, o que aumenta o valor de WOC em $n_{op.interface}+n_{param.int}$ unidades. No segundo caso, os valores de WOC de cada classe interna serão eliminados do sistema, diminuindo assim o valor total dessa métrica. Supondo que cada classe tenha um valor de WOC inicial $woc_{inicial}$, a diminuição total dessa métrica pela unificação das classes internas será de $(n_{cl.ext}-1)*woc_{inicial}$ unidades.

Assim, a variação total de WOC na extração de várias classes internas será de $-((n_{cl.ext}-1)*woc_{inicial})+(n_{param.const}+1)$ até $-((n_{cl.ext}-1)*woc_{inicial})+(n_{op.interface}+n_{param.int})+(n_{param.const}+1+\sum_{i=1}^{n_{trechos}}p_i+6*n_{atrib.ext})$. Isso significa que esta refatoração poderá diminuir ou aumentar o valor de WOC dependendo do código em que a refatoração for aplicada.

CBC: $(-(n_{cl.ext}-1)*acop_{cl.int}+2)$ a $(-(n_{cl.ext}-1)*acop_{cl.int}+acop_{int}+2)$
Ao extrair apenas uma classe interna são criados dois acoplamentos entre os componentes envolvidos na refatoração. No caso da extração de diversas classes, tanto a criação da interface no passo 2 quanto a unificação das classes internas em uma única classe autônoma modificam o número de acoplamentos totais do sistema. Se for necessário criar a interface, serão criados acoplamentos entre ela e outros componentes do sistema pelas declarações de suas operações e de seus atributos. Já a unificação das classes internas remove acoplamentos do sistema porque o número de classes internas iguais será reduzido de $n_{cl.ext}$ para 1, portanto os acoplamentos dessas classes com os outros componentes do sistema serão eliminados. Supondo que a interface esteja acoplada a $acop_{int}$ e que cada classe interna tenha $acop_{cl.int}$ acoplamentos, a variação total do valor de CBC ao aplicar esta refatoração será entre $-(n_{cl.ext}-1)*acop_{cl.int}+2$ e $-(n_{cl.ext}-1)*acop_{cl.int}+acop_{int}+2$ unidades. Isso significa que a extração de várias classes internas pode tanto aumentar quanto diminuir o valor de CBC, dependendo do

código em que ela for feita.

LCOO: -X a +X

Na extração de uma única classe interna o valor de LCOO pode tanto aumentar como diminuir, dependendo de características dos componentes envolvidos na extração. A diferença para a extração de várias classes internas se dá pela unificação das classes internas em uma única classe autônoma, pois isso faz com que os valores de LCOO de cada uma das classes seja eliminado do valor total de LCOO do sistema, diminuindo assim este valor. No entanto, essa diminuição pode não compensar o aumento feito pela extração de métodos no passo 1 e pelo encapsulamento de atributos no passo 5 e a refatoração continuar a ter um impacto positivo no valor de LCOO. Por causa disso, LCOO poderá variar tanto positiva quanto negativamente ao aplicar esta refatoração para extrair diversas classes internas.

Análise dos Resultados

Tabela 5.34: Variação das métricas em *Extrair Classe Interna para Autônoma*

	Separação de Interesses			Tamanho				Acoplamento		Coesão
classes extraídas	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
1	=	$>_{(1aX)}$	$<_{(2aX)}$	=	$>_{(X)}$	$>_{(1)}$	$>_{(X)}$	$>_{(2)}$	=	$\gtrless$
+1	$\leq_{(X)}$	$\gtrless$	$<_{(X)}$	$\leq_{(X)}$	$\gtrless$	$\gtrless$	$\gtrless$	$\gtrless$	=	$\gtrless$

Analisando os resultados mostrados na tabela 5.34, é possível observar que os impactos desta refatoração nos valores das métricas são bastante diferentes caso sejam extraídas uma ou várias classes internas. Isso ocorre principalmente porque no segundo caso as classes internas são unificadas em uma única classe autônoma, o que retira das somas totais das métricas as parciais das classes internas removidas.

A extração de uma única classe interna tem, no caso mais desfavorável, uma melhora de uma das métricas da separação de interesses (CDLOC) e uma piora de outra métrica deste atributo (CDO) e dos demais atributos internos do software, enquanto que no melhor caso ocorre um aumento da coesão dos componentes e uma diminuição do entrelaçamento de código (menor CDLOC), porém com o custo de um maior espalhamento de código pelas operações (maior CDO), aumento de tamanho e do acoplamento dos componentes. Isso mostra que mesmo na situação mais favorável a extração de uma classe interna terá conseqüências ruins em alguns atributos internos do software. É importante relembrar, no entanto, que esta refatoração pode ser usada em um processo maior de reorganização do software, o qual poderá reverter posteriormente as perdas temporárias obtidas nesta refatoração.

No caso da extração de diversas classes internas, em uma situação mais favorável esta refatoração melhora todos os atributos internos avaliados, enquanto que no pior caso a coesão e o acoplamento dos componentes pioram e a separação dos interesses e o tamanho pioram em algumas medidas e melhoram em outras. Isso mostra que as conseqüências da extração de múltiplas classes internas são bastante distintas para as diferentes situações em que ela pode ser feita. Ressalta-se, no entanto, que a extração de diversas classes elimina códigos duplicados, o que é sempre um mal-cheiro a ser evitado (FOWLER, 1999).

5.2.8 Internalizar Classe em Aspecto

Quando uma pequena classe autônoma é usada somente por um determinado aspecto, recomenda-se movê-la para dentro dele. Isso pode ser necessário, por exemplo, quando uma ou mais pequenas classes são relacionadas a um interesse transversal que está sendo modularizado em um aspecto. Outro caso onde isso ocorre é durante o processo de migração de uma classe interna originalmente presente em uma classe externa para dentro de um aspecto, após usar *Extrair Classe Interna para Autônoma* (5.2.7). A refatoração *Internalizar Classe em Aspecto* (MONTEIRO, 2005, seção 6.3.5) possui 3 passos para efetuar essa movimentação:

1. Criar uma cópia da classe dentro do aspecto, substituindo "public" por "static" antes do nome da classe;
2. Compilar e testar;
3. Remover a classe independente, então compilar e testar novamente.

Decomposição da Refatoração em Passos Básicos

O passo 1 da refatoração pode ser decomposto em três passos básicos: "Criar cópia interna de componente autônomo" (46), "Substituir `public` por `static` em classe" (98) e "Trocar referência a componente externo por referência a componente interno" (97).

Já o 2o. passo usa apenas o passo básico "Compilar e testar" (101).

Por fim, o último passo remove a classe usando "Remover cópia autônoma de componente interno" (50), e compila e testa o software usando novamente "Compilar e testar" (101).

A tabela 5.35 mostra quantas vezes e onde (em que passo) cada passo básico é utilizado por esta refatoração.

Tabela 5.35: Quantidade de vezes que cada passo básico é usado em *Internalizar Classe em Aspecto*

Passo da refatoração	Quantidade	Passo básico
1	1	Criar cópia interna de componente autônomo
	1	Substituir `public` por `static` em classe
	1	Trocar referência a componente externo por referência a componente interno
2	1	Compilar e testar
3	1	Remover cópia autônoma de componente interno
	1	Compilar e testar

Cálculo da Variação Total das Métricas

Uma vez decompostos os passos da refatoração, deve-se calcular qual o impacto total das alterações nas métricas escolhidas para avaliar a refatoração. A tabela 5.36 mostra o impacto de cada passo básico usado pela refatoração. Nesta tabela é possível observar que apenas **DIT** não é modificada por nenhum dos passos, portanto conclui-se que essa métrica não terá seu valor alterado quando *Internalizar Classe em Aspecto* for aplicada. A variação total de cada uma das demais métricas é explicada a seguir.

Tabela 5.36: Impacto nas métricas de cada passo básico de *Internalizar Classe em Aspecto*

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Criar cópia interna de componente autônomo	$\geq_{(1)}$	$\geq_{(X)}$	$\geq_{(X)}$	$>_{(1)}$	$>_{(X)}$	$\geq_{(X)}$	$\geq_{(X)}$	$\geq_{(X)}$	=	$\geq_{(X)}$
Substituir `public` por `static` em classe	=	=	=	=	=	=	=	=	=	=
Trocar referência a componente externo por referência a componente interno	=	=	=	=	=	=	=	$<_{(1)}$	=	=
Compilar e testar	=	=	=	=	=	=	=	=	=	=
Remover cópia autônoma de componente interno	$\leq_{(1)}$	$\leq_{(X)}$	$\leq_{(X)}$	$<_{(1)}$	$<_{(X)}$	$\leq_{(X)}$	$\leq_{(X)}$	$\leq_{(X)}$	=	$\leq_{(X)}$

CDC e CDO: =

A classe interna criada no passo 1 e a classe autônoma removida no passo 3 são praticamente idênticas – apenas a primeira possui `static` no lugar de `public` em sua declaração, mas esta diferença não tem efeitos sobre as métricas escolhidas para a avaliação. Dessa forma, ambas implementam as mesmas funcionalidades, estão relacionadas aos mesmos interesses transversais e têm as mesmas operações. Conseqüentemente, se CDC e CDO aumentarem com a criação da cópia interna, os valores dessas métricas serão reduzidos na mesma medida quando a cópia autônoma for removida, ou seja, a refatoração não modifica o valor de CDC e CDO.

CDLOC: =

Como as classes interna e autônoma são iguais, ambas possuem o mesmo número de pontos de transição em seu interior. Dessa forma, ao criar a primeira no passo 1 e remover a segunda no passo 3, serão criados pontos de transição no sistema e em seguida o mesmo número de pontos será eliminado, mantendo assim o valor final de CDLOC igual ao seu valor inicial.

É importante notar que não é criado nenhum ponto de transição entre a cópia interna da classe e o aspecto. Isto ocorre porque o sombreamento da classe é o mesmo do aspecto, já que ela é usada apenas pelo aspecto e, portanto, ambos os componentes são necessários para implementar o mesmo interesse transversal.

VS: =

Ao criar a cópia interna da classe no passo 1 o número de componentes do sistema aumenta em uma unidade, porém este número retorna ao seu valor inicial quando a cópia autônoma é removida no passo 3.

LOC e NOA: =

Como as classes interna e autônoma são iguais, ambas têm o mesmo número de linhas de código e atributos. Assim, com a criação da primeira no passo 1 são adicionados ao sistema algumas linhas de código e atributos, porém a remoção da segunda remove o mesmo número de linhas de código e atributos do sistema e, portanto, mantém os valores de LOC e NOA inalterados.

WOC e LCOO: =

As classes interna e autônoma possuem métodos idênticos porque uma é cópia da outra, portanto ambas têm valores de WOC e LCOO iguais. Ao criar a primeira no passo 1 os valores totais de WOC e LCOO do sistema aumentam tanto quanto diminuem com a remoção da segunda no passo 3. Assim, o valor destas métricas também não são modificados por esta refatoração.

CBC: -2 a -1

Pela definição de CBC, o acoplamento a um componente interno não é contado no cál-

culo dessa métrica. Dessa forma, antes da refatoração o acoplamento do aspecto com a classe autônoma e desta classe com o aspecto (se esse acoplamento também existir) eram contados no cálculo de CBC porque uma não estava definida dentro do outro, mas após a refatoração estes acoplamentos não serão mais contados porque a classe usada passou a ser a cópia interna. Com isso, o valor de CBC poderá diminuir uma ou duas unidades.

Tabela 5.37: Variação das métricas em *Internalizar Classe em Aspecto*

Separação de Interesses			Tamanho				Acoplamento		Coesão
CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
=	=	=	=	=	=	=	$<_{(1a2)}$	=	=

Análise dos Resultados

Analisando os resultados mostrados na tabela 5.37, é possível observar que apenas CBC terá seu valor alterado pela refatoração. Isso mostra que os impactos desta refatoração nos atributos de qualidade são pequenos, reduzindo apenas o acoplamento dos componentes. Ainda assim, nota-se que sempre que é possível aplicar esta refatoração os resultados são positivos (diminuição de uma ou duas unidades de CBC), portanto quando houver uma classe usada apenas por um aspecto deve-se usar esta refatoração para movê-la para dentro do aspecto, sem o risco de deteriorar alguma outra métrica.

5.2.9 Internalizar Interface em Aspecto

Quando uma ou mais interfaces são usadas apenas por um aspecto e não são definidas dentro dele, pode ser um sinal que elas estejam no lugar errado. Uma sugestão neste caso é movê-las para dentro do aspecto. A refatoração *Internalizar Interface em Aspecto* (MONTEIRO, 2005, seção 6.3.6) possui três passos para efetuar essa movimentação:

1. Criar dentro do aspecto uma cópia privada da interface;
2. Remover a interface original;
3. Compilar e testar.

Esta refatoração é bastante parecida com *Internalizar Classe em Aspecto* (seção 5.2.8): trata-se em ambos os casos de um componente usado apenas por um aspecto, declarado fora desse aspecto e que deve ser movido para dentro do mesmo. Por causa disso, as decomposições em passos básicos de ambas as refatorações serão bastante similares, e conseqüentemente também o serão os impactos das refatorações nos valores das métricas.

Decomposição da Refatoração em Passos Básicos

O passo 1 da refatoração pode ser decomposto em dois passos básicos: "Criar cópia interna de componente autônomo" (46) e "Trocar referência a componente externo por referência a componente interno" (97).

Já o 2o. passo usa "Remover cópia autônoma de componente interno" (50), enquanto que o último passo usa apenas o passo básico "Compilar e testar" (101).

A tabela 5.38 mostra quantas vezes e onde (em que passo) cada passo básico é utilizado por esta refatoração.

Tabela 5.38: Quantidade de vezes que cada passo básico é usado em *Internalizar Interface em Aspecto*

Passo da refatoração	Quantidade	Passo básico
1	1	Criar cópia interna de componente autônomo
	1	Trocar referência a componente externo por referência a componente interno
2	1	Remover cópia autônoma de componente interno
3	1	Compilar e testar

Cálculo da Variação Total das Métricas

Uma vez decompostos os passos da refatoração, deve-se calcular qual o impacto total das alterações nas métricas escolhidas para avaliar a refatoração. A tabela 5.39 mostra o impacto de cada passo básico usado pela refatoração. Nesta tabela é possível observar que, assim como em *Internalizar Classe em Aspecto*, apenas **DIT** não é modificada por nenhum dos passos, portanto conclui-se que essa métrica não terá seu valor alterado quando *Internalizar Interface em Aspecto* for aplicada. A variação total de cada uma das demais métricas é explicada a seguir.

Tabela 5.39: Impacto nas métricas de cada passo básico de *Internalizar Interface em Aspecto*

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Criar cópia interna de componente autônomo	$\geq_{(1)}$	$\geq_{(X)}$	$\geq_{(X)}$	$>_{(1)}$	$>_{(X)}$	$\geq_{(X)}$	$\geq_{(X)}$	$\geq_{(X)}$	=	$\geq_{(X)}$
Trocar referência a componente externo por referência a componente interno	=	=	=	=	=	=	=	$<_{(1)}$	=	=
Remover cópia autônoma de componente interno	$\leq_{(1)}$	$\leq_{(X)}$	$\leq_{(X)}$	$<_{(1)}$	$<_{(X)}$	$\leq_{(X)}$	$\leq_{(X)}$	$\leq_{(X)}$	=	$\leq_{(X)}$
Compilar e testar	=	=	=	=	=	=	=	=	=	=

CDC e CDO: =

A interface interna criada no passo 1 e a interface autônoma removida no passo 2 são idênticas. Dessa forma, ambas estão relacionadas aos mesmos interesses transversais e têm as mesmas operações. Conseqüentemente, se CDC e CDO aumentarem com a criação da cópia interna, os valores dessas métricas serão reduzidos na mesma medida quando a cópia autônoma for removida, ou seja, a refatoração não modifica o valor de CDC e CDO.

CDLOC: =

Como as interfaces interna e autônoma são iguais, ambas possuem o mesmo número de pontos de transição em seu interior. Dessa forma, ao criar a primeira no passo 1 e remover a segunda no passo 2, serão criados pontos de transição no sistema e em seguida o mesmo número de pontos será eliminado, mantendo assim o valor final de CDLOC igual ao seu valor inicial.

É importante notar que não é criado nenhum ponto de transição entre a cópia interna da interface e o aspecto. Isto ocorre porque o sombreamento da interface é o mesmo do aspecto, já que ela é usada apenas pelo aspecto e, portanto, ambos os componentes são necessários para implementar o mesmo interesse transversal.

VS: =

Ao criar a cópia interna da interface no passo 1 o número de componentes do sistema

aumenta em uma unidade, porém este número retorna ao seu valor inicial quando a cópia autônoma é removida no passo 2.

LOC e NOA: =

Como as interfaces interna e autônoma são iguais, ambas têm o mesmo número de linhas de código e atributos. Assim, com a criação da primeira no passo 1 são adicionados ao sistema algumas linhas de código e atributos, porém a remoção da segunda remove o mesmo número de linhas de código e atributos do sistema e, portanto, mantém os valores de LOC e NOA inalterados.

WOC: =

As interfaces interna e autônoma possuem métodos idênticos porque uma é cópia da outra, portanto ambas têm valor de WOC igual. Ao criar a primeira no passo 1 o valor total de WOC do sistema aumenta tanto quanto diminui com a remoção da segunda no passo 2. Assim, o valor desta métrica também não é modificados por esta refatoração.

CBC: -2 a -1

Pela definição de CBC, o acoplamento a um componente interno não é contado no cálculo dessa métrica. Dessa forma, antes da refatoração o acoplamento do aspecto com a interface autônoma e desta interface com o aspecto (se esse acoplamento também existir) eram contados no cálculo de CBC porque uma não estava definida dentro do outro, mas após a refatoração estes acoplamentos não serão mais contados porque a interface usada passou a ser a cópia interna. Com isso, o valor de CBC poderá diminuir uma ou duas unidades.

LCOO: =

Como esta refatoração trata apenas de modificações de operações de interfaces, nenhuma operação concreta é modificada, logo os pares de operações dos componentes do sistema não se alteram e, portanto, o valor de LCOO se mantém o mesmo.

Tabela 5.40: Variação das métricas em *Internalizar Interface em Aspecto*

Separação de Interesses			Tamanho				Acoplamento		Coesão
CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
=	=	=	=	=	=	=	$<_{(1a2)}$	=	=

Análise dos Resultados

Analisando os resultados mostrados na tabela 5.40, é possível observar que apenas CBC terá seu valor alterado pela refatoração. Isso mostra que os impactos desta refatoração nos atributos de qualidade são pequenos, reduzindo apenas o acoplamento dos componentes. Ainda assim, nota-se que sempre que é possível aplicar esta refatoração os resultados são positivos (diminuição de uma ou duas unidades de CBC), portanto quando houver uma interface usada apenas por um aspecto deve-se usar esta refatoração para movê-la para dentro do aspecto, sem o risco de deteriorar alguma outra métrica.

5.2.10 Particionar Assinatura de Construtor

Durante a extração de um interesse para um aspecto, pode ser que algum construtor possua parâmetros que sejam desnecessários caso o interesse não esteja presente no sistema. Neste caso, a refatoração *Particionar Assinatura de Construtor* (MONTEIRO, 2005, seção 6.6.1) sugere criar um construtor na classe sem nenhum código relacionado ao interesse e substituir a parte do código não relacionada ao interesse no construtor ori-

ginal por uma chamada ao novo construtor, além de mover o construtor alterado para o aspecto.

Esta solução é necessária quando a interface da classe não pode ser modificada ou quando se deseja evitar impactos na código do cliente, sendo por isso classificada por Monteiro (2005) como “uma refatoração para lidar com código legado”.

Os passos desta refatoração são os seguintes:

1. Criar um novo construtor na classe apenas com os parâmetros sem relação com o interesse transversal;
2. Mover para o novo construtor todos os comandos não-relacionados ao interesse;
3. Criar uma chamada a ele como primeiro comando no construtor original, passando apenas os parâmetros não-relacionados ao interesse;
4. Mover o construtor original modificado para o aspecto;
5. Adicionar `.new` depois do nome do construtor movido no passo anterior, caracterizando uma declaração inter-tipo relativa ao construtor da classe;
6. Compilar e testar.

Exemplo:

Para mostrar o funcionamento desta refatoração, Monteiro (2005, p.138) apresenta o exemplo de uma pilha entrelaçada ao interesse de representá-la visualmente em uma janela:

```
public class TangledStack {
   (...)
   public TangledStack(JFrame frame) {
       elements = new Object[S_SIZE];
       frame.getContentPane().add(_label);
       _text.setText("[]");
       frame.getContentPane().add(_text);
   }
   (...)
}
```

```
public aspect WindowView {
   (...)
}
```

⇓

```
public class TangledStack {
   (...)
   public TangledStack(~~JFrame frame~~) {
       elements = new Object[S_SIZE];
       ~~frame.getContentPane().add(_label);~~
       ~~_text.setText("[]");~~
       ~~frame.getContentPane().add(_text);~~
   }
   (...)
}
```

```
public aspect WindowView {
   public TangledStack.new(JFrame frame) {
       this();
       frame.getContentPane().add(_label);
       _text.setText("[]");
       frame.getContentPane().add(_text);
   }
   (...)
}
```

Decomposição da Refatoração em Passos Básicos

Para executar o primeiro passo da refatoração, usa-se uma vez o passo básico "Criar operação vazia" (58) para criar o construtor, e "Adicionar parâmetro a operação" (60) $n_{param.nao.rel}$ vezes para adicionar os $n_{param.nao.rel}$ parâmetros não-relacionados ao interesse transversal.

O 2o. passo move os comandos sem relação com o interesse para o construtor criado no passo anterior através do passo básico "Mover código entre operações do mesmo componente" (67). Este passo básico deve ser usado tantas vezes quanto forem os $n_{trechos}$ trechos contínuos de comandos não-relacionados ao interesse.

No passo 3 é adicionada no construtor original uma chamada ao novo construtor, alteração que é feita pelo passo básico "Adicionar chamada a operação" (79).

Os passos 4 e 5 fazem, juntos, uma modificação só: transformar o novo construtor em uma declaração inter-tipo. Isso é feito (1) criando a declaração inter-tipo de operação no aspecto e (2) removendo o construtor da classe. A primeira etapa usa uma vez o passo básico "Adicionar declaração inter-tipo de operação" (87) para copiar a definição do construtor dentro do aspecto, $n_{param.rel}$ vezes o passo básico "Adicionar parâmetro a operação" (60) para criar os parâmetros necessários à declaração inter-tipo, n_{var} vezes "Adicionar variável local a operação" (62) para incluir as variáveis usadas e, por fim, usa uma vez "Adicionar linhas de código a operação" (76) para preencher o corpo da declaração inter-tipo. A segunda etapa é feita pelos seguintes passos básicos: "Remover linhas de código de operação" (78) para eliminar os comandos do construtor; "Remover variável local de operação" (62) para apagar cada uma das n_{var} variáveis do construtor; "Remover parâmetro de operação" (61) para eliminar os $n_{param.rel}$ parâmetros usados no construtor; e, finalmente, "Remover operação vazia" (59) para retirar o construtor da classe.

Por fim, o passo 6 compila e testa o programa usando o passo básico "Compilar e testar" (101).

A tabela 5.41 mostra quantas vezes e onde (em que passo) cada passo básico é utilizado por esta refatoração.

Tabela 5.41: Quantidade de vezes que cada passo básico é usado em *Particionar Assinatura de Construtor*

Passo da refatoração	Quantidade	Passo básico
1	1	Criar operação vazia
	$n_{param.nao.rel}$	Adicionar parâmetro a operação
2	$n_{trechos}$	Mover código entre operações do mesmo componente
3	1	Adicionar chamada a operação
4 e 5	1	Adicionar declaração inter-tipo de operação
	$n_{param.rel}$	Adicionar parâmetro a operação
	n_{var}	Adicionar variável local a operação
	1	Adicionar linhas de código a operação
	1	Remover linhas de código de operação
	n_{var}	Remover variável local de operação
	$n_{param.rel}$	Remover parâmetro de operação
	1	Remover operação vazia
6	1	Compilar e testar

Cálculo da Variação Total das Métricas

Uma vez decompostos os passos da refatoração, deve-se calcular qual o impacto total das alterações nas métricas escolhidas para avaliar a refatoração. A tabela 5.42 mostra o impacto de cada passo básico usado pela refatoração. Nesta tabela é possível observar que as métricas **VS**, **NOA** e **DIT** não são modificadas por nenhum dos passos, portanto conclui-se que essas métricas não terão seu valor alterado quando *Particionar Assinatura de Construtor* for aplicada. A variação total de cada uma das demais métricas é explicada a seguir.

Tabela 5.42: Impacto nas métricas de cada passo básico de *Particionar Assinatura de Construtor*

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Criar operação vazia	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(2)}$	=	$>_{(1)}$	$\geq_{(1)}$	=	$\geq_{(X)}$
Adicionar parâmetro a operação	=	=	=	=	=	=	$>_{(1)}$	$\geq_{(1)}$	=	=
Mover código entre operações do mesmo componente	=	$\gtrless_{(-1a1)}$	$\gtrless_{(-2a2)}$	=	=	=	=	=	=	$\gtrless$
Adicionar chamada a operação	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(1)}$	=	=	$\geq_{(1)}$	=	=
Adicionar declaração inter-tipo de operação	=	$>_{(1)}$	=	=	$>_{(2)}$	=	$>_{(1)}$	$\geq_{(1)}$	=	$\geq_{(X)}$
Adicionar variável local a operação	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(1)}$	=	=	$\geq_{(1)}$	=	=
Adicionar linhas de código a operação	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(X)}$	=	$>_{(X)}$	=	=	$\geq_{(X)}$	=	$\leq_{(X)}$
Remover linhas de código de operação	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(X)}$	=	$<_{(X)}$	=	=	$\leq_{(X)}$	=	$\geq_{(X)}$
Remover variável local de operação	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(2)}$	=	$<_{(1)}$	=	=	$\leq_{(1)}$	=	=
Remover parâmetro de operação	=	=	=	=	=	=	$<_{(1)}$	$\leq_{(1)}$	=	=
Remover operação vazia	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(2)}$	=	$<_{(2)}$	=	$<_{(1)}$	$\leq_{(1)}$	=	$\leq_{(X)}$
Compilar e testar	=	=	=	=	=	=	=	=	=	=

CDC: -1 a 0

Esta refatoração transfere o código do construtor (parâmetros e comandos) relacionado ao interesse transversal para o aspecto que implementa esse interesse. Se ao final da refatoração não houver na classe nenhum local relacionado ao interesse, então a refatoração terá removido a relação da classe com o interesse, diminuindo assim o número de componentes pelas quais este interesse transversal está disperso. Como resultado, o valor de CDC diminuirá uma unidade.

CDO: =

Como a refatoração retira o código do construtor relacionado ao interesse transversal, esta operação deixará de estar relacionada ao interesse. No entanto, a refatoração cria uma nova operação no aspecto para a qual será transferido o código removido do construtor original: o novo construtor da classe, definido em uma declaração inter-tipo. Assim, enquanto uma operação deixa de estar relacionada ao interesse, outra passa a possuir essa relação, mantendo inalterado o número de operações pelas quais o interesse transversal está disperso e, conseqüentemente, não modificando o valor de CDO.

CDLOC: $-2 * n_{trechos}$

Originalmente o construtor da classe possuía algumas linhas sombreadas, as quais foram removidas para a declaração inter-tipo por esta refatoração. Para cada trecho movido do construtor original para o novo construtor no passo 2, serão removidos dois pontos de transição – um no início e outro no final do trecho movido. Isso diminui o valor de CDLOC em $2 * n_{trechos}$ unidades.

LOC: +3

As linhas de código do construtor original da classe serão divididas entre o construtor que

for transferido para o aspecto e o construtor que permanecer na classe. Assim, as únicas modificações que alteram o número de linhas de código do sistema são a criação do novo construtor (aumento de duas unidades) e a adição da chamada a esse construtor (aumento de uma unidade). Somando as alterações parciais, obtém-se um aumento do valor de LOC de três unidades.

WOC: $n_{param.nao.rel} + 1$

Todas as operações existentes originalmente continuarão existindo e terão o mesmo número de parâmetros após a refatoração, apesar de uma delas ter sua localização alterada. Além disso, é criada uma nova operação com $n_{param.nao.rel}$ parâmetros: o construtor sem relação com o interesse transversal. Essa nova operação adicionará uma parcela ao cálculo de WOC de valor $n_{param.nao.rel}+1$, o que aumenta o valor total desta métrica nesta mesma magnitude.

CBC: $-n_{acopl.cl}$ a $n_{acopl.asp}$

A criação do novo construtor na classe não cria nenhum acoplamento porque todos os componentes usados na composição desse novo construtor já são usados no construtor original – seus parâmetros são um subconjunto dos parâmetros do construtor original e seus comandos foram movidos deste outro construtor. No entanto, ao mover o construtor original para uma declaração inter-tipo no aspecto, poderão ser removidos acoplamentos da classe e criados acoplamentos no aspecto. Isso ocorre, respectivamente, se algum dos componentes usados na definição do construtor não for mais usado no restante da classe, e se o aspecto originalmente não referenciar algum dos componentes usados no código movido.

Supondo que a classe perca $n_{acopl.cl}$ acoplamentos e o aspecto ganhe $n_{acopl.asp}$ acoplamentos, a variação de CBC é dada por $n_{acopl.asp} - n_{acopl.cl}$. Isso significa que esta refatoração pode diminuir CBC em até $-n_{acopl.cl}$ unidades (nenhum acoplamento criado no aspecto) ou pode aumentar CBC em até $n_{acopl.asp}$ unidades (nenhum acoplamento removido da classe).

LCOO: -X a +X

Esta refatoração retira do construtor da classe as linhas de código relacionadas ao interesse transversal do aspecto. Se nessas linhas algum atributo da classe for acessado e não houver outra linha no construtor que acesse esse mesmo atributo, e se outra operação da classe compartilhar apenas esse atributo com o construtor, então o par formado por essas duas operações deixará de ser contado na parcela negativa (pares de operações com atributos em comum) e passará a ser contado na parcela positiva (pares de operações que não compartilham atributos) de LCOO. Para cada uma das $n_{op.cl}$ operações da classe em que isso ocorre, o valor desta métrica aumentará duas unidades, totalizando um aumento de $2 * n_{op.cl}$ unidades.

No aspecto, a refatoração transfere o construtor original da classe. Essa operação formará novos pares com as demais operações do aspecto, pares esses que poderão aumentar ou diminuir o valor de LCOO dependendo de sua classificação: se forem criados mais pares de operações que compartilham atributos do que pares de operações sem atributos em comum, então o valor de LCOO diminui; caso contrário, aumenta. O quanto essa métrica é alterada depende das operações pré-existentes no aspecto e dos atributos acessados nessas operações e no construtor movido.

Análise dos Resultados

Analisando os resultados mostrados na tabela 5.43, é possível observar que esta refatoração diminui a separação dos interesses, tendo como custo o aumento do tamanho do

Tabela 5.43: Variação das métricas em *Particionar Assinatura de Construtor*

Separação de Interesses			Tamanho				Acoplamento		Coesão
CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
$\leq_{(1)}$	=	$<_{(X)}$	=	$>_{(3)}$	=	$>_{(X)}$	$\gtrless$	=	$\gtrless$

software. Além disso, a coesão e o acoplamento dos componentes poderão melhorar ou piorar dependendo das características dos componentes envolvidos na refatoração. Na situação mais favorável, são melhorados três atributos internos (separação de interesses, acoplamento e coesão), enquanto que na situação mais adversa essa refatoração melhora apenas a separação de interesses e piora os outros três atributos internos. Isso mostra que de fato a refatoração melhora a dispersão dos interesses transversais, mas aponta possíveis conseqüências indesejáveis que podem ocorrer: aumento do tamanho e acoplamento, e diminuição da coesão.

5.2.11 Extrair Funcionalidade para Aspecto

Quando o código relacionado a uma funcionalidade está espalhado por diversas classes e operações e entrelaçado a código não-relacionado a ela, recomenda-se extrair todos os elementos relacionados a esta funcionalidade para um aspecto, de forma a aproveitar melhor as novas formas de modularização e composição que a OA proporciona. Com essa finalidade, Monteiro (2005, seção 6.3.2) propõe a refatoração *Extrair Funcionalidade para Aspecto*, a qual possui os seguintes passos:

1. Criar um aspecto vazio no pacote adequado;

2. Se houver classes internas relacionadas ao interesse extraído, usar *Extrair Classe Interna para Autônoma* (seção 5.2.7) para extraí-las;

3. Para cada atributo do interesse extraído, aplicar *Move Field From Class to Inter-type* (seção 5.2.4). Como campos normalmente são privados, talvez seja necessário declarar o aspecto temporariamente como privilegiado a fim de manter o código compilável e testável;

4. Os trechos de inicialização presente em construtores devem ser movidos por *Extrair Fragmento para Adendo* (seção 5.2.2). Caso algum parâmetro do construtor seja usado nestes trechos, deve-se primeiro reestruturar essa parte do código: se for possível substituir o parâmetro por uma chamada a um método separado, então deve-se usar *Extrair Método* (seção 5.1.1); caso não seja possível fazer essa substituição ou se o construtor fizer parte de uma interface publicada que não pode ser mudada, aplicar *Particionar Assinatura de Construtor* (seção 5.2.10);

5. Para cada método relativo ao interesse extraído, usar *Mover Método de Classe para Inter-tipo* (seção 5.2.3);

6. Quando apenas parte do método for relacionada ao interesse, existem duas alternativas: (1) usar *Extrair Método* e então aplicar *Mover Método de Classe para Inter-tipo*, ou (2) usar apenas *Extrair Fragmento para Adendo*. Se o fragmento usar um parâmetro do método de origem, é mais simples usar esta última opção;

7. Aplicar *Internalizar Classe em Aspecto* (seção 5.2.8) para cada classe que for usada apenas dentro do aspecto – como as classes extraídas no passo 2. Da mesma forma,

aplicar *Internalizar Interface em Aspecto* (seção 5.2.9) para cada interface que não seja mais usada fora do aspecto;

8. Trocar para privado o tipo de acesso de todos os membros do aspecto que são agora usados apenas dentro deste componente;

9. Tornar o aspecto não-privilegiado assim que ele deixar de acessar membros não-públicos do resto do código.

Após aplicar esses passos e extrair todos os elementos relativos ao interesse, é possível melhorar a estrutura interna do aspecto resultante aplicando *Tidy Up Internal Aspect Structure* (MONTEIRO, 2005, seção 6.4.6). Ou seja, nas palavras de Monteiro (2005), "nem tudo é feito assim que um interesse transversal é movido para um aspecto".

Exemplo:

Para mostrar as transformações que ocorrem ao aplicar esta refatoração, será reproduzido aqui um exemplo dado por Monteiro (2005, pp. 92-94). Trata-se de uma classe que implementa uma pilha com dois interesses transversais: (1) visualização do estado da pilha em uma janela e (2) verificação de pré-condições.

A classe original é mostrada abaixo, com os interesses transversais destacados em cores diferentes:

```
public class TangledStack {
    private int _top = -1;
    private Object[] _elements;
    private final int S_SIZE = 10;
    private JLabel _label = new JLabel("Stack ");
    private JTextField _text = new JTextField(20);
    public TangledStack(JFrame frame) {
        _elements = new Object[S_SIZE];
        frame.getContentPane().add(_label);
        _text.setText("[]");
        frame.getContentPane().add(_text);
    }
    public String toString() {
        StringBuffer result = new StringBuffer("[");
        for(int i=0;i<=_top;i++) {
            result.append(_elements[i].toString());
            if(i!=_top)
                result.append(", ");
        }
        result.append("]");
        return result.toString();
    }
    private void display() {
        _text.setText(toString());
    }
    public void push(Object element) {
        if(isFull())
            throw new PreConditionException("push when stack full.");
        _elements[++_top] = element;
        display();
    }
    public void pop() {
        if(isEmpty())
            throw new PreConditionException("pop when stack empty.");
        _top--;
        display();
    }
    public Object top() {
        if(isEmpty())
            throw new PreConditionException("top when stack empty.");
        return _elements[_top];
    }
    public boolean isFull() {
        return (_top == S_SIZE-1);
    }
    public boolean isEmpty() {
        return (_top<0);
    }
}
```

Após aplicar *Extrair Funcionalidade para Aspecto* duas vezes – uma para cada interesse transversal extraído –, são retirados da classe os trechos relativos aos dois interesses. Como resultado, a classe original é alterada e transforma-se na seguinte classe:

```
public class TangledStack {
    private int _top = -1;
    private Object[] _elements;
    private final int S_SIZE = 10;
    public TangledStack() {
        _elements = new Object[S_SIZE];
    }
    public String toString() {
        StringBuffer result = new StringBuffer("[");
        for(int i=0;i<=_top;i++) {
            result.append(_elements[i].toString());
            if(i!=_top)
                result.append(", ");
        }
        result.append("]");
        return result.toString();
    }
    public void push(Object element) {
        _elements[++_top] = element;
    }
    public void pop() {
        _top--;
    }
    public Object top() {
        return _elements[_top];
    }
    public boolean isFull() {
        return (_top == S_SIZE-1);
    }
    public boolean isEmpty() {
        return (_top<0);
    }
}
```

Já os dois interesses transversais foram extraídos para os aspectos `WindowView` e `PreConditionChecking`, criados durante a refatoração, e cujos códigos-fonte são mostrados abaixo.

```
public aspect WindowView {
    private JLabel TangledStack._label = new JLabel("Stack ");
    private JTextField TangledStack._text = new JTextField(20);
    public TangledStack.new(JFrame frame) {
        this();
        frame.getContentPane().add(_label);
        _text.setText("[]");
        frame.getContentPane().add(_text);
    }
    private void TangledStack.display() {
        _text.setText(toString());
    }
    pointcut stateChange(TangledStack stack):
        (execution(public void stack.TangledStack.push(Object))
        ||
        execution(public void stack.TangledStack.pop()))
        && this(stack);
    after(TangledStack _this) returning :
        stateChange(_this) {
        _this.display();
    }
}
```

```
public aspect PreConditionChecking {
    pointcut checkPush(TangledStack stack):
        execution(public void TangledStack.push(Object))
            && this(stack);
    before(TangledStack _this): checkPush(_this) {
        if(_this.isFull())
            throw new PreConditionException("push when stack full");
    }
    pointcut checkPop(TangledStack stack):
        execution(public void TangledStack.pop())
            && this(stack);
    before(TangledStack _this): checkPop(_this) {
        if(_this.isEmpty())
            throw new PreConditionException("pop when stack empty");
    }
    pointcut checkTop(TangledStack stack):
        execution(public Object TangledStack.top())
            && this(stack);
    before(TangledStack _this): checkTop(_this) {
        if(_this.isEmpty())
            throw new PreConditionException("top when stack empty");
    }
}
```

Decomposição da Refatoração em Passos Básicos

O primeiro passo da refatoração usa o passo básico "Criar aspecto vazio" (39) para criar o aspecto para onde as funcionalidades do interesse transversal serão extraídas.

No passo 2, as $n_{cl.int}$ classes internas que tiverem que ser extraídas de outras classes serão movidas pela refatoração *Extrair Classe Interna para Autônoma* (149).

Já o 3o. passo extrai os n_{atrib} atributos usando *Move Field From Class to Inter-type* (134), uma vez para cada atributo. Se for necessário tornar o aspecto temporariamente privilegiado, então usa-se o passo básico "Tornar aspecto privilegiado" (93) para fazer isso.

No passo 4 são extraídos os trechos relacionados ao interesse localizados em construtores de classes. Para cada um dos n_{cons} construtores com trechos relacionados ao interesse, existem três possibilidades: se o trecho não usar nenhum parâmetro do construtor, basta usar *Extrair Fragmento para Adendo* (124); senão, deve-se tentar substituir a passagem dos parâmetros por uma chamada ao método extraído por *Extrair Método* (104) ou aplicar *Particionar Assinatura de Construtor* (166) para separar a parte relacionada ao interesse transversal em outro construtor. Cada uma dessas refatorações pode ser usada até n_{cons} vezes.

O passo 5 move os n_{met} métodos relacionados ao interesse transversal da classe para o aspecto, através do uso de *Mover Método de Classe para Inter-tipo* (130), uma vez para cada método.

No 6o. passo, há duas possibilidades para mover apenas parte de um método: usar *Extrair Método* (104) e em seguida *Mover Método de Classe para Inter-tipo* (130), ou usar apenas *Extrair Fragmento para Adendo* (124). Para cada um dos $n_{met.parc}$ métodos em que apenas parte de seu código seja relacionada ao interesse do aspecto, uma dessas possibilidades deve ser aplicada.

O 7o. passo move para dentro do aspecto classes e interfaces usadas apenas por ele. Cada uma das $n_{cl.aut}$ classes deste conjunto deve ser movida por *Internalizar Classe em Aspecto* (162) e cada uma das $n_{int.aut}$ interfaces deve ser movida por *Internalizar Interface em Aspecto* (164). Note que o número de classes movidas é maior ou igual ao número de classes extraídas no passo 2, pois o conjunto de classes movidas para dentro do aspecto inclui as classes originalmente definidas dentro de outros componentes e que foram extraídas para classes autônomas.

No passo 8 os $n_{comp.priv}$ componentes, as $n_{op.priv}$ operações e os $n_{atrib.priv}$ atributos usados apenas dentro do aspecto devem se tornar privados. Para fazer isso, são usados os passos básicos "Tornar componente privado" (89), "Tornar operação privada" (91) e "Tornar atributo privado" (92).

Por fim, o último passo da refatoração retira o privilégio do aspecto através do passo básico "Tornar aspecto não-privilegiado" (93).

Apesar de não ser descrito explicitamente, ao final da refatoração as mudanças do código devem ser compiladas e testadas para garantir a preservação do comportamento da refatoração, o que é feito por "Compilar e testar" (101).

A tabela 5.44 mostra quantas vezes e onde (em que passo) cada passo básico é utilizado por esta refatoração.

Cálculo da Variação Total das Métricas

Uma vez decompostos os passos da refatoração, deve-se calcular qual o impacto total das alterações nas métricas escolhidas para avaliar a refatoração. A tabela 5.45 mostra o

Tabela 5.44: Quantidade de vezes que cada passo básico é usado em *Extrair Funcionalidade para Aspecto*

Passo da refatoração	Quantidade	Passo básico
1	1	Criar aspecto vazio
2	$n_{cl.int}$	*Extrair Classe Interna para Autônoma*
3	n_{atrib}	*Move Field From Class to Inter-type*
	1	Tornar aspecto privilegiado
4	n_{cons}	*Extrair Fragmento para Adendo*
	n_{cons}	*Extrair Método*
	n_{cons}	*Particionar Assinatura de Construtor*
5	n_{met}	*Mover Método de Classe para Inter-tipo*
6	$n_{met.parc}$	*Extrair Método*
	$n_{met.parc}$	*Mover Método de Classe para Inter-tipo*
	$n_{met.parc}$	*Extrair Fragmento para Adendo*
7	$n_{cl.aut}$	*Internalizar Classe em Aspecto*
	$n_{int.aut}$	*Internalizar Interface em Aspecto*
8	$n_{comp.priv}$	Tornar componente privado
	$n_{op.priv}$	Tornar operação privada
	$n_{atrib.priv}$	Tornar atributo privado
9	1	Tornar aspecto não-privilegiado
	1	Compilar e testar

impacto de cada passo básico usado pela refatoração. Nesta tabela é possível observar que apenas a métrica **DIT** não é modificada por nenhum dos passos, portanto conclui-se que essa métrica não terão seu valor alterado quando *Extrair Funcionalidade para Aspecto* for aplicada. A variação total de cada uma das demais métricas é explicada a seguir.

CDC: $-n_{comp} + 1$

A criação do aspecto no passo 1 aumenta temporariamente o número de componentes pelos quais o interesse transversal está disperso. No entanto, com a extração dos códigos relacionados a esse interesse para dentro do aspecto, este aumento será neutralizado. Supondo que originalmente os códigos relacionados ao interesse estejam dispersos em n_{comp} componentes, então após a refatoração o valor de CDC será $n_{comp} - 1$ unidades menor que seu valor inicial. Isso mostra que se o interesse estiver originalmente presente em apenas um componente, a refatoração não altera o valor de CDC, mas se ele estiver disperso em mais do que um componente, então esta refatoração diminui o valor de CDC.

CDO: -X a +X

Diversas etapas da refatoração aumentam ou diminuem o número de operações pelas quais o interesse transversal está disperso. Na extração de classes internas, é criado um construtor dentro de cada classe extraída. Se várias classes forem extraídas para uma única classe autônoma, pode ser necessário criar uma interface que exponha as operações usadas por esses componentes, interface essa que contém operações relacionadas ao interesse do aspecto. Por outro lado, a unificação de classes internas também elimina diversas operações, pois os métodos iguais entre as classes serão transformados em apenas um na classe unificada.

Ao mover um atributo para o aspecto pode ser necessário encapsulá-lo, o que cria duas novas operações neste componente. Já a extração de fragmentos de código cria uma nova operação no aspecto – o adendo – mas pode fazer com as operações onde os frag-

Tabela 5.45: Impacto nas métricas de cada passo básico de *Extrair Funcionalidade para Aspecto*

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Criar aspecto vazio	$>_{(1)}$	=	=	$>_{(1)}$	$>_{(2)}$	=	=	=	=	=
Extrair Classe Interna para Autônoma p/ extrair 1 classe	=	$>_{(1aX)}$	$<_{(2aX)}$	=	$>_{(X)}$	$>_{(1)}$	$>_{(X)}$	$>_{(2)}$	=	$\gtrless$
Extrair Classe Interna para Autônoma p/ extrair várias classes	$\leq_{(X)}$	$\gtrless$	$<_{(X)}$	$\leq_{(X)}$	$\gtrless$	$\gtrless$	$\gtrless$	$\gtrless$	=	$\gtrless$
Move Field From Class to Inter-type	$\leq_{(1)}$	$\gtrless$	$\gtrless$	=	$\gtrless$	=	$\geq_{(X)}$	$\gtrless$	=	$\gtrless$
Tornar aspecto privilegiado	=	=	=	=	=	=	=	=	=	=
Extrair Fragmento para Adendo	$\leq_{(X)}$	$\gtrless_{(-Xa+1)}$	$\leq_{(X)}$	=	$\gtrless$	=	$\geq_{(X)}$	$\gtrless$	=	$\gtrless$
Extrair Método	=	$\geq_{(1)}$	$\leq_{(X)}$	=	$>_{(3a4)}$	=	$>_{(X)}$	=	=	$\gtrless$
Particionar Assinatura de Construtor	$\leq_{(1)}$	=	$<_{(X)}$	=	$>_{(3)}$	=	$>_{(X)}$	$\gtrless$	=	$\gtrless$
Mover Método de Classe para Inter-tipo	$\leq_{(1)}$	=	$\leq_{(2)}$	=	=	=	=	$\gtrless$	=	$\gtrless$
Internalizar Classe em Aspecto	=	=	=	=	=	=	=	$<_{(1a2)}$	=	=
Internalizar Interface em Aspecto	=	=	=	=	=	=	=	$<_{(1a2)}$	=	=
Tornar componente privado	=	=	=	=	=	=	=	=	=	=
Tornar operação privada	=	=	=	=	=	=	=	=	=	=
Tornar atributo privado	=	=	=	=	=	=	=	=	=	=
Tornar aspecto não-privilegiado	=	=	=	=	=	=	=	=	=	=
Compilar e testar	=	=	=	=	=	=	=	=	=	=

mentos estavam originalmente deixem de estar relacionadas ao interesse. Por fim, se for necessário substituir o uso de parâmetros em construtores por métodos, novas operações serão criadas por *Extrair Método*.

O quanto CDO será alterado por esta refatoração depende de quantas operações fora do aspecto perderão sua relação com o interesse transversal, do número de operações criadas pela refatoração e de quantas operações são unificadas ao usar *Extrair Classe Interna para Autônoma* para extrair diversas classes iguais.

CDLOC: $-n_{pts}$

Esta refatoração move todos os códigos relacionados ao interesse transversal para o aspecto. Ao adicionar esses códigos no aspecto, não é criado nenhum ponto de transição porque o aspecto inteiro é sombreado para o interesse transversal que ele implementa. No entanto, a remoção dos códigos relacionados ao interesse de seus locais de origem elimina todos os n_{pts} pontos de transição existentes entre códigos de outros interesses e os códigos do interesse extraído. Isso quer dizer que o valor de CDLOC diminuirá tantas unidades quanto forem os pontos de transição (entre o interesse transversal extraído e os demais interesses) existentes originalmente no software.

VS: $-n_{cl.unificadas} + 2$ ou $+1$

A criação do aspecto no passo 1 aumenta o número de componentes do sistema em uma unidade. No entanto, caso sejam unificadas diversas classes internas em uma única classe autônoma no passo 2, este aumento será anulado por causa da remoção das classes internas iguais. Supondo que sejam unificadas $n_{cl.unificadas}$ classes, então a variação de VS é dada por $-n_{cl.unificadas} + 2 - n_{cl.unificadas}$ classes internas removidas, e uma classe autônoma e um aspecto criados.

LOC: -X a +X

Esta refatoração move as linhas de código relacionadas ao interesse dos componentes onde o interesse estava disperso originalmente para o aspecto criado. Para fazer isso, pode ser necessário adicionar novas linhas de código ao software para definir métodos auxiliares (novos construtores, métodos de acesso a atributos, conjuntos de pontos de junção, etc.), assim como linhas podem ser removidas pela extração de diversas classes internas iguais ou pela unificação de comportamento ao definir um adendo (veja o adendo "`after(TangledStack _this) returning : stateChange(_this)`" do aspecto `WindowView` do exemplo desta refatoração).

Assim, esta refatoração pode aumentar ou diminuir o valor de LOC dependendo das etapas necessárias para extrair o aspecto e dos componentes pelos quais o interesse transversal está originalmente espalhado.

NOA: -X a +X

A única maneira desta refatoração alterar a quantidade total de atributos do sistema é se for necessário extrair alguma classe interna relacionada ao interesse do aspecto. Cada vez que uma única classe interna for extraída, será adicionado um novo atributo ao sistema; e cada vez que foram extraídas diversas classes internas iguais em uma só classe autônoma, serão adicionados atributos à classe extraída (como na extração de uma só classe) e à interface comum a essas classes (caso seja necessário criar uma), e removidos os atributos iguais entre as classes unificadas. Assim, esta refatoração poderá aumentar ou diminuir o número de atributos do sistema em várias unidades.

WOC: -X a +X

Diversas etapas desta refatoração aumentam o valor de WOC em algumas unidades, com exceção da unificação de diversas classes internas iguais em uma classe autônoma, que pode aumentar este valor ou diminui-lo. Por causa desta última etapa, esta refatoração poderá aumentar ou diminuir o valor de WOC em várias unidades, dependendo das operações criadas pela refatoração em todas as suas etapas e do valor original de WOC das classes internas iguais extraídas.

CBC: -X a +X

Esta refatoração cria um novo aspecto e adiciona a ele todos os códigos relacionados ao interesse transversal que ele implementa. Por causa disso, todos os componentes referenciados por este aspecto serão considerados acoplamentos adicionados ao sistema pela refatoração.

No entanto, os componentes onde estavam originalmente os códigos movidos para o aspecto poderão perder acoplamentos caso algum componente só seja referenciado por esses componentes nos trechos movidos para o aspecto. Neste caso, o acoplamento deixará de existir após a remoção dos códigos e, portanto, o valor de CBC do componente de origem diminuirá.

Outra forma de diminuir o valor de CBC é pela unificação de diversas classes internas em uma única classe autônoma. Esta alteração faz com que os valores de CBC das classes unificadas deixe de ser considerado no cálculo do valor total dessa métrica, removendo parcelas não-negativas (CBC é sempre maior ou igual a zero) da soma que calcula o valor total de CBC.

Para saber se a variação total do valor de CBC será positiva ou negativa, é preciso saber se serão criados mais acoplamentos no aspecto do que removidos dos componentes de origem e das classes unificadas, ou o contrário. No primeiro caso, o valor de CBC aumenta, enquanto que no segundo esse valor diminui.

LCOO: -X a +X

Diversas etapas desta refatoração criam, removem ou transferem operações e atributos do sistema, bem como movimentam trechos de código que acessam atributos entre operações. Essas mudanças podem alterar a classificação dos pares de operações dos componentes, transformando pares de operações que originalmente compartilhavam atributos em pares de operações sem atributos em comum, e vice-versa; ou criando e eliminando pares de operações dos componentes. Para saber como o valor de LCOO é alterado pela refatoração, é necessário saber se serão criados e transformados mais pares de operações com atributos em comum do que pares de operações que não compartilham atributos, o no contrário. No primeiro caso, o valor de LCOO diminui; no segundo, este valor aumenta.

Tabela 5.46: Variação das métricas em *Extrair Funcionalidade para Aspecto*

Separação de Interesses			Tamanho				Acoplamento		Coesão
CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
$\leq_{(X)}$	$\gtrless$	$<_{(X)}$	$\gtrless_{(-Xa1)}$	$\gtrless$	$\gtrless$	$\gtrless$	$\gtrless$	=	$\gtrless$

Análise dos Resultados

O principal objetivo desta refatoração é reunir códigos relacionados a um mesmo interesse transversal em um único aspecto. Não há, no entanto, nenhuma preocupação explícita de melhorar o tamanho do software ou o acoplamento e a coesão dos componentes. Ao analisar os resultados mostrados na tabela 5.46, observa-se que o objetivo principal foi alcançado: há uma tendência a reduzir o espalhamento dos interesses pelos componentes do sistema (CDC sempre diminui ou se mantém) e sempre ocorre a redução do entrelaçamento dos interesses transversais (CDLOC é sempre menor).

Quanto aos demais atributos internos do software, os impactos desta refatoração podem ser positivos ou negativos, dependendo das características dos componentes envolvidos na refatoração. Em situações mais favoráveis, todos os atributos internos avaliados melhorarão; já nos casos mais adversos, haverá impactos negativos no tamanho, na coesão e no acoplamento, além de uma das métricas da separação de interesses (CDO) também piorar.

Isso mostra que as conseqüências dessa refatoração variam muito de acordo com o caso em que for aplicada, fazendo com que ela seja recomendada apenas em algumas situações.

5.3 Considerações Finais

As refatorações OA avaliadas neste capítulo impactam de maneiras variadas no tamanho do software, na separação dos seus interesses transversais e no acoplamento e coesão de seus componentes. A tabela 5.47 recupera os impactos das refatorações OA descritos separadamente ao longo deste capítulo e os reúne lado-a-lado para facilitar a comparação de seus valores. Nessa tabela é possível observar que grande parte das refatorações tem impacto no valor das métricas que medem a separação dos interesses do software, impactos esses que na maioria das vezes tornam os interesses menos separados, isto é, melhoram a avaliação deste atributo de qualidade. Este resultado quantitativo está de acordo com os objetivos que refatorações para extração de aspectos têm: modularizar interesses transversais em aspectos, reunindo trechos de código de uma mesma funcionalidade que se encontram espalhados pelos componentes do software em um único local – o aspecto extraído.

Já os impactos nos outros atributos de qualidade avaliados (tamanho, acoplamento e coesão) não seguem um padrão de comportamento: ora eles são positivos, ora negativos. Isso mostra que esses atributos de qualidade não são os alvos principais de melhoria das refatorações avaliadas neste capítulo, o que fica ainda mais evidente ao comparar esses impactos com os impactos na separação de interesses. De fato, dentre as refatorações propostas por Monteiro e Fernandes (2005a; 2006), existem outras refatorações que não as responsáveis pela extração de aspectos cujo objetivo é melhorar outros atributos de qualidade como tamanho e acoplamento: Monteiro e Fernandes afirmam que após extrair um aspecto pode ser necessário melhorar sua estrutura interna através das refatorações para reestruturação do interior de aspectos. Por exemplo, uma dessas refatorações, *Generalizar Tipo-Alvo com Interface de Marcação* (MONTEIRO, 2005, p. 114), "contribui para reduzir o acoplamento entre o aspecto e seu código-alvo e também pode ser usada para expor e eliminar código duplicado (...)" (MONTEIRO, 2005), ou seja, melhora o acoplamento do software e reduz seu tamanho[5]. Dessa forma, em um processo de extração de aspectos através das refatorações propostas por Monteiro e Fernandes, a aplicação das refatorações avaliadas neste capítulo compreendem apenas a parte inicial do processo, sendo possível melhorar a avaliação de outros atributos de qualidade (além da separação de interesses) após aplicar refatorações para reestruturar internamente os aspectos.

[5]Essas contribuições devem ser comprovadas quantitativamente em trabalhos futuros

Tabela 5.47: Impactos de todas as refatorações OA avaliadas

	Separação de Interesses			Tamanho				Acoplamento		Coesão
Refatoração	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Trocar `implements`...	$\leq_{(1)}$	=	$\leq_{(2)}$	=	$>_{(1)}$	=	=	$\gtrless_{(-1a+2)}$	=	=
Extrair Fragmento...	$\leq_{(X)}$	$\gtrless_{(-Xa+1)}$	$\leq_{(X)}$	=	$\gtrless$	=	$\geq_{(X)}$	$\gtrless$	=	$\gtrless$
Mover Método...	$\leq_{(1)}$	=	$\leq_{(2)}$	=	=	=	=	$\gtrless$	=	$\gtrless$
Mover Atributo...	$\leq_{(1)}$	$\gtrless$	$\gtrless$	=	$\gtrless$	=	$\geq_{(X)}$	$\gtrless$	=	$\gtrless$
Trocar Classe Abstrata...	=	=	=	=	=	=	=	=	$\leq_{(X)}$	=
Dividir Classe Abstrata...	$>_{(1)}$	$>_{(X)}$	=	$>_{(1)}$	$>_{(2aX)}$	=	$\geq_{(X)}$	$>_{(1aX)}$	$\leq_{(X)}$	$\gtrless$
Extrair Classe Interna... p/ 1 cl.	=	$>_{(1aX)}$	$<_{(2aX)}$	=	$>_{(X)}$	$>_{(1)}$	$>_{(X)}$	$>_{(2)}$	=	$\gtrless$
Extrair Classe Interna... p/ +1 cl.	$\leq_{(X)}$	$\gtrless$	$<_{(X)}$	$\leq_{(X)}$	$\gtrless$	$\gtrless$	$\gtrless$	$\gtrless$	=	$\gtrless$
Internalizar Classe...	=	=	=	=	=	=	=	$<_{(1a2)}$	=	=
Internalizar Interface...	=	=	=	=	=	=	=	$<_{(1a2)}$	=	=
Particionar Assinatura...	$\leq_{(1)}$	=	$<_{(X)}$	=	$>_{(3)}$	=	$>_{(X)}$	$\gtrless$	=	$\gtrless$
Extrair Funcionalidade...	$\leq_{(X)}$	$\gtrless$	$<_{(X)}$	$\gtrless_{(-Xa1)}$	$\gtrless$	$\gtrless$	$\gtrless$	$\gtrless$	=	$\gtrless$

6 CONCLUSÃO

Este trabalho apresentou um processo para avaliar quantitativamente refatorações OA, de forma a permitir que seus impactos nos valores de métricas de software sejam conhecidos sem a necessidade de aplicá-las. Ele torna possível obter evidências quantitativas dos benefícios trazidos pela refatoração avaliada e permite descobrir possíveis desvantagens de seu uso não previstas em sua definição, provendo o desenvolvedor com informações que auxiliam na tomada de decisão sobre a aplicação ou não da refatoração. Isso é particularmente importante no caso das refatorações sem opostas definidas, pois desfazer uma refatoração desse tipo pode ser bastante complexo e propenso à inserção de erros.

O processo proposto é usado para avaliar um conjunto de refatorações OA, tendo como critério métricas de software que medem tamanho, acoplamento, coesão e separação de interesses. Os resultados desta avaliação mostram que a maioria das refatorações diminui a separação dos interesses transversais, porém em alguns casos a melhoria desse atributo de qualidade tem como conseqüência impactos negativos (isto é, aumento no valor das métricas) no acoplamento, coesão ou tamanho do software.

Trabalhos Relacionados

No capítulo 2 foram apresentadas diversos trabalhos que listam impactos de refatorações por meio do uso de métricas de software. Benn et al. (2005) avaliam refatorações OA por meio de um estudo de caso, no qual são comparados os valores de algumas métricas coletados antes e depois de refatorar o código do estudo. Apesar de o trabalho permitir conhecer a magnitude exata do impacto das refatorações, esses valores são válidos apenas para o caso estudado e não podem ser generalizados para todos os contextos em que as refatorações forem usadas, limitando a abrangência das conclusões do trabalho ao caso estudado.

Tahvildari e Kontogiannis (2004) agrupam refatorações OO em meta-padrões de transformação e avaliam se cada meta-padrão melhora ou piora algumas métricas de software. Esta abordagem permite saber facilmente se as refatorações que se enquadram em um dos meta-padrões definidos pelos autores têm impacto positivo ou negativo, porém impede a obtenção dos impactos das refatorações que não podem ser enquadradas em nenhum desses meta-padrões, como refatorações propostas para outros paradigmas de programação. Além disso, os autores não detalham o processo que foi usado para obter os impactos, o que impede que outros meta-padrões de refatoração tenham seus impactos estimados de maneira similar ao que foi feito com os meta-padrões apresentados no trabalho.

Bois e Mens (2003) definem um formalismo para descrever o impacto de refatorações na estrutura de programas OO, representando programas em árvores sintáticas abstratas

estendidas e descrevendo as modificações feitas pelas refatorações através de alterações nas árvores. Esta proposta permite conhecer os impactos da refatoração avaliada para qualquer caso em que ela for usada, porém restringe o universo de refatorações que podem ser avaliadas às refatorações OO, já que o modelo utilizado para representar os programas contempla apenas as estruturas presentes neste paradigma de programação. Esta limitação pode ser contornada modificando-se o modelo para representação de programas, porém "isso o torna mais complexo", nas palavras dos próprios autores. Além do aumento da complexidade, a adaptação do modelo à OA tem como limitação a obtenção de impactos apenas de métricas que levam em consideração informações contidas na estrutura de programas, o que exclui por exemplo métricas como CDLOC.

Além de propostas que avaliam refatorações, este trabalho também está relacionado ao trabalho de Figueiredo et al. (2005). Nele, os autores propõem um método de avaliação quantitativa de artefatos a ser aplicado durante o desenvolvimento de software orientado a aspectos. Este método prevê uma etapa de refatorações do código para melhorar atributos de qualidade identificados como deficientes em medições feitas em etapas anteriores. Uma forma de escolher que refatorações aplicar é usar como critério os impactos que as refatorações têm nos atributos de qualidade medidos, o que pode ser obtido usando o processo de avaliação de refatorações definido no presente trabalho. Desta forma, o trabalho de Figueiredo et al. é considerado complementar a este.

Contribuições

Dentre as contribuições deste trabalho, a principal delas é o processo de avaliação de refatorações OA definido no capítulo 3, uma vez que não foi encontrada na literatura nenhuma maneira de obter impactos abrangentes de refatorações para esse paradigma. Apesar de este processo ser proposto com o objetivo de permitir a avaliação de refatorações OA, ele é definido sem utilizar estruturas específicas deste paradigma de programação, evitando que o processo fique restrito à avaliação apenas de refatorações OA. Este trabalho apresenta evidências (capítulo 5) que o processo proposto pode ser usado também para avaliar refatorações de outros paradigmas de programação, como o OO.

A avaliação das refatorações OA descritas no capítulo 5 traz como contribuição as medições dos impactos que as refatorações têm no tamanho, acoplamento, coesão e separação de interesses do software. Esses impactos comprovam quantitativamente a existência de alguns dos benefícios das refatorações citados nas definições das mesmas, mas também indicam que atributos de qualidade podem ser deteriorados caso as refatorações sejam usadas – o que poucas vezes é previsto na descrição das refatorações.

Outra contribuição deste trabalho é o catálogo de passos básicos avaliados no capítulo 4. Este catálogo provê a definição e avaliação (dos atributos de qualidade selecionados) de um extenso conjunto de pequenas modificações do software que podem ser usadas na decomposição de diferentes refatorações. Em avaliações futuras de outras refatorações, os passos básicos definidos no catálogo podem ser reaproveitados na decomposição das refatorações, diminuindo o esforço necessário para completar as avaliações.

Limitações

Este trabalho, no entanto, também possui algumas limitações. Em primeiro lugar, o processo de avaliação só pode ser usado com as chamadas "métricas estáticas" (SOMMERVILLE, 2007). Isso ocorre porque as métricas dinâmicas, coletadas a partir de medições de programas em execução (SOMMERVILLE, 2007), não podem ser medidas analisando apenas alterações do código-fonte, o que torna impossível obter o impacto

de passos básicos nesse tipo de métrica. Esta limitação também está presente em outras propostas para obter impactos abrangentes de refatorações no código do software (BOIS; MENS, 2003; TAHVILDARI; KONTOGIANNIS, 2004) e não permite que o processo seja usado para medir características como desempenho ou confiabilidade do software.

Além disso, em algumas avaliações pode ser necessário usar diversos passos básicos para decompor uma simples modificação do código, o que torna algumas refatorações bastante difíceis de serem avaliadas. Por exemplo, apenas para remover uma classe, são necessários os passos básicos "Remover atributo de componente", "Remover variável local de operação", "Remover linhas de código de operação", "Remover parâmetro de operação", "Remover operação vazia", "Remover implementação de interface de componente", "Remover herança de classe ou aspecto", etc. No entanto, por causa da possibilidade de reaproveitamento de resultados obtidos em avaliações anteriores, à medida que novas avaliações são feitas a decomposição de refatorações se torna mais simples e são necessários menos esforços para completar a avaliação.

Trabalhos futuros

O uso do processo proposto neste trabalho para avaliar algumas refatorações OA permite o levantamento de algumas vantagens e limitações deste processo. No entanto, essas aplicações do processo não têm como objetivo avaliá-lo, ficando a avaliação do processo como um trabalho futuro. Isso permitiria esclarecer, por exemplo, se o processo pode ser usado para avaliar refatorações para qualquer paradigma de programação, ou o quanto podem ser reaproveitados os resultados obtidos em aplicações anteriores do processo.

Outro trabalho futuro diz respeito à automação do processo de avaliação, através da construção de uma ferramenta semi-automática que auxilie o desenvolvedor na obtenção dos impactos de uma refatoração em um conjunto de métricas. Para avaliar uma refatoração, bastaria o desenvolvedor escolher o conjunto de métricas usadas como critério e efetuar a decomposição da mesma, selecionando quais passos básicos (e quantas vezes) são necessários para fazer um alteração no código equivalente à refatoração original. Outras funcionalidades englobariam, por exemplo, a definição de novos passos básicos e a inserção de dados sobre os impactos de passos básicos (para reavaliar passos básicos por outras métricas ou na própria inserção de novos passos no programa). Como conjunto de dados iniciais da ferramenta, este trabalho fornece a definição e avaliação de 82 passos básicos, bem como das refatorações OA e OO.

REFERÊNCIAS

ASPECTJ: Home Page. Disponível em: <http:/www.eclipse.org/aspectj>. Acesso em: jun. 2007.

ASPECTWERKZ: Home Page. Disponível em: <http://aspectwerkz.codehaus.org/>. Acesso em: jun. 2007.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. The Goal Question Metric Approach. In: MARCINIAK, J. J. (Ed.). **Encyclopedia of Software Engineering**. [S.l.]: John Wiley and Sons, 1994. v.2, p.528–532.

BENN, J. et al. Reasoning on Software Quality Improvement with Aspect-Oriented Refactoring: a case study. In: SOFTWARE ENGINEERING AND APPLICATIONS, 2005, Phoenix, AZ, USA. **Proceedings...** [S.l.: s.n.], 2005.

BOEHM, B. W.; BROWN, J. R.; LIPOW, M. Quantitative evaluation of software quality. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 1976, San Francisco, CA, USA. **Proceedings...** Long Beach, CA: IEEE Computer Society Press, 1976. p.592–605.

BOIS, B. D.; MENS, T. Describing the impact of refactoring on internal program quality. In: INTERNATIONAL WORKSHOP ON EVOLUTION OF LARGE-SCALE INDUSTRIAL SOFTWARE APPLICATIONS, ELISA, 2003, Amsterdam, The Netherlands. **Proceedings...** [S.l.: s.n.], 2003. p.37–48.

BONéR, J. **AspectWerkz - dynamic AOP for Java**. Trabalho submetido a International Conference on Aspect-Oriented Software Development, AOSD, 2004. Disponível em: <http://codehaus.org/jboner/papers/aosd2004_aspectwerkz.pdf>. Acesso em: dez. 2005.

CHIDAMBER, S. R.; KEMERER, C. F. A Metrics Suite for Object Oriented Design. **IEEE Trans. Softw. Eng.**, Piscataway, NJ, USA, v.20, n.6, p.476–493, 1994.

CHIKOFSKY, E. J.; CROSS, J. H. Reverse Engineering and Design Recovery: a taxonomy. **IEEE Software**, [S.l.], v.7, n.1, p.13–17, 1990.

COPPICK, J. C.; CHEATHAM, T. J. Software metrics for object-oriented systems. In: ACM ANNUAL CONFERENCE ON COMMUNICATIONS, CSC, 1992. **Proceedings...** New York: ACM Press, 1992. p.317–322.

FENTON, N.; PFLEEGER, S. L. **Software metrics**: a rigorous and practical approach. 2nd ed. Boston, MA, USA: PWS Publishing, 1997.

FIGUEIREDO, E. et al. Assessing Aspect-Oriented Artifacts: towards a tool-supported quantitative method. In: WORKSHOP ON QUANTITATIVE APPROACHES IN OO SOFTWARE ENGINEERING, QAOOSE, 2005, Glasgow, Scotland. **Proceedings...** [S.l.: s.n.], 2005.

FIGUEIREDO, E. M. L.; STAA, A. von. **Avaliação de um modelo de qualidade para implementações orientadas a objetos e orientadas a aspectos**. Rio de Janeiro, RJ: PUC-RJ, 2005.

FOWLER, M. **Refactoring**: improving the design of existing code. [S.l.]: Addison Wesley, 1999.

GARCIA, A. et al. Modularizing design patterns with aspects: a quantitative study. In: ASPECT-ORIENTED SOFTWARE DEVELOPMENT, AOSD, 4., 2005. **Proceedings...** New York: ACM Press, 2005. p.3–14.

GARCIA, V. C. et al. Manipulating Crosscutting Concerns. In: LATIN AMERICAN CONFERENCE ON PATTERNS LANGUAGES OF PROGRAMMING, 4., 2004. **Proceedings...** [S.l.: s.n.], 2004.

HANENBERG, S.; OBERSCHULTE, C.; UNLAND, R. Refactoring of Aspect-Oriented Software. In: NET.OBJECTDAYS, 2003, Erfurt, Germany. **Proceedings...** [S.l.: s.n.], 2003.

HANNEMANN, J.; MURPHY, G. C.; KICZALES, G. Role-based refactoring of crosscutting concerns. In: ASPECT-ORIENTED SOFTWARE DEVELOPMENT, AOSD, 4., 2005. **Proceedings...** New York: ACM Press, 2005. p.135–146.

HYPERJ: Home Page. Disponível em: <http://www.alphaworks.ibm.com/tech/hyperj>. Acesso em: jun. 2007.

IWAMOTO, M.; ZHAO, J. Refactoring Aspect-Oriented Programs. In: AOSD MODELING WITH UML WORKSHOP, 2003. **Proceedings...** [S.l.: s.n.], 2003.

JAVA: Home Page. Disponível em: <http://java.sun.com>. Acesso em: jun. 2007.

KICZALES, G. et al. Aspect-Oriented Programming. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 1997. **Proceedings...** Berlin: Springer-Verlag, 1997. p.220–242.

KICZALES, G. et al. An Overview of AspectJ. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 15., 2001. **Proceedings...** [S.l.]: Springer-Verlag, 2001. p.327–353.

KOGURE, M.; AKAO, Y. Quality function deployment and CWQC in Japan. **Quality Progress**, Milwaukee, WIS, v.16, n.10, p.25–29, 1983.

LI, W.; HENRY, S. Object-oriented metrics that predict maintainability. **J. Syst. Softw.**, New York, NY, USA, v.23, n.2, p.111–122, 1993.

MENS, T.; TOURWÉ, T. A Survey of Software Refactoring. **IEEE Trans. Softw. Eng.**, Piscataway, NJ, USA, v.30, n.2, p.126–139, 2004.

MONTEIRO, M. P. **Refactorings to Evolve Object-Oriented Systems with Aspect-Oriented Concepts**. 2005. Tese (Doutorado em Ciência da Computação) — Universidade do Minho.

MONTEIRO, M. P.; FERNANDES, J. M. Towards a catalog of aspect-oriented refactorings. In: ASPECT-ORIENTED SOFTWARE DEVELOPMENT, AOSD, 4., 2005. **Proceedings...** New York: ACM Press, 2005. p.111–122.

MONTEIRO, M. P.; FERNANDES, J. M. Refactoring a Java Code Base to AspectJ: an illustrative example. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, ICSM, 21., 2005. **Proceedings...** Los Alamitos, CA: IEEE Computer Society, 2005. p.17–26.

MONTEIRO, M. P.; FERNANDES, J. M. Towards a Catalogue of Refactorings and Code Smells for AspectJ. In: RASHID, A.; AKSIT, M. (Ed.). **Transactions on Aspect-Oriented Software Development I**. Berlin: Springer, 2006. p.214–258.

MOSER, R. et al. Does Refactoring Improve Reusability? In: INTERNATIONAL CONFERENCE ON SOFTWARE REUSE, ICSR, 9., 2006, Turin, Italy. **Proceedings...** [S.l.: s.n.], 2006. p.287–297.

OLIVEIRA, P. W.; DIVERIO, T. A.; CLAUDIO, D. M. **Fundamentos da Matemática Intervalar**. 2. ed. Porto Alegre: Instituto de Informática da UFRGS: Sagra-Luzzatto, 2005.

OSSHER, H.; TARR, P. Hyper/J: multi-dimensional separation of concerns for java. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 23., 2001. **Proceedings...** [S.l.]: IEEE Computer Society, 2001. p.821–822.

QFD: Quality Function Deployment. Disponível em: <http://www.qfdi.org/>. Acesso em: jul. 2007.

REFACTORING: Home Page. Disponível em: <http://www.refactoring.com>. Acesso em: jun. 2007.

SANT'ANNA, C. et al. On the Reuse and Maintenance of Aspect-Oriented Software: an assessment framework. In: BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING, 2003, Manaus. **Proceedings...** [S.l.: s.n.], 2003. p.19–34.

SANT'ANNA, C. N. **Manutenibilidade e Reusabilidade de Software Orientado a Aspectos**: um framework de avaliação. 2004. Dissertação (Mestrado em Ciência da Computação) — PUC-RJ, Rio de Janeiro, RJ.

SOMMERVILLE, I. **Software engineering**. 6th ed. Boston, MA, USA: Addison-Wesley Longman, 2001.

SOMMERVILLE, I. **Software engineering**. 8th ed. Boston, MA, USA: Addison-Wesley Longman, 2007.

STROGGYLOS, K.; SPINELLIS, D. Refactoring: does it improve software quality? In: INTERNATIONAL WORKSHOP ON SOFTWARE QUALITY, 5., 2007. **Proceedings...** New York: ACM Press, 2007.

TAHVILDARI, L.; KONTOGIANNIS, K. Improving design quality using meta-pattern transformations: a metric-based approach. **Journal of Software Maintenance and Evolution: Research and Practice**, New York, NY, USA, v.16, n.4-5, p.331–361, 2004.

TARR, P. et al. N degrees of separation: multi-dimensional separation of concerns. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 1999. **Proceedings...** Los Alamitos, CA: IEEE Computer Society Press, 1999. p.107–119.

WASP: Home Page. Disponível em: <http://twiki.im.ufba.br/bin/view/AOSDbr/TermosEmPortugues>. Acesso em: 20 dez. 2005.

APÊNDICE A PASSOS BÁSICOS EM ORDEM ALFABÉTICA

Para facilitar a referência aos passos básicos definidos neste trabalho, as tabelas abaixo mostram como as métricas usadas na avaliação (seção 2.3.1) são alteradas por cada passo básico, listando-os em ordem alfabética e indicando em que página está descrita a respectiva definição.

Tabela 6.1: Variação das métricas ao usar os passos básicos (A-A)

		Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	pág.	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Adicionar atributo a componente	82	$\geq_{(1)}$	=	$\geq_{(2)}$	=	$>_{(1)}$	$>_{(1)}$	=	$\geq_{(1)}$	=	=
Adicionar chamada a operação	79	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(1)}$	=	=	$\geq_{(1)}$	=	=
Adicionar declaração inter-tipo de atributo	86	=	=	=	=	$>_{(1)}$	$>_{(1)}$	=	$\geq_{(1a2)}$	=	=
Adicionar declaração inter-tipo de herança	85	=	=	=	=	$>_{(1)}$	=	=	$\geq_{(1a2)}$	$\geq_{(X)}$	=
Adicionar declaração inter-tipo de implementação	84	=	=	=	=	$>_{(1)}$	=	=	$\geq_{(1a2)}$	=	=
Adicionar declaração inter-tipo de operação	87	=	$>_{(1)}$	=	=	$>_{(2)}$	=	$>_{(1)}$	$\geq_{(1)}$	=	$\geq_{(X)}$
Adicionar herança a classe ou aspecto	45	$\geq_{(1)}$	=	$\geq_{(2)}$	=	=	=	=	$\geq_{(1)}$	$\geq_{(X)}$	=
Adicionar herança a interface	43	$\geq_{(1)}$	=	$\geq_{(2)}$	=	=	=	=	$\geq_{(1)}$	=	=
Adicionar implementação de interface a componente	41	$\geq_{(1)}$	=	$\geq_{(2)}$	=	=	=	=	$\geq_{(1)}$	=	=
Adicionar importações	95	=	=	=	=	=	=	=	=	=	=
Adicionar linha de leitura/escrita de valor de atributo a operação	73	=	=	=	=	$>_{(1)}$	=	=	=	=	$\leq_{(X)}$
Adicionar linhas de código a operação	76	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(X)}$	=	$>_{(X)}$	=	=	$\geq_{(X)}$	=	$\leq_{(X)}$
Adicionar palavra-chave `abstract` a componente	100	=	=	=	=	=	=	=	=	=	=
Adicionar palavra-chave `abstract` a operação de interface	101	=	=	=	=	=	=	=	=	=	=
Adicionar parâmetro a conjunto de pontos de junção	52	=	=	=	=	=	=	=	$\geq_{(1)}$	=	=
Adicionar parâmetro a operação	60	=	=	=	=	=	=	$>_{(1)}$	$\geq_{(1)}$	=	=
Adicionar ponto de junção a conjunto de pontos de junção	53	=	=	=	=	$\gtrless$	=	=	$\geq_{(1)}$	=	=

Tabela 6.2: Variação das métricas ao usar os passos básicos (A-R)

		Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	pág.	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Adicionar variável local a operação	62	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(1)}$	=	=	$\geq_{(1)}$	=	=
Alterar referência a componente em `within` de conjunto de pontos de junção	57	=	=	=	=	=	=	=	$\gtrless_{(-1a1)}$	=	=
Capturar contexto de ponto de junção	56	=	=	=	=	$\geq_{(X)}$	=	=	=	=	=
Compilar e testar	101	=	=	=	=	=	=	=	=	=	=
Criar `declare error` de proteção de acesso a atributo	94	=	=	=	=	$>_{(6)}$	=	=	=	=	=
Criar `declare warning` de sinalização de acesso a atributo	93	=	=	=	=	$>_{(5)}$	=	=	=	=	=
Criar aspecto vazio	39	$>_{(1)}$	=	=	$>_{(1)}$	$>_{(2)}$	=	=	=	=	=
Criar classe ou interface vazia	40	$\geq_{(1)}$	=	=	$>_{(1)}$	$>_{(2)}$	=	=	=	=	=
Criar conjunto de pontos de junção	51	=	=	=	=	$>_{(X)}$	=	=	$\geq_{(X)}$	=	=
Criar cópia autônoma de componente interno	50	$\geq_{(1)}$	$\geq_{(X)}$	$\geq_{(X)}$	$>_{(1)}$	$>_{(X)}$	$\geq_{(X)}$	$\geq_{(X)}$	$\geq_{(X)}$	=	$\geq_{(X)}$
Criar cópia interna de componente autônomo	46	$\geq_{(1)}$	$\geq_{(X)}$	$\geq_{(X)}$	$>_{(1)}$	$>_{(X)}$	$\geq_{(X)}$	$\geq_{(X)}$	$\geq_{(X)}$	=	$\geq_{(X)}$
Criar operação vazia	58	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$>_{(2)}$	=	$>_{(1)}$	$\geq_{(1)}$	=	$\geq_{(X)}$
Marcar operação como obsoleta	80	=	=	=	=	=	=	=	=	=	=
Mover código entre operações de componentes diferentes	63	$\gtrless_{(-1a1)}$	$\gtrless_{(-1a1)}$	$\gtrless_{(-2a2)}$	=	=	=	=	$\gtrless$	=	$\gtrless$
Mover código entre operações do mesmo componente	67	=	$\gtrless_{(-1a1)}$	$\gtrless_{(-2a2)}$	=	=	=	=	=	=	$\gtrless$
Mudar palavra-chave `class` para `interface`	99	=	=	=	=	=	=	=	=	=	=
Mudar palavra-chave `interface` para `class`	99	=	=	=	=	=	=	=	=	=	=
Remover `declare warning` de sinalização de acesso a atributo	94	=	=	=	=	$<_{(5)}$	=	=	=	=	=
Remover aspecto vazio	39	$<_{(1)}$	=	=	$<_{(1)}$	$<_{(2)}$	=	=	=	=	=
Remover atributo de componente	83	$\leq_{(1)}$	=	$\leq_{(2)}$	=	$<_{(1)}$	$<_{(1)}$	=	$\leq_{(1)}$	=	=
Remover classe ou interface vazia	40	$\leq_{(1)}$	=	=	$<_{(1)}$	$<_{(2)}$	=	=	=	=	=

Tabela 6.3: Variação das métricas ao usar os passos básicos (R-T)

		Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	pág.	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Remover conjunto de pontos de junção	52	=	=	=	=	$<_{(X)}$	=	=	$\leq_{(X)}$	=	=
Remover cópia autônoma de componente interno	50	$\leq_{(1)}$	$\leq_{(X)}$	$\leq_{(X)}$	$<_{(1)}$	$<_{(X)}$	$\leq_{(X)}$	$\leq_{(X)}$	$\leq_{(X)}$	=	$\leq_{(X)}$
Remover cópia interna de componente autônomo	48	$\leq_{(1)}$	$\leq_{(X)}$	$\leq_{(X)}$	$<_{(1)}$	$<_{(X)}$	$\leq_{(X)}$	$\leq_{(X)}$	$\leq_{(X)}$	=	$\leq_{(X)}$
Remover declaração inter-tipo de atributo	86	=	=	=	=	$<_{(1)}$	$<_{(1)}$	=	$\leq_{(1a2)}$	=	=
Remover declaração inter-tipo de herança	85	=	=	=	=	$<_{(1)}$	=	=	$\leq_{(1a2)}$	$\leq_{(X)}$	=
Remover declaração inter-tipo de implementação	84	=	=	=	=	$<_{(1)}$	=	=	$\leq_{(1a2)}$	=	=
Remover declaração inter-tipo de operação	88	=	=	=	=	$<_{(2)}$	=	$<_{(1)}$	$\leq_{(1)}$	=	$\leq_{(X)}$
Remover herança de classe ou aspecto	45	$\leq_{(1)}$	=	$\leq_{(2)}$	=	=	=	=	$\leq_{(1)}$	$\leq_{(X)}$	=
Remover herança de interface	44	$\leq_{(1)}$	=	$\leq_{(2)}$	=	=	=	=	$\leq_{(1)}$	=	=
Remover implementação de interface de componente	42	$\leq_{(1)}$	=	$\leq_{(2)}$	=	=	=	=	$\leq_{(1)}$	=	=
Remover importações	96	=	=	=	=	=	=	=	=	=	=
Remover linha de leitura/escrita de valor de atributo a operação	74	=	=	=	=	$<_{(1)}$	=	=	=	=	$\geq_{(X)}$
Remover linhas de código de operação	78	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(X)}$	=	$<_{(X)}$	=	=	$\leq_{(X)}$	=	$\geq_{(X)}$
Remover marcação de operação obsoleta	80	=	=	=	=	=	=	=	=	=	=
Remover operação vazia	59	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(2)}$	=	$<_{(2)}$	=	$<_{(1)}$	$\leq_{(1)}$	=	$\leq_{(X)}$
Remover palavra-chave `abstract` de componente	100	=	=	=	=	=	=	=	=	=	=
Remover palavra-chave `abstract` de operação de interface	100	=	=	=	=	=	=	=	=	=	=
Remover parâmetro de conjunto de pontos de junção	53	=	=	=	=	=	=	=	$\leq_{(1)}$	=	=
Remover parâmetro de operação	61	=	=	=	=	=	=	$<_{(1)}$	$\leq_{(1)}$	=	=
Remover ponto de junção de conjunto de pontos de junção	55	=	=	=	=	$\gtrless$	=	=	$\leq_{(1)}$	=	=
Remover variável local de operação	62	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(2)}$	=	$<_{(1)}$	=	=	$\leq_{(1)}$	=	=
Substituir chamada a operação de leitura ou escrita por referência direta a atributo	75	=	$\leq_{(1)}$	=	=	=	=	=	=	=	$\leq_{(X)}$
Substituir `public` por `static` em classe	98	=	=	=	=	=	=	=	=	=	=
Substituir referência direta a atributo por chamada a operação de leitura ou escrita	74	=	$\geq_{(1)}$	=	=	=	=	=	=	=	$\geq_{(X)}$
Substituir `static` por `public` em classe	98	=	=	=	=	=	=	=	=	=	=
Tornar aspecto não-privilegiado	93	=	=	=	=	=	=	=	=	=	=
Tornar aspecto privilegiado	93	=	=	=	=	=	=	=	=	=	=

Tabela 6.4: Variação das métricas ao usar os passos básicos (T-Z)

		Separação de Interesses			Tamanho				Acoplamento		Coesão
Passo básico	pág.	CDC	CDO	CDLOC	VS	LOC	NOA	WOC	CBC	DIT	LCOO
Tornar atributo privado	92	=	=	=	=	=	=	=	=	=	=
Tornar atributo protegido	92	=	=	=	=	=	=	=	=	=	=
Tornar atributo público	92	=	=	=	=	=	=	=	=	=	=
Tornar componente privado	89	=	=	=	=	=	=	=	=	=	=
Tornar componente protegido	90	=	=	=	=	=	=	=	=	=	=
Tornar componente público	89	=	=	=	=	=	=	=	=	=	=
Tornar operação abstrata	81	=	=	=	=	$<_{(1)}$	=	=	=	=	$\gtrless$
Tornar operação concreta	81	=	=	=	=	$>_{(1)}$	=	=	=	=	$\gtrless$
Tornar operação privada	91	=	=	=	=	=	=	=	=	=	=
Tornar operação protegida	91	=	=	=	=	=	=	=	=	=	=
Tornar operação pública	90	=	=	=	=	=	=	=	=	=	=
Trocar chamada a operação sobrecarregada por versão com mais parâmetros	69	$\geq_{(1)}$	$\geq_{(1)}$	$\geq_{(2)}$	=	$\geq_{(X)}$	=	=	$\geq_{(X)}$	=	=
Trocar chamada a operação sobrecarregada por versão com menos parâmetros	71	$\leq_{(1)}$	$\leq_{(1)}$	$\leq_{(X)}$	=	$\leq_{(X)}$	=	=	$\leq_{(X)}$	=	=
Trocar passagem de parâmetro em construtor por chamada de método de escrita	72	=	=	=	=	$>_{(X)}$	=	=	=	=	=
Trocar referência a `this` por parâmetro de operação	68	=	=	=	=	=	=	=	=	=	=
Trocar referência a componente externo por referência a componente interno	97	=	=	=	=	=	=	=	$<_{(1)}$	=	=
Trocar referência a componente interno por referência a componente externo	97	=	=	=	=	=	=	=	$>_{(1)}$	=	=

APÊNDICE B MODELO GQM UTILIZADO

Tabela 6.5: Modelo GQM usado nas avaliações deste trabalho

Objetivo: Melhorar tamanho, acoplamento, coesão e separação de interesses transversais de um programa, através da aplicação de refatorações orientadas a aspectos do ponto de vista do desenvolvedor.	
Perguntas:	Quão conciso é o programa? 1. Quantas classes, interfaces e aspectos existem no programa? 2. Quantas linhas de código existem no programa? 3. Quantos atributos existem no programa? 4. Quantos métodos e adendos existem no programa?
Métricas:	Tamanho do Vocabulário (VS) Número de Linhas de Código (LOC) Número de Atributos (NOA) Peso de Operações por Componente (WOC)
Perguntas:	Quão bem localizados estão os interesses transversais? 1. Quão espalhada e entrelaçada está a definição de <nome do interesse transversal 1> 2. Quão espalhada e entrelaçada está a definição de <nome do interesse transversal 2> 3. ...
Métricas:	Difusão do Interesse por Componentes (CDC) Difusão do Interesse por Operações (CDO) Difusão do Interesse por Linhas de Código (CDLOC)
Pergunta:	Quão baixo é o acoplamento entre os componentes?
Métricas:	Acoplamento Entre Componentes (CBC) Profundidade da Árvore de Herança (DIT)
Pergunta:	Quão alta é a coesão dos componentes do programa?
Métrica:	Falta de Coesão nas Operações (LCOO)