Pontifícia Universidade Católica
do Rio de Janeiro

Eduardo Magno Lages Figueiredo

Uma Abordagem Quantitativa para Desenvolvimento de Software Orientado a Aspectos

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio.

Orientadores: Carlos José Pereira de Lucena
Alessandro Fabricio Garcia

Rio de Janeiro

Março de 2006

Pontifícia Universidade Católica
do Rio de Janeiro

Eduardo Magno Lages Figueiredo

Uma Abordagem Quantitativa para Desenvolvimento de Software Orientado a Aspectos

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Carlos José Pereira de Lucena
Orientador
PUC-Rio

Prof. Alessandro Fabricio Garcia
Co-Orientador
Universidade de Lancaster – UK

Prof. Arndt von Staa
PUC-Rio

Prof. Renato Fontoura de Gusmão Cerqueira
PUC-Rio

Prof. José Eugenio Leal
Coordenador Setorial do Centro Técnico Científico – PUC-Rio

Rio de Janeiro, 29 de março de 2006

Eduardo Magno Lages Figueiredo

Graduou-se em Ciência da Computação na Universidade Federal de Ouro Preto (UFOP) em 2003.

Ficha Catalográfica

Figueiredo, Eduardo Magno Lages

Uma abordagem quantitativa para desenvolvimento de software orientado a aspectos / Eduardo Magno Lages Figueiredo ; orientador: Carlos José Pereira de Lucena; co-orientador: Alessandro Fabricio Garcia. – Rio de Janeiro : PUC-Rio, Departamento de Informática, 2006.

140 f. ; 30 cm

Dissertação (mestrado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Teses. 2. Desenvolvimento de software orientado a aspectos. 3. Método de avaliação. 4. Engenharia de software experimental. 5. Software. 6. Padrões de projeto. I. Lucena, Carlos José Pereira de. II. Garcia, Alessandro Fabrício Garcia. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

Agradecimentos

Em primeiro lugar, gostaria de agradecer aos meus orientadores Carlos Lucena e Alessandro Garcia, pois sem eles este trabalho não teria sido possível. O professor Lucena trouxe apoio, confiança, ensinamentos e liberdade para escolher o trabalho, enquanto o amigo Alessandro complementou com incentivo, criatividade, dinamismo e adrenalina (nos *deadlines*).

Aos professores do Departamento de Informática que contribuíram para a minha formação. Em especial, ao professor Arndt von Staa por ter me acompanhado desde o meu primeiro relatório técnico pelo mundo de medições e qualidade de software. Também ao professor Renato Cerqueira pelas discussões que permitiram melhorias neste documento.

Aos participantes dos estudos experimentais que contribuíram diretamente para os resultados alcançados: Alessandro Garcia, Cláudio Sant'Anna, Carlos Lucena, Uirá Kulesza, Arndt von Staa, Nélio Cacho, Thaís Batista, Fernando Castor, Cecília Rubira e Thiago Bartolomei. Em particular, ao amigo Cláudio Sant'Anna que foi fundamental para os caminhos traçados e compartilhou cada idéia desta dissertação.

Agradeço também ao professor Marcelo Maia (UFOP) pelos primeiros passos na Engenharia de Software e pelas conversas sobre manipulação de programas.

Aos colegas do Laboratório de Engenharia de Software, onde encontrei um espaço privilegiado para desenvolvimento de meu trabalho, em especial ao grupo “Aspectos@PUC-Rio” pelas frutíferas discussões.

À Vera Menezes pelo carinho e apoio, especialmente em questões administrativas.

À CAPES, à PUC-Rio e à Fundação Padre Leonel Franca, pelo apoio financeiro.

Agradeço aos meus pais José e Sônia, aos meus irmãos Ângelo, Rosane e Marcelo e à minha princesinha Marina, pelo incentivo, carinho e amor incondicional.

À Deus, por me agraciar com tantas conquistas.

Resumo

Figueiredo, Eduardo. **Uma Abordagem Quantitativa para Desenvolvimento de Software Orientado a Aspectos.** Rio de Janeiro, 2006. 140p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

O desenvolvimento de software orientado a aspectos é um paradigma recente que introduz novas abstrações e mecanismos com o objetivo de melhorar a modularidade de interesses que se espalham pelo sistema. Entretanto, a satisfação de atributos de qualidade em sistemas orientados a aspectos não é tarefa simples e a utilização equivocada destas novas abstrações pode resultar em efeitos colaterais relacionados a princípios importantes da Engenharia de Software, tais como elevado acoplamento, baixa coesão dos módulos e incompleta modularidade dos interesses em aspectos. Problemas como estes não são facilmente verificáveis em sistemas de médio e grande porte sem um método adequado e, geralmente, consomem muito tempo e recursos. Portanto, torna-se necessário um método de avaliação que auxilie engenheiros de software na análise de sistemas orientados a aspectos. Este trabalho de mestrado propõe uma abordagem que provê suporte à avaliação quantitativa de implementações orientadas a aspectos. A abordagem incluiu: (i) um método de avaliação organizado em etapas, e (ii) uma ferramenta de medição e avaliação, chamada AJATO, que dá suporte ao método proposto. O método é composto por um conjunto de métricas e regras heurísticas. As métricas fornecem informações quantitativas e as heurísticas contribuem com algum raciocínio semântico dos números. A ferramenta AJATO é composta por quatro módulos que efetuam o *parser* do código, mapeamento de estruturas sintáticas em interesses, medição e avaliação heurística. Um conjunto de cinco estudos de caso envolvendo domínios de aplicação distintos foi realizado para avaliar a utilidade e usabilidade da abordagem proposta.

Palavras-chave

Desenvolvimento de software orientado a aspectos, método de avaliação, métricas de software, engenharia de software experimental, padrões de projeto.

Abstract

Figueiredo, Eduardo. **A Quantitative Approach to Aspect Oriented Software Development.** Rio de Janeiro, 2006. 140p. Master Thesis – Computer Science Department, Pontifical Catholical University of Rio de Janeiro.

Aspect-oriented software development is an emerging paradigm that provides new abstractions and mechanisms to support the modularization of crosscutting concerns through the software development lifecycle. However, the achievement of high-quality aspect-oriented software is not trivial. The inappropriate use of aspect-oriented abstractions and mechanisms potentially leads to the violation of important design principles, such as low coupling, high cohesion, incomplete modularization of crosscutting concerns into aspects, and so forth. These problems are not easily detectable and an *ad hoc* analysis of large designs and implementations is often expensive and time-consuming. Hence there is a need for an assessment method that assists software engineers in the analysis of their aspect-oriented implementations. This work proposes the development of a systematic approach to support the quantitative assessment of aspect-oriented software. The approach is organized in a stepwise fashion and is founded on a metrics suite and a comprehensive set of complementary rules. Our proposal is supported by a measurement and assessment tool. A set of five case studies from different application domains have been carried out in order to evaluate the usability and usefulness of our proposed approach.

Keywords

Aspect-Oriented Software Development, Assessment Method, Software Metrics, Empirical Software Engineering, Design Patterns.

Sumário

Lista de figuras

Lista de tabelas

Lista de métricas

Sigla	Nomes (Inglês e Tradução)	Referências
CBC	Coupling Between Components Acoplamento entre Componentes	[31] [57]
CBO	Coupling Between Objects Acoplamento entre Objetos	[14]
CDC	Concern Diffusion over Components Difusão de Interesse em Componentes	[31] [57]
CDLOC	Concern Diffusions over Lines of Code Difusão de Interesse em Linhas de Código	[31] [57]
CDO	Concern Diffusion over Operations Difusão de Interesse em Operações	[31] [57]
DIT	Depth Inheritance Tree Profundidade da Árvore de Herança	[13] [14]
LCOO	Lack of Cohesion in Operations Perda de Coesão em Operações	[13] [57]
LCOM	Lack of Cohesion in Methods Perda de Coesão em Métodos	[14]
LOC	Lines of Code Número de Linhas de Código	[20] [47]
NOA	Number of Attributes Número de Atributos	[20] [57]
NOC	Number of Children Número de Filhos	[14] [22]
NOO	Number of Operations Número de Operações	[20] [22]
NOS	Number of Statements Número de Comandos	[20] [22]

VS	Vocabulary Size Tamanho do Vocabulário	[31] [57]
WOC	Weighted Operations per Component Peso das Operações por Componente	[31] [57]
WMC	Weighted Methods per Class Peso dos Métodos por Classe	[13] [14]
NOA$_{concern}$	Number of Attributes per Concern Número de Atributos do Interesse	-
NOO$_{concern}$	Number of Operations per Concern Número de Operações do Interesse	-
LOC$_{concern}$	Number of Lines of Code per Concern Número de Linhas de Código do Interesse	-

Neste documento os nomes das métricas são usados em português, entretanto, as suas respectivas siglas fazem referência aos nomes originais.

1
Introdução

O desenvolvimento de software orientado a aspectos (DSOA) [42] [43] [45] [66] é um paradigma recente que propõe mecanismos e abstrações para modularizar interesses transversais. Tais interesses são propriedades que naturalmente se espalham através dos módulos de um sistema e não são bem capturados por outros paradigmas, como o orientado a objetos (OO). O DSOA introduz uma nova unidade modular, chamado aspecto, para capturar o código espalhado de um interesse. Além disso, tal paradigma provê mecanismos para dar suporte à composição entre aspectos e outros módulos do sistema, como classes e interfaces. Alguns exemplos tradicionais de interesses transversais são: tratamento de exceções, persistência, distribuição, segurança, auditoria, entre outros. Os benefícios esperados da utilização de técnicas do DSOA são: superior separação de interesses, menor replicação de código, maior coesão dos módulos, menor acoplamento e, como conseqüência, aumento potencial de reutilização e evolução no desenvolvimento de sistemas de software complexos.

1.1.
Definição do Problema

Apesar dos benefícios prometidos com a utilização de DSOA, os próprios aspectos podem levar a complexidade adicional, ou ainda reduzir a qualidade das classes afetadas por eles [27] [28] [30] [32]. Na verdade, alcançar elevada qualidade em artefatos de software orientado a aspectos (OA) é um desafio por cinco razões principais. Primeiro, os desenvolvedores de software podem usar inadequadamente as construções das linguagens de programação OA, uma vez que estas exibem uma série de abstrações e mecanismos de decomposição e composição adicionais. Segundo, a separação de interesses alcançada por técnicas orientadas a aspectos pode ter impacto negativo em outros importantes atributos, como acoplamento e coesão [23] [27] [30]. Terceiro, a identificação de aspectos específicos do domínio e o projeto adequado de um sistema na presença destes

aspectos não são tarefas triviais, e requerem raciocínio adicional sobre decisões de projeto e implementação. Quarto, mesmo quando todos os aspectos são identificados, a separação completa de um determinado interesse transversal não é direta, como também não é trivial capturar todo o código que implementa tal comportamento. Finalmente, o projeto interno do próprio aspecto também pode conter problemas relacionados à separação de interesses [51].

Melhorar a qualidade no DSOA pode ser conseguido por meio de avaliações durante o ciclo de desenvolvimento. Estas avaliações são usualmente baseadas em métricas. Porém, a interpretação dos números gerados no processo de medição não é uma tarefa trivial e análises equivocadas são freqüentemente feitas. Além disso, fazer análise de qualidade em projeto e implementação sem qualquer automatização é um processo caro. Especialmente, no caso de DSOA, em que os aspectos exibem características particulares tais como (i) afetar vários módulos, incluindo outros aspectos, o que torna a análise de tais interações mais complicada e (ii) a propriedade desejável de inconsciência das classes em relação aos aspectos, o que torna ainda mais complicado entender como certas classes estão sendo afetadas por aspectos. Desta forma, um requisito fundamental para o processo de avaliação neste paradigma é o suporte adequado de ferramentas de software.

1.2. Limitações dos Trabalhos Relacionados

A Separação de Interesses (SI) sempre foi tida como de importância primária no processo de desenvolvimento, pois é um instrumento básico para se reduzir a complexidade de software [18]. Entretanto, ainda não existem abordagens que apresentam métodos de avaliação efetivos e suportados por ferramentas para as fases de projeto e implementação em DSOA. Também não se tem conhecimento de trabalhos que efetivamente apresentem boas práticas de desenvolvimento voltadas especificamente para o paradigma de aspectos. As únicas diretivas documentadas são explicitamente voltadas ao uso de construções de uma linguagem de programação em particular [34] [36], tais como AspectJ [66]. Além disso, a literatura inclui geralmente estudos de caso [30] [36] com foco apenas em SI para avaliar a qualidade de implementações orientadas a aspectos e, um dos motivos desta limitação, é ser difícil avaliar múltiplas características de

um software simultaneamente. Entretanto, a qualidade de software não depende exclusivamente de SI e vários outros importantes atributos da Engenharia de Software devem ser considerados, em especial coesão, acoplamento e tamanho.

Muitas ferramentas de software vêm sendo desenvolvidas para DSOA, mas a literatura ainda é escassa de abordagens de avaliação da qualidade. Um *framework* de avaliação bastante difundido na comunidade de DSOA e utilizado em diversos estudos de caso [30] [33] [60] é proposto por Sant'Anna *et al*. [57]. Este *framework* propõe um conjunto de métricas OA e um modelo para mapear os atributos mensuráveis do software em características de qualidade, como reusabilidade e manutenibilidade. Os atributos mensuráveis incluem coesão, acoplamento, tamanho e SI. Entretanto, a abordagem de Sant'Anna *et al*. não deixa claro como ocorre este mapeamento. Além disso, ela não apresenta um conjunto de atividades que possa ser automatizado para guiar o engenheiro de software no processo de avaliação. Em relação a ferramentas de suporte ao DSOA, estão surgindo propostas interessantes (como CME [15] e FEAT [56]) e apenas uma delas cobre o processo de medição [13]. Entretanto, tal ferramenta apóia somente a coletânea de dados associados com as métricas definidas. Nenhuma das ferramentas encontradas na literatura está associada a um método de avaliação que auxilie engenheiros de software na interpretação dos resultados de medições.

1.3. Solução Proposta

Esta dissertação de mestrado propõe um método para avaliação quantitativa de sistemas orientados a aspectos e uma ferramenta para automatizar as atividades definidas no método. O método proposto é fundamentado principalmente no princípio de separação de interesses e pode ser usado na avaliação de artefatos nas fases de projeto detalhado e implementação. Além disso, ele é organizado em etapas, baseado em métricas de software e apoiado por um conjunto de regras heurísticas. As métricas avaliam atributos bem conhecidos da Engenharia de Software como coesão, acoplamento, tamanho e SI. As regras heurísticas auxiliam na interpretação dos números e provêem suporte à identificação de problemas nos artefatos OA de projeto e implementação. Outro objetivo central das regras é

fornecer informações mais abstratas ao desenvolvedor, evitando que este trabalhe diretamente com o resultado das métricas.

Como o método propõe uma avaliação em iterações, uma ferramenta torna-se importante para aplicação efetiva do método em sistemas de médio a grande porte e aumento na confiabilidade dos resultados. Desta forma, este trabalho apresenta também uma ferramenta, chamada AJATO, que pode ser usada na avaliação de sistemas implementados em Java [17] e AspectJ [45]. O principal objetivo desta ferramenta é automatizar as atividades de medição e aplicação das regras definidas no método de avaliação. Para isso, ela disponibiliza um conjunto de métricas e regras heurísticas e oferece ainda mecanismos de extensão que tornam possível adicionar novos elementos. Tanto o método quanto a ferramenta foram definidos com base na experiência obtida em DSOA durante o mestrado no Laboratório de Engenharia de Software (LES) desta instituição de ensino.

Neste período, também foram desenvolvidos cinco estudos experimentais. Os experimentos contaram com a colaboração de equipes de outros grupos de pesquisa da Universidade de Lancaster, Universidade Federal do Rio Grande do Norte (UFRN), Universidade Federal de Pernambuco (UFPE) e Universidade Estadual de Campinas (UNICAMP). Estes estudos foram utilizados na avaliação e evolução dos três principais elementos da abordagem: método de avaliação, regras heurísticas e ferramenta. Em nossos estudos experimentais, a abordagem baseada em regras tem se mostrado útil para apontar problemas não triviais resultantes de uma análise equivocada das medições. Tanto problemas de SI, como de acoplamento e coesão podem ser identificados pelo método de avaliação. Entretanto, a abordagem é orientada a interesses e assume que problemas em outros atributos são relevantes quando estes estiverem relacionados a uma ineficiente SI.

1.4. Organização do Texto

O restante deste documento está estruturado da seguinte forma. No Capítulo 2, são detalhados os principais conceitos e abstrações definidos para o paradigma de desenvolvimento orientado a aspectos, sendo também apresentado AspectJ, uma linguagem que estende Java para realização deste paradigma. No Capítulo 3, o trabalho é contextualizado com o estado da arte em publicações e pesquisas

desta área, enfatizando principalmente qualidade e métricas de software, abordagens de avaliação e ferramentas de DSOA. No Capítulo 4, é apresentado o método de avaliação proposto nesta dissertação e três novas métricas de separação de interesses. As regras heurísticas são detalhadas no Capítulo 5, descrevendo suas contribuições em apontar problemas de SI, acoplamento e coesão que não são facilmente identificados por uma avaliação direta sobre as métricas. No Capítulo 6, é apresentada a ferramenta de medição e avaliação que dá suporte automatizado ao método, incluindo decisões arquiteturais e detalhes de implementação. Os cinco estudos experimentais que contribuíram com a avaliação e evolução da abordagem são explorados no Capítulo 7. Finalmente, o Capítulo 8 apresenta as principais contribuições deste trabalho e possíveis direções para trabalhos futuros.

2
Desenvolvimento de Software Orientado a Aspectos

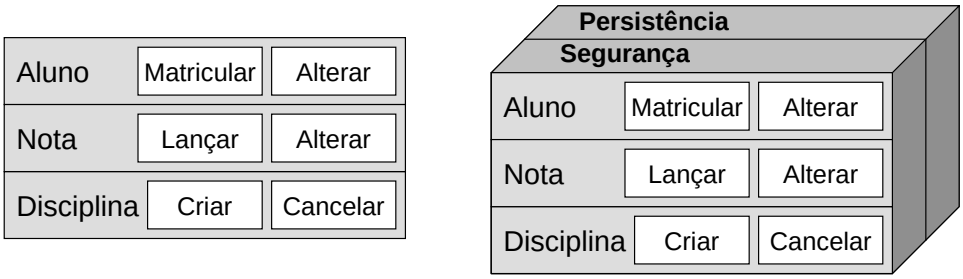
A divisão em partes é um importante instrumento para se reduzir a complexidade de sistemas de software. É muito difícil para o ser humano compreender um sistema de grande porte se este for monolítico, sem fronteiras claras que definam suas funcionalidades. Neste sentido, Edsger Dijkstra [18] em 1974 foi o primeiro a falar em *separação de interesses*[1] para denotar o princípio que guia a divisão em partes. Todo sistema de software lida com diferentes interesses, sejam eles dados, operações ou outros requisitos do sistema. O ideal seria que a parte do programa dedicada a satisfazer a um determinado interesse estivesse concentrada em uma única localidade física, separada de outros interesses. Desta forma, cada interesse pode ser estudado e compreendido com mais facilidade.

No desenvolvimento estruturado, a separação de interesses (SI) ocorre através das diferentes funcionalidades oferecidas pelo software. Cada função é implementada em um único módulo, ou procedimento. Neste paradigma de desenvolvimento, apesar dos interesses relativos a funcionalidades ficarem separados, interesses relativos a dados ficam distribuídos em diversos módulos. Para sanar esta deficiência, o desenvolvimento orientado a objetos (OO) apresenta uma forma mais eficiente para SI. Neste paradigma a separação ocorre em duas dimensões, ou seja, em termos dos dados e das funções que utilizam cada tipo de dados. A Figura 1 (a) ilustra como ocorre SI no desenvolvimento OO.

Na verdade, interesses são modularizados por meio de diferentes abstrações providas pelas linguagens e paradigmas de programação. As abstrações básicas do desenvolvimento OO são as classes, os métodos e os atributos. No entanto, essas abstrações podem não ser suficientes para separar certos tipos de interesses que inerentemente atravessam componentes responsáveis pela modularização de outros interesses. Alguns exemplos tradicionais de tais interesses, chamados

[1] Do termo em inglês *separation of concerns*.

interesses transversais[2], são: persistência, segurança, auditoria e tratamento de exceções. Sem meios apropriados para sua separação em um sistema OO, os interesses transversais tendem a ficar *espalhados* e *entrelaçados*[3] a outros interesses. Um interesse é dito espalhado quando este afeta vários componentes do sistema e entrelaçado quando se mistura com outros interesses dentro de um módulo. Estas duas características dos interesses são indesejáveis porque causam maior dificuldade de entendimento, evolução e reuso dos artefatos de software.



(a) Separação em dados e funções

(b) Separação em dados, funções e aspectos

Figura 1 – Separação de interesses (a) bidimensional e (b) tridimensional

O Desenvolvimento de Software Orientado a Aspectos (DSOA) [42] [43] [66] tem emergido como um novo paradigma, complementar aos existentes, cujo objetivo é prover suporte à separação adequada de interesses transversais em módulos fisicamente separados. Pensando em termos abstratos, a orientação a aspectos introduz uma terceira dimensão de decomposição representada na Figura 1 (b). Além de decompor o sistema em dados e funções, o sistema é decomposto de acordo com os interesses transversais em módulos denominado aspectos. Aspectos são unidades modulares de interesses transversais que se associam aos outros módulos do sistema. A Figura 2 (a) apresenta graficamente um interesse transversal espalhado e entrelaçado pelos módulos de um sistema orientado a objetos, enquanto o diagrama da Figura 2 (b) representa a modularização deste interesse em um aspecto.

[2] Do termo em inglês *crosscutting concerns*.

[3] Dos termos em inglês *scattering* e *tangling*.

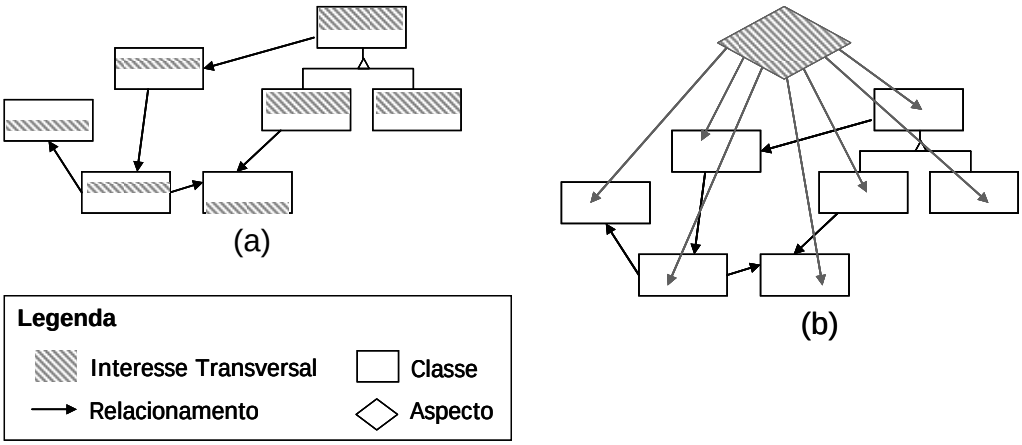


Figura 2 – Interesse transversal em sistemas (a) OO e (b) OA

Além desta nova unidade de modularização, as linguagens para o DSOA provêm mecanismos para compor classes e aspectos. Neste sentido, o paradigma de aspectos envolve duas etapas principais: (i) decomposição do sistema em partes não entrelaçadas; (ii) juntar essas partes novamente de forma significativa para se obter o sistema desejado. O processo de juntar as partes é chamado *combinação*[4]. Outro conceito importante neste paradigma são os *pontos de junção*[5] que especificam pontos específicos nos quais os módulos podem ser combinados. Um ponto de junção é um ponto bem definido na definição estática ou na execução do programa. Outros conceitos importantes deste paradigma são abordados na seção a seguir. Neste momento, torna-se importante ressaltar que as traduções utilizadas neste documento para os termos técnicos de DSOA foram especificadas em um workshop brasileiro da área e encontram-se disponíveis em [68].

2.1. AspectJ: uma Extensão de Java para Orientação a Aspectos

No contexto de DSOA, AspectJ [45] [66] é a linguagem de propósito geral mais conhecida e usada atualmente. AspectJ estende Java [17] com novas abstrações e mecanismos para combinar elementos destas duas linguagens. A combinação de elementos Java e AspectJ é feita pelo compilador de AspectJ (*ajc*). O compilador *ajc* transforma um programa escrito em AspectJ em *bytecode* Java, que pode ser executado por qualquer máquina virtual Java (JVM). Além deste

[4] Do termo em inglês *weaving*.

[5] Do termo em inglês *join point*.

compilador, AspectJ acrescenta novos conceitos e construções à linguagem Java, tais como: aspectos, *conjuntos de pontos de junção*[6], *adendos*[7] e *declarações intertipos*[8]. Estas construções são apresentadas a seguir.

Aspecto

O elemento principal de AspectJ é o aspecto. Cada aspecto assemelha-se a uma classe Java e pode ter tudo que uma classe tem, ou seja, definições de variáveis, métodos, etc. Além desses elementos, os aspectos podem conter outros elementos específicos de AspectJ, como conjunto de pontos de junção, adendos e declarações intertipos. Um aspecto pode afetar a estrutura estática ou dinâmica de uma ou mais classes e objetos de diferentes maneiras. A estrutura dinâmica é alterada pela especificação de conjuntos de pontos de junção e adendos, enquanto a estrutura estática é alterada através de declarações intertipos.

Conjunto de Pontos de Junção

Como mencionado anteriormente neste capítulo, um ponto de junção é qualquer ponto identificável e bem definido durante a execução de um programa. Diversos tipos de pontos de junção são permitidos na linguagem AspectJ, tais como: entradas e saídas de métodos, tratamento de exceções, acessos a variáveis de objetos, construtores, entre outros. Um conjunto de pontos de junção, ou conjunto de junção, é uma construção sintática de AspectJ para agrupar pontos de junção. Esta estrutura não tem nenhum comportamento, ela apenas denota onde e quando os aspectos terão efeito no programa. Para especificar um conjunto de junção podem ser usadas expressões regulares, permitindo grande flexibilidade em sua definição. Sua sintaxe básica é ilustrada no exemplo a seguir:

```
public pointcut printObject() : call (String *.toString());
```

O exemplo acima mostra os quatro elementos principais de um conjunto de pontos de junção. O primeiro é a declaração de restrição de acesso, nesse caso público (`public`), mas conjuntos de junção também podem ser privados

---

[6] Do termo em inglês *pointcut*.

[7] Do termo em inglês *advice*.

[8] Da expressão em inglês *inter-type declaration*.

(`private`) ou protegidos (`protected`). Em seguida, a palavra reservada de AspectJ `pointcut` denota que estamos declarando um conjunto de pontos de junção. O terceiro elemento desta estrutura é um identificador (`printObject`, em nosso exemplo), que pode ou não ser acompanhado por parâmetros. Finalmente, depois dos dois (`:`), vem o tipo e uma expressão regular que representa os pontos de junção. Nesse caso, os pontos de junção são todas as chamadas a métodos `toString()` do sistema. É importante ressaltar que este documento não tem a intenção de oferecer um completo entendimento desta e outras construções sintáticas de AspectJ, mas apenas familiarizar o leitor com a sintaxe da linguagem. Para maiores detalhes devem-se consultar livros [45] e manuais [66] de programação AspectJ.

Adendo

Adendo, ou comportamento transversal, é uma estrutura que denota o que um aspecto deve fazer, ou seja, qual o comportamento do aspecto. Em termos mais formais, esta estrutura designa a semântica comportamental do aspecto. Adendos são comportamentos e estão associados a conjuntos de junção. Eles especificam não só o que será feito, na forma de uma seqüência de operações Java, mas também o momento em que serão feitas estas operações. Há três tipos de adendos:

1. *Anterior*: executa antes do ponto de junção;
2. *Posterior*: executa depois do ponto de junção;
3. *De contorno*: executa “em volta” do ponto de junção.

O adendo do tipo “anterior” é o mais simples. Ele simplesmente executa antes do ponto de junção. Um detalhe a observar é que se este adendo levantar exceção, o seu respectivo ponto de junção não será executado. A seguir um exemplo de adendo do tipo “anterior”:

```
before() : printObject() {
   System.out.println(“Antes de imprimir!”);
}
```

O adendo do tipo “posterior” executa após o ponto de junção. Existem três variações desse adendo que dependem de como terminou a execução do ponto de junção: se terminou normalmente, com uma exceção, ou de qualquer forma. Os exemplos a seguir ilustram estas variações:

```
after() : printObject() {
   // executa independente de como retornar
   // o ponto de junção
}

after() returning : printObject() {
   // executa se o ponto de junção terminar normalmente
}

after() throwing : printObject() {
   // executa se o ponto de junção sair com uma exceção
}
```

O adendo do tipo “de contorno” é utilizado para substituir a execução do ponto de junção pela execução do adendo. No entanto, o ponto de junção pode ser executado de dentro do corpo do adendo usando-se o comando `proceed()`. Todo adendo de contorno deve declarar um tipo a ser retornado. O exemplo a seguir apresenta este tipo de estrutura:

```
String around() : printObject() {
  String s = proceed();
  return s.toLowerCase();
}
```

Declaração Intertipo

Além de modificar o comportamento das classes de um sistema pelo uso de conjuntos de junção e adendos, um aspecto pode alterar a estrutura estática dos módulos do sistema. Isto é feito pelo mecanismo chamado de declaração intertipo que é uma forma do aspecto introduzir mudanças em classes, interfaces e outros aspectos. Exemplos de elementos que podem ser adicionados são variáveis, métodos e construtores. Um aspecto pode ainda acrescentar relacionamentos de herança a classes existentes de duas formas: (i) fazendo com que classes implementem (uma ou mais) interfaces; (ii) fazendo com que classes estendam outras classes. O exemplo de código abaixo mostra como a declaração intertipo pode ser utilizada para adicionar, respectivamente, uma variável, um método e uma superclasse a classe `Aluno`.

```
private int Aluno.senha;

public boolean Aluno.verificar(int s) {
   return (senha == s);
}

declare parents : Aluno extends Pessoa;
```

2.2. Alguns Exemplos de Utilização de Aspectos

Esta seção apresenta três exemplos de aspectos [66] que ilustram o uso da tecnologia de DSOA. O primeiro exemplo consiste em um aspecto para monitorar as falhas de um servidor. O segundo tem como objetivo permitir que objetos possam ser clonados. E o último exemplo desta seção utiliza um aspecto para registrar tempos de conexão entre clientes e servidor. Todos os três exemplos são demonstrados em trechos de código AspectJ e possuem dois objetivos principais: revisar as estruturas sintáticas de AspectJ em contextos de aplicação e motivar o uso do DSOA em diferentes problemas que envolvem interesses transversais.

```
01  public aspect FaultHandler {
02
03    private boolean Server.disabled = false;
04
05    private void reportFault() {
06      System.out.println("Failure! Please fix it.");
07    }
08
09    public static void fixServer(Server s) {
10      s.disabled = false;
11    }
12
13    pointcut services(Server s):
        target(s) && call(public * *(..));
14
15    before(Server s): services(s) {
16      if (s.disabled) throw new DisabledException();
17    }
18
19    after(Server s) throwing (FaultException e):
        services(s) {
20      s.disabled = true;
21      reportFault();
22    }
23  }
```

Figura 3 – Exemplo de aspecto para tratamento de falhas

O aspecto `FaultHandler` [66] apresentado na Figura 3 possui três das principais construções da linguagem AspectJ: declaração intertipo, conjunto de pontos de junção e adendos. A declaração intertipo (linha 03) é utilizada para introduzir o atributo `disabled` na classe `Server`. Este atributo informa se o servidor se encontra disponível ou não. Todas as chamadas a métodos do servidor são interceptadas pelo conjunto de junção `services` (linha 13). A idéia é que comportamentos de falha podem ser disparados em chamadas de métodos

públicos da classe `Server` e, portanto, estas chamadas são eventos interessantes para o aspecto e devem ser capturadas pelo conjunto de junção. A ocorrência destes eventos causa a execução do adendo anterior (linhas 15-17) e do adendo posterior (linhas 19-22). Além destas três construções de AspectJ, o aspecto `FaultHandler` também possui os métodos `reportFault()` (linhas 05-07) e `fixServer()` (linhas 09-11).

```
public interface Shape { }

public class Point implements Shape {
   ...
}

public class Line implements Shape {
   ...
}

public aspect CloneableShape {

   declare parents: Shape extends Cloneable;

   public Object Shape.clone()
        throws CloneNotSupportedException {
      makePolar();
      return super.clone();
   }
}
```

Figura 4 – Exemplo de aspecto para associação de papel a classes

A Figura 4 apresenta a interface `Shape`, duas classes que implementam esta interface (`Point` e `Line`) e o aspecto `CloneableShape` [66]. Este aspecto é responsável por associar o papel de clonável às classes fazendo uso de declarações intertipo. Lembrando que em Java todo objeto herda o método `clone()` da classe `Object`, entretanto, ele só é clonável se implementar a interface `Cloneable`. Além disso, frequentemente o método `clone()` é reescrito nas classes para adicionar algum comportamento específico, como o método `makePolar()` do exemplo. Desta forma, o aspecto `CloneableShape` associa a interface `Cloneable` e garante a implementação do método `clone` como requerido pelas classes sem replicação de código.

O último exemplo desta seção, apresentado na Figura 5, é um aspecto para monitorar o tempo de total de conexão de cada cliente com o servidor [66]. Neste exemplo, o aspecto `Timing` utiliza uma declaração intertipo para introduzir o atributo `totalConnectTime` (linha 3) na classe `Customer`. Este atributo armazena

o tempo total de conexão do cliente. O aspecto também declara em cada objeto do tipo `Connection` (linha 9) um `Timer` para registrar este tempo de conexão. Além dos mecanismos de declaração intertipo, o aspecto `Timing` também possui conjuntos de junção (linha 19) e adendos (linhas 15-17 e 21-25). Os adendos garantem que a contagem de tempo é iniciada assim que a conexão é completada e pára quando a conexão termina. É interessante notar que o adendo das linhas 15-17 utiliza um conjunto de pontos de junção anônimo enquanto o adendo das linhas 21-25 utiliza um nomeado `endTiming`.

```
01  public aspect Timing {
02
03     public long Customer.totalConnectTime = 0;
04
05     public long getTotalConnectTime(Customer cust) {
06        return cust.totalConnectTime;
07     }
08
09     private Timer Connection.timer = new Timer();
10
11     public Timer getTimer(Connection conn) {
12        return conn.timer;
13     }
14
15     after (Connection c):
        target(c) && call(void Connection.complete()) {
16        getTimer(c).start();
17     }
18
19     pointcut endTiming(Connection c):
        target(c) && call(void Connection.drop());
20
21     after(Connection c): endTiming(c) {
22        getTimer(c).stop();
23        c.getCaller().totalConnectTime +=
       getTimer(c).getTime();
24        c.getReceiver().totalConnectTime +=
       getTimer(c).getTime();
25     }
26  }
```

Figura 5 – Exemplo de aspecto para registrar tempo de conexão com servidor

3
Estado da Arte e Trabalhos Relacionados

Neste capítulo resumimos alguns trabalhos existentes na literatura que se relacionam à abordagem de avaliação proposta nesta dissertação. O objetivo de todo método quantitativo de avaliação é garantir a qualidade de um processo ou produto de software. Na Seção 3.1 são abordados alguns conceitos relacionados à qualidade de software. Em seguida, apresentamos as principais métricas de software (Seção 3.2) e estado da arte em avaliação quantitativa para DSOA (Seção 3.3). Na Seção 3.4 é feita uma revisão de ferramentas que dão suporte a este paradigma e para finalizar o capítulo são apontadas algumas limitações dos trabalhos relacionados.

3.1.
Qualidade de Software

A definição de qualidade de software não é simples, pois o termo qualidade é bastante subjetivo e também porque software é intangível. Qualidade, de uma maneira geral, pode ser definida como “atendimento aos requisitos” ou “adequado para uso” [40] [61]. Entretanto, qualidade para o produto de software requer uma definição mais específica do que as definições tradicionais. Assim, alguns autores como Stephen Kan [40] relacionam qualidade de software à ausência de *bugs*, podendo ser medida pela taxa de erros (erros por milhares de linhas de código) ou pela confiabilidade (horas de operação sem falhas). Staa [62] define qualidade de um artefato de software como um conjunto de propriedades a serem satisfeitas em determinado grau, de modo que este satisfaça às necessidades de seus usuários e clientes. Em relação às definições acima, pode-se observar que elas geralmente abordam características explícitas do software, ou seja, características que podem ser claramente documentadas na especificação de requisitos. Por outro lado, há um conjunto de propriedades ou requisitos implícitos que deve ser observado e frequentemente não é mencionado como, por exemplo, desejo de dispor-se de elevada manutenibilidade.

Em [40], Kan coloca qualidade de software como “conformidade a requisitos funcionais e de desempenho explicitamente declarados, a padrões de desenvolvimento claramente documentados e a características implícitas que são esperadas de todo software profissionalmente desenvolvido”. Ou seja, se o software se adequar aos seus requisitos explícitos, mas deixar de cumprir seus requisitos implícitos, sua qualidade será suspeita [40]. Desta forma, para o contexto desta dissertação, várias características implícitas devem estar presentes em um software para que este seja considerado um produto de qualidade. Algumas destas características são frequentemente descritas e trabalhadas na literatura [20] [40] [57] [61] como: manutenibilidade, flexibilidade, testabilidade, reusabilidade, concisão, modularidade, simplicidade ou facilidade de compreensão.

Como mencionado no Capítulo 2, o uso de técnicas de DSOA promete melhoria de muitos destes requisitos de qualidade. Entretanto, alcançar um nível satisfatório de qualidade, mesmo em um sistema orientado a aspectos, não é trivial e requer um acompanhamento por medição e por abordagens sistemáticas de avaliação. Além disso, um processo automatizado torna-se crucial devido ao grande volume de informação que pode ser extraído do sistema. Nas seções a seguir são apresentadas métricas de software (Seção 3.2) e como estas vêm sendo usadas na avaliação e melhoria da qualidade de software orientado a aspectos (Seção 3.3).

3.2. Métricas de Software

O desenvolvimento de grandes sistemas é uma atividade que consome muito tempo e recursos. Sabendo-se que cada etapa deste processo requer esforço, é preciso prover informações para que a equipe tome decisões, trace planos, agende atividades e aloque recursos. Desta forma as métricas de software tornam-se necessárias para identificar onde é necessário efetuar as buscas por melhoria nos artefatos gerados no processo de desenvolvimento, sendo ainda uma fonte crucial para a tomada de decisões [20]. Muitas métricas têm sido propostas na literatura [14] [37] [47] [57], porém, neste trabalho é dado enfoque em métricas de produto. Métricas de produto devem ser fáceis de serem obtidas, pois com diferentes versões de um software, há bastante informação a analisar. Preferencialmente,

estas métricas devem ser passíveis de serem automatizadas. Nesta seção são abordadas as principais métricas que podem ser aplicadas a sistemas implementados sobre o paradigma de desenvolvimento orientado a objetos (OO) ou orientado a aspectos (OA). Elas são classificadas em quatro categorias: tamanho, acoplamento, coesão e separação de interesses.

3.2.1. Métricas de Tamanho

Algumas métricas são utilizadas para identificar a concisão, ou tamanho, de um sistema e estas tipicamente contam o número de ocorrência de um determinado elemento, como o número de subsistemas existentes (pacotes em Java ou AspectJ) ou o número de classes. Em certos casos, o número de classes pode ser generalizado para o **Tamanho do Vocabulário (VS)** [57], o que inclui tanto classes, quanto interfaces e aspectos. Métricas de tamanho também podem contar o número de membros de um componente do sistema [40]. Nesta contagem podem ser feito distinção de tipos específicos de membros, como ocorre nas métricas **Número de Atributos (NOA)** [47] [57], **Número de Operações (NOO)** [20] [22] e **Peso das Operações por Componentes (WOC)** [13] [14] [57].

O **Número de Linhas de Código (LOC)** [20] [47] é uma das medidas mais simples que pode ser obtida de um software. Entretanto, vários fatores podem gerar uma diferença significativa no resultado de sua contagem. Na programação em assembler, em que cada linha de código corresponde a um comando, a definição desta métrica é clara. Com as linguagens de mais alto nível, a contagem de linhas de código pode ser influenciada, dentre outras coisas, pela consideração ou não de comentários, linhas em branco e declarações que não são executadas – como delimitadores, por exemplo. Portanto, dependendo da forma que é contado, o número de linhas de código pode ser fortemente influenciado pelo estilo de programação adotado. Existem padrões para contagem de linhas de código [20] que diminuem a influência de estilo do programador. Entretanto, estes padrões nem sempre são implementados pelas ferramentas. Uma métrica de tamanho menos influenciada pelo estilo de programação e que pode ser utilizada como alternativa a LOC é a contagem do **Número de Comandos (NOS)** [20] [22].

3.2.2. Métricas de Acoplamento

Métricas de acoplamento indicam o número de outros componentes que se relacionam a um determinado componente. A forma mais genérica de se medir acoplamento é feita pela métrica de **Acoplamento entre Componentes (CBC)** [13] [14]. Em CBC são considerados de forma homogênea todos os tipos de relacionamentos entre classes e aspectos como dependências por atributos, operações, heranças, conjuntos de junção ou declarações intertipo. Um tipo especial de acoplamento que é tratado de forma diferenciada por algumas métricas ocorre na hierarquia de herança entre classes e aspectos. Dois exemplos bem conhecidos são **Profundidade da Árvore de Herança (DIT)** [13] [14] [57] e **Número de Filhos (NOC)** [13] [14] [22]. DIT mede o número de níveis da hierarquia até que se atinja a raiz da árvore e NOC conta subtipos imediatos de um componente.

3.2.3. Métricas de Coesão

Existem muitos estudos na literatura em relação a medidas de coesão interna de componentes [13] [14] [37] [57]. A maior parte das métricas propostas se baseia em relacionamentos entre métodos. Um exemplo clássico é a métrica **Perda de Coesão em Operações (LCOO)** [13] [14] [57] que mede o quão interligado estão as operações em relação ao compartilhamento de atributos. O valor desta métrica pode ser obtido pela contagem do número de pares de operações que compartilham atributos, menos o número de pares de operações que não compartilham nenhum atributo.

3.2.4. Métricas de Separação de Interesses

Como visto no Capítulo 2, a motivação para o desenvolvimento orientado a aspectos é a modularização de interesses que podem não ser bem capturados por mecanismos e abstrações de outros paradigmas. Desta forma, novas métricas têm sido propostas recentemente para indicar o nível de espalhamento e entrelaçamento dos interesses através dos artefatos de software [22] [57]. Uma característica comum destas métricas, chamadas métricas de separação de

interesses (SI), é que elas precisam de uma atividade prévia de identificação dos elementos pertencentes aos interesses. Esta atividade é chamada de *sombreamento do interesse* [31] [57]. Além disso, tais métricas oferecem possibilidade de serem aplicadas tanto em sistemas OA como em sistemas OO e este fato viabiliza a comparação de resultados obtidos a partir de sistemas implementados segundo os dois paradigmas de desenvolvimento.

A métrica **Difusão de Interesse em Componentes (CDC)** [31] [57] conta o número de componentes cujo propósito principal é contribuir para a implementação do interesse avaliado. Além disso, CDC conta também o número de componentes que fazem referência aos componentes principais do interesse. Para melhor entender esta métrica, utilizamos o diagrama de classe da Figura 6 que apresenta a implementação orientada a objetos de um editor de figuras. Os elementos sombreados destacam o código utilizado para implementar o padrão de projeto *Observer* [26]. Supondo que se queira verificar o espalhamento do interesse referente ao padrão *Observer* neste sistema é obtido valor 5 para CDC, uma vez que todas as classes e interfaces do diagrama possuem algum código referente a este interesse.

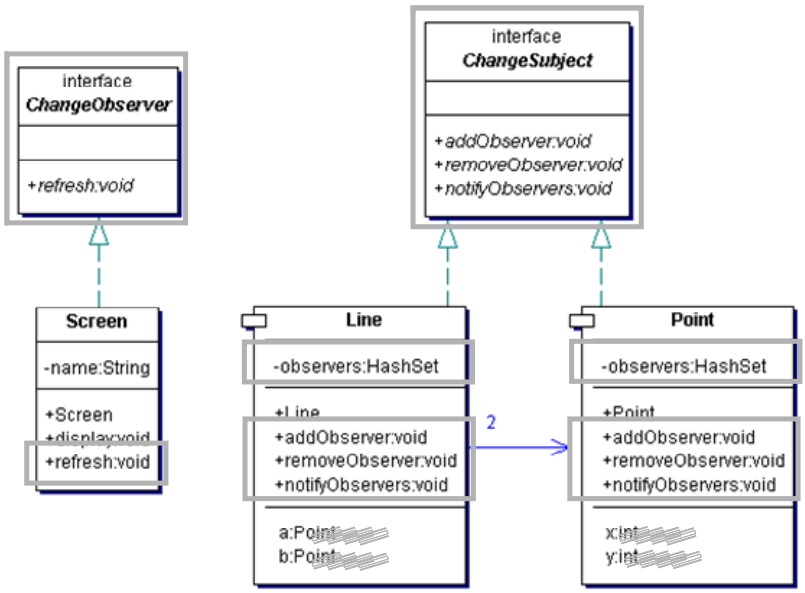


Figura 6 – Diagrama de classes OO do editor de figuras

Outras duas métricas de SI propostas na literatura [31] [57] são: **Difusão de Interesse em Operações (CDO)** e **Difusão de Interesse em Linhas de Código (CDLOC)**. A primeira (CDO) conta o número de operações cujo propósito principal é contribuir par a implementação do interesse avaliado. Esta métrica

conta ainda os métodos, construtores e adendos que acessam alguma das operações principais do interesse. A contagem do espalhamento de interesse referente ao padrão *Observer* sobre as operações no sistema da Figura 6 resulta em quinze. A segunda métrica (CDLOC) conta o número de pontos de transição existentes no código entre o interesse avaliado e os demais interesses do sistema. Para melhor entendimento desta métrica é apresentado o sombreamento de código da classe `Line` na Figura 7. Este sombreamento divide o código em áreas sombreadas e não sombreadas. Os pontos de transição são os pontos em que ocorre a mudança entre áreas sombreadas e não-sombreadas ou vice-versa. No caso da classe `Line` o número de pontos de transição é 8, ou seja, a contagem de CDLOC para esta classe resulta em oito.

```
public class Line
                  implements ChangeSubject {
   private HashSet observers = new HashSet();
   private Point a, b;

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

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

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

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

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

1 2 3 4 5 6 7 8

Figura 7 – Sombreamento da classe `Line` do editor de figuras

3.3. Avaliação de Software Baseado em Métricas

Como discutido na Seção 3.1, a qualidade de software depende de características implícitas como reusabilidade e manutenibilidade. No entanto, tais características são normalmente difíceis de serem avaliadas e só podem ser medidas tardiamente no processo de desenvolvimento. Para contornar este problema, as abordagens de avaliação existentes que são baseadas em métricas procuram usar atributos mensuráveis dos artefatos de software para predizer características de qualidade. Uma forma comum de se fazer este mapeamento é utilizando modelos de qualidade [4] [22] [57].

Modelos de qualidade são geralmente construídos sob a forma de uma árvore, em que os vértices mais próximos da raiz representam as características de qualidade que se deseja avaliar e os vértices mais próximos das folhas representam atributos de mais fácil medição. Pela bibliografia revisada, os primeiros modelos de qualidade construídos desta forma foram propostos na década de setenta por McCall *et al*. [50] e Boehm *et al*. [4]. Recentemente, entretanto, Sant'Anna *et al*. [57] propuseram um modelo de qualidade, revisado em [22], para avaliação de software orientado a aspectos. Este modelo utiliza métricas de software para predizer características de manutenibilidade e reusabilidade como apresentado a seguir.

3.3.1. Um *Framework* de Avaliação Orientado a Aspectos

Sant'Anna *et al*. [31] [57] propõem um *framework* baseado em métricas para avaliação de software orientado a aspectos. Além do conjunto de métricas, seu *framework* possui um modelo de qualidade para medir graus de reusabilidade e manutenibilidade. O modelo mapeia tais características de qualidade em atributos mensuráveis a partir do código fonte, como separação de interesses, coesão, acoplamento e tamanho. Este modelo é composto por quatro níveis: características externas, fatores, atributos internos e métricas. A figura a seguir ilustra o modelo de qualidade proposto por Sant'Anna.

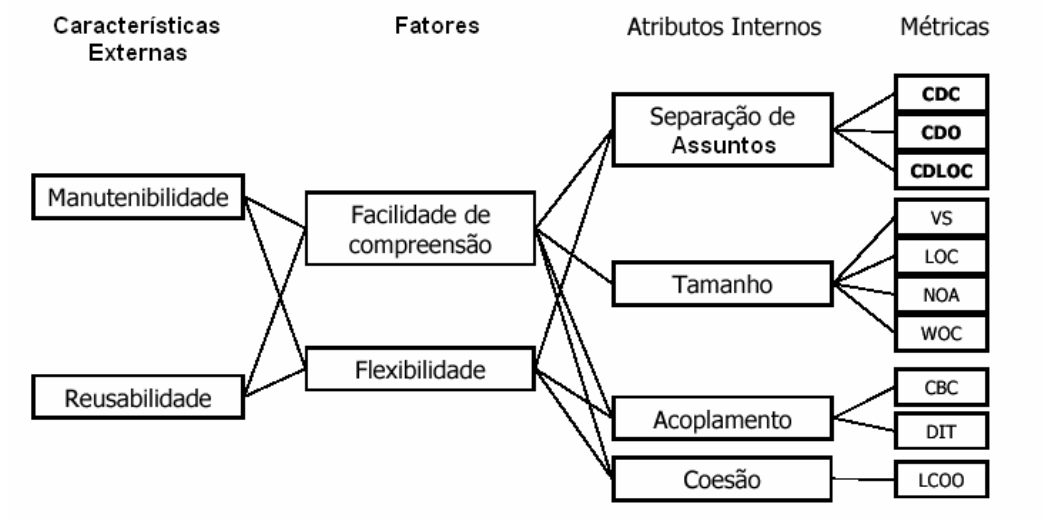


Figura 8 – Modelo de qualidade proposto por Sant'Anna *et al*. [57]

As características de qualidade são colocadas nos dois primeiros níveis do modelo de qualidade apresentado na Figura 8. Manutenibilidade e a reusabilidade estão presentes no primeiro nível e são denominadas características externas. Flexibilidade e simplicidade (ou facilidade de compreensão), presentes no segundo nível, são denominadas fatores. Esta distinção é feita porque, segundo os autores [31] [57], simplicidade e flexibilidade influenciam as duas características de qualidade do nível anterior. Além disso, o modelo de qualidade é originalmente proposto para avaliar a manutenibilidade e a reusabilidade de software orientado a aspectos.

No terceiro nível do modelo são colocados os quatro atributos mensuráveis a partir de artefatos de software: separação de interesses, tamanho, acoplamento e coesão. Estes atributos se baseiam em princípios bem estabelecidos da Engenharia de Software e conectam aos fatores do nível anterior e a um conjunto de métricas no nível seguinte. De acordo com o atributo que se propõe medir, as métricas se dividem em quatro categorias relativas aos quatro atributos internos. A separação de interesse é medida por **Difusão de Interesse em Componentes (CDC)**, **Difusão de Interesse em Operações (CDO)** e **Difusão de Interesse em Linhas de Código (CDLOC)**. O tamanho do sistema é medido em função do **Tamanho do Vocabulário (VS)**, **Número de Linhas de Código (LOC)**, **Número de Atributos (NOA)** e **Peso das Operações por Componentes (WOC)**. O grau de acoplamento entre os componentes é medido por **Acoplamento entre Componentes (CBC)** e **Profundidade da Árvore de Herança (DIT)**.

Finalmente, a coesão de cada módulo do sistema é medida pela **Perda de Coesão em Operações (LCOO)**. Métricas de software, incluindo os presentes neste modelo de qualidade, são abordadas na seção anterior deste documento.

3.4. Ferramentas de Desenvolvimento Orientado a Aspectos

No contexto de suporte automatizado para o DSOA, uma grande quantidade de ferramentas está disponível e pode ser encontrada na literatura [13] [15] [56] [58]. Entretanto, a maioria destas ferramentas é destinada à visualização [56] e *mineração*[9] [15] [58] de interesses transversais. Três ferramentas de suporte ao DSOA são brevemente apresentadas nesta seção: *Concern Manipulation Environment (CME)* [15], *Feature Exploration & Analysis Tool (FEAT)* [56] e uma ferramenta de medição [13]. As duas primeiras propõem formas de encontrar e visualizar elementos que implementam um interesse transversal específico.

3.4.1. Concern Manipulation Environment (CME)

O *Concern Manipulation Environment (CME)* [15] é na verdade um conjunto de ferramentas que promete um suporte ao DSOA através das várias fases do ciclo de vida do software. Este ambiente de programação é apoiado pela IBM [15] e permite que o desenvolvedor utilize diversas ferramentas e tecnologias simultaneamente. Inicialmente, o CME provê suporte a duas importantes tecnologias de DSOA: AspectJ [45] e separação multidimensional de interesses com Hyper/J [64]. Além destas, a proposta do ambiente é que este seja aberto para incorporar outras tecnologias e ferramentas.

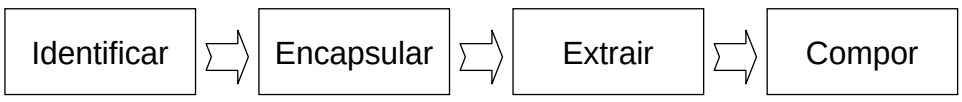


Figura 9 – Atividades do *Concern Manipulation Environment*

O CME trabalha para melhorar a separação de interesses em sistemas de software oferecendo suporte a quatro atividades que ajudam os desenvolvedores a amenizar problemas de código espalhado e entrelaçado. As duas primeiras

[9] Tradução para o termo *mining*.

atividades do CME são não-invasivas, ou seja, os desenvolvedores podem usá-las para adquirir melhor entendimento e organização do software sem ter que fazer uso explicitamente de DSOA. As duas últimas atividades são aplicações de técnicas do DSOA para extração e composição de interesses do sistema. Estas quatro atividades são apresentadas na Figura 9 e descritas a seguir:

1. **Identificação de interesses**: códigos existentes frequentemente envolvem uma variedade de interesses e muitos deles não foram adequadamente encapsulados em módulos. A atividade de identificação de interesses procura responder a questões como: "Quais partes do software pertencem a este ou aquele interesse?". Em CME, a identificação de um interesse envolve exploração dos artefatos usando uma combinação de navegação, consultas, análise e mineração.
2. **Encapsulamento de interesse**: uma vez que os elementos sintáticos do interesse tenham sido identificados, o desenvolvedor pode encapsular o interesse em uma estrutura lógica. Ou seja, os elementos sintáticos do interesse e seus relacionamentos são representados em um modelo do CME como elementos de primeira ordem.
3. **Extração de Interesse**: como mencionado na atividade anterior, o encapsulamento de um interesse resulta em uma separação lógica. Para muitos propósitos este grau de separação é suficiente, mas para outros o desenvolvedor pode querer sua separação física, extraindo o código em um artefato separado.
4. **Composição de interesses**: os artefatos dos interesses que foram fisicamente separados podem ser integrados (ou compostos) de diferentes formas. O resultado desta composição é um software que contenha as funcionalidades de todos os interesses envolvidos.

3.4.2. Feature Exploration & Analysis Tool (FEAT)

Feature Exploration & Analysis Tool (FEAT) [56] é uma ferramenta implementada como um *plugin* da plataforma Eclipse [66] que permite localizar, descrever e analisar o código que implementa um interesse em um sistema Java. O ambiente CME e a ferramenta FEAT se propõem a resolver problemas semelhantes do desenvolvimento de software. Ambos abordam a identificação dos

elementos de um determinado interesse e sua visualização de forma mais estruturada. Entretanto, utilizando a estrutura de navegação de FEAT pode-se localizar o código que implementa um interesse e salvá-lo em uma representação abstrata chamada *grafo de interesses*[10] [56]. Esta representação pode também ser usada para investigar os relacionamentos existentes entre o interesse capturado e o código base da aplicação.

O desenvolvedor cria um grafo de interesses pela aplicação de iterativas consultas no modelo do programa e determinando quais elementos e relacionamentos retornados pelas consultas contribuem para implementação do interesse desejado. FEAT disponibiliza um conjunto predefinido de consultas que se classificam em dois grupos:

- *Fan-in*: retorna todos os vértices no modelo do programa que depende do elemento selecionado.
- *Fan-out*: retorna todos os vértices ligados ao elemento selecionado através de arestas que saem deste elemento.

3.4.3. Uma Ferramenta de Medição

Ceccato e Tonella [13] argumentam sobre as diferenças existentes entre os desenvolvimentos OO e OA, especialmente no que se referem às novas formas de acoplamento introduzidas pelo paradigma de aspectos. Estes autores apresentam um conjunto de métricas e uma ferramenta para sua automatização. As métricas são basicamente extensões de métricas OO de Chidamber e Kemerer [14], exceto pelas de acoplamento que são revisadas para se adequar às novas abstrações do DSOA. A ferramenta proposta somente aplica medições em programas descritos na linguagem AspectJ [66] e a análise sintática do código é feita utilizando a linguagem de transformação de programas TXL [16].

A Figura 10 apresenta os cinco módulos que compõem a ferramenta de medição orientada a aspectos de Ceccato e Tonella. O primeiro módulo da ferramenta recebe como entrada toda classe, interface e aspecto do programa e faz uma engenharia reversa OO. Ou seja, detecta a estrutura dos componentes em termos de seus atributos, operações e relacionamentos de herança. Toda essa

[10] Tradução do inglês *Concern Graph.*

informação é armazenada em uma estrutura de dados. O segundo módulo faz a engenharia reversa mais avançada em que cada aspecto é processado para detecção de declarações intertipo. O resultado desta atividade também é armazenado na estrutura de dados utilizada pelo módulo anterior. O módulo seguinte da ferramenta detecta os relacionamentos de chamada de método e acesso a atributo. No quarto módulo é feita a resolução dos pontos de junção existentes no código dos aspectos, e então, os adendos são associados aos elementos interceptados pelos pontos de junção. Finalmente, o último módulo da ferramenta é responsável pela aplicação das métricas. O cálculo do valor de cada métrica é feito apenas pela aplicação de consultas à estrutura de dados.

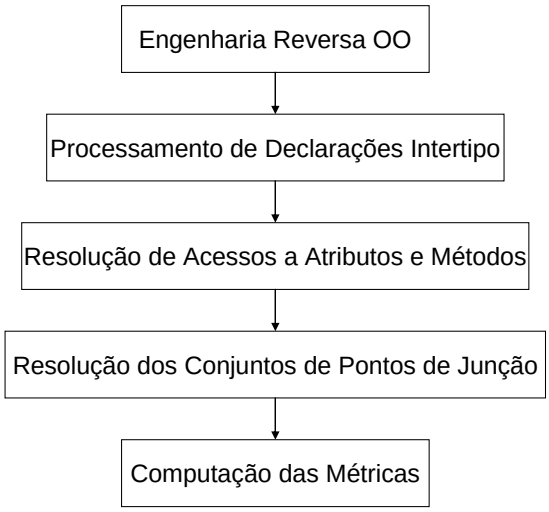


Figura 10 – Módulos da ferramenta de Ceccato e Tonella [13]

3.5. Limitações dos Trabalhos Relacionados

A qualidade de software deve ser acompanhada durante todo o processo de desenvolvimento. Nas seções 3.1 e 3.2 vimos que as abordagens de avaliação mais difundidas utilizam medições de atributos quantificáveis para predizer as características de qualidade dos artefatos. Sant'Anna *et al.* [57] propõem um *framework* de avaliação para desenvolvimento orientado a aspectos baseado em métricas (Subseção 3.3.1). Além do conjunto de métricas, seu *framework* possui um modelo de qualidade que mede graus de reusabilidade e manutenibilidade. Por outro lado, tal abordagem não inclui um conjunto de regras explícitas para auxiliar a interpretação dos resultados das medições e que possam ser automatizadas. Esta

ausência deixa uma lacuna entre o processo de medição e a interpretação dos valores. Além disso, Sant'Anna *et al*. [57] não apresentam um método sistemático para guiar o uso das métricas e uma forma de coordenar a seqüência de atividades para análise dos valores obtidos. Neste contexto, o trabalho apresentado nesta dissertação se caracteriza como uma extensão do trabalho de Sant'Anna *et al*. e tem o intuito de preencher lacunas deixadas pelo trabalho anterior.

Diversas ferramentas de suporte ao DSOA têm sido propostas e implementadas [13] [51] [56] [58], sendo três delas brevemente apresentadas na Seção 3.4. Duas destas ferramentas, CME e FEAT, têm como propósito auxiliar a identificação de elementos que implementam um determinado interesse. Este tipo de ferramenta pode ser considerado como complementar à abordagem desta dissertação, uma vez que elas tratam uma parte do problema não endereçado neste trabalho. A terceira ferramenta (Subseção 3.4.3), proposta por Ceccato e Tonella [13], aborda a medição de software orientado a aspectos. Essa ferramenta se asemelha ao módulo de medição da ferramenta proposta nesta dissertação (Capítulo 6). Porém, Ceccato e Tonella não apresentam um método organizado em etapas para interpretação dos valores e avaliação dos sistemas. Tal ferramenta se limita ao processo de medição, não dando suporte automatizado à identificação de eventuais problemas nas implementações sendo avaliadas. Este suporte pode ser dado pela definição e implementação de regras heurísticas que auxiliem a análise das medidas obtidas. Outra diferença importante entre a ferramenta de Ceccato e Tonella [13] e a proposta nesta dissertação diz respeito ao conjunto de métricas suportadas, por exemplo, medições de separação de interesses não são suportadas pela ferramenta daqueles autores.

4
O Método de Avaliação

O DSOA tem se tornado um importante tópico de pesquisa nos últimos anos, entretanto, pouca atenção tem sido dada à avaliação de software orientada a aspectos nas fases de projeto e implementação. Geralmente são encontrados na literatura apenas alguns estudos [28] [30] [36] que avaliam a qualidade de implementações AspectJ com foco em características relacionadas à separação de interesses. A principal razão para este problema é o fato de ser difícil avaliar múltiplos fatores sem uma abordagem sistemática de análise. Como resultado, engenheiros de software têm assumido que a propriedade que mais impacta a qualidade no DSOA é a separação de interesses. Por outro lado, como visto no Capítulo 3, alcançar elevada qualidade em software não é trivial e depende, além da separação de interesses, de outros importantes atributos da Engenharia de Software como coesão, acoplamento e tamanho.

Neste capítulo apresentamos um método quantitativo de avaliação de software orientado a aspectos que cobre as fases de projeto detalhado e implementação. Este método está apoiado sobre duas hipóteses principais:

a) avaliação de atributos relevantes é um pré-requisito para se alcançar um software de alta qualidade; e
b) avaliação abrangendo diversos fatores é essencial para que engenheiros de software façam um julgamento mais justo de diferentes soluções alternativas.

O método proposto é progressivo e estruturado em passos bem definidos, como apresentado na Figura 11. Este método é dito progressivo porque a cada iteração é obtido um novo conjunto de artefatos que compõe a solução, permitindo uma melhoria contínua até que se alcance uma solução de qualidade satisfatória. Cada iteração do método é composta de quatro passos, ou atividades, chamados medição, aplicação de regras heurísticas, análise de possíveis problemas e refatoração. Estas atividades devem ser sequencialmente seguidas e se dividem em processo de avaliação e processo de melhoria, como ilustrado na Figura 11.

Além das atividades, o método é suplementado por três conjuntos de recursos orientados a aspectos. São eles: métricas, regras heurísticas e refatorações. Estes recursos e os artefatos de entrada e saída são apresentados na Seção 4.1 e as atividades na Seção 4.2.

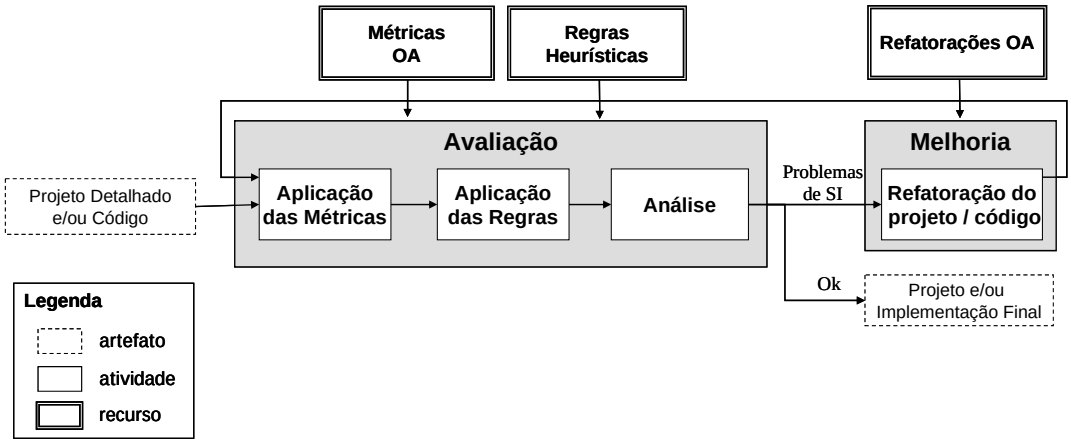


Figura 11 – Método progressivo para avaliação de software

Como mencionado anteriormente, o método proposto permite a avaliação de artefatos de software em duas fases do processo de desenvolvimento: projeto detalhado e implementação. Na fase de projeto detalhado, as informações disponíveis são mais abstratas e, portanto, as métricas, regras heurísticas e refatorações são mais genéricas. Mesmo neste nível, as quatro atividades do método devem ser seguidas. É importante ressaltar que a utilização do método na fase de projeto é independente da linguagem de programação. Mesmo sendo aplicado na fase de projeto detalhado, o método deve ser re-aplicado na fase de implementação, uma vez que os artefatos são refinados nesta fase e alguns atributos de qualidade podem ter sido violados. Na fase de implementação, os recursos do método são dependentes de características da linguagem de programação utilizada. No caso desta dissertação, as métricas e heurísticas para a fase de implementação são definidas em função das características de Java [17] e AspectJ [45].

O método emergiu de estudos empíricos [8] [27] [44] realizados no Laboratório de Engenharia de Software (LES) desta instituição de ensino. Estes estudos foram comumente utilizados na avaliação da qualidade de software orientado a aspetos e na comparação entre versões OO e OA de um mesmo sistema. Após a definição da primeira versão do método [21], outros estudos

foram importantes para a sua evolução. Alguns destes estudos experimentais foram realizados no decorrer deste curso de mestrado e estes são abordados no Capítulo 7. Como mencionado no Capitulo 1, alguns dos estudos contaram com a colaboração de outros grupos de pesquisa.

4.1. Artefatos e Recursos do Método

Dependendo da fase em que o método é aplicado (projeto detalhado ou implementação), os artefatos de entrada e saída podem variar. No nível de projeto detalhado os artefatos avaliados são diagramas baseados em UML [5] para representação do sistema. Estes diagramas devem incluir tanto diagramas estruturais (e.g. diagrama de classes e componentes [5]) como comportamentais (e.g. diagramas de seqüência e colaboração [5]). Além disso, é importante uma representação dos elementos que constituem cada interesse para que se possam calcular as métricas de separação de interesses, ou seja, a representação dos diagramas deve suportar a atividade de sombreamento como aparece na Figura 6 (Seção 3.2). Para utilização do método no nível de implementação, pode ser usado como artefato de entrada o próprio código que implementa o sistema, além de uma representação dos interesses neste artefato. Tanto no nível de projeto como no nível de implementação, os artefatos de saída do método são os artefatos de entrada que podem, ou não, ter sido reestruturados durante o processo.

Além dos artefatos de entrada e saída, o método proposto depende de três tipos de recursos orientados a aspectos. No processo de avaliação, são necessários conjuntos de métricas e de regras heurísticas. As métricas introduzem informações quantitativas enquanto as regras complementam com algum raciocínio qualitativo. Para o processo de melhoria é preciso que sejam definidas refatorações orientadas a aspectos para solucionar os problemas detectados na fase anterior. Estes três recursos são discutidos nas subseções a seguir. Na Subseção 4.1.1 são definidas três novas métricas orientadas a aspectos utilizadas em nossos estudos. As regras heurísticas são brevemente motivadas na Subseção 4.1.2, pois uma discussão mais detalhada é postergada para o Capítulo 5. Na Subseção 4.1.3 é motivado o uso de refatorações orientadas a aspectos.

4.1.1. Métricas Orientadas a Aspectos

O método de avaliação depende de um conjunto de métricas orientadas a aspectos para separação de interesses, coesão, acoplamento e tamanho. A maioria das métricas utilizadas em nossos estudos experimentais (Capítulo 7) encontra-se definida na literatura e aparece na Seção 3.2. Entretanto, três novas métricas são propostas neste trabalho para complementarem as métricas utilizadas. Estas três novas métricas apresentadas neste trabalho foram elaboradas a partir de estudos experimentais e têm como objetivo fornecer mais informações ao método de avaliação. Uma vez que várias métricas têm granularidade fina, um grande volume de informação é fornecido para as regras heurísticas (Capítulo 5). Para cada uma das três métricas desta subseção é apresentada uma definição, a relevância de sua utilização e um exemplo de como esta métrica é aplicada.

Número de Atributos do Interesse (NOAconcern)

Definição: Dado um interesse *I*, um componente *C* e um conjunto *A* de *n* atributos definidos em *C*, $A = \{ a_1, a_2, ..., a_n \}$. O número de atributos do interesse *C*, *NOAconcern(C)*, é definido como a cardinalidade do conjunto *B*, $|B|$, tal que $B \subseteq A$, $a_i \in B$ se e somente se a_i implementa *I*. Ou seja, NOAconcern conta o número de atributos de um componente cujo propósito principal é a implementação do interesse avaliado.

Relevância: Esta métrica mede o grau de espalhamento de um interesse pelos atributos de um componente. Além disso, ela mede o quanto os atributos deste componente são destinados à implementação do interesse avaliado. Uma intuição por trás disso é que quanto menos atributos são destinados ao interesse, menor é a dedicação do componente àquele interesse e, portanto, este interesse provavelmente deve ser modularizado em outro componente.

Exemplo: A Figura 12 mostra o diagrama de classes de uma implementação OO do padrão *Mediator* [26] que é parte de um dos estudos experimentais descritos no Capítulo 7. A área destacada da figura mostra os elementos que contribuem para a implementação do papel *Colleague* deste padrão. O papel *Colleague* é o interesse que se deseja avaliar neste sistema. O propósito principal da interface `GUIColleague` é contribuir para implementação deste interesse e a

classe `Button` que implementa `GUIColleague` também contribui parcialmente. A classe `Button` possui o atributo `mediator` que é parte do papel *Colleague*, portanto, na contagem da métrica NOAconcern para o interesse destacado no diagrama, a classe `Button` recebe valor 1 (um). As demais classes do sistema recebem valor 0 (zero), pois elas não possuem nenhum atributo destinado à implementação do papel *Colleague* do padrão *Mediator*.

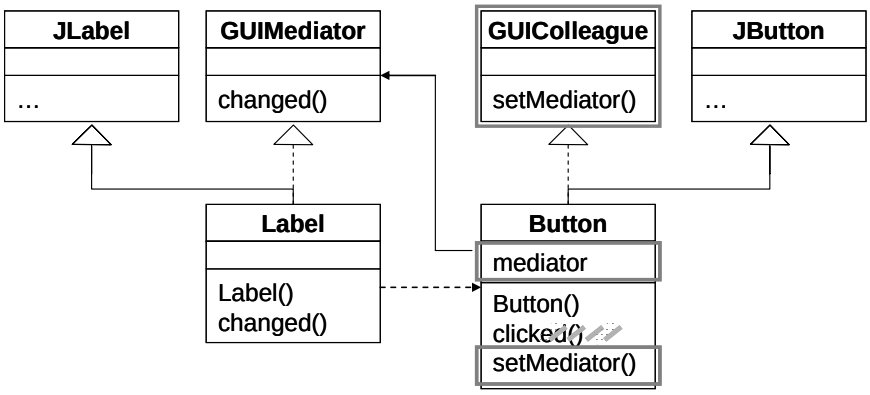


Figura 12 – Diagrama de classes do padrão *Mediator*

Número de Operações do Interesse (NOOconcern)

Definição: Dado um interesse *I*, um componente *C* e um conjunto *O* de *n* operações definidas em *C*, $O = \{ o_1, o_2, ..., o_n \}$, onde cada operação pode ser um método, um construtor ou um adendo. O número de operações do interesse *C*, *NOOconcern(C)*, é definido como a cardinalidade do conjunto *D*, $|D|$, tal que $D \subseteq O$, $o_i \in D$ se e somente se o_i implementa *I*. Ou seja, NOOconcern conta o número de operações de um componente cujo propósito principal é implementar o interesse avaliado.

Relevância: NOOconcern mede o grau de espalhamento de um interesse pelos métodos, construtores e adendos de um componente. Além disso, esta métrica mede o quanto deste interesse é implementado nas operações do componente. Assim como na métrica NOAconcern, a intuição é que quanto menos elementos do componente são destinados ao interesse, menor é a dedicação do componente àquele interesse. Ou seja, se uma classe possui poucas operações cujo propósito é implementar um determinado interesse, provavelmente, este interesse possa ser separado em outro componente, como um aspecto.

Exemplo: Observando o diagrama de classes da Figura 12 notamos que algumas operações são total, ou parcialmente, dedicadas à implementação do papel *Colleague* do padrão *Mediator* [26]. Os elementos que implementam parcialmente este interesse encontram-se rabiscados (). O método `clicked()` da classe `Button` é um exemplo de operação que não é totalmente dedicado à implementação do papel *Colleague*, o código desta classe pode ser verificado na Figura 13. Na contagem da métrica NOOconcern, são contadas apenas operações que têm como propósito implementar exclusivamente o interesse, portanto, o método `clicked()`não é considerado. O resultado desta métrica é 1 (um) para os componentes `Button` e `GUIColleague`, pois estes possuem o método `setMediator()` que implementa exclusivamente o interesse avaliado. Os outros componentes do sistema não possuem elementos dedicados ao interesse e recebem valor 0 (zero) para esta métrica.

Número de Linhas de Código do Interesse (LOCconcern)

Definição: Dado um interesse *I*, um componente *C* e um conjunto *L* que inclui as *n* linhas de código definidas em *C*, $L = \{ l_1, l_2, ..., l_n \}$. O número de linhas de código do interesse *C*, *LOCconcern(C)*, é definido como a cardinalidade do conjunto *M*, $|M|$, tal que $M \subseteq L$, $l_i \in M$ se e somente se l_i implementa *I*. Ou seja, LOCconcern conta o número de linhas de código de um componente cujo propósito principal é implementar o interesse avaliado.

Relevância: As métricas NOAconcern e NOOconcern medem o grau de espalhamento de um interesse pelos elementos do componente já no nível de projeto. Entretanto, só no nível de implementação que se pode verificar o espalhamento do interesse pelas linhas de código utilizando a métrica LOCconcern. Neste nível, as três métricas são complementares e, em conjunto, informam melhor o quanto de um componente é dedicado ao interesse avaliado. Se todos os elementos (atributos, operações e linhas de código) de um componente implementam o mesmo interesse, pode-se dizer que este componente é modular. Por outro lado, se poucos elementos do componente são destinados ao interesse avaliado, é provável que este interesse seja mais bem separado utilizando técnicas de DSOA.

Exemplo: A Figura 13 mostra o código da classe `Button`, em que a parte sombreada destaca as linhas de código necessárias para implementar o papel *Colleague* do padrão *Mediator* [26]. O número de linhas de código deste interesse (LOCconcern) na classe `Button` é 6 (seis). É importante destacar que esta métrica não conta comentário nem linhas em branco.

```
public class Button extends JButton
                  implements GUIColleague {
   private GUIMediator mediator;

   public Button(String name) {
      super(name);
      this.setActionCommand(name);
      this.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            clicked();
         }
      });
   }

   public void clicked() {
      mediator.changed(this);
   }

   public void setMediator(GUIMediator mediator) {
      this.mediator = mediator;
   }
}
```

Figura 13 – Sombreamento da classe `Button` do padrão *Mediator*

4.1.2. Regras Heurísticas de Avaliação

Além de um conjunto de métricas, o método de avaliação proposto é baseado em regras heurísticas que são aplicadas sobre o resultado das medições. O objetivo destas regras é apontar potenciais problemas no projeto ou implementação relacionados aos interesses transversais e que não são trivialmente detectáveis. Estes problemas incluem:

a) Interesses que não são facilmente identificados como interesses transversais.
b) Partes de um interesse transversal que não foi modularizado em um projeto OA.

c) Interesses espalhados e entrelaçados pela própria estrutura dos aspectos, como replicação de métodos na hierarquia ou repetição do uso de declaração intertipo.
d) Aspectos com problemas de coesão, acoplamento e/ou concisão devido a um critério de decomposição OA equivocado.

A aplicação das regras adverte o desenvolvedor de possíveis problemas causados por interesses transversais. Entretanto, isso não garante que os problemas existam realmente ou que o software precise ser mudado. Caso as regras levantem alguma advertência, o desenvolvedor deve analisar os artefatos do sistema para encontrar a razão desta advertência. Estas regras avaliam, de forma associada a SI, questões relacionadas a outros atributos de software. Ou seja, elas indicam quando um interesse transversal está afetando negativamente outras características de qualidade do software. Na realidade, a grande contribuição das regras é que elas carregam informações que ajudam o desenvolvedor a se concentrar em certas partes do sistema que são possivelmente problemáticas. O conjunto de regras heurísticas que compõe o método é apresentado no Capítulo 5.

4.1.3. Refatorações Orientadas a Aspectos

Refatorações são alterações feitas a artefatos de software que não alteram o comportamento observável do sistema [25] [52], ou seja, o programa deve produzir as mesmas saídas para um mesmo conjunto de entradas, antes e após a refatoração. O uso deste mecanismo tem como propósito melhorar a estrutura do sistema de tal forma a fazê-lo mais reutilizável, manutenível e fácil de ser entendido. A forma mais comum de se identificar oportunidades de refatoração são os *bad smells* [25]. *Bad smells* são sintomas que sugerem problemas no software e que podem ser removidos pelo uso de refatoração. Entretanto, estes sintomas não fornecem critérios precisos de quando uma refatoração é necessária e o desenvolvedor deve avaliar quando existe necessidade ou não de mudanças no software.

Refatorações têm sido propostas e são frequentemente utilizadas em sistemas orientados a objetos [25]. No contexto de DSOA, também existe a necessidade de refatorações que permitam a manipulação de artefatos na presença de aspectos. As refatorações orientadas a aspectos devem tratar dois novos tipos

de problemas provenientes deste paradigma: (i) extração de interesses transversais para aspectos e (ii) reestruturação interna dos aspectos. Felizmente, muitas refatorações também têm sido propostas para possibilitar a remoção de *bad smells* em sistemas orientados a aspectos [35] [38] [52]. O método proposto nesta dissertação utiliza refatorações orientadas a aspetos para retirada de problemas (ou *bad smells*) apontados pela regras heurísticas. Entretanto, a definição de um conjunto de refatorações vai além do escopo deste trabalho. Em nossos estudos experimentais (Capítulo 7), quando necessárias, as refatorações para evolução do software foram feitas manualmente pelos próprios desenvolvedores da aplicação ou por uma equipe experiente no domínio da aplicação.

4.2. Atividades do Método

A partir da identificação dos interesses, o método apresentado propõe uma seqüência de atividades para avaliar o quão bem modularizados estão estes interesses no sistema. As atividades do método se classificam em duas categorias de processos: avaliação e melhoria. No processo de avaliação as atividades que devem ser seguidas são: medição, aplicação das regras heurísticas e análise. Caso seja encontrado algum problema, deve ser feita a atividade de refatoração no processo de melhoria do software.

4.2.1. Atividade de Medição

A primeira atividade do método é a aplicação de métricas nos artefatos de entrada. Esta atividade fornece informação sobre os atributos internos do software, tais como separação de interesse, acoplamento, coesão e tamanho. O principal objetivo do método de avaliação é ajudar o projetista a determinar se o uso de aspectos pode contribuir para a melhoraria da separação de interesses. Sendo assim, são aplicadas métricas de separação de interesses para se obter informações de quais interesses podem ser classificados como transversais. Antes da aplicação destas métricas, é necessário que os interesses estejam identificados e anotados nos artefatos do projeto detalhado ou código. Além das métricas de separação de interesses, são aplicadas medições auxiliares que fornecem

características mais gerais do sistema. A Tabela 1 mostra o conjunto atual de métricas usadas em nossos estudos experimentais.

Tabela 1 – Conjunto de métricas orientadas a aspectos do método

Atributos	Métricas	Ref.	Definições
Separação de Interesses	Difusão de Interesse em Componente (CDC)	[31] [57]	Conta o número de classes, interfaces e aspectos que contribuem para implementar o interesse avaliado.
	Difusão de Interesse em Linhas de Código (CDLOC)	[31] [57]	Conta o número de pontos em que existe uma transição entre o interesse avaliado e os outros interesses do sistema.
	Número de Atributos do Interesse (NOAconcern)	Seção 4.1.1	Para cada componente, conta o número de atributos que implementam o interesse avaliado.
	Número de Operações do Interesse (NOOconcern)	Seção 4.1.1	Para cada componente, conta o número de operações cujo propósito principal é implementar o interesse avaliado.
	Número de Linhas de Código do Interesse (LOCconcern)	Seção 4.1.1	Para cada componente, conta o número de linhas de código que implementam o interesse avaliado.
Tamanho	Tamanho do Vocabulário (VS)	[31] [57]	Conta o número de classes, interfaces e aspectos do sistema.
	Número de Atributos (NOA)	[31] [57]	Conta o número de atributos de cada classe, interface e aspecto.
	Número de Operações (NOO)	[22] [40]	Conta o número de operações de cada classe, interface e aspecto.
	Número de Linhas de Código (LOC)	[40] [57]	Conta o número de linhas de código de cada classe, interface e aspecto.
Acoplamento	Acoplamento entre Componentes (CBC)	[14] [57]	Conta o número de outras classes, interfaces e aspectos com os quais um componente se relaciona.
	Profundidade da Árvore de Herança (DIT)	[14] [57]	Mede o comprimento máximo da hierarquia de herança de um componente para a raiz da árvore.
	Número de Filhos (NOC)	[14] [22]	Conta o número de classes, interfaces ou aspectos que herdar diretamente do componente.
Coesão	Perda de Coesão em Operações (LCOO)	[14] [57]	Conta os pares de operações que não acessam atributos em comum, menos os pares de operações que acessam pelo menos um atributo em comum.

Como discutido no Capítulo 7, houve algumas variações no conjunto de métricas utilizadas em cada um dos cinco estudos experimentais. Estas variações ocorreram de forma a avaliar as vantagens e desvantagens das possíveis combinações de métricas. Por exemplo, no estudo *Telestrada* (Subseção 7.1.5) realizado em parceria com pesquisadores da Universidade Estadual de Campinas,

a métrica **Peso das Operações por Componentes (WOC)** foi utilizada em alternativa a **Número de Operações (NOO)**.

4.2.2. Atividade de Aplicação das Regras

Normalmente, a aplicação de métricas gera uma grande quantidade de números cuja interpretação não é trivial e consome tempo. Desta forma, terminada esta atividade de medição, regras heurísticas são aplicadas para ajudar na interpretação dos números, apontando os valores que possam indicar oportunidade de melhoria no software. A Tabela 2 apresenta um conjunto de regras heurísticas que são diretamente derivadas de algumas das métricas listadas na Tabela 1.

Tabela 2 – Conjunto de regras heurísticas baseadas em uma única métrica

	Regras	Métricas
01	**Se** um INTERESSE afeta vários componentes do sistema **então** este é um INTERESSE ESPALHADO	CDC
02	**Se** o código um INTERESSE se mistura ao código de outros interesses **então** este é um INTERESSE ENTRELAÇADO	CDLOC
03	**Se** os atributos e operações de um COMPONENTE não são fortemente relacionados **então** este é um COMPONENTE POUCO COESO	LCOO
04	**Se** um COMPONENTE depende de muitos outros componentes **então** este é um COMPONENTE ALTAMENTE ACOPLADO	CBC
05	**Se** um COMPOENTE está muito distante da raiz na hierarquia de herança **então** este é um COMPONENTE DE HERANÇA PROFUNDA	DIT

As duas primeiras regras da Tabela 2 podem ser usadas para alertar o desenvolvedor da existência de um interesse, possivelmente transversal, que se encontra espalhado e entrelaçado no código. Estas duas regras são suportadas, respectivamente, pelas métricas **Difusão de Interesse em Componente (CDC)** e **Difusão de Interesse em Linhas de Código (CDLOC)**. Esta tabela também apresenta outras três regras que são usadas para identificar componentes com baixa coesão (Regra 3) ou alto acoplamento (Regras 4 e 5). As três últimas regras são suportadas pelas métricas **Perda de Coesão em Operações (LCOO)**, **Acoplamento entre Componentes (CBC)** e **Profundidade da Árvore de Herança (DIT)**. A avaliação de outros atributos do sistema, como coesão e acoplamento, é importante porque a separação de interesses pode afetar tais

atributos [21] [27]. Ou seja, mesmo que um sistema possua boa separação de interesses, se muitos de seus componentes apresentarem baixa coesão e alto acoplamento, o projetista pode decidir fazer uma refatoração.

4.2.3. Atividade de Análise e Identificação de Problemas

As regras heurísticas propostas neste trabalho apóiam a interpretação do resultado de medições. Elas apontam possíveis problemas no projeto ou implementação do sistema e alertam o engenheiro de software. Porém, o uso destas regras não dispensa a análise feita por parte de um especialista humano. As regras indicam os pontos potencialmente problemáticos, mas os desenvolvedores têm que se certificar de que estes pontos são realmente problemas, e caso sejam, avaliar se uma refatoração de software é benéfica. Esta tarefa do método não pode ser completamente automatizada e é representada pela terceira atividade da Figura 11.

4.2.4. Atividade de Refatoração

Como definido na Subseção 4.1.3, refatorações são alterações aplicadas aos artefatos de software e muitas refatorações orientadas a aspectos têm sido propostas na literatura [25] [35] [38] [52]. A última atividade definida no método de avaliação é aplicação destas refatorações no software com o objetivo de melhorar sua estrutura. Esta atividade é utilizada para eliminação de problemas detectados no processo de avaliação, em especial, apontados pelas regras heurísticas. Para exemplificar o uso de refatorações orientadas a aspectos no processo de melhoria do software é apresentada uma implementação do padrão *Prototype* [14]. Esta implementação foi feita por Hannemann e Kiczales [36] e é parte de um dos nossos estudos experimentais.

A intenção do padrão *Prototype* é especificar a criação de objetos pela clonagem de uma instância da classe [14]. Este padrão define um único papel ou interesse, chamado *Prototype*, que define o comportamento para a clonagem de objetos. A instância do padrão, ilustrada na Figura 14, é usada para definir objetos `String` que podem ser clonados [36]. O diagrama de classes desta figura mostra a implementação OO em que as classes `StringPrototypeA` e `StringPrototypeB`

implementam a interface `Cloneable` e definem o método `clone()` para permitir a clonagem.

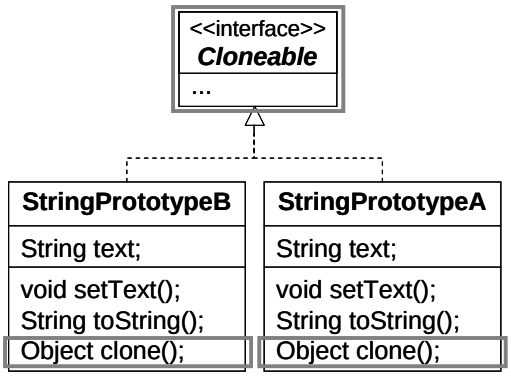


Figura 14 – Diagrama de classes OO do padrão *Prototype*

O resultado das atividades de avaliação na versão OO do padrão *Prototype* sugere a existência de um problema de separação de interesses. A Figura 14 destaca os elementos do padrão espalhados e entrelaçados pelo diagrama, o que confirma este resultado. Para solucionar o problema, desenvolvedores do sistema podem decidir usar refatorações OA e separar o interesse do padrão. Uma possível solução após a refatoração do sistema é apresentada na Figura 15. Nesta versão OA, o código que implementa o padrão *Prototype* está localizado nos dos aspectos `StringPrototypes` e `PrototypeProtocol`.

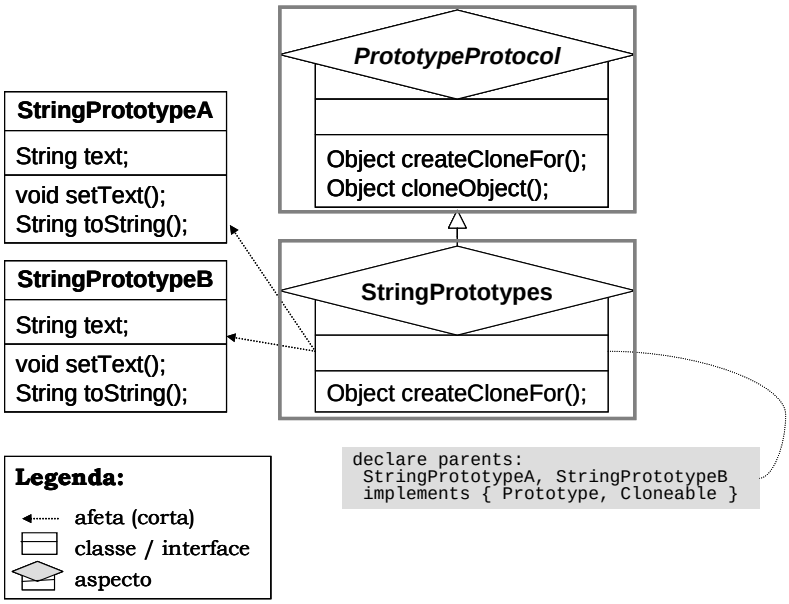


Figura 15 – Diagrama de classes OA do padrão *Prototype*

Um possível conjunto de refatorações aplicadas à versão OO (Figura 14) do padrão *Prototype* para transformá-la em uma solução OA (Figura 15) é sumarizado da seguinte forma:

1. Criar um aspecto vazio e nomeá-lo `StringPrototypes`;
2. Mover os métodos `clone()` das classes `StringPrototypeA` e `StringPrototypeB` para o aspecto `StringPrototypes` (*Move Method* [25]);
3. Criar um super-aspecto de `StringPrototypes` e nomeá-lo `PrototypeProtocol`;
4. Mover o método `clone()` do aspecto `StringPrototypes` para o aspecto pai `PrototypeProtocol` (*Pull up Method* [25]);
5. Criar uma interface `Prototype` interna ao aspecto `PrototypeProtocol` e usar *container introduction* [34] para adicionar o método `clone()` às classes `StringPrototypeA` e `StringPrototypeB`;
6. Substituir a implementação explicita da interface `Cloneable` feita pelas classes `StringPrototypeA` e `StringPrototypeB` por declaração intertipo (*Replace Implements with Declare Parents* [52]).

5
Regras Heurísticas

O conjunto de regras heurísticas apresentado neste capítulo apóia a interpretação dos resultados das medições na avaliação de sistemas. As regras (e o método apresentado no capítulo anterior) não assumem que a solução OA é sempre uma boa solução. Mesmo na presença de interesses transversais, as regras são abrangentes o suficiente para apontar problemas tanto em soluções OO quanto OA. Além disso, a abordagem também detecta, pelo uso de regras de acomplamento e coesão, os potenciais interesses transversais que não deveriam ser modularizados em aspectos. Cada regra é uma expressão baseada em métricas que indica certas características dos artefatos avaliados e podem apontar problemas potenciais. O principal objetivo das regras é fornecer informação mais abstrata ao desenvolvedor, evitando que este trabalhe diretamente com números e, portanto, facilitar a interpretação dos resultados. No contexto desta dissertação, o conjunto de regras pode ser usado na identificação de problemas nos artefatos de projeto detalhado e código. Tanto problemas de separação de interesses (SI), como de acoplamento e coesão podem ser apontados pelas regas heurísticas. Entretanto, o foco deste estudo é avaliação orientada a interesses e problemas de acoplamento e coesão só são relevantes quando estes problemas estão relacionados a uma ineficiente SI.

Como mencionado no capítulo anterior, a aplicação de regras heurísticas alerta o projetista sobre possíveis problemas causados por interesses transversais, no entanto, elas não garantem que os problemas existam realmente e que o software precise ser alterado. Portanto, mesmo quando as regras geram alertas, o projetista deve analisar o projeto e / ou código para verificar quais são as razões destes alertas. Essa tarefa representa a atividade de análise no método de avaliação apresentado no Capítulo 4. Na verdade, o benefício principal trazido pela utilização destas regras é que seus alertas contêm informações que ajudam o projetista a se concentrar em alguns artefatos do software que são possivelmente problemáticos. Depois da análise, se o projetista detectar que realmente existem

problemas causados por interesses transversais, ele pode reestruturar o sistema no sentido de melhor modularizar tais interesses. A reestruturação do software pode ser feito usando refatorações orientadas a aspecto como demonstrado na Subseção 4.2.4.

As regras heurísticas encontradas neste capítulo foram definidas a partir da experiência adquirida em estudos empíricos [8] [21] [27] que usam métricas. Estes estudos tanto avaliam a separação de interesses em sistemas quanto comparam projetos orientados a objetos (OO) a orientado a aspecto (OA) do mesmo sistema. Tais regras tornam-se muito importantes nas fases de projeto detalhado e código, devido ao grande número de elementos existentes nos artefatos (classes, métodos, atributos etc.) e, consequentemente, ao grande volume de dados a avaliar. Cada regra que pode ser expressa como um comando condicional usando a seguinte estrutura:

SE <condição> **ENTÃO** <conseqüência>

A *condição* envolve uma combinação de métricas e valores limites (Seção 5.4) para avaliação de interesses e artefatos, enquanto a *conseqüência* se apresenta como uma classificação do interesse ou artefato avaliado. O grupo de regras é dividido em duas categorias: regras de separação de interesses e regras de acoplamento e coesão. A primeira categoria de regras identifica problemas de projeto diretamente relacionados a uma imprópria modularização de interesses, enquanto a segunda categoria dá suporte à identificação de problemas de acoplamento e coesão que se relacionam aos problemas de SI. Apresentamos estas duas categorias de regras nas seções 5.2 e 5.3, respectivamente. Antes disso, na seção a seguir apontamos alguns problemas não triviais que podem ser resultantes de uma análise equivocada em atividades de medições. É importante ressaltar que estes problemas podem ser minimizados pela utilização do conjunto de regras heurísticas. Ao final deste capítulo são levantados alguns pontos relacionados à definição de valores limites para as regras.

5.1. Problemas de Interpretação em Medições

Uma grande dificuldade em se trabalhar com métricas é como lidar com seus resultados, ou seja, como toda aquela quantidade de dados pode ajudar na

melhoria da qualidade de software. Nesta seção examinamos alguns problemas que impactam a qualidade do sistema e estão relacionados a uma interpretação equivocada dos resultados de medições. Foram identificados seis destes problemas no decorrer do estudo: (a) alerta falso por interesse espalhado e entrelaçado; (b) alerta falso por alto acoplamento e/ou baixa coesão; (c) resultados não mostram um problema existente; (d) resultados não indicam onde está o problema; (e) conflitos entre valores medidos; e (f) métricas não relacionam os problemas existentes. Estes problemas, que ocorrem especialmente em avaliações orientadas a aspectos, são apresentados e exemplificados a seguir.

Problema A: Alerta falso por interesse espalhado e entrelaçado

O primeiro problema ocorre quando o resultado das métricas de SI adverte o desenvolvedor de um possível interesse que se espalha e entrelaça pelos artefatos do sistema. O desenvolvedor poderia pensar que se trata de um interesse transversal, entretanto, em uma análise mais cuidadosa verifica-se que, na verdade, tal interesse se encontra bem modularizado. Neste caso, os valores apontam um problema que não existe no sistema. Para ilustrar uma instância deste problema, é usado o padrão de projeto *Factory Method* [26].

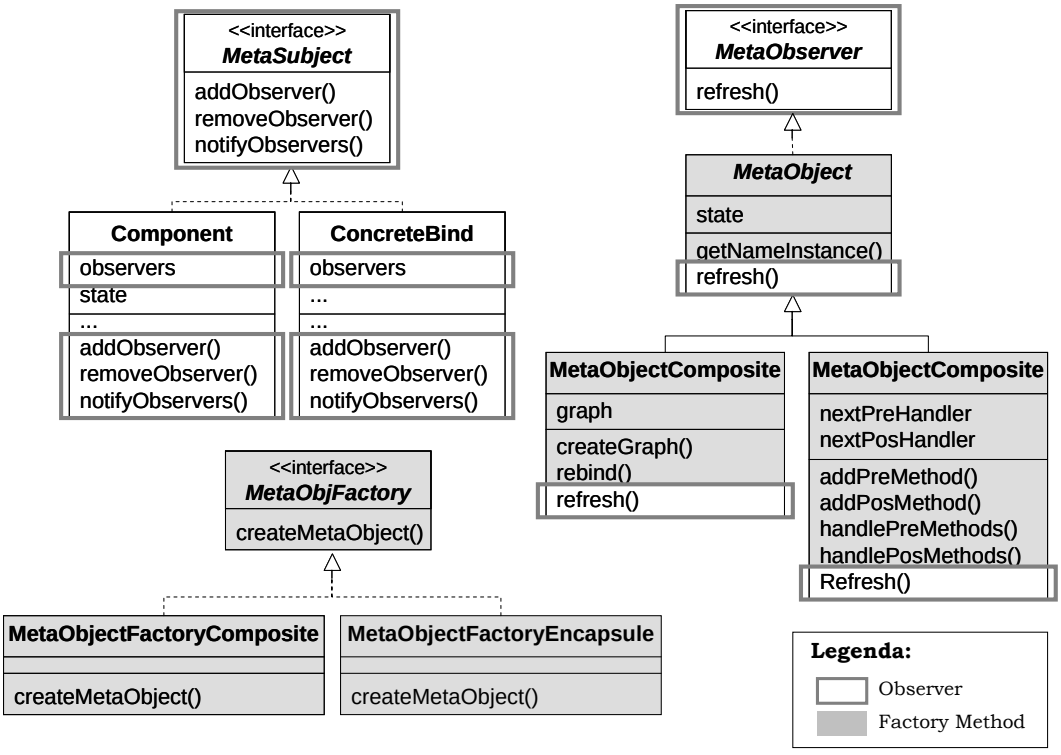


Figura 16 – Diagrama de classes OO destacando *Observer* e *Factory Method*

A Figura 16 apresenta o diagrama de classes parcial de um dos nossos estudos experimentais envolvendo um sistema de *middleware* reflexivo baseado no modelo *OpenOrb* [8] (Subseção 7.1.2). Neste diagrama são destacados os elementos que implementam os padrões *Factory Method* e *Observer* [26]. A aplicação de quatro métricas de SI neste subsistema resulta no conjunto de dados da Tabela 3. Estas métricas são CDC, CDLOC, NOAconcern e NOOconcern e encontram-se definidas na Tabela 1 da Subseção 4.2.1.

Tabela 3 – Resultado das métricas de SI para *Observer* e *Factory Method*

Componentes	**Observer**		**Factory Method**	
	CDC = 7	**CDLOC = 18**	**CDC = 6**	**CDLOC = 8**
	NOAconcern	**NOOconcern**	**NOAconcern**	**NOOconcern**
Component	1	3	0	0
ConcreteBind	1	3	0	0
MetaObject	0	1	1	1
MetaObjectComposite	0	1	1	2
MetaObjectEncapsu	0	1	2	4
MetaObjectFactoryComposite	0	0	0	1
MetaObjectFactoryEncapsule	0	0	0	1
MetaObjFactory	0	0	0	1
MetaObserver	0	1	0	0
MetaSubject	0	3	0	0

Em uma análise descuidada dos resultados apresentados na tabela acima, o desenvolvedor pode vir a acreditar que ambos os padrões são exemplos de interesses transversais – o que não é verdade para o padrão *Factory Method* [27] [36]. Esta interpretação errada é induzida pelos altos valores obtidos por estes padrões nas métricas. Em especial, os valores das métricas CDC e CDLOC para o padrão *Factory Method* são 6 e 8, respectivamente. Tais valores significam que seis componentes (dos 10 apresentados no diagrama) possuem código relativo ao padrão *Factory Method* e existem oito pontos em que há uma troca entre o interesse deste padrão e os outros interesses do sistema. Portanto, o padrão *Factory Method* encontra-se espalhado e entrelaçado pelos elementos do diagrama. Entretanto, isto não significa que este seja um interesse transversal, como na verdade não o é. Uma análise minuciosa do diagrama de classes (Figura 16) mostra que o padrão *Factory Method* encontra-se bem modularizado nos seis componentes em que ele se encontra. Além disso, o principal propósito destes seis componentes é implementar tal padrão [36]. Podemos verificar ainda, que existem alguns elementos do padrão *Observer* nos componentes que implementam o

padrão *Factory Method* causando entrelaçamento de interesses. Por todos esses fatores, pode-se concluir que os valores altos para o padrão *Factory Method* é resultado da incompleta modularização do padrão *Observer* e que as métricas não apresentam isto com clareza. Ou seja, caso o padrão *Observer* seja separado, o problema de entrelaçamento entre estes padrões é resolvido.

Problema B: Alerta falso por acoplamento e coesão

Na abordagem orientada a interesses proposta neste trabalho, um dos objetivos é conquistar uma melhoria em outros atributos do sistema, como acoplamento e coesão, a partir de uma melhor separação de interesses. Neste sentido, problemas de acoplamento e coesão somente são relevantes quando estes podem ser resolvidos por uma separação mais adequada dos interesses. Por outro lado, o elevado valor de algumas métricas de acoplamento e/ou coesão pode alertar o desenvolvedor de um possível problema. Em um caso menos grave, este alerta levaria o desenvolvedor a verificar se realmente existe algum problema que pode ser solucionado pela melhor SI. Em contrapartida, uma situação mais grave seria a busca de solução, possivelmente utilizando refatorações OA no projeto, quando o alerta não retrata problema grave ou que deva ser resolvido por técnicas de DSOA.

Tabela 4 – Resultado de acoplamento e coesão para o padrão *Factory Method*

Componentes	**Factory Method**			
	LCOO	**CBC**	**DIT**	**NOC**
ButtonCreator		3	2	0
ButtonCreator1		3	2	0
ButtonCreator2		3	2	0
GUIComponentCreator	**3**	**6**	1	**6**
LabelCreator		2	2	0
LabelCreator1		2	2	0
LabelCreator2		2	2	0
Main		1	1	0

Para ilustrar este problema de interpretação de números, utilizamos os resultados de medições feitas em uma implementação OO do padrão *Factory Method* [26] [36]. Esta implementação é uma extensão da versão proposta por Hannemann e Kiczales [36] e parte de nosso primeiro estudo experimental (Subseção 7.1.1). Os dados da Tabela 4 apresentam o resultado das medições neste estudo. Este resultado mostra valores elevados para o componente

`GUIComponentCreator` para as métricas LCOO, CBC e NOC em relação aos demais componentes do sistema. Em uma avaliação orientada a interesses, este fato pode levar o desenvolvedor a procurar uma forma de melhorar tais valores pela separação de interesses. Entretanto, o componente `GUIComponentCreator` não apresenta problema de SI, uma vez que seu único propósito é implementar o papel *Creator* do padrão *Factory Method*.

Problema C: Resultados não mostram o problema

Às vezes, as métricas não revelam os problemas existentes. Um caso típico ocorre quando um determinado interesse é conhecido e classificado como transversal, mas os resultados das métricas de SI não mostram problemas para este interesse. Este fato ocorre porque mesmo sendo um interesse transversal, pode haver situações em que não há um espalhamento e entrelaçamento muito intenso deste interesse. Uma instância do problema em que as métricas não mostram problemas aparece em nosso segundo estudo experimental (Subseção 7.1.2), quando avaliamos o padrão *Singleton* [26]. O diagrama de classes da Figura 17 destaca os elementos necessário para implementar este padrão e o padrão *Façade* [26] no estudo da *middleware* reflexiva baseada em *OpenOrb* [8] [9].

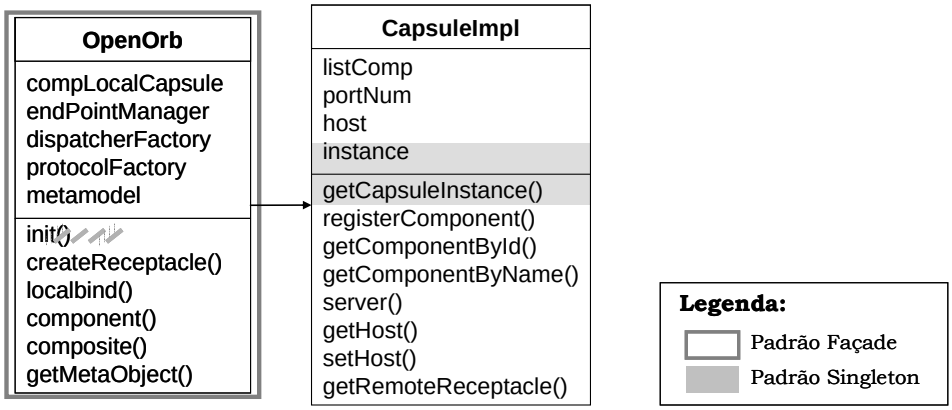


Figura 17 – Diagrama de classe OO destacando padrões *Façade* e *Singleton*

Tabela 5 – Resultado das métricas de SI para padrões *Façade* e *Singleton*

Componentes	**Façade**		**Singleton**	
	CDC = 1	**CDLOC = 2**	**CDC = 2**	**CDLOC = 4**
	NOAconcern	**NOOconcern**	**NOAconcern**	**NOOconcern**
CapsuleImpl	0	0	1	1
OpenOrb	5	6	0	0

O padrão *Singleton* [26] é classificado na literatura como um interesse transversal [9] [27] [28] [36], por dois motivos principais: (i) ele, tipicamente, se espalha pelos componentes e entrelaça aos outros interesses [27] e (ii) os componentes que implementam este interesse possuem outras funcionalidades na aplicação além de implementar o padrão [36]. Entretanto, olhando os dados da Tabela 5 temos a impressão de que o padrão *Singleton* se encontra bem modularizado, assim como o padrão *Façade*. Esta interpretação errada é resultado dos baixos valores apontados em todas as métricas de SI (CDC = 2, CDLOC=4, NOAconcern = 1, NOOconcern = 1). Note que mesmo o valor 4 da métrica CDLOC pode ser considerado baixo, pois este valor significa que existem apenas dois blocos de elementos que implementam tal interesse. Portanto, podemos concluir que neste caso as métricas de SI não mostram o problema de interesse transversal existente na aplicação.

Problema D: Onde está o problema?

Uma situação interessante relacionada à métricas de SI é que, se por um lado elas indicam um problema de interesse transversal, por outro, elas nem sempre mostram onde está o problema. Ou seja, os números não apontam quais partes do projeto ou implementação devem ser alteradas para resolver o problema. Um exemplo ocorre em nosso terceiro estudo experimental, Portalware [30], quando avaliado o interesse Colaboração. Este estudo é descrito na Subseção 7.1.3 e o resultado das medições é apresentado no Apêndice C. Em uma avaliação do resultado das métricas de SI, o interesse Colaboração é provavelmente classificado como um interesse transversal devido ao seu alto grau de espalhamento e entrelaçamento. Os valores apontados pelas métricas CDC = 15 e CDLOC = 14 confirmam esta intuição. É importante notar que o número de componentes do sistema é 60 (métrica VS no Apêndice C), ou seja, 25 % dos componentes possuem código relativo ao interesse avaliado.

A questão levantada é quantos e quais componentes devem ser alterados para separar este interesse e resolver o problema. Em uma impressão inicial, acreditaríamos ter que alterar os 15 componentes que possuem código do interesse Colaboração. Por outro lado, em uma análise mais aprofundada constatamos que 9 destes 15 componentes são totalmente dedicados à realização do interesse e que em apenas 6 existe entrelaçamento. Neste caso, o número de componentes que o

desenvolvedor precisa verificar a existência de problemas de SI reduz de 15 para 6. Apesar de todas estas informações serem extraídas dos resultados das medições, elas não são facilmente identificadas em uma análise *ad hoc* e só podem ser obtidas de um profundo raciocínio dos números de fina granularidade.

Problema E: Resultados conflitantes entre as métricas

A dura atividade de análise em resultados de medição pode se tornar ainda mais difícil quando os números apontam para caminhos contrários. Torna-se difícil classificar um interesse como transversal quando as métricas indicam espalhamento, mas não entrelaçamento, ou vice-versa. Em nosso estudo do Portalware (Subseção 7.1.3) este problema ocorre quando avaliamos o interesse Adaptação. Os dados deste estudo são detalhados no Apêndice C e podemos verificar uma contradição entre os valores das métricas CDLOC = 14 e CDC = 3. Enquanto a primeira métrica aponta um alto grau de entrelaçamento entre interesses, a segunda indica pouco espalhamento deste interesse. Mesmo olhando para outras métricas, como NOAconcern e NOOconcern, concluir se o interesse Adaptação pode ser considerado transversal neste sistema não é fácil.

Problema F: Métricas não relacionam os problemas

O último problema de medição levantado nesta seção é o fato de ser geralmente difícil relacionar problemas apontados pelas métricas de SI a problemas indicados em outras métricas. Este mapeamento é interessante neste estudo porque (i) estudos recentes mostram que outros atributos de software devem ser observados juntamente com separação de interesse na avaliação da qualidade [8] [21] [27] e (ii) o método de avaliação proposto nesta dissertação se baseia no princípio de que a melhoria de SI é acompanhada por ganhos em outros atributos, como acoplamento e coesão. É ilustrado a seguir como um interesse transversal pode levar à perda de coesão e aumento de acoplamento nos componentes afetados por este interesse.

A Tabela 6 apresenta o resultado de medições de SI para o papel *Subject* do padrão *Observer* [26] em uma implementação OO do padrão. Esta tabela também retrata valores de acoplamento e coesão medidos neste software que é parte do primeiro estudo experimental apresentado na Subseção 7.1.1. É interessante

observar que a maior parte dos componentes que implementam o papel *Subject* (`Point`, `Point1`, etc.) apresenta altos valores para as métricas de perda de coesão (LCOO) e de acoplamento (CBC). Uma análise detalhada da implementação mostra que estas classes apresentam maior perda de coesão porque precisam implementar métodos do padrão que não têm relação com os outros métodos da classe. Os métodos do padrão adicionam às classes funcionalidades para incluir, remover e notificar observadores. Além disso, o aumento na medição de acoplamento para as classes que desempenham o papel de *Subject* é justificado porque este papel é altamente acoplado aos componentes que desempenham papéis de *Observer* no mesmo padrão. Portanto, a conclusão é que o papel *Subject* leva a deterioração de outros atributos de qualidade do sistema, como coesão e acoplamento.

Tabela 6 – Resultado das métricas para o papel *Subject* do padrão *Observer*

Componentes	***Subject* (Padrão *Observer*)**			
	CDC = 7	**CDLOC = 56**	**VS = 12**	
	NOAconcern	**NOOconcern**	**LCOO**	**CBC**
Observer	0	0		1
Point	**1**	**3**	**15**	**4**
Point1	**1**	**3**	**15**	**4**
Point2	**1**	**3**	**15**	**4**
Point3	**1**	**3**	**15**	**4**
Point4	**1**	**3**	**15**	**4**
Screen	1	3	1	3
Screen1	0	0	1	2
Screen2	0	0	1	2
Screen3	0	0	1	2
Screen4	0	0	1	2
Subject	0	3		1

5.2. Regras Heurísticas de Separação de Interesses

As regras apresentadas nesta seção utilizam uma combinação de métricas de separação de interesses (SI) e tamanho para avaliar os interesses do sistema e identificar possíveis problemas relacionados a interesses transversais. As métricas de SI utilizadas são CDC, CDLOC, NOAconcern, NOOconcern e LOCconcern; enquanto as métricas de tamanho são VS, NOA, NOO e LOC. Todas estas métricas são encontradas na Tabela 1 da Subseção 4.2.1. O objetivo principal do conjunto de regras de SI é classificar o interesse avaliado em “Modularizado”,

“Primário” ou “Secundário”. As duas primeiras classificações de interesses não indicam problemas de SI, enquanto a última pode ser problemática e, portanto, indicar que o projeto deva ser refatorado para melhor modularizar este tipo de interesse. Além das três categorias mencionadas, as regras também classificam os interesses em possivelmente primário, entrelaçado, de baixo espalhamento, de elevado espalhamento ou possivelmente secundário.

A Figura 18 retrata o diagrama com as possíveis classificações de um interesse e os relacionamentos de transição entre elas. As classificações são representadas por elipses, enquanto as transições são arcos direcionados ligando duas classificações. Além disso, as transições estão associadas às regras e estas definem quando ocorre uma mudança de categoria do interesse. As elipses do diagrama podem indicar três tipos de estados de classificação: inicial quando tracejado, intermediário quando representado em linha simples e final quando em linha dupla. Como discutimos no decorrer desta seção, um estado intermediário pode se tornar a classificação final do interesse quando nenhuma das transições que saem deste estado for validada pelas regras. O significado de cada uma das categorias presentes na Figura 18 é discutido a seguir.

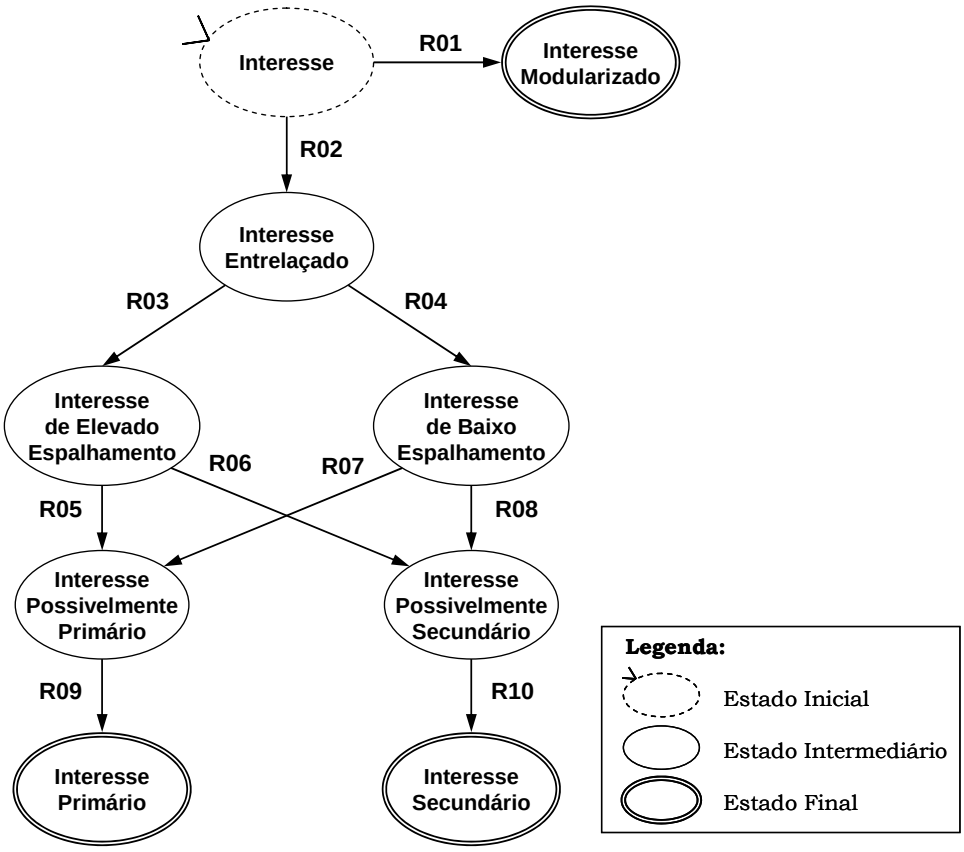


Figura 18 – Diagrama de estados dos interesses e regras de transição

Interesse Modularizado. Um interesse é classificado como *modularizado* quando todos os componentes responsáveis pela sua implementação são totalmente dedicados a este interesse. Ou seja, o interesse não possui entrelaçamento com nenhum outro interesse do sistema. Em um projeto ideal, todos os interesses deveriam estar totalmente separados e, portanto, seriam classificados como interesses modulares.

Interesse (Possivelmente) Primário. Em cada componente do sistema é definido um conjunto de dados e funções que, idealmente, atendem a um mesmo propósito. Este propósito é o que chamamos interesse primário do componente. Um interesse que é primário em um componente pode não ser primário em outros. Desta forma, um interesse somente é classificado como *primário* (no sistema) se ele é primário em todos os componentes em que ele se encontra. As regras classificam o interesse como *possivelmente primário*, quando este é um bom candidato a ser primário, mas não apresenta todas as características para tal.

Interesse Entrelaçado. Um interesse é classificado como *entrelaçado* quando este se encontra misturado a outros interesses dentro de um componente, ou seja, parte das operações e atributos do componente é destinada a um interesse e outra parte é dedicada a outro. O grau de entrelaçamento do interesse é fundamental para diferir sua classificação entre modularizado ou primário, pois, enquanto o “Interesse Modularizado” não se entrelaça a outros interesses em nenhum componente, o “Interesse Primário” pode se entrelaçar.

Interesse (Possivelmente) Secundário. Apesar de cada componente do sistema ser dedicado a um propósito principal, o “Interesse primário”, eles freqüentemente implementam outros interesses. Estes outros interesses são chamados secundários, pois incluem responsabilidades que não são as principais do componente. Mesmo o interesse que é primário em um componente pode ser secundário em outros. Assim, o interesse é classificado como *secundário* (no sistema) se ele é secundário em pelo menos um componente.

Interesse de Elevado Espalhamento e Interesse de Baixo Espalhamento. O espalhamento é a característica do interesse relacionada ao número de componentes afetados por ele. O grau de espalhamento de um interesse é variado e, portanto, são definidas duas classificações: (i) *elevado espalhamento*, quando vários componentes implementam o interesse; e (ii) *baixo espalhamento*, quando apenas alguns componentes são dedicados ao interesse. Quanto maior é o

espalhamento pelo sistema do interesse, maiores são as chances deste interesse ser transversal.

A Tabela 7 apresenta o conjunto de regras de separação de interesses que classifica os interesses de acordo com as categorias apresentadas. As regras que compõem este grupo devem ser aplicadas sequencialmente, na ordem em que elas são enumeradas, de tal forma que a classificação do interesse é gradativamente refinada. É importante ressaltar que as categorias não são exclusivas, ou seja, a nova classificação dada ao interesse por uma regra é cumulativa com a classificação dada pela regra anterior. Por exemplo, se a Regra 02 classifica um determinado interesse como “Entrelaçado”, a Regra 03 como “Elevado Espalhamento” e a Regra 06 como “Possivelmente Secundário”, então este interesse possui as três categorias. Em outras palavras, é um interesse “Entrelaçado, de Elevado Espalhamento e Possivelmente Secundário”. Como mencionado anteriormente, as regras utilizam as métricas de SI e tamanho listadas na Tabela 1 (Subseção 4.2.1).

Tabela 7 – Regras heurísticas de separação de interesses

R01	**Se** CDLOC é 2 **então** INTERESSE MODULARIZADO
R02	**Se** CDLOC é maior que 2 **então** INTERESSE ENTRELAÇADO
R03	**Se** CDC / VS de INTERESSE ENTRELAÇADO é alto **então** INTERESSE DE ELEVADO ESPALHAMENTO
R04	**Se** CDC / VS de INTERESSE ENTRELAÇADO não é alto **então** INTERESSE DE BAIXO ESPALHAMENTO
R05	**Se** (NOAconcern / NOA é alto) **e** (NOOconcern / NOO é alto) para todos os componentes de INTERESSE DE ELEVADO ESPALHAMENTO **então** INTERESSE POSSIVELMENTE PRIMÁRIO
R06	**Se** (NOAconcern / NOA é baixo) **e** (NOOconcern / NOO é baixo) para pelo menos um componente de INTERESSE DE ELEVADO ESPALHAMENTO **então** INTERESSE POSSIVELMENTE SECUNDÁRIO
R07	**Se** (NOAconcern / NOA é alto) **e** (NOOconcern / NOO é alto) para todos os componentes de INTERESSE DE BAIXO ESPALHAMENTO **então** INTERESSE POSSIVELMENTE PRIMÁRIO
R08	**Se** (NOAconcern / NOA é baixo) **e** (NOOconcern / NOO é baixo) para pelo menos um componente de INTERESSE DE BAIXO ESPALHAMENTO **então** INTERESSE POSSIVELMENTE SECUNDÁRIO
R09	**Se** (LOCconcern / LOC é alto) para todos os componentes de POSSÍVEL INTERESSE PRIMÁRIO **então** INTERESSE PRIMÁRIO
R10	**Se** (LOCconcern / LOC é baixo) para pelo menos um componente de POSSÍVEL INTERESSE SECUNDÁRIO **então** INTERESSE SECUNDÁRIO

As duas primeiras regras da Tabela 7 (R01 e R02) utilizam a métrica CDLOC para classificar o interesse em “Modularizado” ou “Entrelaçado”. Se o valor de CDLOC é dois, isto significa que o interesse está localizado em um único bloco sombreado. Mesmo que este interesse esteja em vários componentes, o valor 2 indica que estes componentes não possuem código relativo a nenhum outro interesse do sistema. Conseqüentemente, o interesse está modularizado e não existe entrelaçamento com outros interesses. Por outro lado, se CDLOC for maior que dois, o interesse avaliado se entrelaça a pelo menos um outro interesse. Isto é fácil de ser verificado, pois quando CDLOC é 4, existem dois blocos disjuntos sombreados para o interesse e, entre estes blocos, há um segundo interesse que causa o entrelaçamento.

As regras R03 e R04 são usadas para verificar se o interesse, quando classificado anteriormente como entrelaçado, se espalha por um grande número de componentes ou não. As métricas de CDC e VS provêem informações sobre este espalhamento através da medição do percentual de componentes afetados pelo interesse em relação ao número total de componentes no sistema (CDC/VS). Baseado nesta percentagem, o interesse é classificado em “Interesse de Baixo Espalhamento” ou “Interesse de Elevado Espalhamento”. O desenvolvedor deve estar consciente de interesses que se espalham por muitos componentes, pois qualquer alteração em algum destes interesses pode impactar em grande reestruturação do sistema.

Tanto um “Interesse de Elevado Espalhamento”, quanto um “Interesse de Baixo Espalhamento” pode ser primário ou secundário nos componentes do sistema. As regras R05 e R06 são usadas para decidir se um “Interesse de Elevado Espalhamento” é “Possivelmente Primário” ou “Possivelmente Secundário”, enquanto as regras R07 e R08 fazem análise semelhante para um “Interesse de Baixo Espalhamento”. O raciocínio aplicado nestas quatro regras é que um componente dedica a maior parte dos elementos internos para implementação de seu interesse primário. Desta forma, estas regras utilizam as métricas NOA, NOAconcern, NOO e NOOconcern para verificar a percentagem de atributos (NOAconcern / NOA) e de operações (NOOconcern / NOO) que cada componente dedica ao interesse avaliado. Se a percentagem de atributos e de operações é alta em todos os componentes, o interesse é classificado como “Possivelmente Primário”. Entretanto, se ambas as percentagens são baixas para

pelo menos um componente, o interesse é classificado como “Possivelmente Secundário”.

Completando o grupo de SI, as duas regras seguintes são responsáveis por classificar o interesse em “Primário” ou “Secundário”. O interesse “Possivelmente Primário” pode ser classificado como “Primário” pela regra R09, enquanto o interesse “Possivelmente Secundário” pode ser classificado como “Secundário” pela regra R10. Nestas duas últimas regras, o comportamento do interesse é inspecionado no contexto das linhas de código destinadas à sua implementação. O raciocínio é similar ao desenvolvido para as regras R05 a R08 e parte do princípio que um componente dedica a maior parte de suas linhas de código ao interesse primário. Portanto, a regra R09 classifica um interesse como “Primário” se este é “Possivelmente Primário” e a maior parte das linhas de código é destinada à implementação deste interesse nos componentes em que ele se encontra. Da mesma forma, um interesse é classificado como “Secundário” pela regra R10 se este é “Possivelmente Secundário” e, em pelo menos um componente, a maior parte das linhas de código não é destinada a sua implementação.

Mesmo o interesse classificado pela regra R09 como “Primário” se encontra entrelaçado a outros interesses (note que ele passou pela regra R02). Mas isto não significa que interesses primários sejam problemas de SI no sistema, pois, provavelmente o entrelaçamento é causado pelos outros interesses que são secundários. Um exemplo de interesse primário e entrelaçado pelo sistema é o padrão *Factory Method* [26] explorado na implementação da Seção 5.1 (Figura 16) em uma combinação com o padrão *Observer* [26]. Se naquele sistema todos os interesses forem sombreados e avaliados pelas regras, o padrão *Factory Method* seria classificado como “Interesse Primário” enquanto o *Observer* seria classificado como “Interesse Secundário”. Neste caso, o *Observer* é o interesse que possivelmente mereça ser melhor modularizado.

É importante notar que algumas das métricas utilizadas pelas regras da Tabela 7 só podem ser obtidas em artefatos de código e isto as torna restrita ao nível de implementação. As regras R09 e R10 são dependentes de métricas do nível de implementação e, portanto, estas regras não podem ser aplicadas caso os interesses estejam sendo avaliados no nível de projeto. Este fato justifica a divisão nos dois últimos níveis da avaliação. Se apenas artefatos de projeto detalhado estiverem disponíveis para a avaliação da separação de interesses do sistema, as

regras R01 a R08 podem ser utilizadas e os estados finais passam a ser “Interesse Possivelmente Primário” e “Interesse Possivelmente Secundário”.

5.3. Regras Heurísticas de Acoplamento e Coesão

O principal objetivo das regras apresentadas na seção anterior é identificar “Interesses Secundários” e “Possivelmente Secundários”, pois estes podem representar oportunidades para refatorações OA. Além disso, estas regras de SI apontam os componentes que devem ser alterados para melhor separar tais interesses. Em outras palavras, a aplicação das regras de SI identifica um conjunto de componentes (possivelmente vazio) que possuem um ou mais “Interesses Secundários” ou “Possivelmente Secundários”. Desta forma, após a aplicação das regras de SI, um próximo passo na avaliação é identificar os possíveis efeitos destes interesses em outros atributos dos componentes do software, como acoplamento e coesão.

No decorrer dos estudos experimentais (Capítulo 7) foi verificado que a separação mais adequada dos interesses, em especial interesses secundários, tem como conseqüência a melhoria no acoplamento e coesão dos componentes do sistema. A melhora de coesão dos componentes após a retirada dos interesses secundários se justifica porque estes componentes possuíam responsabilidades não relacionadas. Ou seja, os componentes possuíam um conjunto de operações e atributos que implementava o interesse primário (ou propósito principal do componente) e outros conjuntos, provavelmente disjuntos, de operações e atributos que atendiam aos interesses secundários. Além disso, o acoplamento de um componente diminui após a retirada dos interesses secundários. Isto ocorre porque os componentes que implementam um mesmo interesse precisam se comunicar. Consequentemente, quando um componente é responsável por vários interesses, este também é responsável pelos relacionamentos com componentes destes interesses.

Nesta seção é apresentado um conjunto de regras de acoplamento e coesão que é complementar ao de SI (Seção 5.2). A aplicação deste segundo conjunto de regras constitui um passo seguinte às regras de SI e somente avaliam componentes que possuem algum “Interesse Secundário” ou “Possivelmente Secundário”. Ou seja, após a aplicação das regras de SI em todos os interesses do sistema, se alguns

deles forem classificados como “Secundário”, ou “Possivelmente Secundário”, as regras de acoplamento e coesão são utilizadas nos componentes em que exista tal interesse. Assim, o objetivo destas regras é auxiliar o desenvolvedor na identificação de componentes com problemas de acoplamento e coesão causados por interesses transversais. As regras utilizam as métricas CBC, LCOO e VS apresentadas na Tabela 1 (Subseção 4.2.1). A métrica VS aparece indiretamente nas regras, uma vez que ela é usada para calcular a média de acoplamento e perda de coesão dos componentes.

O conjunto de regras de acoplamento e coesão pode classificar os componentes em dois grupos em seu estado final: “Componente Candidato a Reestruturação Global” e “Componente Candidato a Extração de Interesses”. Além destas duas categorias, as regras de acoplamento e coesão também classificam os componentes nos seguintes estados intermediários: “Componente de Baixa Coesão”, “Componente de Elevada Coesão”, “Componente de Baixo Acoplamento” e “Componente de Elevado Acoplamento”. A Figura 19 apresenta o diagrama de categorias dos componentes e suas transições. Como no diagrama de interesses (Figura 18), neste diagrama as transições de estados também são definidas pelas as regras. A seguir são definidos os seis estados (finais e intermediários) da Figura 19.

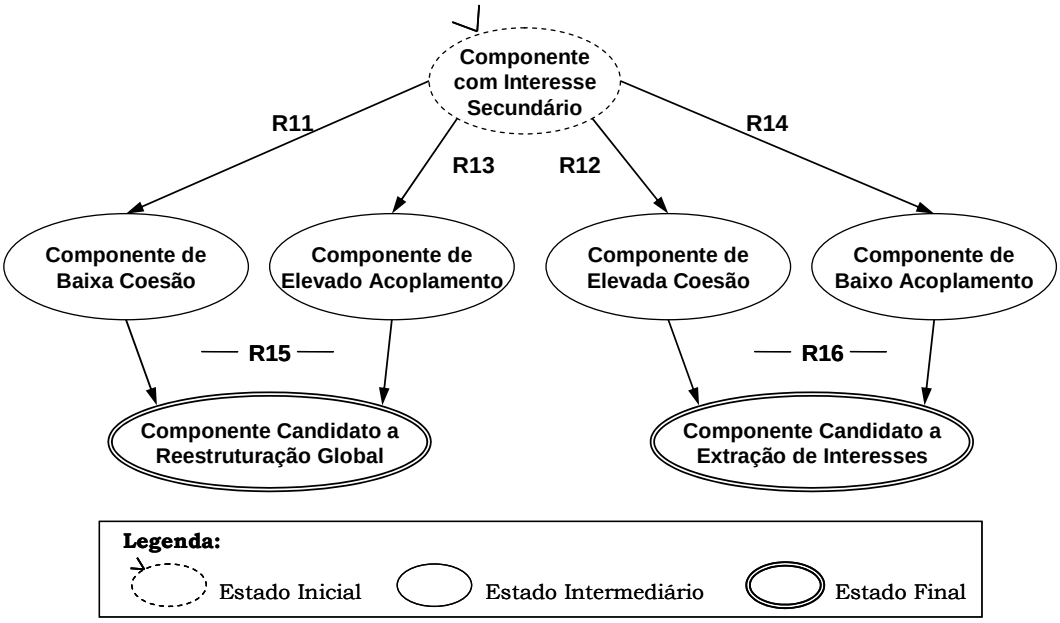


Figura 19 – Diagrama de estados dos componentes e regras de transição

Componente de Elevada Coesão e Componente de Baixa Coesão. A coesão interna de um componente é geralmente avaliada pelos relacionamentos existentes entre operações e atributos [14] [37]. Desta forma, as regras heurísticas propostas utilizam a métrica LCOO [14] [57] para classificar os componentes em *elevada coesão* ou *baixa coesão*. Um "Componente de Elevada Coesão" possui superior valor de coesão quando comparado à média apresentada pelos demais componentes do sistema. Em contrapartida, um "Componente de Baixa Coesão" apresenta valor inferior à média do sistema.

Componente de Elevado Acoplamento e Componente de Baixo Acoplamento. O acoplamento de um componente indica sua dependência em relação a outros componentes do sistema. Para avaliar o grau de acoplamento são verificadas as conexões com outros componentes, como chamadas a atributos e operações de outros componentes. No conjunto de regras proposto, o acoplamento é avaliado pela métrica CBC [14] [57] e os componentes podem ser classificados como *elevado acoplamento* ou *baixo acoplamento*. Um "Componente de Elevado Acoplamento" apresenta grande dependência de outros componentes, enquanto em um "Componente de Baixo Acoplamento" esta dependência é baixa. No contexto das regras apresentadas nesta seção, o acoplamento de um componente pode ser considerado baixo, ou alto, quando comparado com a média de acoplamento dos demais componentes do sistema.

Componente Candidato a Reestruturação Global. Quando um componente é classificado pelas regras como *candidato a reestruturação global* significa que este componente apresenta interesses secundários, baixa coesão e alto acoplamento. Estas características indicam problemas na definição do componente e, portanto, sugerem que este deva ser totalmente reestruturado. O principal objetivo das regras é encontrar oportunidades de aperfeiçoamento do sistema pela melhor separação de interesses, portanto, uma possível ação a ser tomada para resolver os problemas identificados pelas regras é utilizar refatorações OA para modularizar os interesses secundários destes componentes. É importante ressaltar que o "Componente Candidato a Reestruturação Global" representa o principal alerta de problema apontado pelas regras heurísticas deste trabalho.

Componente Candidato a Extração de Interesses. Um componente pode possuir interesses secundários e estes interesses não causarem problemas

aparentes de acoplamento e coesão. Quando isto ocorre, ou seja, quando as regras não detectam problemas nestes atributos dos componentes com interesses secundários, os componentes são classificados como *candidato a extração de interesses*. Esta classificação não descarta a existência de problemas nos módulos, apenas sugere que eles não são tão graves quanto na classificação anterior. Na verdade, o único possível problema verificado nestes componentes é a existência de interesse secundário e foi apontado pelas regras de SI. Quando comparado ao alerta feito pelas regras de SI "Interesse (Possivelmente) Secundário" ao alerta de "Componente Candidato a Extração de Interesses", pode-se dizer que este último é menos grave, pois não foi detectado problema de acoplamento e coesão.

A Tabela 8 apresenta o conjunto de regras de acoplamento e coesão proposto neste trabalho. Estas regras são complementares às de SI (Seção 5.2) e classificam os componentes que possuem interesses secundários em uma das seis categorias mencionadas acima. Assim como ocorre nas regras de SI, a aplicação destas regras deve ser feita sequencialmente, na ordem em que elas são enumeradas. As regras de acoplamento e coesão utilizam as métricas CBC, LCOO e, indiretamente, VS para calcular as médias das duas métricas anteriores no sistema. O grupo é composto de 6 regras, numeradas de R11 a R16, que discutimos a seguir.

Tabela 8 – Regras heurísticas de acoplamento e coesão

R11	**Se** LCOO do COMPONENTE COM INTERESSE SECUNDÁRIO é alto em relação à média de LCOO dos componentes do sistema **então** COMPONENTE DE BAIXA COESÃO
R12	**Se** LCOO do COMPONENTE COM INTERESSE SECUNDÁRIO é baixo em relação à média de LCOO dos componentes do sistema **então** COMPONENTE DE ELEVADA COESÃO
R13	**Se** CBC do COMPONENTE COM INTERESSE SECUNDÁRIO é alto em relação à média de CBC dos componentes do sistema **então** COMPONENTE DE ELEVADO ACOPLAMENTO
R14	**Se** CBC do COMPONENTE COM INTERESSE SECUNDÁRIO é baixo em relação à média de CBC dos componentes do sistema **então** COMPONENTE DE BAIXO ACOPLAMENTO
R15	**Se** (COMPONENTE DE BAIXA COESÃO) **e** (COMPONENTE DE ELEVADO ACOPLAMENTO) **então** COMPONENTE CANDIDATO A REESTRUTURAÇÃO GLOBAL
R16	**Se** (COMPONENTE DE ELEVADA COESÃO) **e** (COMPONENTE DE BAIXO ACOPLAMENTO) **então** COMPONENTE CANDIDATO A EXTRAÇÃO DE INTERESSES

As duas primeiras regras da Tabela 8, R11 e R12, avaliam a coesão dos componentes que possuem interesses secundários. Estas regras utilizam as

métricas LCOO e VS para classificar os componentes em “Baixa Coesão” ou “Elevada Coesão”. A métrica VS aparece implicitamente na regra para o cálculo de média de perda de coesão do sistema, ou seja, o somatório da perda de coesão de todos os componentes dividido por VS. Um componente é classificado como “Componente de Baixa Coesão” quando seu valor de LCOO é maior que a média do sistema, enquanto um “Componente de Elevada Coesão” possui valor abaixo da média. Note que a métrica LCOO mede a perda de coesão, portanto, quanto maior o valor desta métrica, menor é a coesão do componente.

As regras R13 e R14 fazem uma avaliação semelhante às regras anteriores, porém, considerando o critério de acoplamento. Os componentes que possuem algum interesse secundário podem ser classificados em “Baixo Acoplamento” ou “Elevado Acoplamento” por estas regras. Na primeira categoria, estão os componentes que possuem valor de CBC abaixo da média do sistema e na segunda categoria os componentes com valor acima da média. Tanto as regras R11 e R12 quanto as regras R13 e R14 criam dois grupos complementares de componentes. A diferença está no critério utilizado para alocar os componentes nos grupos, coesão para o primeiro par de regras e acoplamento para o segundo par.

As últimas duas regras da Tabela 8, R15 e R16, não utilizam nenhum resultado de medição diretamente. O principal propósito destas regras é unificar estados nos quais os componentes apresentam problemas de acoplamento e coesão (R15) e estados nos quais os componentes não apresentam problemas nestes atributos (R16). Quando a aplicação das regras anteriores detecta que um componente é pouco coeso (R11) e muito acoplado (R13), este componente é classificado como “Candidato a Reestruturação Global” pela regra R15. Em contrapartida, o componente muito coeso (R12) e pouco acoplado (R14) é classificado como “Candidato a Extração de Interesses” pela regra R16. É importante ressaltar que em ambos os casos, os componentes possuem interesses secundários identificados durante a aplicação das regras de SI.

5.4. Definição de Valores Limite

Nesta seção discutimos um ponto importante em relação às regras heurísticas apresentadas no decorrer do capítulo, os valores limites. Tanto regras

de separação de interesses quanto regras de acoplamento e coesão dependem da definição de valores limites que, quando ultrapassados, disparam as conseqüências destas regras. Os valores limite são parâmetros importantes para a eficácia das regras e, portanto, devem ser muito bem definidos. Quando a conseqüência da regra depende de um valor abaixo do limite e este limite possui um valor acima do ideal, o resultado é muitos falsos positivos. Por outro lado, se o limite configurado na regra está abaixo do ideal, o resultado são muitos falsos negativos. O problema de encontrar o valor limite ideal não é novo e intrinsecamente caracteriza qualquer abordagem baseada em medição [40] [47].

Duas estratégias podem ser usadas na definição de valores limites: boas sugestões encontradas na literatura e configuração empírica. No primeiro caso, a escolha do valor limite é guiada por experiências semelhantes do passado e acertos apontados por autores de trabalhos relacionados a medições [47]. Quando estes acertos não estão disponíveis, os valores podem ser configurados de acordo com as características do sistema (e.g. tamanho) e avaliados em estudos empíricos. Por exemplo, se um determinado valor que foi definido gerar muitos falsos positivos ou falsos negativos do que resultados corretos, este valor pode ser alterado e re-avaliado. Na verdade, quando uma ferramenta de suporte às regras se encontra disponível, o ajuste do valor limite para os requisitos específicos de um projeto ou de um desenvolvedor é facilitado. Com uma ferramenta o desenvolvedor pode definir um valor e verificar imediatamente os efeitos desta ação (alertas excessivos ou escassos); variando continuamente este valor até restringir os alertas a um número que pode ser verificado em tempo hábil e verificar se os alertas gerados estão sendo efetivos.

Tabela 9 – Valores limite utilizados nas regras heurísticas

Regras	**Valores Limites**
Regra 01 e 02	2
Regras 03 e 04	50 % (se VS <= 10) 40 % (se 10 < VS <= 20) 30 % (se 20 < VS <= 50) 20 % (se 50 < VS <= 80) 15 % (se 80 < VS)
Regras 05, 06, 07 e 08	50 % (de NOA) 50 % (de NOO)
Regras 09 e 10	50 % (de LOC)
Regras 11 e 12	20 % (da média de LCOO)
Regras 13 e 14	20 % (da média de CBC)
Regras 15 e 16	-

Na abordagem desta dissertação são utilizadas ambas as estratégias para definição de valores limites, sugestões encontradas na literatura e configuração empírica. O ponto de partida são acertos de trabalhos anteriores e, então, os valores são customizados empiricamente para as características individuais de cada projeto. A Tabela 9 apresenta os valores limites para as dezesseis regras propostas neste capítulo e que foram utilizados nos estudos de caso realizados. As regras 15 e 16 são as únicas que não necessitam de valores limites, pois estas regras apenas unificam dois estados de classificação dos componentes (seção 5.3).

Como pode ser notado pela Tabela 9, o valor limite das regras 3 e 4 varia entre 15 % e 50 % dependendo do tamanho do projeto. Ou seja, o valor limite é de 15 % se o sistema avaliado possui mais de 80 componentes, 50 % se possui menos de 10 componentes e outro valor intermediário quando o número de componentes é maior que 10 e menor que 80. Esta variação de valor limite é feita para tentar acompanhar a intuição humana em relação ao alto espalhamento ou baixo espalhamento de um interesse. Por exemplo, em um sistema grande de 100 componentes quando um interesse atinge mais de 15 componentes deste sistema (portanto, mais de 15 %), este interesse será provavelmente classificado como de elevado espalhamento por um usuário humano (e também pelas regras). Por outro lado, em um sistema de 10 componentes é necessário que pelo menos 5 deles (50 %) possuam um determinado interesse para que o interesse seja considerado de elevado espalhamento pela maioria dos programadores.

6
A Ferramenta de Medição e Avaliação

Neste capítulo é apresentada a ferramenta AJATO, acrônimo para *Ferramenta de Avaliação AspectJ*[11]. Esta ferramenta permite medir e avaliar sistemas implementados em Java [17] e AspectJ [45]. O principal objetivo da ferramenta AJATO é automatizar as duas primeiras atividades – medição e aplicação de regras – do método de avaliação proposto no Capítulo 4 deste documento. Para a atividade de medição, a ferramenta disponibiliza um conjunto de métricas orientadas a aspectos (OA), especialmente as métricas listadas na Tabela 1 (Subseção 4.2.1). A atividade de avaliação do software é feita após a medição, baseada nos resultados das métricas e utilizando regras heurísticas. As regras heurísticas suportadas pela ferramenta aparecem na Tabela 7 e Tabela 8 do capítulo anterior. É importante ressaltar que a avaliação proposta neste trabalho consiste em sugestões para melhorar a separação de interesses do sistema, e consequentemente, alguns atributos do software como acoplamento e coesão. Além das métricas e regras disponíveis, a ferramenta oferece mecanismos de extensão o que torna possível adicionar novos elementos.

A ferramenta AJATO pode ser usada apenas na fase de implementação, pois uma decisão deste trabalho foi utilizar como artefato de entrada o código fonte dos sistemas. As medições são efetuadas sobre este código e a aplicação das regras sobre o resultado das medições. Esta decisão se justifica pelo fato do código fonte ser a fonte mais confiável de informação do software, especialmente se estas informações são relativas a métricas de produto como é o foco deste trabalho. Apesar disso, extensões da ferramenta são planejadas para suportar avaliação de diagramas UML [5] no nível de projeto detalhado. A motivação para esta possível extensão é antecipar a identificação de problemas no processo de desenvolvimento de tal forma a solucioná-lo antes da fase de implementação. A

[11] O acrônimo é derivado do nome em inglês, *AspectJ Assessment Tool*.

ferramenta proposta é descrita neste capítulo em função das decisões arquiteturais (Seção 6.1) e detalhes de projeto e implementação (Seção 6.2).

6.1. Modelo Arquitetural

Esta seção descreve em mais alto nível os elementos que compõem a ferramenta de medição e avaliação AJATO. Esta ferramenta é composta por quatro módulos principais: Extrator de Modelo AspectJ, Gerenciador de Interesses, Coletor de Métricas e Analisador de Regras. Os módulos e suas dependências são ilustrados no diagrama da Figura 20. Além dos módulos da ferramenta, esta figura apresenta ainda os recursos de entrada e saída de cada módulo.

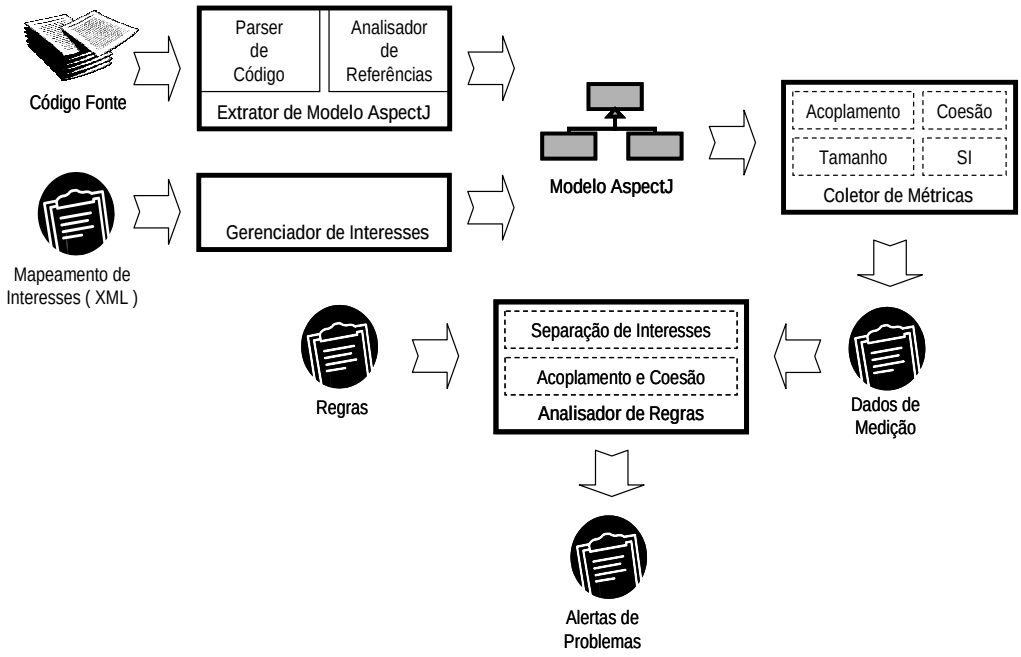


Figura 20 – Representação arquitetural da ferramenta AJATO

A entrada do módulo Extrator de Modelo AspectJ é o código fonte do sistema a ser avaliado e a saída é uma estrutura de dados representativa do sistema que este módulo constrói parcialmente, chamada Modelo AspectJ. O Modelo AspectJ se completa após o módulo Gerenciador de Interesses fazer o mapeamento dos elementos do modelo para os interesses do sistema. A entrada do Gerenciador de Interesses é uma descrição em formato XML [48] [49] deste mapeamento. O Modelo AspectJ, artefato de saída dos dois primeiros módulos, é

a entrada para o módulo Coletor de Métricas. Este terceiro módulo gera os dados das medições que são utilizados pelo Analisador de Regras e este, finalmente, produz um relatório com os alertas de possíveis problemas. Cada um dos quatro módulos da ferramenta é brevemente discutido a seguir em função de seu propósito principal e características.

Extrator de Modelo AspectJ

O módulo Extrator de Modelo AspectJ efetua o *parser* do código fonte de entrada e constrói um modelo representativo do sistema, o Modelo AspectJ (Figura 20). Este módulo detecta a estrutura dos arquivos de entrada em termos dos elementos sintáticos de Java e AspectJ, tais como componentes, atributos, conjuntos de junção, operações e comandos. O Extrator de Modelo é composto de dois submódulos: (i) *Parser* de Código e (ii) Analisador de Referências. O primeiro submódulo utiliza o ambiente de meta-programação MetaJ [54] para auxiliar no processo de extrair as informações do código. Este ambiente permite a manipulação de estruturas sintáticas da árvore de sintaxe abstrata do programa. O segundo submódulo, Analisador de Referências, é responsável por capturar os relacionamentos entre os elementos sintáticos. Exemplos de relacionamentos são importações, herança, associações e chamadas de métodos.

Gerenciador de Interesses

O módulo Gerenciador de Interesses faz o mapeamento de estruturas sintáticas do Modelo AspectJ (construído pelo módulo Extrator de Modelo AspectJ) para os interesses a serem avaliados. Este mapeamento é necessário para que as métricas de separação de interesses (SI) possam ser aplicadas. Desta forma, este módulo recebe como entrada um arquivo XML e então atualiza o modelo com estas informações. O arquivo XML descreve os elementos sintáticos do software e à quais interesses estes elementos pertencem. A atividade exercida pelo módulo Gerenciador de Interesses representa o sombreamento de código (Subseção 3.2.4) e deve preceder as medições de SI.

Coletor de Métricas

O módulo Coletor de Métricas é responsável por efetuar medições no modelo representativo do sistema (Modelo AspectJ). As métricas definidas neste módulo se classificam em quatro categorias principais: acoplamento, coesão, tamanho e separação de interesses. Para computar as métricas das três primeiras categorias não é necessário que tenha sido feito o mapeamento de interesse pelo módulo Gerenciador de Interesses, entretanto, as métricas de SI só podem ser obtidas após este mapeamento. O resultado da aplicação das métricas pode ser gravado em um arquivo de saída, representado na Figura 20 como Dados de Medição.

Analisador de Regras

O último módulo da ferramenta é o Analisador de Regras que tem como propósito principal a aplicação de regras heurísticas como as definidas no Capítulo 5. Estas regras são baseadas em resultados de medições e elaboradas de tal forma a gerarem alertas específicos de possíveis problemas relativos à separação de interesses do sistema. O módulo Analisador de Regras utiliza dois artefatos de entrada: (i) o resultado das métricas coletadas pelo módulo Coletor de Métricas e (ii) um arquivo que define as regras e a ordem em que estas devem ser aplicadas. O resultado da aplicação das regras feita por este módulo é o arquivo de Alertas de Problemas (Figura 20). Este artefato de saída lista os alertas, as regras associadas a eles e os valores resultantes da atividade de medição.

6.2. Projeto e Implementação

O projeto da ferramenta AJATO é orientado a aspectos com implementação em AspectJ. Em sua primeira versão, dois aspectos foram utilizados para separar os interesses de auditoria e persistência. Estes interesses foram implementados em aspectos por serem conhecidamente transversais [44] [45] [66]. A Figura 21 ilustra a tela principal da ferramenta em que podem ser notadas três regiões, divididas em três colunas, para apresentação de informações relativas ao sistema avaliado. As informações da coluna à esquerda da janela encontram-se organizadas em forma de árvore, hierarquizando as estruturas sintáticas do

sistema. A coluna ao centro apresenta informações relativas ao vértice selecionado na árvore e a coluna à direita da janela informa ao usuário da ferramenta sobre a avaliação do sistema.

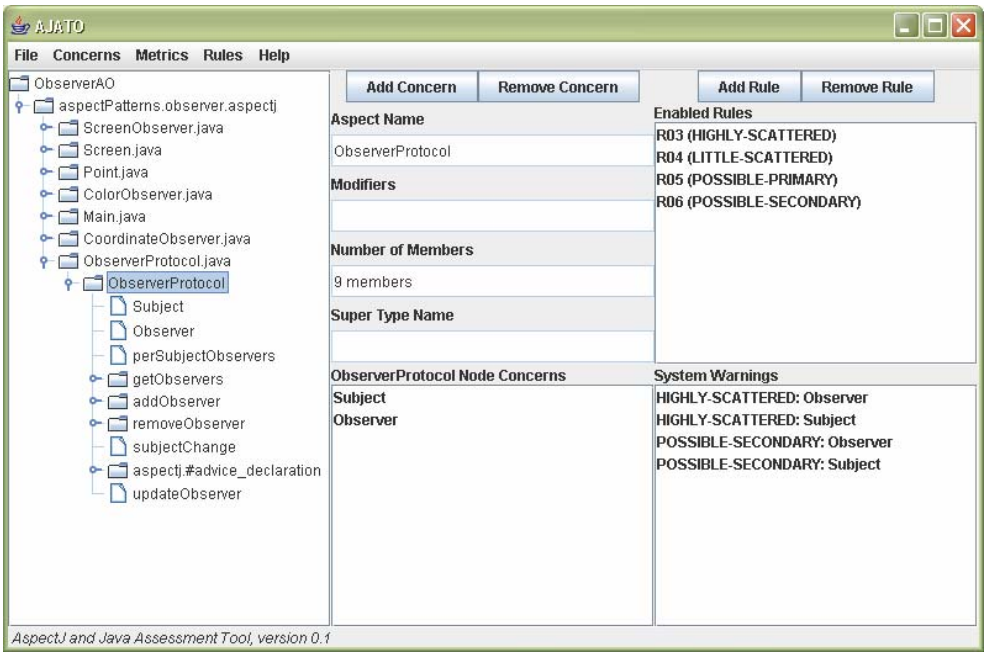


Figura 21 – Interface da ferramenta de avaliação

A hierarquia de estruturas sintáticas, apresentada na coluna esquerda da janela, é organizada da seguinte forma. O sistema como um todo é representado na raiz da árvore, no segundo nível aparecem os pacotes do sistema e no terceiro os arquivos Java e AspectJ. No exemplo da Figura 21 o sistema é chamado “ObserverAO”, o único pacote do sistema é chamado “aspectPatterns.observer.aspectj” e existem ainda sete arquivos dentro deste pacote. Internamente aos arquivos encontram-se os componentes que podem ser classes, interfaces ou aspectos. Cada componente pode conter ainda atributos, operações, conjuntos de junção, declarações intertipo ou outros componentes aninhados. As operações de um componente podem possuir comandos. Na Figura 21 o arquivo “ObserverProtocol.java” encontra-se expandido para mostrar as estruturas internas.

As informações apresentadas ao centro da janela da ferramenta são dependentes da estrutura selecionada na árvore à esquerda (Figura 21). Em especial, os interesses que esta estrutura contribui para a implementação (centro inferior da janela). Como o vértice selecionado é o aspecto “ObserverProtocol”, as informações apresentadas ao usuário na região ao centro são: o nome do aspecto,

modificadores, número de membros internos, super-aspecto (ou superclasse) e os interesses implementados por este componente. O usuário pode adicionar ou remover um interesse ao elemento selecionado na árvore da esquerda utilizando os botões “Add Concern” e “Remove Concern” no centro superior da janela.

Na coluna direita da janela (Figura 21) são apresentadas informações das regras e do resultado da avaliação do sistema. As regras ativas são mostradas na parte superior e o usuário pode ativar ou desativar regras através dos botões “Add Rule” e “Remove Rule”, respectivamente. Quatro regras se encontram ativas na janela da Figura 21: Regra 03, Regra 04, Regra 05 e Regra 06. Estas regras são descritas na Tabela 7 do Capítulo 5. Os alertas levantados pelas regras ativas são listados na região inferior direita da janela, em nosso exemplo são levantados alertas de elevado espalhamento e possivelmente secundário para os interesses *Observer* e *Subject*.

Além das três colunas que fornecem informações sobre o sistema sendo avaliado, a ferramenta disponibiliza um menu de opções na parte superior da janela. As principais opções deste menu são para carregar um sistema, salvar o mapeamento de interesses, configurar as preferências do usuário e disparar a avaliação heurística. Podem ser definidas preferências para métricas e regras e estas preferências incluem opções de visualização dos resultados e definição de valores limites. O menu de ajuda (*Help*, na Figura 21) oferece instrução para utilização das funcionalidades da ferramenta. Opções de ajuda também estão disponíveis em outras janelas da ferramenta como mostrado na Figura 22. A Figura 22 apresenta a janela da ferramenta para carregar um sistema (esquerda) e a ajuda disponível ao usuário para efetuar esta operação (direita). Esta ajuda é acessível pelo botão “Help” da janela apresentada à esquerda da figura.

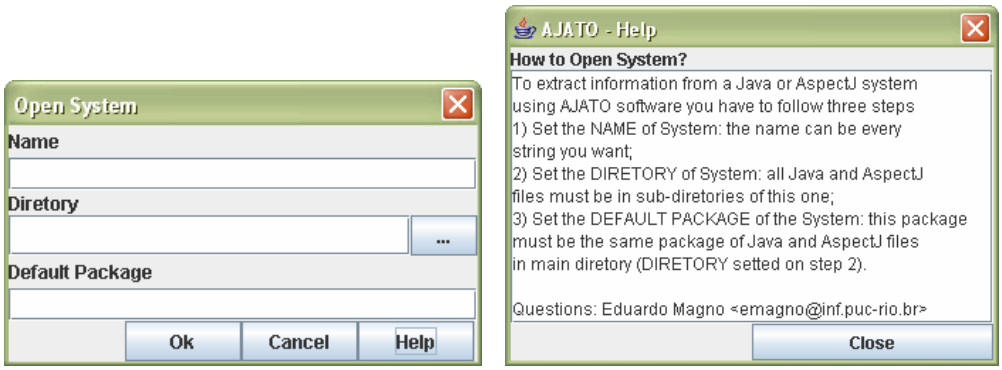


Figura 22 – Janelas para carregar um sistema e respectiva ajuda

A estrutura de diretórios da ferramenta AJATO é apresentada na Figura 23. Esta figura mostra ainda quais diretórios são responsáveis por implementar os quatro módulos (Analisador de Regras, Coletor de Métricas, Gerenciador de Interesses e Extrator de Modelo) e a estrutura de dados Modelo AspectJ da ferramenta. Os arquivos responsáveis pela construção da interface com o usuário da ferramenta estão localizados no diretório “gui”. Além disso, a ferramenta utiliza um ambiente de metaprogramação chamado MetaJ [54] para auxiliar o *parser* do código e o caminhamento na árvore de sintaxe abstrata, como discutido na Subseção 6.2.2. Este ambiente é composto por uma biblioteca (metaenv.jar), um arquivo de configuração (plugins.spec) e um arquivo que permite a compilação de *templates* (TemplateCompiler.bat). Estes três arquivos e o diretório de *templates* MetaJ utilizados pela ferramenta também podem ser vistos na Figura 23.

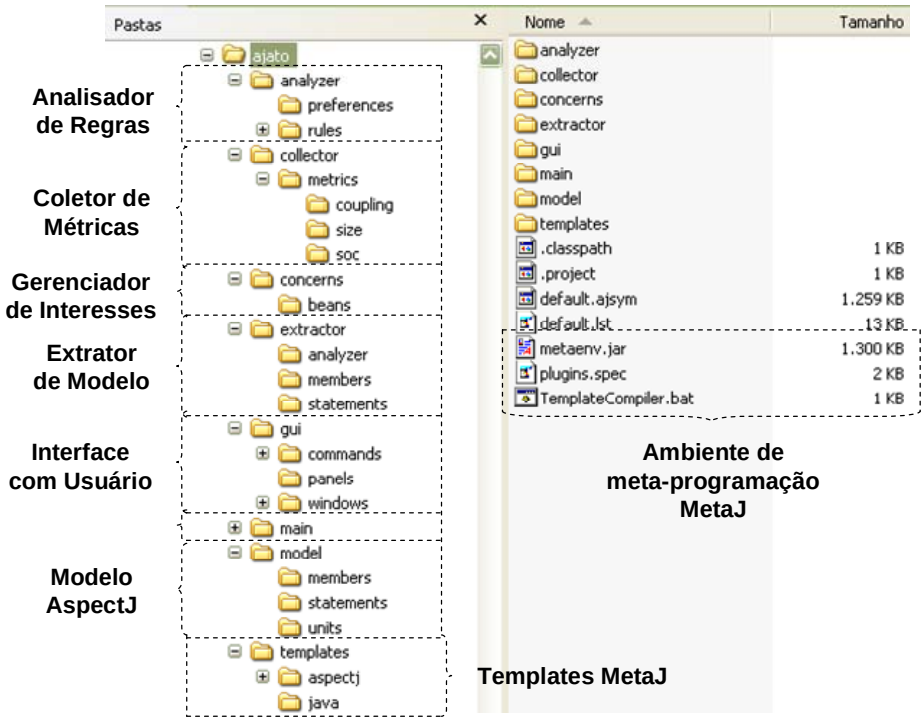


Figura 23 – Estrutura de diretórios da ferramenta AJATO

6.2.1. Modelo de Programas AspectJ

Com o objetivo de facilitar o processo de medição, é construído uma estrutura de dados representativa de programas AspectJ. Nesta estrutura de dados são utilizadas decomposições sucessivas para construção do modelo.

Decomposição [62] é uma técnica adotada para vencer barreiras de complexidade. Esta técnica é freqüentemente utilizada para representar modelos de programas sobre a forma de estrutura de decomposição. Uma estrutura de decomposição é organizada em forma de grafo dirigido no qual cada patamar corresponde a um nível de abstração. Esta estrutura registra as decomposições desde a idéia original até o detalhe mais elementar de sua implementação. Um conceito importante em estruturas de decomposição é a abstração raiz [62]. Abstração raiz é um elemento complexo demais para se dar uma solução direta, sendo dividido em elementos mais simples. Todos os vértices que podem ser decompostos na Figura 24 são considerados abstrações raiz. Quando utilizado recursão em uma decomposição, o problema da abstração raiz pode ser particionado em uma parte a ser resolvida diretamente e em uma parte restante a ser resolvida por recorrência [62]. A Figura 24 apresenta decomposição recursiva nos vértice “Componentes” e “Bloco de Comandos”.

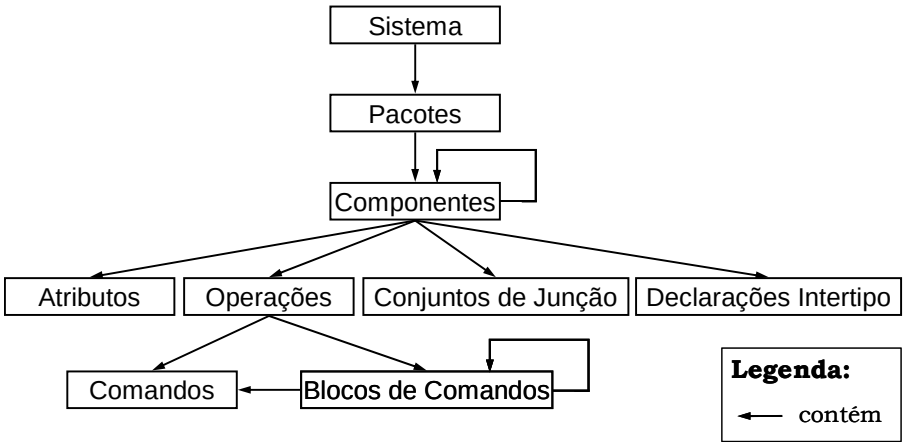


Figura 24 – Modelo de decomposição de sistemas AspectJ

Na Figura 24 é apresentada uma estrutura de decomposição para representar a organização adotada em sistemas implementados na linguagem AspectJ [66]. No nível mais abstrato desta estrutura aparece o sistema como um todo. Em um nível abaixo estão localizados os elementos que representam pacotes existentes no sistema. Assim como em Java, um sistema implementado na linguagem AspectJ deve possuir pelo menos um pacote. Os pacotes podem ser decompostos em componentes (classes, interfaces ou aspectos). Cada componente pode conter outros componentes internos (decomposição recursiva), além de atributos, operações, conjuntos de junção e declarações intertipos. As operações são

decompostas em blocos de comandos ou em comandos. E a decomposição de blocos de comandos pode ser recursiva ou em comandos.

A estrutura de decomposição apresentada na Figura 24 é modelada pelo diagrama de classes da Figura 25. Este diagrama compõe o Modelo AspectJ da ferramenta de medição e avaliação. As classes do diagrama correspondem aos vértices da estrutura de decomposição, exceto por algumas classes e interfaces adicionais que auxiliam na modelagem. Para efeito de simplificação as operações e os atributos foram ocultados no diagrama da Figura 25, bem como os componentes menos relevantes.

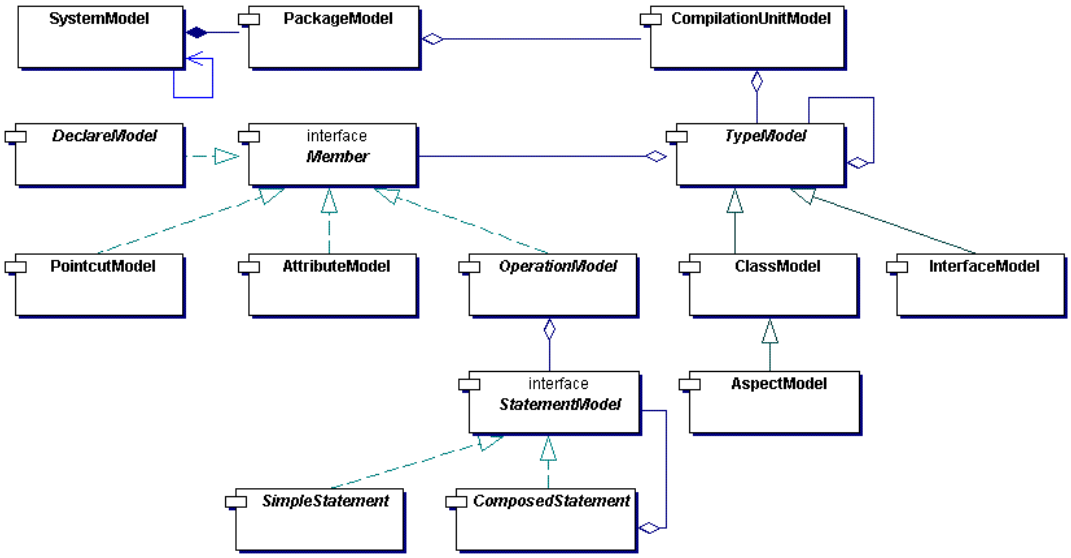


Figura 25 – Diagrama de classes parcial do Modelo AspectJ.

Na Figura 25, a raiz da decomposição é representada pela classe `SystemModel`. Esta classe implementa o padrão *Singleton* [26] para permitir seu acesso a partir de qualquer ponto da aplicação. As decomposições recursivas nos vértices "Componentes" e "Bloco de Comandos" da estrutura de decomposição (Figura 24) são modeladas, respectivamente, pelas classes `TypeModel` e `StatementModel`. Esta última implementa o padrão *Composite* [26] juntamente com as classes `SimpleStatement` e `ComposedStatement` de tal forma a permitir recursão. Outro detalhe interessante do diagrama de classes do Modelo AspectJ é que a classe `AspectModel` é subclasse de `ClassModel`. Note que a primeira classe representa um aspecto e a segunda uma classe do sistema alvo de avaliação. A decisão de estender `ClassModel` para `AspectModel` é coerente porque um aspecto pode possuir todos os elementos de uma classe.

6.2.2.
Módulo de Extração do Modelo

A ferramenta de medição e avaliação AJATO constrói o Modelo AspectJ (Subseção 6.2.1) a partir de informações obtidas do código fonte. Para auxiliar o processo de extrair as informações do código é utilizado MetaJ [54] [55]. MetaJ é um ambiente para meta-programação construído como uma extensão de Java e permite manipular código de programas descritos em qualquer linguagem de programação. Entretanto, na literatura são encontrados experimentos utilizando MetaJ apenas em programas Java. Felizmente, os autores de MetaJ [54] indicam direções para permitir a manipulação de programas escritos em outras linguagens utilizando a gramática da nova linguagem a ser manipulada. Neste trabalho de mestrado é utilizada a gramática de AspectJ para estender MetaJ para esta linguagem. A gramática AspectJ utilizada neste estudo pode ser obtida em [1].

O ambiente MetaJ oferece suporte à análise, transformação e geração de código. Neste trabalho a funcionalidade de análise é utilizada para percorrer o sistema em avaliação de tal forma a construir o Modelo AspectJ. Dois conceitos são importantes para utilização de MetaJ: *templates* e *p-reference*. Os *templates* são abstrações que encapsulam padrões de programas utilizando-se da sintaxe concreta da linguagem objeto (*by example*) [55]. *P-reference* (ou *program-reference*) é um elemento interno aos *templates* que armazena trechos dos programas manipulados. Este elemento esconde a representação interna do código armazenado e oferece apenas operações que garantem a consistência sintática do código. Informações completas sobre a utilização de MetaJ e seus elementos encontram-se disponíveis em [54] [55].

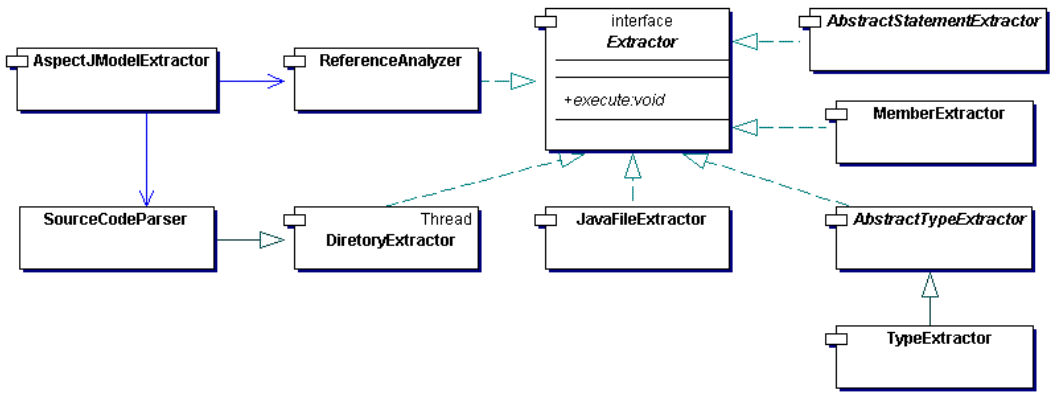


Figura 26 – Diagrama de classes parcial do módulo Extrator de Modelo AspectJ

Além de utilizar MetaJ, o módulo Extrator de Modelo AspectJ é implementado seguindo o padrão de projeto *Interpreter* [26]. A Figura 26 apresenta o diagrama de classes parcial deste módulo. A interface `Extractor` do diagrama desempenha o papel de *AbstractExpression* [26] sendo, portanto, implementada por todas as classes participantes do padrão. Esta interface disponibiliza o método `execute()`, responsável pela interpretação, que deve ser implementado em suas subclasses concretas. As classes `AspectJModelExtractor`, `SourceCodeParser` e `ReferenceAnalyzer` são clientes do padrão *Interpreter* [26]. Elas são responsáveis por iniciar o processo de extração de informação e por criar o Modelo AspectJ. A classe `SourceCodeParser` utiliza MetaJ para efetuar o *parser* do código e a classe `ReferenceAnalyzer` atualiza as referências cruzadas entre as estruturas sintáticas. As demais classes atualizam o Modelo AspectJ com informações obtidas da árvore de sintaxe abstrata do sistema.

6.2.3. Módulo de Gerenciamento de Interesses

O módulo Gerenciador de Interesses da ferramenta AJATO é composto por três componentes principais: as classes `ConcernManager` e `SystemConcerns` e o aspecto `ConcernPersistence`. Estes componentes e seus relacionamentos são apresentados no diagrama de classes da Figura 27.

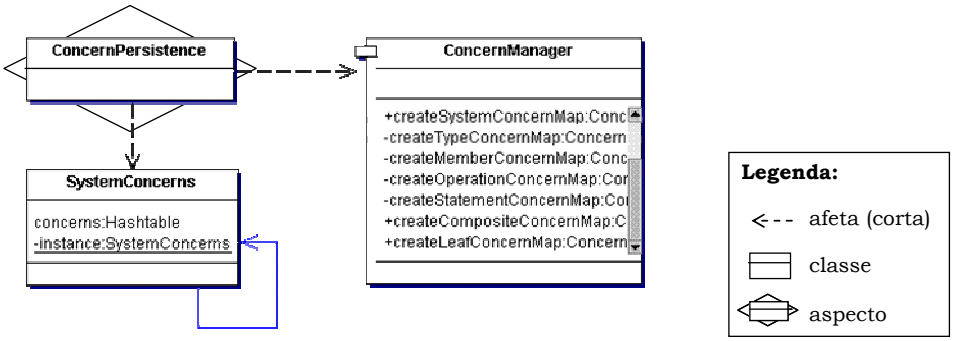


Figura 27 – Diagrama de classes parcial do módulo Gerenciador de Interesses

A classe `ConcernManager` é responsável pelo mapeamento entre as estruturas sintáticas e os interesses do sistema. As estruturas que podem ser mapeadas são todas aquelas que constituem o Modelo AspectJ, ou seja, sistema, pacotes, arquivos, classes, operações, etc. A classe `SystemConcerns` armazena em

uma *hashtable* todos os interesses do sistema e as estruturas que pertencem a cada interesse. Esta classe implementa o padrão *Singleton* [26] de tal forma a ser acessível de qualquer ponto da ferramenta. O terceiro componente deste módulo é o aspecto `ConcernPersistence` que é responsável pelo armazenamento em disco e recuperação do mapeamento de interesses. A decisão de implementar a persistência da ferramenta utilizando aspecto foi tomada porque este interesse (persistência) é tradicionalmente conhecido como transversal [42] [44].

O aspecto `ConcernPersistence` grava e recupera informações de estruturas sintáticas e interesses em um arquivo no formato XML. Este arquivo permite que outras ferramentas interajam com AJATO de tal forma a oferecer extensibilidade para a ferramenta. Para auxiliar na tarefa de gravar e recuperar arquivos XML, o aspecto `ConcernPersistence` utiliza classes do pacote "java.beans" da API de Java [17]. A classe `java.beans.XMLEncoder` é responsável pela conversão de uma estrutura de dados que armazena as estruturas sintáticas e seus respectivos interesses em um arquivo XML. Em contrapartida, a classe `java.beans.XMLDecoder` permite a recuperação das informações do arquivo XML.

6.2.4. Módulo de Medição

As métricas do módulo Coletor de Métricas são implementadas utilizando o padrão *Visitor* [26]. Nesta instância do padrão cada métrica é uma operação executada sobre a estrutura do Modelo AspectJ. A Figura 28 apresenta o diagrama de classes parcial deste módulo da ferramenta. Neste diagrama se pode verificar que a classe `SystemVisitor` é superclasse direta ou indireta de todas as outras classes. Esta classe desempenha o papel de *Visitor* no padrão e disponibiliza os métodos `visit()` para visitar todos os elementos do Modelo AspectJ. As classes concretas (i.e. as métricas) devem re-implementar estes métodos para obter a informação específica que se deseja medir. É importante notar que os componentes que desempenham o papel *Element* [26] do padrão *Visitor* são classes do Modelo AspectJ.

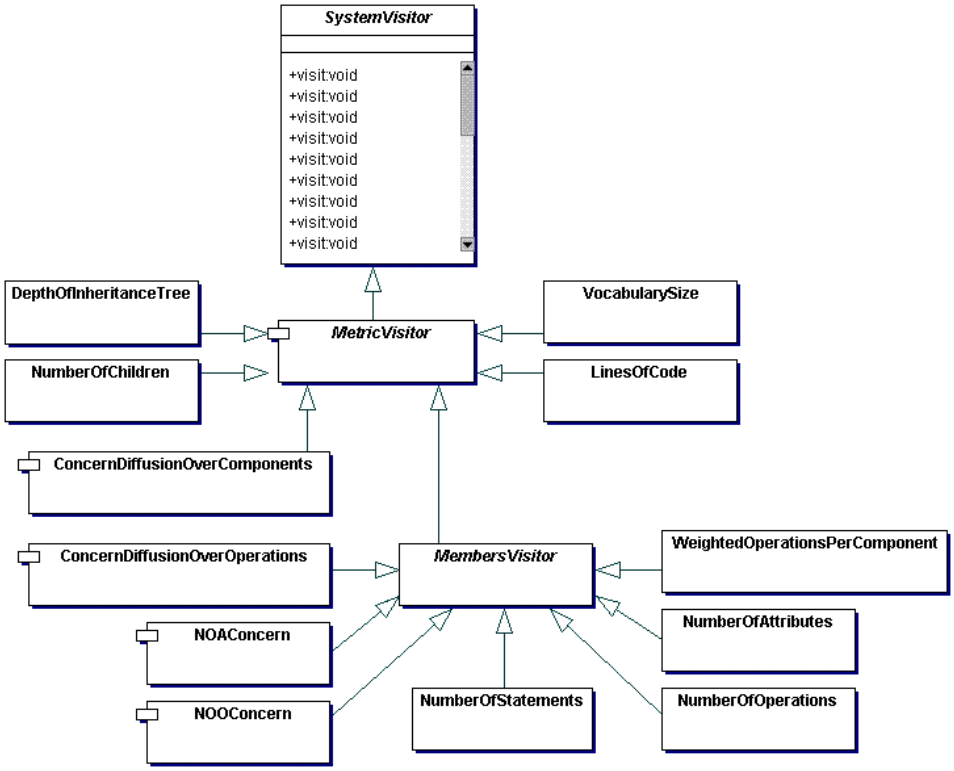


Figura 28 – Diagrama de classes parcial do módulo Coletor de Métricas

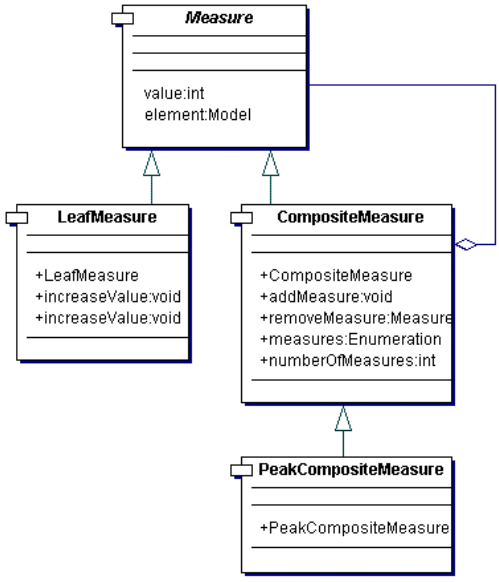


Figura 29 – Diagrama de classes da estrutura de medição

Além das classes que implementam as métricas da ferramenta, este módulo possui uma estrutura de dados para registrar o resultado da medição. Este resultado é armazenado em uma tupla com o elemento do modelo e o valor inteiro medido para este elemento. A estrutura de dados com o resultado da medição implementa o padrão de projeto *Composite* [26] e é ilustrada na Figura 29. A

classe `Measure` desempenha o papel de *Component*, a classe `LeafMeasure` desempenha o papel de *Leaf* e a classe `CompositeMeasure` o papel de *Composite* no padrão. A classe `PeakCompositeMeasure` estende `CompositeMeasure`. A diferença entre elas é que enquanto a primeira retorna o maior valor medido para os elementos da composição, a segunda retorna a soma destes valores.

O módulo Coletor de Métricas disponibiliza uma fachada, padrão *Façade* [26], para localizar em uma única classe o acesso às medições. A classe `MeasurementFacade` implementada a fachada do módulo. Esta classe possui dependência de com a classe `MetricVisitor` (Figura 28) e com todas as classes concretas que implementam métricas. Quando é solicitada qualquer medição através da fachada, o elemento da classe `Measure` retornado corresponde ao resultado da medição.

6.2.5. Módulo de Avaliação Heurística

A classe principal do módulo Analisador de Regras, `RuleAnalyzer`, inicia o processo de avaliação heurística. Esta classe implementa o padrão *Singleton* [26] como apresentado na Figura 30. Além disso, a classe `RuleAnalyzer` possui referência para outras duas classes do módulo: `SoCRuleAnalyzer` e `HeuristicRules`. A classe `SoCRuleAnalyzer` possui um vetor com as regras ativas do sistema, ou seja, aquelas que são utilizadas para alertar o usuário de problemas em interesses transversais. Esta classe possui uma instância de `SoCPreferences` que registra as preferências dos usuários em relação à aplicação das regras. Em especial, possui valores limites definidos para cada regra e um vetor com os interesses que o usuário deseja avaliar no sistema. A classe `HeuristicRules` armazena todas as regras heurísticas (ativas ou não) da ferramenta. As regras que não estão ativas podem ser ativadas pela interface da ferramenta (Figura 21).

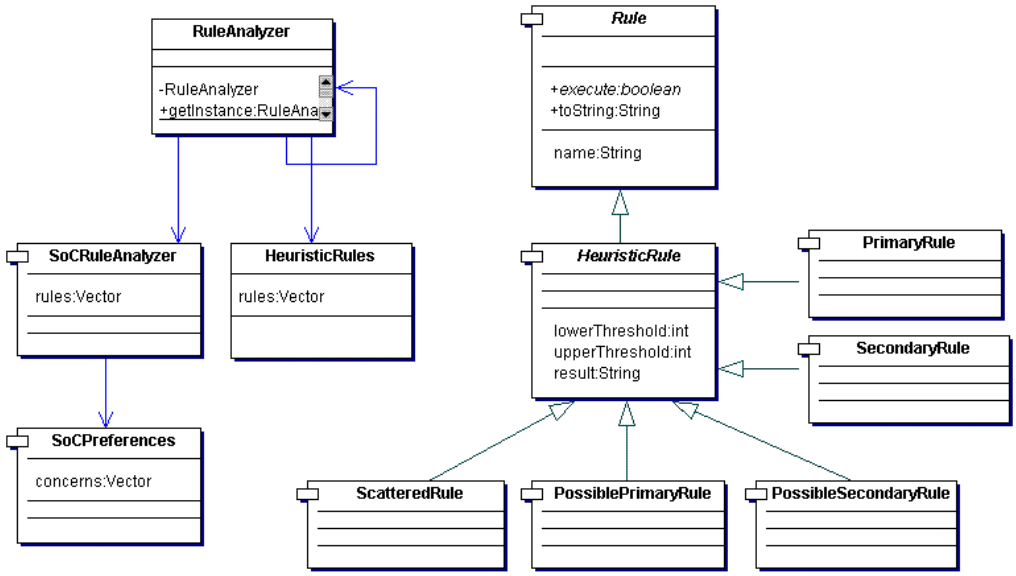


Figura 30 – Diagrama de classes parcial do módulo Analisador de Regras

Além das classes analisadoras, a Figura 30 apresenta também as regras heurísticas implementadas neste módulo. A versão atual da ferramenta disponibiliza regras de separação de interesse (Seção 5.2). As regras de acoplamento e coesão (Seção 5.3) ainda não foram implementadas porque dependem de métricas não disponíveis nesta versão. Como apresentado na Figura 30, a classe abstrata `Rule` é superclasse de todas as regras. Entretanto, as regras heurísticas concretas devem estender da classe `HeuristicRule` (que estende `Rule`). Esta classe define valores limites inferiores e superiores, além de armazenar uma *string* com informação sobre o resultado da aplicação da regra. Regras auxiliares (por exemplo, regras para calcular porcentagem e regras lógicas "e", "ou", "pelo menos um" e "para todos") podem ser subclasses diretas de `Rule`. Para simplificar o diagrama da Figura 30 as regras auxiliares não são mostradas.

7
Estudos Experimentais

A abordagem proposta neste trabalho emergiu principalmente de experimentos práticos realizados nas atividades de pesquisa deste curso de mestrado. No total cinco estudos experimentais foram realizados e estes são apresentados no decorrer deste capítulo. Os dois primeiros estudos envolvem padrões de projeto [26], o primeiro [27] [28] em implementações isoladas e o segundo no contexto de uma aplicação [8]. O terceiro estudo experimental é um sistema multi-agentes [30] para desenvolvimento e gerência de portais na internet. Os dois últimos estudos são sistemas de informação *web* para queixas em relação a serviços de saúde [44] e para acompanhamento de auto-estradas brasileiras [11], respectivamente. Todas as aplicações envolvem implementações orientadas a objetos (OO) e orientadas a aspectos (OA) feitas em Java e AspectJ. É importante observar que estes estudos tiveram um importante papel não só na avaliação da abordagem, mas também em seu amadurecimento.

7.1.
Sistemas Utilizados nos Estudos Experimentais

Os cinco estudos realizados no contexto desta dissertação foram selecionados por quatro razões principais. Em primeiro lugar, porque eles são bastante heterogêneos e envolvem aplicações de diferentes domínios. Além disso, os dois primeiros estudos são largamente baseados em padrões de projeto, e, portanto, fortes candidatos a atenderem requisitos de qualidade relevantes neste trabalho como reusabilidade. A terceira razão por termos selecionado tais estudos é que, a exceção do primeiro, eles refletem aplicações realistas feitas em Java e AspectJ. Finalmente, a quarta razão é que os estudos possuem variados graus de complexidade em relação à natureza de seus interesses.

Os estudos experimentais deste trabalho foram aplicados na avaliação dos três pilares básicos da abordagem proposta: método de avaliação (Capítulo 4), regras heurísticas (Capítulo 5) e ferramenta de medição e avaliação (Capítulo 6).

Entretanto, nem todos os estudos avaliam os mesmos elementos da abordagem. Por exemplo, no sistema Telestrada [11] é utilizada apenas uma versão preliminar da ferramenta. Por outro lado, três dos cinco estudos são utilizados para avaliar e evoluir os principais elementos de abordagem de forma integrada. A Tabela 10 ressalta quais elementos da abordagem recebem contribuições de quais estudos. Nas linhas desta tabela são listados os estudos e nas colunas os elementos da abordagem, i.e., método, regras e ferramenta. Cada célula marcada com "X" significa que no elemento daquela coluna é utilizado o estudo correspondente à linha.

Tabela 10 – Elementos da abordagem avaliados nos estudos experimentais

Estudos	**Elementos Principais da Abordagem**		
	Método	**Regras**	**Ferramenta**
Padrões GoF Individuais	X	X	
Composição de Padrões	X	X	X
Portalware	X	X	X
Health Watcher	X	X	X
Telestrada			X

7.1.1. Padrões de Projetos

O primeiro estudo experimental [27] [28] [29] realizado no contexto desta dissertação compara implementações OO e OA dos 23 padrões de projeto descritos pela *Gang-of-Four* (GoF) [26]. As implementações dos padrões avaliados foram desenvolvidas por Hannemann e Kiczales [36], e estes autores fazem uma comparação qualitativa dos padrões baseada em atributos da engenharia de software que não são bem conhecidos, tais como, "*componibilidade*" e "*conectabilidade*"[12]. Em nosso estudo, entretanto, são utilizados atributos rigorosos da engenharia de software no critério de avaliação, tais como acoplamento, coesão, tamanho e separação de interesses. Desta forma, este estudo experimental pode ser considerado complementar ao trabalho de Hannemann e Kiczales por utilizar critérios diferentes na comparação das soluções OO e OA.

[12] Tradução para os termos em inglês *composability* e *pluggability* usados no trabalho de Hannemann e Kiczales [36].

Com o objetivo de permitir que as distintas implementações de um mesmo padrão sejam comparáveis são feitas pequenas alterações no código original disponibilizado por Hannemann e Kiczales em [67]. Dois exemplos destas alterações são: (i) garantia de um mesmo estilo de programação e (ii) adição (ou remoção) de funcionalidades que ocorre em uma das implementações e não ocorre em outra. A decisão entre adicionar ou remover uma funcionalidade foi tomada dependendo de sua relevância para o contexto do padrão. Esta etapa de garantir implementações comparáveis é chamada de alinhamento do código. Após o alinhamento do código, são aplicadas as três primeiras fases do método de avaliação: medição, aplicação de regras e análise.

Na atividade de medição, o conjunto de métricas definido na Tabela 1 (Subseção 4.2.1) é utilizado sobre as implementações Java e AspectJ dos padrões GoF. Com o resultado das medições, as regras são então aplicadas de tal forma a gerarem alertas. Estes alertas advertem sobre a possibilidade do interesse de algum padrão ser transversal ao sistema. Desta forma, tais alertas servem de guia para a fase de análise, na qual os possíveis interesses transversais são verificados. Neste estudo, a ferramenta de suporte ao método não foi utilizada porque esta ainda se encontrava em fase inicial de desenvolvimento. Os resultados completos da avaliação feita neste estudo encontram-se disponíveis nas referências [27] [28] [29] e no Apêndice A são apresentados os dados detalhados de três padrões: *Observer*, *Factory Method* e *Builder*.

7.1.2. Middleware OpenOrb: Um Estudo de Composição de Padrões

Um dos principais problemas na implementação de múltiplos padrões de projetos em um sistema é que estes padrões não se limitam a afetar as classes base da aplicação. Por outro lado, eles também interagem entre si de forma tão heterogênea que torna difícil a separação dos elementos que implementam cada padrão. No estudo de caso anterior, o objetivo foi avaliar de maneira isolada as implementações Java e AspectJ dos padrões de projeto feitas por Hannemann e Kiczales [36]. Neste segundo estudo, entretanto, os 23 padrões são avaliados no contexto de uma aplicação realística e na presença de grandes interações entre eles. As interações entre os padrões é resultado principalmente da forma em que

estes se compõem, variando desde uma simples chamada de método até o compartilhamento de uma ou mais classes do sistema.

A aplicação avaliada neste estudo compreende um sistema de *middleware* [3] [9] em que duas diferentes versões foram utilizadas, a primeira OO implementada em Java e a segunda OA implementada em AspectJ. A Figura 31 mostra a composição de diversos padrões em um diagrama de classes parcial da versão Java deste sistema. Estes padrões se associam para atingir os requisitos cruciais definidos neste software, como componibilidade e adaptabilidade. Cada número apresentado na figura representa um padrão específico e estes números são associados aos métodos e atributos que implementam o padrão correspondente. Por exemplo, a implementação do padrão *Decorator* (representado pelo número 1) inclui o atributo `bind` e vários métodos como `makeRequest()`, `breakBind()`, `rebind()`, `checkPreMethods()` e `checkPosMethods()` da classe `DecoratorBind`.

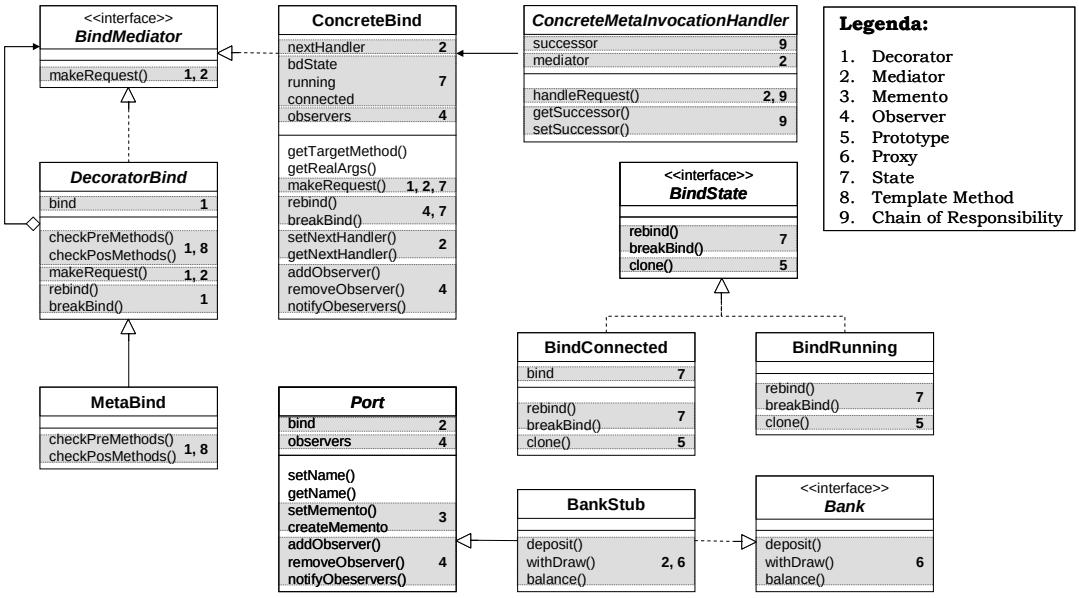


Figura 31 – Diagrama de classes OO parcial da middleware OpenOrb

Neste estudo experimental os padrões são avaliados em composições de dois a dois, ou seja, cada par de padrão que possui algum nível de interação é isolado do restante do sistema para que possa ser estudado. O processo de avaliação segue as atividades definidas no método apresentado no Capítulo 4 para um total de 62 composições. Desta forma, o primeiro passo é medir cada uma das composições utilizando as métricas descritas na Tabela 1 (Subseção 4.2.1). Em seguida, aplicar as regras heurísticas de separação de interesses, acoplamento e

coesão (Capítulo 5). Finalmente, a última atividade do processo de avaliação é avaliar os alertas gerados pela aplicação das regras. Note que, assim como no primeiro estudo, o objetivo é comparar as implementações e, portanto, não foram aplicadas refatorações nas versões do sistema.

A ferramenta apresentada no Capítulo 6 foi utilizada para automatizar as atividades de medição e aplicação das regras neste estudo. É importante ressaltar que este segundo estudo experimental foi de fundamental importância para o amadurecimento da ferramenta, em especial, do módulo Analisador de Regras. O resultado detalhado da avaliação de ambas as versões deste sistema de *middleware* pode ser encontrado nas referências [8] [9]. O Apêndice B deste documento apresenta, entretanto, as informações obtidas para 4 pares de padrões: *Observer* com *Factory Method*, *Singleton* com *Façade*, *Proxy* com *Interpreter* e *Prototype* com *State*.

7.1.3. Portalware

Sistemas Multi-Agentes (SMA) envolvem vários interesses, como Autonomia, Adaptação e Colaboração, que não são bem tratados pelas abstrações dos paradigmas tradicionais de desenvolvimento (por exemplo, OO). O quão reutilizável e manutenível é um SMA depende largamente de quão bem separados estão os interesses dos agentes nas fases de projeto e implementação [30] [31]. Assim, em nosso terceiro estudo experimental é utilizado um SMA para desenvolvimento e gerência de portais na internet, chamado Portalware [30]. Este sistema foi desenvolvido no Laboratório de Engenharia de Software (LES) desta universidade e implementado em duas versões: Java e AspectJ.

O objetivo da utilização do Portalware neste estudo é avaliar como as técnicas de desenvolvimento OO e OA podem ser usadas para separar os interesses de agentes. Para isso, foram comparadas as duas implementações desta aplicação (Java e AspectJ) que possuem exatamente as mesmas funcionalidades. A Figura 32 ilustra o projeto parcial da primeira versão do sistema feito em Java. A versão AspectJ foi obtida a partir de refatorações da implementação original e é parcialmente apresentada no diagrama de classes da Figura 33. Como nos dois estudos de caso anteriores, as três primeiras atividades do método de avaliação foram utilizadas: medição, aplicação de regras heurísticas e análise dos resultados.

A ferramenta de medição e avaliação também foi utilizada neste estudo, suportando as duas primeiras atividades citadas. Os dados de medição e aplicação das regras para os interesses Adaptação, Colaboração e Autonomia são apresentados no Apêndice C.

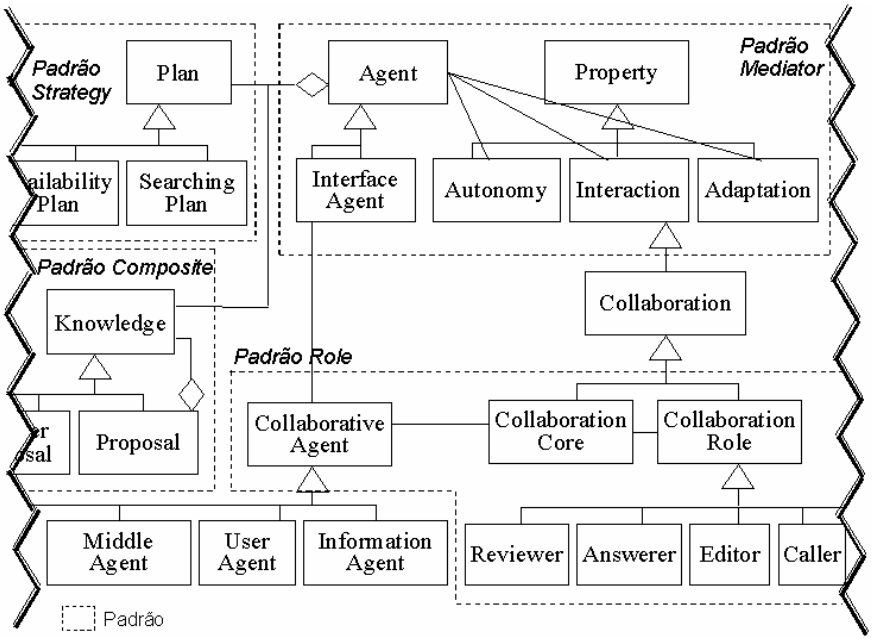


Figura 32 – Diagrama de classes OO parcial do Portalware

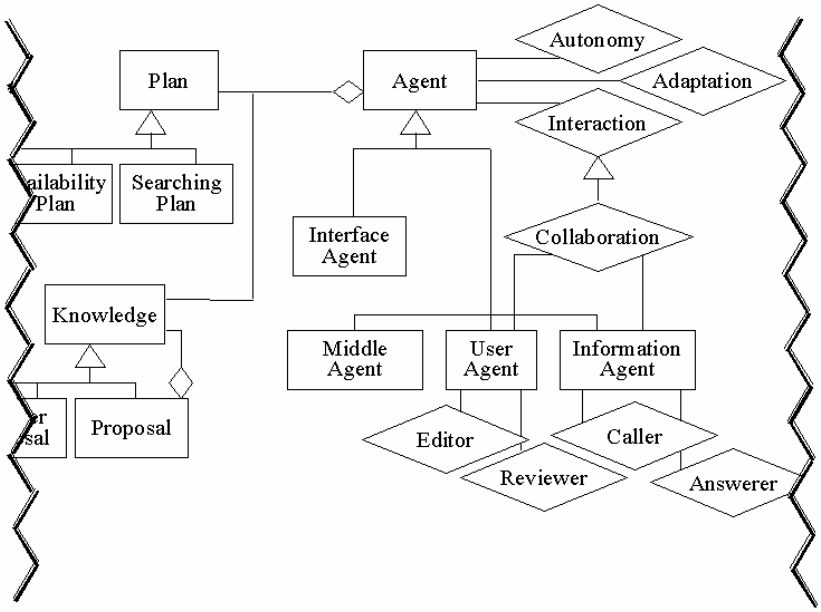


Figura 33 – Diagrama de classes OA parcial do Portalware

7.1.4. Health Watcher

O quarto estudo experimental deste trabalho consiste em comparar as implementações OO e OA de um sistema de informação baseado em *web*. A principal funcionalidade deste sistema, chamado Health Watcher [44], é o registro de queixas para melhoria da qualidade de serviços providos por instituições de saúde. O estudo envolve a avaliação de dois interesses transversais bem conhecidos na literatura – Concorrência e Distribuição – e seus efeitos nos componentes do sistema. A escolha do Health Watcher como estudo de caso se deve por três razões principais. Em primeiro lugar, este sistema pertence a um grupo externo ao nosso ambiente de pesquisa e tanto a versão OO quanto a versão OA foram desenvolvidas por pesquisadores da Universidade Federal de Pernambuco. A segunda razão é que avaliações anteriores, tanto qualitativas [7] quanto baseadas exclusivamente em métricas [44], já haviam sido conduzidas. Estas avaliações anteriores permitem comparação de resultados com a nossa abordagem baseada em regras heurísticas. O terceiro motivo pela escolha deste sistema é que sua implementação envolve várias tecnologias Java comumente adotadas pela indústria, como Invocação Remota de Método (RMI), *Servlets* e Conexão com Banco de Dados (JDBC).

As versões OO e OA do sistema de informação Health Watcher são implementadas nas linguagens Java e AspectJ, respectivamente. Na implementação Java, o padrão arquitetural “Em Camadas” [7] é usado para estruturar os componentes em quatro níveis principais: interface com o usuário, distribuição, negócios e dados. A Figura 34 apresenta o diagrama de classes parcial desta implementação e destaca as quatro camadas do padrão. A implementação AspectJ tem como propósito separar os interesses transversais relativos à Distribuição, Persistência e Concorrência. Esta versão ainda é estruturada seguindo o padrão “Em Camadas”, entretanto, o interesse Distribuição não é mais implementado como uma camada e sim como um aspecto do sistema. O resultado da aplicação das métricas e regras heurísticas para os interesses Concorrência e Distribuição deste estudo é apresentado no Apêndice D.

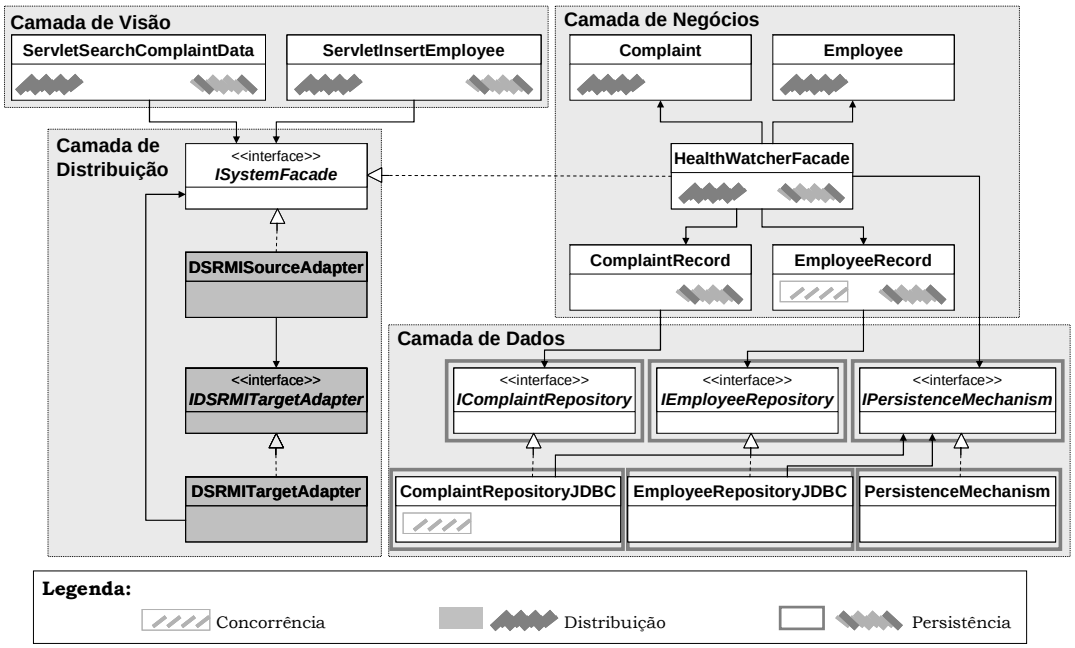


Figura 34 – Diagrama de classes OO parcial do Health Watcher

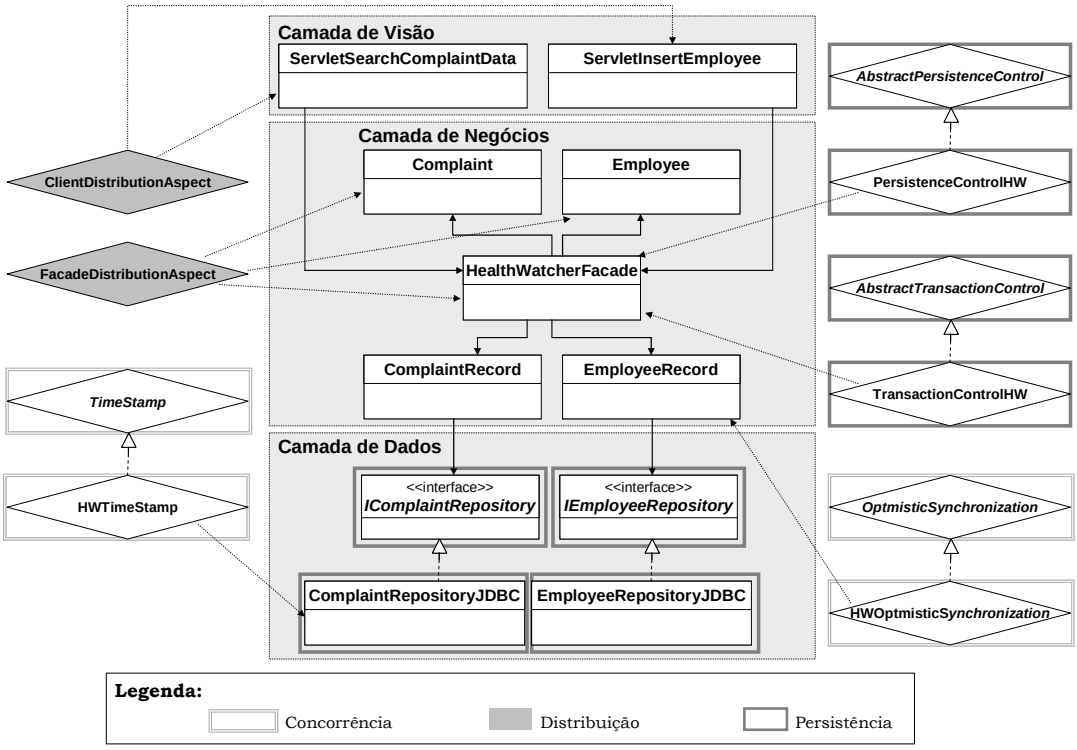


Figura 35 – Diagrama de classes OA parcial do Health Watcher

7.1.5. Telestrada

Em nosso último estudo experimental é utilizado o Telestrada [10] [11], um sistema de informação sobre auto-estradas brasileiras. Este sistema envolve cinco subsistemas: Central de Banco de Dados, Sistema de Informação Geográfica, Centro de Operação de Chamadas (*call-center*), Sistema de Operações em Estradas e Sistema de Gerência de Reclamações. O objetivo deste estudo é verificar se a separação do interesse de tratamento de exceções foi bem sucedida na implementação OA do sistema Telestrada. Para isso são comparadas as implementações antes (OO) e após (OA) este interesse ter sido separado em aspecto. O módulo de medição da ferramenta apresentada no Capítulo 6 foi utilizado e a avaliação foi baseada exclusivamente em métricas. Portanto, o método de avaliação e as regras heurísticas não foram utilizados, o que oferece oportunidade para um estudo futuro mais aprofundado. O resultado da avaliação baseada em métricas é reportado na referência [10].

7.2. Contribuições dos Estudos

Os cinco estudos de caso realizados neste trabalho contribuíram de forma unificada para avaliação e amadurecimento da abordagem. Pelas características individuais de cada estudo, eles puderam exercitar pontos diferentes da abordagem e contribuir de forma mais completa. Por exemplo, a organização do método em etapas emergiu destes estudos. Na verdade, o método de avaliação proposto neste trabalho reflete as atividades feitas por pesquisadores em estudos quantitativos. Desta forma, a grande contribuição do método é organizar e sistematizar as atividades de avaliação desempenhadas em outros estudos semelhantes. Em contra partida, os cinco estudos deste trabalho foram importantes para que pudéssemos perceber as prioridades na avaliação de software OA. Ou seja, percebemos as vantagens da avaliação orientada a interesse, uma vez que este atributo afeta a qualidade de outros atributos do software.

Em relação ao conjunto de regras heurísticas, os cinco estudos de caso realizados neste trabalho permitiram efetuar uma avaliação empírica das mesmas. Os estudos contribuíram para inferir novas regras e melhorar àquelas existentes.

Desta forma, o conjunto de regras foi constantemente reformulado a cada novo estudo de tal forma a se tornar mais eficaz na identificação de problemas. As regras que não foram eficazes nos estudos passaram por aperfeiçoamentos ou foram descartadas. Assim, o conjunto final de regras apresentado no Capítulo 5 tem se mostrado como um importante instrumento para identificar problemas não triviais em medições. A Tabela 11 destaca os principais problemas identificados pela utilização das regras heurísticas nos sistemas avaliados. Estes problemas não são facilmente detectáveis em uma avaliação baseada exclusivamente em métricas, como discutido na Seção 5.1.

Tabela 11 – Problemas identificados pelas regras heurísticas nos sistemas

Estudos de Caso		**Interesses Avaliados**	**Problemas Identificados**					
			A	**B**	**C**	**D**	**E**	**F**
1º	Padrão Builder	Papel Director					X	
	Padrão Factory Method	Papel Creator		X			X	
	Padrão Observer	Papel Observer						X
		Papel Subject						
2º	Composição Factory Method com Observer	Padrão Factory Method	X			X	X	
		Padrão Observer						
	Composição Façade com Singleton	Padrão Façade			X			
		Padrão Singleton						
	Compoição Prototype com State	Padrão Prototype						
		Padrão State					X	X
	Composição Interpreter com Proxy	Padrão Interpreter			X			
		Padrão Proxy						
3º	Health Watcher	Concorrência		X		X		
		Distribuição						
		Persistência						
4º	Portalware	Adaptação						
		Autonomia						
		Colaboração		X		X	X	X

A Tabela 11 apresenta os quatro estudos nos quais as regras heurísticas foram utilizadas e seis problemas associados à interpretação equivocada dos resultados de medições. Para o primeiro estudo destacamos três padrões nesta tabela e os papéis destes padrões avaliados. Para o segundo estudo são mostradas quatro composições de padrões e seus respectivos interesses (padrões) avaliados. As seis últimas colunas da Tabela 11 mostram os problemas identificados pelas regras em nossos estudos: (a) alerta falso por interesse espalhado e entrelaçado; (b) alerta falso por alto acoplamento e/ou baixa coesão; (c) resultados não mostram um problema existente; (d) resultados não indicam onde está o problema; (e) conflitos entre valores medidos; e (f) métricas não relacionam os problemas existentes. Estes problemas são apresentados e exemplificados na Seção 5.1.

Na Tabela 11, cada “X” significa que o conjunto de regras apontou um problema que não é facilmente identificado por avaliação exclusivamente baseada em medição. Note que a célula vazia nem sempre significa que o sistema não possui problemas de interesse transversal, em certos casos a ausência do “X” significa que o problema existente é facilmente identificado a partir dos números. Por exemplo, na avaliação heurística do papel *Subject* do padrão *Observer* [26] este interesse é classificado como “Secundário” (Apêndice A) e ele é realmente um interesse transversal [28] [36]. Entretanto, a Tabela 11 não apresenta nenhum “X” na linha relativa a este interesse porque a avaliação direta sobre os resultados de medição (sem regras) é suficiente para identificar este problema.

O quinto estudo de caso, Telestrada, foi utilizado para avaliar a ferramenta de medição e avaliação. Como neste estudo foi utilizada uma versão inicial da ferramenta, ele foi de fundamental importância para a identificação de *bugs*. Além disso, o Telestrada foi desenvolvido por pesquisadores externos ao nosso grupo de pesquisa o que é importante para garantir a robustez da ferramenta. A ferramenta AJATO também foi utilizada nos outros estudos de caso, exceto no primeiro, permitindo seu amadurecimento durante este trabalho. Atualmente, esta ferramenta vem sendo distribuída e utilizada por pesquisadores em diversas instituições como, por exemplo, Universidade de Lancaster (UK), Universidade Estadual de Campinas (UNICAMP), Universidade Federal do Rio Grande do Norte (UFRN), Universidade de Ciências Aplicadas de Kiel (Alemanha) e Universidade de Toronto (Canadá).

7.3. Restrições dos Estudos

Algumas métricas utilizadas neste estudo (LOC e LCOO, por exemplo) têm sido criticadas na literatura [37]. Na verdade, resultados obtidos por métricas de tamanho podem ser difíceis de interpretar, pois altos valores podem significar tanto replicação de código como melhoria de modularidade. Além disso, a métrica de coesão LCOO é criticada por não apresentar uma boa fundamentação teórica ou validação empírica. Apesar das limitações destas métricas, o maior problema com elas ocorre quando as métricas são utilizadas de forma isolada, o que não é o caso deste trabalho. Em nossos estudos, as métricas de acoplamento, coesão,

tamanho e separação de interesses são utilizadas de forma complementar e têm se apresentado como fontes confiáveis de informação sobre o sistema.

Os cinco estudos de caso explorados neste capítulo são dependentes de características das linguagens de programação utilizadas: Java e AspectJ. Entretanto, a decisão de utilizar estes sistemas foi tomada porque eles possuem boas implementações OO e OA. Além disso, as linguagens Java e AspectJ são as mais difundidas destes paradigmas e possuem ampla utilização tanto na comunidade acadêmica quanto na indústria. Um outro ponto em favor das linguagens utilizadas é que elas implementam os principais conceitos definidos para os paradigmas OO e OA.

8
Considerações Finais

O DSOA é um paradigma recente que promete aumento da qualidade em sistemas de software, por meio de uma melhor separação de interesses. Entretanto, alguns atributos de qualidade importantes podem ser afetados negativamente pelo uso inadequado das abstrações existentes neste paradigma. O DSOA vem ganhando força tanto na academia quanto na indústria, mas, apesar de sua popularidade, pouca atenção tem sido dada a métodos de avaliação que mostrem que sistemas desenvolvidos neste paradigma satisfazem a certas propriedades desejáveis. A pesquisa atual nesta área tem se preocupado principalmente com a construção de linguagens de modelagem [46] e implementação [56] [66]. Apenas alguns estudos experimentais [8] [11] [27] [36] têm sido realizados no contexto de avaliação, mas estes são predominantemente qualitativos e específicos de um domínio de aplicação [36] ou interesse [44]. Neste sentido, este trabalho define uma abordagem sistemática para auxiliar engenheiros de software na avaliação da qualidade de seus artefatos. A abordagem apresentada neste documento é composta de três elementos principais: um método de avaliação, um conjunto de regras heurísticas e uma ferramenta de suporte.

8.1.
Contribuições

A medição de propriedades do sistema (tais como acoplamento, coesão, tamanho e separação de interesses) tem se mostrado uma prática promissora para predizer a qualidade nas fases de projeto e implementação do DSOA [21] [57]. Neste contexto, este trabalho apresenta as seguintes contribuições principais:

- Um método de avaliação de sistemas orientados a aspectos. O método é iterativo, organizado em etapas, apoiado por métricas e por regras heurísticas. Este método encontra-se publicado em [21].
- Um conjunto de regras heurísticas orientado a interesses para projeto e implementação. Tais regras são associadas aos resultados de

medições e oferecem informações semânticas sobre os interesses e componentes do sistema. Algumas regras propostas nesta dissertação foram publicadas em [21].

- Uma ferramenta implementada e documentada para dar suporte às atividades do método de avaliação e às regras. Um dos benefícios centrais da ferramenta é automatizar a geração de alertas de possíveis problemas relativos a interesses transversais, apoiando engenheiros de software na identificação de refatorações que possam ser necessárias. A ferramenta é um projeto *SourceForge* hospedado em [53].
- Cinco estudos experimentais envolvendo implementações OO e OA para avaliação do método, regras heurísticas e ferramenta. É importante ressaltar que estes estudos também contribuíram para o amadurecimento da abordagem de avaliação proposta. Alguns destes estudos encontram-se publicados em [8] [27]. Vale ainda dizer que o primeiro estudo foi convidado especial para publicação na edição inaugural de LNCS Transactions on Aspect-Oriented Software Development [28].

Nos estudos realizados, a abordagem baseada em regras tem se mostrado bastante útil para apontar problemas não triviais resultantes de uma análise equivocada das medições. Como resultado, as métricas, as regras e o método de avaliação têm sido utilizados por pesquisadores em universidades internacionais como Nélio Cacho da Universidade de Lancaster (UK) e Hans-Arno Jacobsen da Universidade de Toronto (Canadá). Além disso, a ferramenta de medição e avaliação que suporta a abordagem vem ganhando rápida visibilidade entre pesquisadores de diferentes instituições. Três exemplos podem ser citados: (i) a ferramenta foi utilizada no último estudo experimental deste trabalho, realizado em parceria com o aluno de mestrado Fernando Castor na Universidade Estadual de Campinas (UNICAMP); (ii) o módulo de medição da ferramenta está sendo estendido para comportar novas métricas pelo aluno de graduação Gary Thewlis, orientado pelo professor Alessandro Garcia, da Universidade de Lancaster – Inglaterra; e (iii) a ferramenta vem sendo estendida para suportar múltiplas linguagens de programação OA em parceria com o aluno de mestrado Thiago Bartolomei da Universidade de Ciências Aplicadas de Kiel – Alemanha. Esta

última extensão da ferramenta será feita em dois estágios: primeiro para suportar CaesarJ [51], e depois a generalização para outras linguagens utilizando mecanismos de *plugins*. Todos estes trabalhos colaborativos que vêm sendo realizados, envolvendo diversas pessoas e instituições, só vêm contribuir para o crescimento da abordagem e da ferramenta.

8.2. Trabalhos Futuros

O trabalho apresentado nesta dissertação pode ser continuado com a realização das atividades apresentadas a seguir.

Método de Avaliação, Métricas e Regras Heurísticas

- Definição de novas métricas e regras heurísticas para extensão do método de avaliação. As novas regras podem combinar uma ou mais das métricas existentes, ou mesmo utilizar novas métricas incorporadas ao método, para identificação de outras características importantes de interesses (ou componentes) na avaliação de qualidade do software. De fato, através da nossa experiência empírica, percebemos que o método deveria também prover suporte explícito a outras dimensões complementares àquelas que já são suportadas, tais como complexidade das interfaces aspectuais, e interação entre aspectos.
- Extensão do método proposto de tal forma a associar as regras definidas pelo método com possíveis sugestões de refatorações, melhorando a automatização dos processos de projeto da qualidade e manutenções periódicas.
- Definição de um conjunto de refatorações orientadas a aspectos para compor o método, ou selecioná-las dentre as existentes na literatura [35] [38] [52]. Estas refatorações devem ser utilizadas no método de avaliação como possíveis soluções para resolver problemas de projeto e implementação apontados pelas regras heurísticas.
- Propor métricas e regras para guiar o engenheiro de software nas decisões que ocorrem antes da fase de implementação, ou seja, nos

níveis prelimindares de desenvolvimento. Uma abordagem à priori é importante porque ela pode evitar que problemas de interesses transversais ocorram durante a implementação. E, portanto, evitar refatorações do sistema que geralmente são custosas.

Ferramenta AJATO

- Extensão da ferramenta para tornar possível medir e avaliar sistemas implementados em outras linguagens de programação. Este trabalho já se encontra em andamento por uma iniciativa de software livre, hospedado em [53]. A idéia é extensão da ferramenta por adição de *plugins* que descrevem as novas linguagens a serem incorporadas. Cada *plugin* deve definir atributos relevantes da linguagem como a sua gramática e seu analisador léxico.
- Extensão da ferramenta para avaliar artefatos no nível de projeto detalhado, além de artefatos de código. Os artefatos de entrada em uma avaliação no nível de projeto devem constar diagramas estruturais e dinâmicos UML [5] como, por exemplo, diagramas de classes e diagramas de seqüência.
- Desenvolvimento de um novo módulo na ferramenta para permitir aplicação de refatorações OA em programas. Esta atividade é muito importante visto que o método de avaliação identifica problemas que podem ser resolvidos por refatorações OA, entretanto, a ferramenta ainda não suporta esta atividade. É importante ressaltar que o ambiente MetaJ [55] permite análise, transformação e geração de código, portanto, as refatorações podem ser implementadas utilizando os recursos deste ambiente.

Estudos Experimentais

- Realização de um estudo mais aprofundado em relação ao quinto estudo de caso, Telestrada (Subseção 7.1.5). A avaliação neste estudo foi baseada exclusivamente em métricas, ou seja, o método de avaliação e as regras heurísticas não foram utilizados. Desta

forma, torna-se natural uma avaliação que aplique todas as etapas da abordagem proposta.

- Realização de novos estudos experimentais para cobrir outras variáveis da abordagem como, por exemplo, sistemas implementados em diferentes linguagens. Os cinco estudos realizados neste trabalho se limitaram a estudos comparativos entre implementações Java e AspectJ e outros estudos tormam-se importantes para maior amadurecimento da abordagem. Entretanto, é importante ressaltar que (i) os sistemas avaliados possuem variados graus de complexidade e cobrem diferentes domínios, e (ii) as linguagens utilizadas são as mais difundidas e realizam as principais abstrações dos paradigmas OO e AO, respectivamente.

9
Referências

[1] AVGUSTINOV, P.; CHRISTENSEN, A. S.; HENDREN, L.; KUZINS, S.; LHOTÁK, J.; LHOTÁK, O.; MOOR, O.; SERENI, D.; SITTAMPALAM, G.; TIBBLE, J. **abc: An extensible AspectJ compiler**. In: ACM International Conference on Aspect-Oriented Software Development (AOSD'05). Proceedings… Chicago, USA, 2005, p. 87-98.

[2] BASILI, V.; BRIAND, L.; MELO, W. A Validation of Object-Oriented Design Metrics as Quality Indicators. **IEEE Transactions on Software Engineering**, v.22, n.10, p. 751-761, 1996.

[3] BLAIR, G. S.; COULSON, G.; ANDERSEN, A.; BLAIR, L.; CLARKE, M.; COSTA, F.; DURAN-LIMON, H.; FITZPATRICK, T.; JOHNSTON, L.; MOREIRA, R.; PARLAVANTZAS, N. The design and implementation of Open ORB. **IEEE Distributed Systems Journal**, v. 2, n. 6, 2001.

[4] BOEHM, B. W. **Characteristics of Software Quality**. TRW Series of Software Technology. Amsterdam, North Holland, 1978.

[5] BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **The Unified Modeling Language User Guide**. Addison-Wesley Professional, 1998. 482p.

[6] BRAND, M.; DEURSEN, A.; HEERING, J.; JONG, H.; JONGE, M.; KUIPERS, T.; KLINT, P.; MOONEN, L.; OLIVIER, P. A.; SCHEERDER, J.; VINJU, J. J.; VISSER, E.; VISSER, J. **The ASF + SDF Meta-Environment: A Component-Based Language Development Environment**. In: Compiler Construction (CC'01). Proceedings… Genova, Italy, 2001, p. 365-370.

[7] BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M.; SOMMERLAD, P.; STAL, M. **Pattern-Oriented Software Architecture: A System of Patterns**. John Wiley & Sons, 1996. 476p.

[8] CACHO, N. F.; SANT'ANNA, C. N.; FIGUEIREDO, E. M. L.; GARCIA, A. F.; BATISTA, T. V.; LUCENA, C. J. P. **Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming**. In: 5th International Conference on Aspect Oriented Software Development (AOSD'06). Proceedings… Bonn, Germany, 2006. (to appear)

[9] CACHO, N. F.; FIGUEIREDO, E. M. L.; SANT'ANNA, C. N.; GARCIA, A. F.; BATISTA, T. V.; LUCENA, C. J. P. **Aspect-Oriented Composition of Design Patterns: A Quantitative Assessment**. Technical Report, n. 34/05, Computer Science Department, PUC-Rio. Rio de Janeiro 2005, 29 p.

[10] FILHO, F. C.; GUERRA, P. A. C.; PAGANO, V. A.; RUBIRA, C. M. F. A Systematic Approach for Structuring Exception Handling in Robust

Component-Based Software. **Journal of the Brazilian Computer Society**, 2005. (to appear)

[11] CASTOR, F.; RUBIRA, C.; GARCIA, A. **A Quantitative Study on the Aspectization of Exception Handling**. In: ECOOP'2005 Workshop on Exception Handling in Object-Oriented Systems. Proceedings… Glasgow, UK, 2005.

[12] CASTOR, F.; OLIVEIRA, K.; SOUZA, A.; SANTOS, G.; E BORBA, P. **JaTS: A Java Transformation System**. In: XV Simpósio Brasileiro de Engenharia de Software. Anais… Rio de Janeiro, Brazil, 2001, p. 374-379.

[13] CECCATO, M.; TONELLA P. **Measuring the Effects of Software Aspectization**. In: 1st Workshop on Aspect Reverse Engineering. Proceedings… The Netherlands, 2004. (CD-ROM)

[14] CHIDAMBER, S.; KEMERER, C. A Metrics Suite for Object Oriented Design. **IEEE Transactions on Software Engineering**, v. 20 n. 6, p. 476-493, 1994.

[15] Concern Manipulation Environment. Disponível em: <http://www.research.ibm.com/cme/> Acesso em: 13 mar. 2006.

[16] CORDY, J.; DEAN, T.; MALTON, A.; SCHNEIDER, K. Source Transformation in Software Engineering using the TXL Transformation System. **Journal of Information and Software Technology**, v. 44, n. 13, p.827-837, 2002.

[17] DEITEL, H. M.; DEITEL, P. J. **Java: How to Programme**. Prentice Hall, 1999. 1500p.

[18] DIJKSTRA, E. **A Discipline of Programming**. Prentice-Hall, Englewood Cliffs, New Jersey, 1976. 217p.

[19] EISENBARTH, T.; KOSCHKE, R.; SIMON, D. Locating Features in Source Code. **IEEE Transactions on Software Engineering**, v. 29, n. 3, p. 210-224, 2003.

[20] FENTON, N.; PFLEEGER, S. **Software Metrics: A Rigorous and Practical Approach**. 2.ed. London: PWS, 1997. 638p.

[21] FIGUEIREDO, E. M. L.; GARCIA, A. F.; SANT'ANNA, C. N.; KULESZA, U.; LUCENA, C. J. P. **Assessing Aspect-Oriented artifacts: Towards a Tool-Supported Quantitative Method**. In: 9th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'05). Proceedings... UK, 2005.

[22] FIGUEIREDO, E. M. L.; STAA, A. **Avaliação de um Modelo de Qualidade para Implementações Orientadas a Objetos e Orientadas a Aspectos**. Monografia em Ciência da Computação nº 14/05, Departamento de Informática, PUC-Rio. Rio de Janeiro, 2005, 29 p.

[23] FILHO, F.; MARANHA, R.; RUBIRA, C.; GARCIA, A. A Quantitative Study on the Aspectization of Exception Handling. **LNCS Advances on Exception Handling Techniques II**, State-of-the-Art Survey Series, Springer, 2006. (to appear)

[24] FILMAN, R. E.; ELRAD, T.; CLARKE, S.; AKSIT, M. **Aspect-Oriented Software Development**. Addison-Wesley Professional, 2004. 800p.

[25] FOWLER, M. **Refactoring: Improving the Design of Existing Code**. 1st ed. Addison Wesley, 1999. 464p.

[26] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley Professional, 1995. 395p.

[27] GARCIA, A. F.; SANT'ANNA, C. N.; FIGUEIREDO, E. M. L.; KULESZA, U.; LUCENA, C. J. P.; STAA, A. Modularizing Design Patterns with Aspects: A Quantitative Study. **LNCS Transactions on Aspect-Oriented Software Development (TAOSD'05)**, v. 31, n. 2, p. 36-74, 2006.

[28] GARCIA, A. F.; SANT'ANNA, C. N.; FIGUEIREDO, E. M. L.; KULESZA, U.; LUCENA, C. J. P.; STAA, A. **Modularizing Design Patterns with Aspects: A Quantitative Study**. In: 4th International Conference on Aspect Oriented Software Development (AOSD'05). Proceedings… USA. 2005.

[29] GARCIA, A. F.; SANT'ANNA, C. N.; FIGUEIREDO, E. M. L.; KULESZA, U.; LUCENA, C. J. P.; STAA, A. **Aspectizing Design Patterns: Rewards and Pitfalls**. Monografia em Ciência da Computação nº 43/04, Departamento de Informática, PUC-Rio. Rio de Janeiro, 2004, 21p.

[30] GARCIA, A.; SANT'ANNA, C.; CHAVEZ, C.; SILVA, V.; LUCENA, C.; STAA, A. **Separation of Concerns in Multi-Agent Systems: An Empirical Study**. In: Software Engineering for Multi-Agent Systems II, Springer, LNCS 2940, 2004.

[31] GARCIA, A. **From objects to agents: an aspect-oriented approach**. Tese de Doutorado, Departamento de Informática, PUC-Rio. Rio de Janeiro, 2004, 319p.

[32] GARCIA, A.; SILVA, V.; CHAVEZ, C.; LUCENA, C. Engineering Multi-Agent Systems with Aspects and Patterns. **Journal of the Brazilian Computer Society**, v. 1, n. 8, p. 57-72, 2002.

[33] GODIL, I.; JACOBSEN H. **Horizontal Decomposition of Prevayler**. In: Conference of the Centre for Advanced Studies on Collaborative Research. Proceedings… Canada, 2005.

[34] HANENBERG, S.; SCHMIDMEIER, A. **AspectJ Idioms for Aspect-Oriented Software Construction**. In: European Conference on Pattern Languages of Programs (EuroPLoP'03). Proceedings… Germany, 2003.

[35] HANNEMANN, J.; MURPHY, G.; KICZALES, G. **Role-Based Refactoring of Crosscutting Concerns**. In: ACM International Conference on Aspect-Oriented Software Development (AOSD'05). Proceedings… USA, 2005, p. 135-146.

[36] HANNEMANN, J.; KICZALES, G. **Design Pattern Implementation in Java and AspectJ**. In: ACM Conference on Object-Oriented

Programming Systems, Languages, and Applications (OOPSLA'02). Proceedings… USA, 2002, p. 161-173.

[37] HENDERSON-SELLERS, B. **Object-Oriented Metrics: Measures of Complexity**. Prentice Hall, 1996. 234p.

[38] IWAMOTO, M.; ZHAO, J. **Refactoring Aspect-Oriented Programs**. In: 4th AOSD Modeling with UML Workshop (UML'03). Proceedings… USA, 2003.

[39] JANZEN, D.; VOLDER, K. **Navigating and Querying Code without Getting Lost**. In: International Conference on Aspect-Oriented Software Development (AOSD'03). Proceedings… USA, 2003, p. 178-187.

[40] KAN, S. H. **Metrics and Models in Software Quality Engineering**. 2nd ed. Pearson Education, 2002. 560 p.

[41] KERSTEN, A.; MURPHY, G. **Atlas: A Case Study in Building a Web-based learning environment using aspect-oriented programming**. In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99). Proceedings… USA, 1999.

[42] KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. Getting Started with AspectJ. **Communication of the ACM**, v. 44, n. 10, p. 59-65, 2001.

[43] KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J.; IRWIN, J. **Aspect-Oriented Programming**. In: European Conference on Object-Oriented Programming (ECOOP'97). Proceedings… Finland, 1997, p. 220-242.

[44] KULESZA, U.; SANT'ANNA, C.; GARCIA, A.; LUCENA, C.; STAA, A. Aspectization of Distribution and Persistence: Quantifying the Effects of AOP. **IEEE Software, Special Issue on AOP**, 2005. (submitted)

[45] LADDAD, R. **AspectJ in Action: Practical Aspect-Oriented Programming**. Manning Publications, 2003. 512p.

[46] LOPES, C. **D: A Language Framework for Distributed Programming**. PhD Thesis, College of Computer Science – Northeastern University, 1997, 295p.

[47] LORENZ, M.; KIDD J. **Object-Oriented Software Metrics, a Practical Guide**. Englewood Cliffs, N.J.: PTR Prentice-Hall, 1994.

[48] MARUYAMA, H.; TAMURA, K.; URAMOTO N. **XML and Java: Developing Web Applications**. Addison Wesley Publishing Company, 1999. 386p.

[49] MARCHAL, B. **XML by Example**. Que, 1999. 425p.

[50] MCCALL, J. A.; RICHARDS, P. K.; WALTERS, G. F. **Factors in Software Quality**. RADC TR-77-369, 1977. vols I, II, III, US Rome Air Development Center Reports NTIS AD/A-049 014, 015, 055, 1977.

[51] MEZINI, M.; OSTERMANN, K. **Conquering Aspects with Caesar**. In: ACM International Conference on Aspect-Oriented Software Development (AOSD'05). Proceedings…USA, 2005, p. 90-99.

[52] MONTEIRO, M.; FERNANDES, J. **Towards a Catalog of Aspect-Oriented Refactorings**. In: ACM International Conference on Aspect-Oriented Software Development (AOSD'05). Proceedings…USA, 2005, p. 111-122.

[53] **MuLATo: A Multi-Language Assessment Tool (SourceForge.net)**. Disponível em: <http://sourceforge.net/projects/mulato>. Acesso em: 17 fev. 2006.

[54] OLIVEIRA, A.; BRAGA, T.; MAIA, M.; BIGONHA, R. MetaJ: An Extensible Environment for Metaprogramming in Java. **Journal of Universal Computer Science**, v. 10, n. 7, p. 872-891, 2004.

[55] OLIVEIRA, A. **MetaJ: Um Ambiente para Meta-Programação em Java**. Dissertação de Mestrado, Departamento de Computação - UFMG. Belo Horizonte, 2004, 174p.

[56] ROBILLARD, M.; MURPHY, G. **Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies**. In: International Conference on Software Engineering (ICSE'02). Proceedings… USA, 2002, p. 406-416.

[57] SANT'ANNA, C.; GARCIA, A.; CHAVEZ, C.; LUCENA, C.; STAA, A. **On the Reuse and Maintenance of Aspect-Oriented Software: An Evaluation Framework**. In: XVII Brazilian Symposium on Software Engineering. Proceedings… Manaus, 2003, p. 19-34.

[58] SHEPHERD, D.; POLLOCK, L. **Ophir: A Framework for Automatic Mining and Refactoring of Aspects**. Technical Report, no.2004-03, Department of Computer and Information Sciences - University of Delaware. Newark, DE, 2003, 7p.

[59] SOARES, S. C. B. **An Aspect-Oriented Implementation Method**. Tese de Doutorado - Universidade Federal de Pernambuco. Recife, 2004. 166p.

[60] SOARES, S.; LAUREANO, E.; BORBA, P. **Implementing Distribution and Persistence Aspects with AspectJ**. In: 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'2002). Proceedings… USA, 2002, p. 174-190.

[61] SOMMERVILLE, I. **Software Engineering**. 6.ed. Harlow, England, Addison-Wesley, 2001, 693p.

[62] STAA, A. v. **Programação modular**. Rio de Janeiro: Campus, 2000, 720 p.

[63] SUTTON JR, S.; ROUVELLOU, I. **Concern Modeling for Aspect-Oriented Software Development.** In: Aspect Oriented Software Development, Addison-Wesley, p 479-505, 2004.

[64] TARR, P.; OSSHER, H.; HARRISON, W.; SUTTON, S. **N Degrees of Separation: Multi-Dimensional Separation of Concerns**. In: 21st International Conference on Software Engineering (ICSE'99). Proceedings… USA, 1999, p. 107-119.

[65] TEKINERDOGAN, B. **ASAAM: Aspectual Software Architecture Analysis Method**. In: IEEE/IFIP Conference on Software Architecture. Proceedings… Norway, 2004, p. 5-14.

[66] The AspecJ Team. The AspectJ™ Programming Guide. Disponível em: <http:/www.eclipse.org/aspectj> Acesso em: 17 fev. 2006.

[67] The Aspect-Oriented Design Pattern Implementation Homepage. Disponível em: <http://www.cs.ubc.ca/~jan/AODPs/> Acesso em: 15 nov. 2005.

[68] WASP'04 - Primeiro Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos. Disponível em: <http://twiki.im.ufba.br/bin/view/WAsp/Termos> Acesso em: 14 mar. 2006.

[69] WONG, E.; GOKHALE, S.; HORGAN, J. **Metrics for Quantifying the Disparity, Concentration, and Dedication between Program Components and Features**. In: 6th IEEE International Symposium on Software Metrics. Proceedings… USA, 1999, p. 189-197.

[70] ZACARIA, A.; HOSNY, H. **Metrics for Aspect-Oriented Software Design**. In: 3rd International Workshop on Aspect-Oriented Modeling. Proceedings… USA, 2003.

[71] ZHAO, J. **Towards a Metrics Suite for Aspect-Oriented Software**. Technical-Report SE-136-25, Information Processing Society of Japan (IPSJ), 2002, 8p.

Apêndice A
Resultados Estudo Experimental dos Padrões

A. Padrão *Observer* Orientado a Objetos

Atividade de Medição

Tabela 12 – Resultados de SI e tamanho para o papel *Subject*

Componentes	***Subject* (Padrão *Observer*)**					
	CDC = 8		**CDLOC = 58**		**VS = 13**	
	NOA	**NOAc**	**NOO**	**NOOc**	**LOC**	**LOCc**
Main	0	0	1	0	54	12
Observer	0	0	1	0	4	0
Point	4	1	10	3	42	17
Point1	4	1	10	3	42	17
Point2	4	1	10	3	42	17
Point3	4	1	10	3	42	17
Point4	4	1	10	3	42	17
Screen	2	1	6	3	29	14
Screen1	1	0	3	0	15	0
Screen2	1	0	3	0	15	0
Screen3	1	0	3	0	15	0
Screen4	1	0	3	0	15	0
Subject	0	0	3	3	6	6

Tabela 13 – Resultados de SI e tamanho para o papel *Observer*

Componentes	***Observer* (Padrão *Observer*)**					
	CDC = 6		**CDLOC = 22**		**VS = 13**	
	NOA	**NOAc**	**NOO**	**NOOc**	**LOC**	**LOCc**
Main	0	0	1	0	54	0
Observer	0	0	1	1	4	4
Point	4	0	10	0	42	0
Point1	4	0	10	0	42	0
Point2	4	0	10	0	42	0
Point3	4	0	10	0	42	0
Point4	4	0	10	0	42	0
Screen	2	0	6	1	29	4
Screen1	1	0	3	1	15	4
Screen2	1	0	3	1	15	4
Screen3	1	0	3	1	15	4
Screen4	1	0	3	1	15	4
Subject	0	0	3	0	6	0

Tabela 14 – Resultado de acoplamento e coesão para o padrão *Observer*

Componentes	**Observer**			
	LCOO	**CBC**	**DIT**	**NOC**
Main		12	1	0
Observer		1	1	5
Point	15	4	2	0
Point1	15	4	2	0
Point2	15	4	2	0
Point3	15	4	2	0
Point4	15	4	2	0
Screen	1	3	2	0
Screen1	1	2	2	0
Screen2	1	2	2	0
Screen3	1	2	2	0
Screen4	1	2	2	0
Subject		1	1	6
Soma	**80**	**45**	**23**	**11**
Média	**6,15**	**3,46**	**1,77**	**0,85**

Aplicação das Regras

Tabela 15 – Regras de SI para o papel *Subject* do padrão *Observer*

Regras	**Valor Retornado**	**Classificações**
R01	FALSO	
R02	VERDADEIRO	Entrelaçado
R03	VERDADEIRO	Elevado Espalhamento
R04	FALSO	
R05	FALSO	
R06	VERDADEIRO	Possivelmente Secundário
R07	Não se Aplica	
R08	Não se Aplica	
R09	Não se Aplica	
R10	VERDADEIRO	Secundário

Tabela 16 – Regras de SI para o papel *Observer* do padrão *Observer*

Regras	**Valor Retornado**	**Classificações**
R01	FALSO	
R02	VERDADEIRO	Entrelaçado
R03	VERDADEIRO	Elevado Espalhamento
R04	FALSO	
R05	FALSO	
R06	VERDADEIRO	Possivelmente Secundário
R07	Não se Aplica	
R08	Não se Aplica	
R09	Não se Aplica	
R10	VERDADEIRO	Secundário

Tabela 17 – Regras de Acoplamento e Coesão para o padrão *Observer*

Componentes	Interesses Secundários	Classificações
Main	Subject	Elevado Acoplamento
Point	Subject	Baixa Coesão
Point1	Subject	Baixa Coesão
Point2	Subject	Baixa Coesão
Point3	Subject	Baixa Coesão
Point4	Subject	Baixa Coesão
Screen	Observer	Extração de Interesses
Screen1	Observer	Extração de Interesses
Screen2	Observer	Extração de Interesses
Screen3	Observer	Extração de Interesses
Screen4	Observer	Extração de Interesses

Análise e Identificação de Problemas

As classes que possuem o interesse do papel *Observer* como secundário (i.e., as classes `ScreenX`) não apresentam problemas com as regras acoplamento e coesão. Em especial, não possuem perda de coesão e, portanto, apesar do papel *Observer* ser secundário, este não causa um problema muito grave ao sistema. Por outro lado, as classes que possuem o papel Subject como secundário (i.e., as classes `PointX`) apresentam problemas com as regras de acoplamento e coesão além do interesse secundário. Em especial, estas classes possuem perda de coesão, a excessão é a classe Main que possui elevado acoplamento. Podemos verificar que o papel *Subject* é realmente mais problemático para as classes que o papel *Observer*. Portanto, as regras foram eficazes ao geram alertas mais fortes (Baixa Coesão ou Elevado Acoplamento) para os componentes que possuem este papel *Subject*.

B. Padrão *Factory Method* Orientado a Objetos

Atividade de Medição

Tabela 18 – Resultados de SI e tamanho para o papel *Creator*

Componentes	***Creator* (Padrão *Factory Method*)**					
	CDC = 8		**CDLOC = 4**		**VS = 8**	
	NOA	**NOAc**	**NOO**	**NOOc**	**LOC**	**LOCc**
ButtonCreator	0	0	2	2	19	19
ButtonCreator1	0	0	2	2	19	19
ButtonCreator2	0	0	2	2	19	19
GUIComponentCreator	1	0	3	2	25	3
LabelCreator	0	0	2	2	12	12
LabelCreator1	0	0	2	2	12	12
LabelCreator2	0	0	2	2	12	12
Main	0	0	1	1	17	12

Tabela 19 – Resultado de acoplamento e coesão para o *Factory Method*

Componentes		**Factory Method**			
		LCOO	**CBC**	**DIT**	**NOC**
ButtonCreator			3	2	0
ButtonCreator1			3	2	0
ButtonCreator2			3	2	0
GUIComponentCreator		3	6	1	6
LabelCreator			2	2	0
LabelCreator1			2	2	0
LabelCreator2			2	2	0
Main			1	1	0
	Soma	**3**	**22**	**14**	**6**
	Média	**0,38**	**2,75**	**1,75**	**0,75**

Aplicação das Regras

Tabela 20 – Regras de SI para o papel *Creator* do padrão *Factory Method*

Regras	Valor Retornado	Estado Classificado
R01	FALSO	
R02	VERDADEIRO	Entrelaçado
R03	VERDADEIRO	Elevado Espalhamento
R04	FALSO	
R05	FALSO	
R06	FALSO	
R07	Não se Aplica	
R08	Não se Aplica	
R09	Não se Aplica	
R10	Não se Aplica	

As regras de acoplamento e coesão não se aplicam ao padrão *Factory Method* porque as regras de SI não classificaram o interesse do papel *Creator* como sencundário.

Análise e Identificação de Problemas

O interesse do papel *Creator* não é secundário, portanto, não gera alerta (problema de SI) nesta avaliação. Note que, este interesse é primário na prática. `GUIComponentCreator` foi o único componente no qual as regras não indicaram o papel *Creator* como primário, entretanto, elas também não indicaram este interesse como secundário nesta classe.

C. Padrão *Builder* Orientado a Objetos

Atividade de Medição

Tabela 21 – Resultados de SI e tamanho para o papel *Director*

Componentes	***Director* (Padrão *Builder*)**					
	CDC = 1		**CDLOC = 4**		**VS = 8**	
	NOA	**NOAc**	**NOO**	**NOOc**	**LOC**	**LOCc**
Creator	1	0	5	0	10	0
Main	0	0	2	1	32	21
TextCreator	0	0	3	0	12	0
TextCreator1	0	0	3	0	12	0
TextCreator2	0	0	3	0	12	0
XMLCreator	2	0	4	0	30	0
XMLCreator1	2	0	4	0	30	0
XMLCreator2	2	0	4	0	30	0

Tabela 22 – Resultado de acoplamento e coesão para o padrão *Builder*

Componentes		**Builder**			
		LCOO	**CBC**	**DIT**	**NOC**
Creator		6	0	1	6
Main			2	1	0
TextCreator			0	2	0
TextCreator1			0	2	0
TextCreator2			0	2	0
XMLCreator		2	0	2	0
XMLCreator1		2	0	2	0
XMLCreator2		2	0	2	0
	Soma	**12**	**2**	**14**	**6**
	Média	**1,50**	**0,25**	**1,75**	**0,75**

Aplicação das Regras

Tabela 23 – Regras de SI para o papel *Director* do padrão *Builder*

Regras	**Valor Retornado**	**Estado Classificado**
R01	FALSO	
R02	VERDADEIRO	Entrelaçado
R03	FALSO	
R04	VERDADEIRO	Baixo Espalhamento
R05	Não se Aplica	
R06	Não se Aplica	
R07	VERDADEIRO	Possivelmente Primário
R08	FALSO	
R09	VERDADEIRO	Primário
R10	Não se Aplica	

As regras de acoplamento e coesão não se aplicam ao padrão *Builder* porque as regras de SI não classificaram o interesse do papel *Director* como sencundário.

Análise e Identificação de Problemas

O interesse do papel *Director* é primário na classe `Main`. O método de avaliação assume que interesses primários não causam problemas de acoplamento e coesão e, portanto, as tais regras não são utilizadas.

Apêndice B
Resultados do Estudo Experimental Middleware OpenOrb

A. Composição *Observer* com *Factory Method* Orientado a Objetos

Atividade de Medição

Tabela 24 – Resultados de SI e tamanho para o padrão *Observer*

Componentes	***Observer* (Composição com *Factory Method*)**					
	CDC = 7		**CDLOC = 20**		**VS = 86**	
	NOA	**NOAc**	**NOO**	**NOOc**	**LOC**	**LOCc**
Component	2	1	9	3	39	14
ConcreteBind	5	1	11	3	55	15
MetaObject	1	0	2	1	9	2
MetaObjectComposite	1	0	3	1	24	4
MetaObjectEncapsu	2	0	5	1	37	3
MetaObjectFactoryComposite	0	0	1	0	17	0
MetaObjectFactoryEncapsule	0	0	1	0	15	0
MetaObjFactory	0	0	1	0	4	0
MetaObserver	0	0	1	1	4	4
MetaSubject	0	0	3	3	6	6

Tabela 25 – Resultados de SI e tamanho para o padrão *Factory Method*

Componentes	***Factory Method* (Composição com *Observer*)**					
	CDC = 6		**CDLOC = 6**		**VS = 86**	
	NOA	**NOAc**	**NOO**	**NOOc**	**LOC**	**LOCc**
Component	2	0	9	0	39	0
ConcreteBind	5	0	11	0	55	0
MetaObject	1	1	2	1	7	5
MetaObjectComposite	1	1	3	2	24	14
MetaObjectEncapsu	2	2	5	4	37	33
MetaObjectFactoryComposite	0	0	1	1	17	17
MetaObjectFactoryEncapsule	0	0	1	1	15	15
MetaObjFactory	0	0	1	1	4	4
MetaObserver	0	0	1	0	4	0
MetaSubject	0	0	3	0	6	0

Tabela 26 – Resultado de acoplamento e coesão para *Observer* com *Factory Method*

Componentes	***Observer* com *Factory Method***			
	LCOO	**CBC**	**DIT**	**NOC**
ConcreteBind	31	6	1	2
MetaObject	1	1	1	2
MetaObjFactory		1	1	2
MetaSubject		0	1	2
MetaObserver		0	1	1
Component	0	6	1	0
MetaObjectComposite	0	0	2	0
MetaObjectEncapsu	6	2	2	0
MetaObjectFactoryComposite		5	1	0
MetaObjectFactoryEncapsule		6	1	0
Soma	**38**	**27**	**12**	**9**
Média	**3,80**	**2,70**	**1,20**	**0,90**

Aplicação das Regras

Tabela 27 – Regras de SI para o padrão *Observer*

Regras	**Valor Retornado**	**Estado Classificado**
R01	FALSO	
R02	VERDADEIRO	Entrelaçado
R03	FALSO	
R04	VERDADEIRO	Baixo Espalhamento
R05	Não se Aplica	
R06	Não se Aplica	
R07	FALSO	
R08	VERDADEIRO	Possivelmente Secundário
R09	Não se Aplica	
R10	VERDADEIRO	Secundário

Tabela 28 – Regras de SI para o padrão *Factory Method*

Regras	**Valor Retornado**	**Estado Classificado**
R01	FALSO	
R02	VERDADEIRO	Entrelaçado
R03	FALSO	
R04	VERDADEIRO	Baixo Espalhamento
R05	Não se Aplica	
R06	Não se Aplica	
R07	VERDADEIRO	Possivelmente Primário
R08	FALSO	
R09	VERDADEIRO	Primário
R10	Não se Aplica	

Tabela 29 – Regras de acoplamento e coesão para composição *Observer* com *Factory Method*

Componentes	Interesses Secundários	Classificações
Component	Observer	Elevado Acoplamento
ConcreteBind	Observer	Reestruturação Global
MetaObject	Observer	Extração de Interesses
MetaObjectComposite	Observer	Extração de Interesses
MetaObjectEncapsu	Observer	Baixa Coesão

Análise e Identificação de Problemas

O interesse do padrão *Observer* é secundário e três classes (de cinco) que possuem este interesse como secundário também apresentam problemas em atribudo como acoplamento e coesão. Alem disso, é importante destacar que a classe de maior problema, `ConcreteBind`, desempenha o papel de *Subject* no padrão. Esta observação confirma o que é dito no Apêndice A de que o interesse do papel *Subject*, além de ser secundário, causa perda de coesão nos componentes que o implementam.

B. Composição *Singleton* com *Façade* Orientado a Objetos

Atividade de Medição

Tabela 30 – Resultados de SI e tamanho para o padrão *Singleton*

Componentes	***Singleton* (Composição com *Façade*)**					
	CDC = 2		**CDLOC = 6**		**VS = 86**	
	NOA	**NOAc**	**NOO**	**NOOc**	**LOC**	**LOCc**
CapsuleImpl	4	1	8	1	72	9
OpenOrb	6	1	6	0	56	2

Tabela 31 – Resultados de SI e tamanho para o padrão *Façade*

Componentes	***Façade* (Composição com *Singleton*)**					
	CDC = 1		**CDLOC = 2**		**VS = 86**	
	NOA	**NOAc**	**NOO**	**NOOc**	**LOC**	**LOCc**
CapsuleImpl	4	0	8	0	72	0
OpenOrb	6	6	6	6	56	56

Tabela 32 – Resultado de acoplamento e coesão para *Singleton* com *Façade*

Componentes		***Singleton* com *Façade***			
		LCOO	**CBC**	**DIT**	**NOC**
CapsuleImpl		6	11	1	0
OpenOrb		13	12	1	0
	Soma	**19**	**23**	**2**	**0**
	Média	**9,50**	**11,50**	**1,00**	**0,00**

Aplicação das Regras

Tabela 33 – Regras de SI para o padrão *Singleton* na composição com *Façade*

Regras	**Valor Retornado**	**Estado Classificado**
R01	FALSO	
R02	VERDADEIRO	Entrelaçado
R03	FALSO	
R04	VERDADEIRO	Baixo Espalhamento
R05	Não se Aplica	
R06	Não se Aplica	
R07	FALSO	
R08	VERDADEIRO	Possivelmente Secundário
R09	Não se Aplica	
R10	VERDADEIRO	Secundário

Tabela 34 – Regras de SI para o padrão *Façade* na composição com *Singleton*

Regras	Valor Retornado	Estado Classificado
R01	VERDADEIRO	Modularizado
R02	FALSO	
R03	Não se Aplica	
R04	Não se Aplica	
R05	Não se Aplica	
R06	Não se Aplica	
R07	Não se Aplica	
R08	Não se Aplica	
R09	Não se Aplica	
R10	Não se Aplica	

Tabela 35 – Regras de acoplamento e coesão para *Singleton* com *Façade*

Componentes	Interesses Secundários	Classificações
CapsuleImpl	Singleton	Extração de Interesses
OpenOrb	Singleton	Baixa Coesão

Análise e Identificação de Problemas

O interesse do parão *Singleton* é secundário, entretanto, umas das classes que possuem este interesse (CapsuleImpl) não apresenta problema de acoplamento ou coesão. Portanto, o alerta é mais fornte na classe que possui perda de coesão além do interesse secundário: `OpenOrb`. Em relação ao padrão *Façade*, apesar deste interesse estar modularizado na classe `OpenOrb`, esta classe apresenta perda de coesão. Isto pode ser atribuído a dois fatores: (i) o padrão *Façade* não garante coesão, pois junta vários métodos com diferentes funções em um mesmo componente e (ii) o padrão *Singleton* (que é secundário) também é implementado por esta classe.

C. Composição *Proxy* com *Interpreter* Orientado a Objetos

Atividade de Medição

Tabela 36 – Resultados de SI e tamanho para o padrão *Proxy*

Componentes	***Proxy* (Composição com *Interpreter*)**					
	CDC = 3		**CDLOC = 4**		**VS = 118**	
	NOA	**NOAc**	**NOO**	**NOOc**	**LOC**	**LOCc**
Extractor	0	0	2	0	4	0
AspectDeclarationExtractor	1	0	2	0	19	0
DiretoryExtractor	2	0	9	0	61	0
ImportsExtractor	2	0	4	0	21	0
JavaFileExtractor	3	0	12	0	67	0
InterfaceDeclarationExtractor	1	0	2	0	15	0
ClassDeclarationExtractor	1	0	2	0	17	0
PackageExtractor	2	0	5	0	16	0
TypeExtractor	3	0	6	0	38	0
SourceCodeParser	0	0	2	0	17	0
SystemModel	0	0	17	17	19	19
SystemModelImpl	2	0	16	0	57	1
SystemModelProxy	4	3	20	18	85	77

Tabela 37 – Resultados de SI e tamanho para o padrão *Interpreter*

Componentes	***Interpreter* (Composição com *Proxy*)**					
	CDC = 9		**CDLOC = 16**		**VS = 86**	
	NOA	**NOAc**	**NOO**	**NOOc**	**LOC**	**LOCc**
AspectDeclarationExtractor	1	0	2	1	19	12
ClassDeclarationExtractor	1	0	2	1	17	10
DiretoryExtractor	2	0	9	3	61	38
Extractor	0	0	2	2	4	4
ImportsExtractor	2	0	4	2	21	11
InterfaceDeclarationExtractor	1	0	2	1	15	8
JavaFileExtractor	3	0	12	5	67	40
PackageExtractor	2	0	5	2	16	9
SourceCodeParser	0	0	2	0	17	0
SystemModel	0	0	17	0	19	0
SystemModelImpl	2	0	16	0	57	0
SystemModelProxy	4	0	20	0	85	0
TypeExtractor	3	1	6	2	38	21

Tabela 38 – Resultado de acoplamento e coesão para *Proxy* com *Interpreter*

Componentes	***Proxy* com *Interpreter***			
	LCOO	**CBC**	**DIT**	**NOC**
AspectDeclarationExtractor	0	5	2	0
ClassDeclarationExtractor	0	4	2	0
DiretoryExtractor	23	9	2	1
Extractor		1	1	4
ImportsExtractor	0	3	1	0
InterfaceDeclarationExtractor	0	3	2	0
JavaFileExtractor	42	9	1	0
PackageExtractor	2	2	1	0
SourceCodeParser		5	3	0
SystemModel		4	1	2
SystemModelImpl	84	4	2	0
SystemModelProxy	0	4	1	0
TypeExtractor	5	6	1	0
Soma	**156**	**59**	**20**	**7**
Média	**12,00**	**4,54**	**1,54**	**0,54**

Aplicação das Regras

Tabela 39 – Regras de SI para o padrão *Proxy* na composição com *Interpreter*

Regras	**Valor Retornado**	**Estado Classificado**
R01	FALSO	
R02	VERDADEIRO	Entrelaçado
R03	FALSO	
R04	VERDADEIRO	Baixo Espalhamento
R05	Não se Aplica	
R06	Não se Aplica	
R07	FALSO	
R08	VERDADEIRO	Possivelmente Secundário
R09	Não se Aplica	
R10	VERDADEIRO	Secundário

Tabela 40 – Regras de SI para o padrão *Interpreter* na composição com *Proxy*

Regras	**Valor Retornado**	**Estado Classificado**
R01	FALSO	
R02	VERDADEIRO	Entrelaçado
R03	FALSO	
R04	VERDADEIRO	Baixo Espalhamento
R05	Não se Aplica	
R06	Não se Aplica	
R07	FALSO	
R08	VERDADEIRO	Possivelmente Secundário
R09	Não se Aplica	
R10	FALSO	

Tabela 41 – Regras de acoplamento e coesão para *Proxy* com *Interpreter*

Componentes	Interesses Secundários*	Classificações
DiretoryExtractor	Interpreter	Reestruturação Global
JavaFileExtractor	Interpreter	Reestruturação Global
PackageExtractor	Interpreter	Extração de Interesses
TypeExtractor	Interpreter	Elevado Acoplamento
SystemModelImpl	Proxy	Baixa Coesão

* É considerado também o interesse possivelmente secundário *Interpreter*.

Análise e Identificação de Problemas

O interesse do padrão *Interpreter* é possivelmente secundário. Das quatro classes que possuem este interesse como secundário, duas apresentam problemas de acoplamento e coesão (Reestruturação Global) e uma apresenta elevado acoplamento. Em relação ao interesse secundário do padrão *Proxy*, a classe que o implementa, `SystemModelImpl`, possui perda de coesão.

D. Composição *Prototype* com *State* Orientado a Objetos

Atividade de Medição

Tabela 42 – Resultados de SI e tamanho para o padrão *State*

Componentes	***State* (Composição com *Prototype*)**					
	CDC = 3		**CDLOC = 4**		**VS = 86**	
	NOA	**NOAc**	**NOO**	**NOOc**	**LOC**	**LOCc**
BindState	0	0	2	2	4	4
BindConnected	2	1	4	2	20	8
BindRunning	0	0	3	2	13	5
ConcreteBind	5	0	11	0	56	0

Tabela 43 – Resultado de acoplamento e coesão para *Prototype* com *State*

Componentes		***Prototype* com *State***			
		LCOO	**CBC**	**DIT**	**NOC**
BindState		0	3	1	0
BindConnected			1	1	0
BindRunning			1	1	2
ConcreteBind		31	6	1	2
	Soma	**31**	**11**	**4**	**4**
	Média	**7,75**	**2,75**	**1,00**	**1,00**

Aplicação das Regras

Tabela 44 – Regras de SI para o padrão *State* na composição com *Prototype*

Regras	**Valor Retornado**	**Estado Classificado**
R01	FALSO	
R02	VERDADEIRO	Entrelaçado
R03	FALSO	
R04	VERDADEIRO	Baixo Espalhamento
R05	Não se Aplica	
R06	Não se Aplica	
R07	VERDADEIRO	Possivelmente Primário
R08	FALSO	
R09	FALSO	
R10	Não se Aplica	

As regras de acoplamento e coesão não se aplicam à composição *Prototype* com *State* porque as regras de SI não classificaram o interesse do padrão *State* como sencundário.

Análise e Identificação de Problemas

O interesse do padrão *State* é possivelmente primário. Como o método de avaliação proposto nesta dissertação assume que interesses primários ou possivelmente primários não causam problemas de acoplamento e coesão, estas regras não são aplicadas.

Apêndice C
Resultados do Estudo Experimental Portalware

Atividade de Medição

Tabela 45 – Resultados de SI e tamanho para o interesse Adaptação

Componentes	Adaptação					
	CDC = 3		CDLOC = 14		VS = 60	
	NOA	NOAc	NOO	NOOc	LOC	LOCc
PAgent	9	1	17	0	118	6
Adaptation	0	0	4	4	72	72
Property	1	1	1	1	7	7

Tabela 46 – Resultados de SI e tamanho para o interesse Colaboração

Componentes	Colaboração					
	CDC = 15		CDLOC = 14		VS = 60	
	NOA	NOAc	NOO	NOOc	LOC	LOCc
SearchingPlan	0	0	2	0	22	2
SearchResultReceivingPlan	0	0	2	0	17	3
AvailabilityPlan	0	0	2	0	16	3
SearchAskAnsweringPlan	0	0	2	0	15	3
ContentDistributionPlan	0	0	2	0	15	3
ResponseReceivingPlan	0	0	2	0	14	4
Collaboration	0	0	7	7	13	13
CollaboratorCore	2	2	11	11	49	49
CollaboratorRole	1	1	8	8	36	36
CollaborativeAgent	1	1	6	6	46	46
SharedObject	4	4	6	6	44	44
Answerer	1	1	4	4	36	36
Caller	1	1	3	3	35	35
ContentSupplier	1	1	2	2	16	16
Editor	0	0	4	4	24	24

Tabela 47 – Resultados de SI e tamanho para o interesse Autonomia

Componentes	Autonomia					
	CDC = 3		CDLOC = 16		VS = 60	
	NOA	NOAc	NOO	NOOc	LOC	LOCc
PAgent	9	1	17	0	118	7
Autonomy	1	1	6	6	38	38
Property	1	1	1	1	7	7

Tabela 48 – Resultado de acoplamento e coesão para o *Portalware*

Componentes	***Portalware***			
	LCOO	**CBC**	**DIT**	**NOC**
Adaptation		15	2	0
Answerer	4	7	5	0
App		19	1	0
Autonomy	15	9	2	0
AvailabilityPlan		8	3	0
Belief	13	0	1	3
BeliefAgentList	0	2	2	0
BeliefMyRole	0	0	2	0
Caller	3	7	5	5
Collaboration		1	3	2
CollaborationPlan		2	2	0
CollaborativeAgent	0	5	2	4
CollaboratorCore	43	5	4	0
CollaboratorRole	0	1	4	0
CompositeBelief	0	1	2	0
CompositeGoal	0	1	2	0
ContentDistributionPlan		8	3	0
ContentProposal	0	0	2	0
ContentSupplier	1	4	5	0
CoordinatorAgent		3	2	0
CProposal		0	2	0
CProposalMsg	0	1	3	2
DecisionPlan	1	3	2	1
EditionGoal	1	0	2	0
EditionWorkDistributionPlan		6	3	0
Editor		7	5	0
Effector	0	2	1	0
Environment	3	6	1	0
EnvironmentThread	34	2	3	0
Goal	15	1	1	7
GoalMsg	0	1	2	0
InformationAgent	1	3	3	0
InformationExchangeGoal	0	0	2	0
Interaction	36	7	2	1
MainThread	43	5	3	0
MakeDecisionGoal		0	2	0
Message	0	0	1	5
NegotiationMsg		0	2	2
NewAgentNotification	0	1	2	0
Notification		0	1	1
NotificationMsg	0	1	2	0
PAgent	50	12	1	2
Plan	35	5	1	3
Property		1	1	3
Proposal		0	1	2
ProposalMsg	2	1	3	0
ReactionPlan		2	2	2
ResourceMsg	0	0	2	0
ResponseCheckingGoal		0	2	0
ResponseMsg	2	1	2	0
ResponseReceivingPlan		8	3	0
SearchAskAnsweringPlan		6	3	0
SearchingGoal		0	2	0
SearchingPlan		9	3	0

SearchResultReceivingGoal		1	0	2	0
SearchResultReceivingPlan			9	3	0
SearchSendPlan			8	3	0
Sensor		0	5	1	0
SharedObject		0	1	1	0
UserAgent		1	3	3	0
	Soma	**304**	**215**	**138**	**45**
	Média	**5,07**	**3,58**	**2,30**	**0,75**

Aplicação das Regras

Tabela 49 – Regras de SI para o interesse Adaptação

Regras	Valor Retornado	Estado Classificado
R01	FALSO	
R02	VERDADEIRO	Entrelaçado
R03	FALSO	
R04	VERDADEIRO	Baixo Espalhamento
R05	Não se Aplica	
R06	Não se Aplica	
R07	FALSO	
R08	VERDADEIRO	Possivelmente Secundário
R09	Não se Aplica	
R10	VERDADEIRO	Secundário

Tabela 50 – Regras de SI para o interesse Colaboração

Regras	Valor Retornado	Estado Classificado
R01	FALSO	
R02	VERDADEIRO	Entrelaçado
R03	VERDADEIRO	Elevado Espalhamento
R04	FALSO	
R05	FALSO	
R06	VERDADEIRO	Possivelmente Secundário
R07	Não se Aplica	
R08	Não se Aplica	
R09	Não se Aplica	
R10	VERDADEIRO	Secundário

Tabela 51 – Regras de SI para o interesse Autonomia

Regras	Valor Retornado	Estado Classificado
R01	FALSO	
R02	VERDADEIRO	Entrelaçado
R03	FALSO	
R04	VERDADEIRO	Baixo Espalhamento
R05	Não se Aplica	
R06	Não se Aplica	
R07	FALSO	
R08	VERDADEIRO	Possivelmente Secundário
R09	Não se Aplica	
R10	VERDADEIRO	Secundário

Tabela 52 – Regras de acoplamento e coesão para o *Portalware*

Componentes	Interesses Secundários	Classificações
SearchingPlan	Colaboração	Elevado Acoplamento
SearchResultReceivingPlan	Colaboração	Elevado Acoplamento
AvailabilityPlan	Colaboração	Elevado Acoplamento
SearchAskAnsweringPlan	Colaboração	Elevado Acoplamento
ContentDistributionPlan	Colaboração	Elevado Acoplamento
ResponseReceivingPlan	Colaboração	Elevado Acoplamento
PAgent	Adaptação, Autonomia	Reestruturação Global

Análise e Identificação de Problemas

Os interesses Adaptação e Autonomia são secundários, mas com baixo espalhamento. Apenas três componentes são afetados por cada um deles. Estes dois interesses são secundários na classe `PAgent` e esta classe também apresentam problemas nos atributos de acoplamento e coesão. Desta forma, o alerta em relação à classe `PAgent` é muito grave por três motivos: (i) esta classe possui o interesse Adaptação como secundário; (ii) esta classe possui o interesse Autonomia como secundário; e (iii) e o componente apresenta a grande perda de coesão e elevado acoplamento.

Diferentemente dos dois anteriores, o interesse Colaboração possui elevado espalhamento além de ser secundário. Em adição, todos os seis componentes em que o interesse é secundário possuem elevado acoplamento.

Apêndice D
Resultados do Estudo Experimental Health Watcher

Atividade de Medição

Tabela 53 – Resultados de SI e tamanho para o interesse Concorrência

Componentes	Concorrência					
	CDC = 16		CDLOC = 86		VS = 89	
	NOA	NOAc	NOO	NOOc	LOC	LOCc
DSRMISourceAdapter	3	0	16	0	203	1
DSRMITargetAdapter	1	0	14	0	185	1
HealthWatcherFacade	3	0	16	0	106	1
IDSRMITargetAdapter	0	0	13	0	102	1
IFachada	0	0	13	0	86	1
CitizenFacade	6	0	12	0	394	8
ComplaintRecord	1	0	5	0	45	1
ServletUpdateComplaintData	1	0	2	0	77	2
ComplaintRepositoryRDBMS	4	0	16	0	486	24
Complaint	12	1	26	2	118	7
IComplaintRepository	0	0	4	0	16	1
ComplaintRepositoryArray	4	0	10	0	78	8
EmployeeRepositoryArray	3	0	10	0	75	8
EmployeeRecord	2	1	4	0	36	4
TimestampException	0	0	1	1	5	5
ConcurrencyManager	1	1	3	3	36	36

Tabela 54 – Resultados de SI e tamanho para o interesse Distribuição

Componentes	Distribuição					
	CDC = 36		CDLOC = 78		VS = 89	
	NOA	NOAc	NOO	NOOc	LOC	LOCc
Date	4	0	21	0	408	2
ServletSearchComplaintData	1	0	2	0	216	2
ServletInsertFoodComplaint	1	0	2	0	126	2
ServletInsertAnimalComplaint	1	0	2	0	120	2
Complaint	12	0	26	0	118	2
ServletInsertSpecialComplaint	1	0	2	0	114	2
ServletUpdateComplaintSearch	1	0	2	0	106	2
ServletLogin	1	0	3	0	88	2
ServletGetDataForSearchByHealthUnit	1	0	2	0	70	2
ServletUpdateComplaintData	1	0	2	0	69	2
ServletGetDataForSearchBySpecialty	1	0	2	0	66	2
ServletGetDataForSearchByDiseaseTyp	1	0	2	0	65	2
ServletUpdateEmployeeData	1	0	2	0	65	2
ServletInsertEmployee	1	0	2	0	62	2
ServletSearchDiseaseData	1	0	2	0	60	2
DiseaseType	6	0	13	0	57	2

ServletSearchHealthUnitsBySpecialty	1	0	2	0	56	2
Address	8	0	11	0	56	2
ServletSearchSpecialtiesByHealthUnit	1	0	2	0	52	2
Employee	3	0	9	0	34	2
HealthUnit	3	0	8	0	32	2
MedicalSpecialty	3	0	9	0	32	2
Symptom	3	0	7	0	26	2
IFachada	0	0	13	0	86	13
UndefinedDistributionException	0	0	0	0	2	2
CommunicationException	0	0	1	1	6	6
ConfigFile	0	0	4	4	38	38
DistributionFactory	0	0	3	3	25	25
DSRMISourceAdapter	3	3	16	16	203	203
DSRMITargetAdapter	1	1	14	14	185	185
I_IteratorRMITargetAdapter	0	0	2	2	6	6
IDSRMITargetAdapter	0	0	13	13	102	102
IteratorHW	0	0	4	4	7	7
IteratorRMISourceAdapter	3	3	7	7	83	83
IteratorRMITargetAdapter	2	2	3	3	24	24
RMIDistributionFactory	1	1	3	3	27	27

Tabela 55 – Resultado de acoplamento e coesão para o *Health Watcher*

Componentes	***Health Watcher***			
	LCOO	**CBC**	**DIT**	**NOC**
Address	39	0	1	0
AddressRepositoryRDBMS	9	10	1	0
AdministratorFacade	0	6	1	0
AnimalComplaint	21	3	2	0
CitizenFacade	0	15	1	0
CommunicationException		0	4	0
Complaint	257	3	1	3
ComplaintRecord	0	10	1	0
ComplaintRepositoryArray	0	3	1	0
ComplaintRepositoryRDBMS	0	20	1	0
ConcreteIterator	4	1	1	0
ConcurrencyManager	0	2	1	0
ConfigFile		3	1	0
Date	24	4	1	0
DiseaseRecord	0	5	1	0
DiseaseRepositoryArray	0	3	1	0
DiseaseRepositoryRDBMS	0	11	1	0
DiseaseType	46	1	1	0
DistributionFactory		7	1	1
DSRMISourceAdapter	0	15	1	0
DSRMITargetAdapter	0	20	5	0
Employee	12	0	1	0
EmployeeRecord	0	8	1	0
EmployeeRepositoryArray	0	2	1	0
EmployeeRepositoryRDBMS	0	8	1	0
FacadeFactory		8	1	0
FoodComplaint	78	3	2	0
Funcoes		0	1	0
HealthUnit	12	1	1	0
HealthUnitRecord	0	6	1	0
HealthUnitRepositoryArray	0	5	1	0

HealthUnitRepositoryRDBMS	0	15	1	0
HealthWatcherFacade	0	17	1	0
Horario	17	3	1	0
HorarioInvalidoException		0	4	0
HTMLCode	159	1	1	0
I_IteratorRMITargetAdapter		1	2	1
IComplaintRepository		6	1	2
IDiseaseRepository		4	1	2
IDSRMITargetAdapter		13	2	1
IEmployeeRepository		5	1	2
IFachada		12	1	2
InvalidDateException		0	4	0
InvalidSessionException		0	4	0
IPersistenceMechanism		2	1	1
IRepositorioEspecialidade		3	1	2
IRepositorioIterable		0	1	6
IRepositorioUnidadeSaude		5	1	2
ISymptomRepository		5	1	1
IteratorHW		1	2	2
IteratorRMISourceAdapter	0	6	1	0
IteratorRMITargetAdapter	0	2	5	0
Library		5	1	0
LocalIterator		0	3	1
MedicalSpecialty	20	0	1	0
MedicalSpecialtyRecord	0	4	1	0
ObjectAlreadyInsertedException		0	4	0
ObjectNotFoundException		0	4	0
ObjectNotValidException		0	4	0
PersistenceMechanismException		0	4	0
PersistenceMechanismRDBMS	31	6	1	0
RepositorioException		0	4	0
RMIDistributionFactory	1	8	2	0
ServletGetDataForSearchByDiseaseType	0	12	1	0
ServletGetDataForSearchByHealthUnit	0	14	1	0
ServletGetDataForSearchBySpecialty	0	13	1	0
ServletInsertAnimalComplaint	0	12	1	0
ServletInsertEmployee	0	12	1	0
ServletInsertFoodComplaint	0	12	1	0
ServletInsertSpecialComplaint	0	12	1	0
ServletLogin	1	14	1	0
ServletSearchComplaintData	0	16	1	0
ServletSearchDiseaseData	0	12	1	0
ServletSearchHealthUnitsBySpecialty	0	12	1	0
ServletSearchSpecialtiesByHealthUnit	0	11	1	0
ServletUpdateComplaintData	0	16	1	0
ServletUpdateComplaintSearch	0	12	1	0
ServletUpdateEmployeeData	0	13	1	0
SpecialComplaint	21	3	2	0
SpecialtyRepositoryArray	0	3	1	0
SpecialtyRepositoryRDBMS	0	9	1	0
StatusClosedException		0	4	0
Symptom	15	0	1	0
SymptomRepositoryArray	0	2	1	0
TimestampException		0	4	0
TransactionException		0	4	0
UndefinedDistributionException		0	4	0
Soma	**767**	**517**	**145**	**29**
Média	**8,62**	**5,81**	**1,63**	**0,33**

Aplicação das Regras

Tabela 56 – Regras de SI para o interesse Concorrência do *Health Watcher*

Regras	Valor Retornado	Estado Classificado
R01	FALSO	
R02	VERDADEIRO	Entrelaçado
R03	VERDADEIRO	Elevado Espalhamento
R04	FALSO	
R05	FALSO	
R06	VERDADEIRO	Possivelmente Secundário
R07	Não se Aplica	
R08	Não se Aplica	
R09	Não se Aplica	
R10	VERDADEIRO	Secundário

Tabela 57 – Regras de SI para o interesse Distribuição do *Health Watcher*

Regras	Valor Retornado	Estado Classificado
R01	FALSO	
R02	VERDADEIRO	Entrelaçado
R03	VERDADEIRO	Elevado Espalhamento
R04	FALSO	
R05	FALSO	
R06	VERDADEIRO	Possivelmente Secundário
R07	Não se Aplica	
R08	Não se Aplica	
R09	Não se Aplica	
R10	VERDADEIRO	Secundário

Tabela 58 – Regras de acoplamento e coesão para o *Health Watcher*

Componentes	Interesses Secundários	Classificações
Address	Distribuição	Baixa Coesão
CitizenFacade	Concorrência	Elevado Acoplamento
Complaint	Concorrência, Distribuição	Baixa Coesão
ComplaintRecord	Concorrência	Elevado Acoplamento
ComplaintRepositoryArray	Concorrência	Extração de Interesses
ComplaintRepositoryRDBMS	Concorrência	Elevado Acoplamento
Date	Distribuição	Baixa Coesão
DiseaseType	Distribuição	Baixa Coesão
DSRMISourceAdapter	Concorrência	Elevado Acoplamento
DSRMITargetAdapter	Concorrência	Elevado Acoplamento
Employee	Distribuição	Baixa Coesão
EmployeeRecord	Concorrência	Elevado Acoplamento
EmployeeRepositoryArray	Concorrência	Extração de Interesses
HealthUnit	Distribuição	Baixa Coesão
HealthWatcherFacade	Concorrência	Elevado Acoplamento
IComplaintRepository	Concorrência	Extração de Interesses
IFachada	Concorrência, Distribuição	Elevado Acoplamento
IDSRMITargetAdapter	Concorrência	Elevado Acoplamento
MedicalSpecialty	Distribuição	Baixa Coesão

ServletSearchComplaintData	Distribuição	Elevado Acoplamento
ServletInsertFoodComplaint	Distribuição	Elevado Acoplamento
ServletInsertAnimalComplaint	Distribuição	Elevado Acoplamento
ServletInsertSpecialComplaint	Distribuição	Elevado Acoplamento
ServletUpdateComplaintSearch	Distribuição	Elevado Acoplamento
ServletLogin	Distribuição	Elevado Acoplamento
ServletGetDataForSearchByHealth	Distribuição	Elevado Acoplamento
ServletUpdateComplaintData	Concorrência, Distribuição	Elevado Acoplamento
ServletGetDataForSearchBySpecia	Distribuição	Elevado Acoplamento
ServletGetDataForSearchByDiseas	Distribuição	Elevado Acoplamento
ServletUpdateEmployeeData	Distribuição	Elevado Acoplamento
ServletInsertEmployee	Distribuição	Elevado Acoplamento
ServletSearchDiseaseData	Distribuição	Elevado Acoplamento
ServletSearchHealthUnitsBySpecia	Distribuição	Elevado Acoplamento
ServletSearchSpecialtiesByHealth	Distribuição	Elevado Acoplamento
Symptom	Distribuição	Baixa Coesão

Análise e Identificação de Problemas

Tanto o interesse Concorrência quanto o Distribuição são classificados como secundários neste estudo de caso. É interessante observar que quase todas as classes (32 de 35) que possuem pelo menos um destes interesses como secundário também apresentam algum problema de acoplamento ou coesão. Algumas classes possuem ambos os interesses como secundário, como é o caso de `Complaint`, `IFachada` e `ServletUpdateComplaintData`. Estas três classes possuem problemas de acoplamento ou coesão.