UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RECUPERAÇÃO DE MODELOS DE CLASSES ORIENTADOS A ASPECTOS A PARTIR DE SISTEMAS ORIENTADOS A OBJETOS USANDO REFATORAÇÕES DE MODELOS

PAULO AFONSO PARREIRA JÚNIOR

ORIENTADORA: PROFª. DRª. ROSÂNGELA APARECIDA DELLOSSO PENTEADO

CO-ORIENTADOR: PROF. DR. VALTER VIEIRA DE CAMARGO

São Carlos - SP
Maio/2011

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RECUPERAÇÃO DE MODELOS DE CLASSES ORIENTADOS A ASPECTOS A PARTIR DE SISTEMAS ORIENTADOS A OBJETOS USANDO REFATORAÇÕES DE MODELOS

PAULO AFONSO PARREIRA JÚNIOR

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software.
Orientadora: Profª. Drª. Rosângela Aparecida Dellosso Penteado.

São Carlos - SP
Maio/2011

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

P259rm

Parreira Junior, Paulo Afonso.
Recuperação de modelos de classes orientados a aspectos a partir de sistemas orientados a objetos usando refatorações de modelos / Paulo Afonso Parreira Junior. -- São Carlos : UFSCar, 2011.
194 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2011.

1. Reengenharia de software. 2. Software - manutenção. 3. Orientação a aspectos. 4. Refatoração. 5. Modelagem de software. I. Título.

CDD: 005.1 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"Recuperação de Modelos de Classes Orientados a Aspectos a partir de Sistemas Orientados a Objetos usando Refatorações de Modelos"

PAULO AFONSO PARREIRA JÚNIOR

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Membros da Banca:

R.A.Penteado

Profa. Dra. Rosângela Ap. Dellosso Penteado
(Orientadora - DC/UFSCar)

Prof. Dr. Valter Vieira de Camargo
(Co-orientador - DC/UFSCar)

Daniel Lucrédio

Prof. Dr. Daniel Lucrédio
(DC/UFSCar)

Eduardo M.F. Figueiredo

Prof. Dr. Eduardo Magno Lages Figueiredo
(UFMG)

São Carlos
Maio/2011

A minha esposa Flávia, aos meus pais Paulo e Eliana
e ao meu irmão Ricardo.

AGRADECIMENTO

Agradeço primeiramente a Deus pelas bênçãos e pelo discernimento concedidos a mim durante esta caminhada. Aos meus familiares, em especial, aos meus pais Paulo e Eliana e ao meu irmão Ricardo por todo apoio e confiança depositados em meus objetivos pessoais e profissionais. À minha esposa Flávia pela paciência, compreensão e companheirismo ao longo desses dois anos de trabalho. À minha orientadora Profª. Drª. Rosângela Penteado pela amizade e pela excelência apresentada na orientação desta pesquisa. Ao meu co-orientador Profº. Drº. Valter Camargo pelas contribuições dadas ao trabalho, que certamente agregaram muitos pontos fortes ao mesmo. Ao amigo e ex-orientador Profº. Drº. Heitor Augustus pela participação nas fases iniciais deste projeto, o que gerou excelentes ideias para o desenvolvimento das fases posteriores. Aos amigos do Grupo de Desenvolvimento e Manutenção de Software pelo apoio e prontidão para ajudar a solucionar dúvidas e dificuldades que tive ao longo deste período. Por fim, agradeço a toda equipe da coordenação do Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos pelo suporte técnico-administrativo oferecido durante o curso de mestrado.

RESUMO

Orientação a Aspectos (OA) permite encapsular Interesses Transversais (ITs) - requisitos de software cuja implementação encontra-se entrelaçada e espalhada pelos módulos funcionais - em novas abstrações, tais como, Aspectos, Conjuntos de Junção, Adendos e Declarações Inter-tipo. A reengenharia de um software OO para um OA não é uma atividade trivial em consequência da existência de abstrações diferentes entre as tecnologias envolvidas. Neste trabalho é proposto um conjunto de refatorações que pode ser aplicado sobre modelos de classes OO anotados com indícios de ITs para obtenção de modelos de classes OA. Modelos de classes anotados são diagramas de classes da UML cujos elementos (classes, interfaces, atributos e métodos) são anotados com estereótipos referentes aos ITs existentes no software. O conjunto de refatorações desenvolvido é subdivido em: i) refatorações genéricas; e ii) refatorações específicas. As refatorações genéricas são responsáveis por transformar um modelo OO anotado com indícios de ITs em um modelo OA parcial - modelo cujos ITs existentes não são completamente modularizados. Essas refatorações são aplicáveis a qualquer tipo de IT existente no software, considerando o cenário que esses interesses apresentam no modelo de classes. As refatorações específicas são responsáveis por transformar modelos de classes OA parciais em modelos de classes OA finais - modelos nos quais os ITs foram completamente modularizados em aspectos. Para que isso aconteça, cada refatoração possui um conjunto de passos específicos para modularização de um determinado tipo de interesse. Três refatorações genéricas e seis refatorações específicas foram elaboradas para os interesses de persistência (subdividido em: gerenciamento de conexões, de transações e sincronização), de logging e para os padrões de projeto Singleton e Observer. Um *plug-in* Eclipse, denominado MoBRe, foi desenvolvido para auxiliar o Engenheiro de Software na tarefa de aplicação das refatorações. Como principal contribuição, a utilização das refatorações propostas neste trabalho pode permitir a obtenção de modelos OA que representam bons projetos arquiteturais, pois: i) fornecem um guia para modularização de determinados ITs, podendo evitar que Engenheiros de Software escolham estratégias inadequadas para modularização; e ii) foram elaboradas com base em boas práticas de projeto OA preconizadas pela comunidade científica. Assim, além dos modelos a utilização das refatorações pode levar à geração de códigos de melhor qualidade, por exemplo, livre da presença de *bad smells*. Um estudo de caso foi conduzido para verificar a aplicabilidade das refatorações propostas e os modelos OA resultantes foram equivalentes aos modelos obtidos na literatura.

Palavras-chave: Manutenção de Software, Reengenharia de Software, Orientação a Aspectos, Modelagem Orientada a Aspectos, Refatorações Orientadas a Aspectos.

ABSTRACT

Aspect-Oriented Programming allows encapsulating the so-called "Crosscutting Concerns (CCC)" - software requirements whose implementation is tangled and scattered throughout the functional modules - into new abstractions, such as Aspects, Pointcuts, Advices and Inter-type Declarations. The reengineering of an OO software to an AO is not an easy task due to the existence of different abstractions in these technologies. We develop a set of nine refactorings of annotated OO class models to AO class models. In the context of this work, "annotated class models" are UML class diagrams whose elements (classes, interfaces, attributes and methods) are annotated with stereotypes representing the existing CCC in the application source code. The set of refactorings developed is divided into: i) generic refactorings; and ii) specific refactorings. Three generic refactorings and six specific refactorings to the persistence (which is divided into management and connections, transaction and synchronization) and logging concerns and to the Singleton and Observer design patterns were created. The generic refactorings are responsible for transforming an annotated OO model with indications of CCC into a partial AO model. This model is called partial because it is usually not fully modularized, i.e., there are remaining software elements stereotyped with indications of particular concerns. These refactorings are applicable to any kind of CCC; this is possible, because what is taken into consideration is the scenario in which these concerns appear in the class model. The specific refactorings are responsible for transforming partial AO models into final ones, whose CCC have been fully modularized in aspects. For that, each refactoring has a set of specific steps for modularization of a particular kind of concern. An Eclipse plug-in, called MoBRe was developed to assist the software engineer in the tasks of refactoring application. As a major contribution, the refactorings proposed in this paper allow obtaining well designed AO models. This is so because: i) they provide a step-by-step guide to the modularization of certain CCC and can avoid that software engineers choose inappropriate strategies for modularization of these CCC; and ii) they were prepared based on good design practices recommended by the scientific community. Thus, besides, the models the use of refactorings can lead to generation of better-quality code, for example, free of bad smells. A case study was conducted to assess the applicability of the proposed refactorings in order to compare an AO model generated by them with an AO model obtained from the literature.

Keywords: Software Maintenance, Software Reengineering, Aspect Orientation, Aspect-Oriented Modeling, Aspect-Oriented Refactorings.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CASE – *Computer Aided Software Engineering*

ComSCId - *Computational Support for Concern Identification*

DMAsp – *Design Model to Aspect*

DSOA – *Desenvolvimento de Software Orientado a Aspectos*

MOA – *Modelagem Orientada a Aspectos*

MoBRe – *MOdel-Based Refactoring*

OA – *Orientação a Aspectos (Orientada(o) a Aspectos)*

OCL – *Object Constraint Language*

OO – *Orientação a Objetos (Orientada(o) a Objetos)*

POA – *Programação Orientada a Aspectos*

POO – *Programação Orientada a Objetos*

RF – *Requisito Funcional*

RNF – *Requisito Não-Funcional*

UML - *Unified Modeling Language*

XMI - *XML Metadata Interchanges*

XML – *EXtensible Markup Language*

XSL - *eXtensible Stylesheet Language for Transformation*

SUMÁRIO

Capítulo 1

INTRODUÇÃO

1.1 Contextualização

A manutenção de software é uma atividade custosa na prática das organizações que utilizam e desenvolvem software, correspondendo a aproximadamente 90% das atividades de desenvolvimento de software (Pressman, 2010). Mesmo assim, os desenvolvedores, muitas vezes por falta de conhecimento, não utilizam técnicas, métodos e tecnologias preconizadas pela engenharia de software, produzindo software com baixo nível de qualidade. O software produzido geralmente atende aos objetivos dos clientes, mas apresenta diversos problemas quando alguma modificação deve ser realizada - seja para incluir uma funcionalidade adicional ou para corrigir algum defeito encontrado.

Software legado é todo aquele que precisa ser alterado para que seu tempo de vida útil seja estendido, pois embora atenda às necessidades dos usuários, foi desenvolvido com tecnologias/técnicas que dificultam as atividades de manutenção (Pressman, 2010).

Há várias técnicas, tecnologias e métodos que têm o objetivo de melhorar a manutenção de um software (Lieberherr *et al.*, 2001; Ossher e Tarr, 1999; Kiczales *et al.*, 1997; Aksit *et al.*, 1992). A Programação Orientada a Aspectos (POA) (Kiczales *et al.*, 1997), por exemplo, é uma tecnologia que tem como objetivo a elaboração de um sistema mais modular, sendo uma possibilidade a ser utilizada no processo de reengenharia de sistemas existentes, especialmente quando o software legado foi desenvolvido com Programação Orientada a Objetos (POO) (Costa *et al.*,

2009; Kawakami, 2007; Nassau e Valente, 2007; Kawakami e Penteado, 2005; Binkley *et al.*, 2005; Ramos, 2004). A POA tem como objetivo encapsular “interesses transversais” em módulos criados especificamente para esse fim, fazendo com que fiquem fisicamente separados do restante do código. Interesses transversais referem-se aos requisitos de software que produzem representações entrelaçadas e espalhadas com os módulos funcionais do sistema. Os novos módulos propostos pela POA para encapsular os interesses transversais são denominados “aspectos”.

A POA é uma tecnologia complementar à POO, entretanto aplicar um processo de reengenharia para a conversão de um sistema Orientado a Objetos (OO) para um sistema Orientado a Aspectos (OA) não é trivial. Esse processo deve ser conduzido de forma que os passos sejam bem definidos e documentados, para que o software resultante tenha melhores índices de qualidade do que a versão anterior.

1.2 Motivação e Objetivos

Muitas vezes o único artefato disponível para a análise do sistema legado é o seu código fonte, com todas as regras de negócio nele embutidas, sem qualquer documentação adicional.

Em geral, a obtenção direta automática/manual de um código fonte OA a partir de um código fonte OO pode tornar o projeto rígido e não adequado às diferentes situações/contextos/domínios. Esse é o caso de algumas abordagens que realizam a refatoração de um código fonte OO para um código fonte OA, no qual as decisões de migração de tecnologia podem resultar em código de baixa qualidade e/ou mal organizado e estruturado (Kawakami, 2007, Binkley *et al.*, 2006, Garcia *et al.*, 2004, Iwamoto e Zhao, 2003).

De modo análogo, a obtenção de um modelo de classes OA a partir de um código fonte OO não é trivial, pois a tomada de decisões durante essa atividade pode influenciar a qualidade do projeto recuperado. Por exemplo, recuperar um modelo totalmente OA a partir de um software OO, que possua o interesse transversal de persistência, consiste em decidir como esse interesse será removido e como será projetado nesse modelo. Para isso, podem existir várias alternativas de

projeto que podem ser influenciadas pelo domínio do software, pelo contexto no qual o software será usado ou por políticas internas das organizações desenvolvedoras.

Com base nesses problemas, o objetivo desta dissertação é a elaboração de um conjunto de refatorações em nível de modelos para obtenção de modelos OA a partir de modelos de classes OO anotados com indícios de interesses transversais. Os modelos de classes OO anotados são diagramas de classes da UML (*Unified Modeling Language*) com estereótipos localizados em seus elementos (classes, interfaces, atributos e métodos) que simbolizam a presença de um determinado interesse transversal no software representado pelo diagrama. Por exemplo, um atributo `conn` do tipo `Connection` pode representar um indício do interesse de persistência, por isso, receberá um estereótipo correspondente a esse interesse no modelo de classes do software ao qual ele pertence.

As principais razões para se utilizar refatorações apoiadas por modelos são:

1. geralmente, refatorações existem em nível de código fonte e não consideram o contexto em que podem ser aplicadas. Refatorações baseadas em interesses permitem que porções maiores de um sistema sejam transformadas em soluções mais bem projetadas utilizando um número menor de passos do que quando isso é feito via código. A identificação dos cenários candidatos a serem refatorados torna-se mais fácil quando existe um apoio visual para essa identificação. Assim, identificar tais cenários em código fonte é uma tarefa mais onerosa do que fazê-lo em modelos. Por exemplo, identificar todos os elementos de um código fonte afetados por um determinado interesse transversal é mais difícil do que fazer a mesma identificação em um modelo de classes.
2. refatorações baseadas em modelo possuem a vantagem de serem independentes de plataforma. Assim, modelos podem ser transformados e bons projetos podem ser produzidos sem a dependência da linguagem alvo, permitindo que ela seja escolhida de acordo com requisitos do ambiente operacional e dos stakeholders.
3. refatorações em nível de código fonte podem ser aplicadas para converter um software OO em um OA. Entretanto, essa transformação é geralmente feita em apenas um passo, em que tem-se como entrada o codigo OO e como saída o código OA. Isso faz com que esse processo tenha pouca flexibilidade recaindo sobre a qualidade das refatorações a responsabilidade de gerar um

codigo OA que siga bons estilos arqueteturais. Fazer essa transformação utilizando refatorações baseada em contexto e apoiadas por modelos introduz, pelo menos, um passo a mais no processo antes de se obter o código fonte final. Esse passo intermediário consiste na obtenção de um modelo OA, em que o engenheiro de software pode refletir sobre ele e modificá-lo de acordo com os requisitos do ambiente operacional e dos stakeholders. Isso permite que esse processo se torne mais flexível.

4. Além disso, as diferenças semânticas e sintáticas existentes entre OO e OA fazem com que haja um gap muito grande entre os paradigmas o que dificulta a transformação em apenas um passo.

Para obtenção automática dos indícios de interesses transversais e dos modelos de classes OO anotados serão utilizadas, respectivamente, as ferramentas ComSCId (*Computational Support for Concern Identification*) (Parreira Júnior *et al.*, 2010a) e DMAsp (em inglês, *Design Model to Aspect*) (Costa *et al.*, 2009). ComSCId é um *plug-in* Eclipse (Eclipse, 2011) que implementa a técnica de identificação de interesses baseada em tipos e texto. Com ele é possível reconhecer convenções de nomenclaturas de métodos, variáveis e classes e determinados tipos de dados em códigos fonte Java. DMAsp também é um *plug-in* Eclipse e permite obter um modelo de classes OO anotado com indícios de interesses transversais a partir do código fonte Java. Tais indícios são representados no diagrama de classe por estereótipos aplicados aos seus elementos (classes, atributos e métodos).

Com base nos modelos de classes OO anotados obtidos com o auxílio das ferramentas ComSCId e DMAsp, refatorações são aplicadas sobre esses modelos a fim de se obter modelos de classes OA. As refatorações propostas nesta dissertação foram desenvolvidas com base em boas práticas para projeto de software OA preconizadas pela comunidade científica (Piveta *et al.,* 2007; Sant'anna *et al.*, 2007; Hannemann, 2006; Monteiro e Fernandes, 2006; Piveta *et al.,* 2006; Hannenberg *et al.*, 2003). Além das refatorações, desenvolveu-se ainda um *plug-in* Eclipse, denominado MoBRe, com o objetivo de automatizar as atividades necessárias para aplicação das refatorações e minimizar o esforço necessário para aplicá-las em softwares de médio e grande porte e a possibilidade da ocorrência de falhas humanas durante esse processo.

O modelo OO gerado pelo DMAsp é utilizado como intermediário antes da obtenção de um modelo OA, visando a melhorar a qualidade do processo de

reengenharia de software OO para OA. Os modelos OA obtidos a partir da aplicação das refatorações propostas neste trabalho são construídos em conformidade com o perfil para modelagem de software OA proposto por Evermann (2007) e podem ser visualizados no ambiente Eclipse.

A refatoração é uma das técnicas essenciais utilizadas para mitigar problemas relacionados à evolução (Mens e Tourwe, 2004). Zhang *et al.* (2005) afirmam que a aplicação de refatorações em fases anteriores à implementação possibilita melhoria na qualidade do projeto, garantindo redução na complexidade e no custo durante as fases subsequentes do ciclo de desenvolvimento.

Muitos trabalhos têm sido propostos para refatoração de software OO para software OA, porém na maior parte deles essas refatorações são aplicadas apenas em nível de código, ou seja, têm como entrada um código OO e com a aplicação das refatorações a saída é um código OA. Esse é o caso dos trabalhos de Silva *et al.* (2009); Murphy *et al.* (2009); Monteiro e Fernandes (2006); Hannemann *et al.* (2005); Marin *et al.* (2005) e Iwamoto e Zhao (2003).

Além disso, após pesquisa na literatura especializada, notou-se escassez de trabalhos relacionados à refatoração em nível de modelos. Boger *et al.* (2002) desenvolveram um *plug-in* para a ferramenta CASE (*Computer Aided Software Engineering*) ArgoUML[1] que apoia a refatoração de modelos UML. Com ele é possível refatorar diagramas de classe, de estados e de atividades permitindo ao usuário aplicar refatorações que não são simples de se aplicar em nível de código. Van Gorp *et al.* (2003) propuseram um perfil UML para expressar pré e pós-condições de refatoração de código fonte utilizando restrições OCL (*Object Constraint Language*). O perfil proposto permite que uma ferramenta CASE: i) verifique pré e pós-condições para composição de sequências de refatorações; e ii) use o mecanismo de consulta OCL para detectar *bad smells*[2]. Além desses, há trabalhos que tratam do uso de refatorações de modelos para introdução de Padrões de Projeto (*Design Patterns*) (Gamma *et al.*, 1995) em software orientado a objetos (Scherlis, 1998; Genssler *et al.*, 1998; Tokuda e Batory, 1995). Embora esses trabalhos tenham enfoque na refatoração de modelos, nenhum deles apresenta

[1] http://argouml.tigris.org/

[2] *Bad smells* podem ser vistos como sinais ou alertas de que problemas existem no software (Elssamadisy e Schalliol, 2002) e podem ser corrigidos através da aplicação de refatorações.

estudos que consideram, especificamente, modelos orientados a aspectos. O diferencial deste projeto em relação aos demais é a proposta de construção de um modelo OA considerando modelos de classes OO anotados com estereótipos representando interesses transversais.

1.3 Organização da Dissertação

Esta dissertação está organizada em oito capítulos. Neste primeiro foi feita a contextualização de como orientação a aspectos pode auxiliar o processo de manutenção de software, apresentaram-se os objetivos e a motivação para a presente pesquisa de mestrado.

No Capítulo 2 a Orientação a Aspectos é introduzida com a descrição de seus termos mais comumente usados e a apresentação da abordagem proposta por Evermann (2007) para modelagem visual de software Orientado a Aspectos.

No Capítulo 3 a identificação e a classificação de interesses transversais são discutidas como uma alternativa para a reengenharia de software Orientado a Objetos para software Orientado a Aspectos. Discussões relacionadas à refatoração orientada a aspectos são conduzidas e ferramentas e técnicas são apresentadas.

No Capítulo 4 são apresentadas as refatorações para interesses transversais com base em modelos de classes OO anotados desenvolvidas neste projeto de pesquisa. Essas refatorações são classificadas em refatorações: genéricas, as que podem ser aplicadas a qualquer tipo de interesse e específicas, as que são elaboradas especialmente para um determinado tipo de interesse, por exemplo, o de persistência. Além das refatorações, também são apresentados exemplos que ilustram a aplicabilidade dessas.

No Capítulo 5 as ferramentas ComSCId (Parreira Júnior *et al*., 2010a) e DMAsp (Costa *et al*., 2009), utilizadas neste trabalho para obtenção de modelos de classes OO anotados com indícios de interesses transversais a partir de códigos Java, e a ferramenta MoBRe, construída para automatizar a tarefa de aplicação das refatorações OA, são apresentadas, destacando-se as suas principais características. A ferramenta MoBRe é flexível, ou seja, pode ser estendida pelo usuário que implementa novas refatorações OA a fim de ampliar sua funcionalidade.

Além da funcionalidade da ferramenta, exemplos da expansão de sua funcionalidade também são apresentados.

No Capítulo 6 um estudo de caso foi conduzido com o objetivo de apresentar a aplicabilidade das refatorações elaboradas. As refatorações foram aplicadas a três aplicações desenvolvidas por terceiros, *Health Watcher* (Greenwood et al., 2007), *JSpider* (2011) e *JAccounting* (2011) para os interesses de persistência, *logging* e para os padrões de projeto *Observer* e *Singleton* (Gamma *et al.*, 1995).

No Capítulo 7 é discutida uma avaliação dos resultados obtidos com o estudo de caso. Um conjunto de métricas OA foi utilizado para comparação entre a solução de modularização de interesses transversais, obtida a partir da aplicação das refatorações desenvolvidas neste projeto de mestrado, com a solução elaborada por pesquisadores experientes em modularização de interesses transversais sem a utilização das refatorações aqui propostas.

Por fim, no Capítulo 8, são apresentadas as contribuições e limitações deste projeto e possíveis trabalhos futuros.

Capítulo 2

ORIENTAÇÃO A ASPECTOS

2.1 Considerações Iniciais

Orientação a Aspectos é uma das tecnologias que visam à melhor modularização do software, encapsulando requisitos cuja implementação pode tornar-se espalhada e entrelaçada pelos módulos funcionais do software em módulos específicos denominados aspectos. Neste capítulo são apresentados os principais conceitos relacionados à orientação a aspectos e é descrita uma abordagem que permite a modelagem visual de software Orientado a Aspectos (OA). A terminologia de OA utilizada neste trabalho segue a nomenclatura adotada pela Comunidade Brasileira de Desenvolvimento de Software Orientado a Aspectos (AOSDbr) para a Língua Portuguesa (AOSDbr, 2008).

Na Seção 2.2 há um breve histórico sobre a Programação Orientada a Aspectos (POA), abordando sua origem e sua importância, bem como apresentando os seus conceitos básicos. Na Seção 2.3 é descrito o Perfil UML para modelagem de software OA proposto por Evermann (2007) que será utilizado neste trabalho. As considerações finais estão apresentadas na Seção 2.4.

2.2 Programação Orientada a Aspectos

O conceito de interesses (*concerns*) foi apresentado por Dijkstra (1976) e refere-se a requisitos presentes no processo de construção de um software, que envolvem desde assuntos de alto nível aos relacionados a implementações de baixo nível.

Um software pode ser visto como uma implementação combinada de múltiplos interesses, tais como regras de negócios, desempenho, persistência, registro de informações (*logging*), autenticação, segurança, gerenciamento de memória, balanceamento de carga e replicação (Kiczales *et al.*, 2001). Na Figura 2.1 é apresentado um exemplo de software visto como a implementação de um conjunto de interesses. Os módulos de implementação são compostos por outros módulos, como o de persistência (*persistence*), o de lógica de negócio (*business logic*) e o de registro de informações (*logging*).

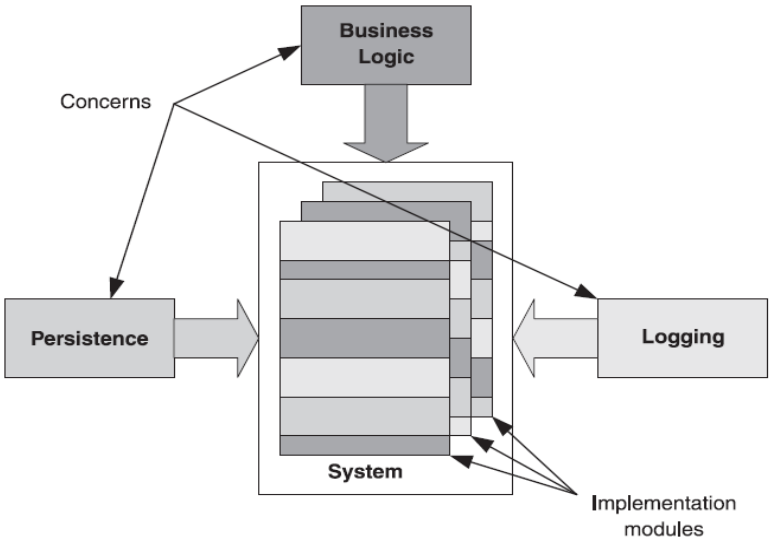

Figura 2.1 – Exemplo de um Sistema visto como um Conjunto de Interesses (Laddad, 2003).

A Programação Orientada a Objetos (POO) trouxe avanços para o desenvolvimento de software, permitindo a construção de projetos de modo mais fácil, com maior reusabilidade de componentes, modularidade, componentização, implementações mais robustas e redução do custo de manutenção (Junior e Winck, 2006). Entretanto, apesar de tais avanços, o aumento da complexidade do software gerou novos problemas, difíceis de serem resolvidos com os elementos de abstração oferecidos pela orientação a objetos (OO). Na concepção de Kiczales *et al.* (1997)

as técnicas de POO e da programação estruturada não são suficientes para implementar importantes decisões de projeto. Isto conduz a uma implementação espalhada (*scattered*) e entrelaçada (*tangled*), resultando em um emaranhado de código que se torna difícil de desenvolver e manter.

A necessidade de novas abstrações capazes de propiciar maior clareza na separação de interesses motivou a definição de novas técnicas de programação e de desenvolvimento de software. Dentre as principais técnicas pode-se citar: *Adaptive Programming* (AP) (Lieberherr *et al.*, 2001); *Composition Filters* (CF) (Aksit *et al.*, 1992); *Subject-Oriented Programming* (SOP) (Ossher e Tarr, 1999); e *Aspect-Oriented Programming* (AOP, em português, POA) (Kiczales *et al.*, 1997). POA tem se mostrado mais promissora e é abordada neste trabalho, pois não trabalha isoladamente, complementando o paradigma orientado a objetos. Alguns termos relacionados à POA são (Elrad *et al.*, 2001a):

- **Pontos de Junção (*Join Points*)**. Referem-se a pontos bem definidos no fluxo de execução de um programa, onde um comportamento adicional pode ser anexado. Um modelo de pontos de junção provê um conjunto de referências que propicia a definição de estruturas de aspectos. Os elementos mais comuns de um modelo de pontos de junção são as chamadas de métodos, apesar das linguagens OA terem definido pontos de junção para outras circunstâncias (por exemplo, acesso, modificação e definição de atributos, exceções e execução de eventos e estados). Se uma linguagem OA permite chamadas de métodos no seu modelo de pontos de junção, um programador pode designar um código adicional para ser executado antes, após ou durante uma chamada de método;
- **Interesses transversais (*CrossCutting Concerns*)**. Um interesse que entrecorta outros é denominado interesse transversal. Um interesse transversal produz representações entrelaçadas e espalhadas que podem ser inseridas em diversas etapas do ciclo de vida de um software.
- **Aspecto (*Aspects*).** Representam unidades modulares projetadas para implementação de interesses. Uma definição de aspectos pode conter códigos e instruções sobre onde, quando e como será invocado;
- **Conjuntos (de pontos) de junção (*Pointcut*s)**. Essa característica provê um mecanismo que possibilita a declaração de pontos bem definidos no

fluxo de execução de um software para implementação de comportamentos adicionais;

- **Adendos (*Advice*)**. Referem-se a construções similares aos métodos de objetos do paradigma OO e compreende o comportamento que deve ser executado em cada ponto de junção (*join point*) especificado por um conjunto de junção (*pointcut*). Por exemplo, o código de segurança responsável por realizar a autenticação e o controle de acesso. Muitas linguagens OA possuem mecanismos para executar o adendo antes (*before*), depois (*after*), em substituição ou acerca (*around*) dos pontos de junção convenientes;
- **Declarações inter-tipo (*Inter-type Declarations*)**. São utilizadas para alterar a estrutura da classe, introduzindo novos atributos, métodos, construtores, *getters*, *setters*, herança e interfaces;
- **AspectJ**. É uma extensão de propósito geral para orientação a aspectos da linguagem Java (Kiczales *et al.*, 2001).
- **Combinação (*Weaving*)**. É o processo de costura de um ou mais aspectos com os componentes não aspectuais (componentes OO, por exemplo). Um combinador (*weaver*) processa o código, compondo-o de forma correta, e produz a interação desejada. O processo de combinação pode ser: i) estático, o qual consiste em modificar o código fonte original, inserindo sentenças nos pontos de combinação, que representam o código de aspectos, em tempo de compilação (exemplo: AspectJ); ou ii) dinâmico, no qual as combinações são feitas em tempo de execução e os aspectos devem existir em tempo de compilação e em tempo de execução, por exemplo, AOP/ST (Böllert, 1999).

2.3 Modelagem de Software Orientado a Aspectos

Há várias abordagens na literatura para modelagem de software orientado a aspectos (Schauerhuber *et al.*, 2007). Algumas são baseadas na ideia de composição de modelos, em que os interesses são modelados de forma independente e depois compostos para a geração do sistema todo. Outras são

baseadas em extensões leves (*light weight*) e pesadas (*heavy weight*) da UML. Extensões leves estendem as metaclasses da UML ou restringem os elementos existentes. Extensões pesadas permitem alterações mais profundas, como novas metaclasses ou alterações profundas nas metaclasses existentes (OMG, 2011).

Nesta seção é apresentado o perfil UML para AspectJ proposto por Everman (2007), referenciado, daqui em diante, por ProAJ/UML (*Profile for AspectJ in UML*). Tal abordagem é retratada neste trabalho, pois: i) é uma extensão leve da UML; ii) mostrou-se mais adequada ao contexto deste trabalho; iii) continua sendo evoluída (mantida); e iv) apresenta suporte ferramental.

2.4 Perfil UML para AspectJ (ProAJ/UML)

Utilizando mecanismos de extensão disponíveis na UML, Evermann (2007) propôs um perfil que provê elementos de modelagem para todos os conceitos implementados pela linguagem AspectJ. Algumas vantagens desse perfil são:

- **Suporte ferramental**: por ser uma abordagem de extensão leve da UML, o perfil proposto não necessita de software especial. Permite que a modelagem de aspectos seja realizada na maioria das ferramentas de modelagem disponíveis que estejam em conformidade com o padrão UML 2.0;
- **Compatibilidade com o padrão XMI (*XML Metadata Interchange*)**: permite que um modelo gerado em uma determinada ferramenta de modelagem possa ser exportado para um arquivo no formato XMI (OMG, 2011) que será posteriormente importado em outra ferramenta;
- **Consistência do metamodelo proposto**: a maioria dos conceitos especificados pela linguagem AspectJ pode ser representada utilizando os elementos definidos pelo metamodelo;
- **Separação entre aplicação base e interesses transversais**: a abordagem proposta mantém clara distinção entre as classes base e os interesses transversais que afetam estas classes.

O perfil completo proposto por Evermann (2007) é apresentado na Figura 2.2. Cada classe ilustrada nessa figura é um estereótipo que pode ser aplicado em nível

de modelo. Cada estereótipo estende uma determinada metaclasse da UML que é graficamente representada entre colchetes ([ ]). Por exemplo, o texto `[Class]` apresentado abaixo do nome do estereótipo `Aspect` denota que esse estereótipo estende a metaclasse `Class` do metamodelo da UML. A seguir é apresentada breve descrição de alguns estereótipos disponíveis no perfil proposto pelo autor:

- ***CrossCuttingConcern***: o estereótipo `<<CrossCuttingConcern>>` estende a metaclasse `Package` e foi criado com o objetivo de encapsular aspectos relacionados. O conceito de pacotes foi utilizado, pois da mesma forma que um pacote pode conter várias classes, um interesse transversal (*CrossCuttingConcern*) pode conter vários aspectos. Além disso, como *CrossCuttingConcern* é extensão de um pacote da UML, o mesmo poderá conter aspectos e classes;
- ***Aspect***: o estereótipo `<<Aspect>>` é uma extensão da metaclasse `Class` da UML. Um aspecto contém características estáticas (conjuntos de junção) e dinâmicas (adendos). Além disso, aspectos podem ser relacionados por meio de herança com classes ou outros aspectos, bem como podem implementar (*realize*) interfaces. Tais características são similares às de uma classe UML, e desta forma, um aspecto pode ser modelado como um estereótipo que estende a metaclasse `Class`;
- ***Advice***: o estereótipo `<<Advice>>` é uma extensão da metaclasse `BehavioralFeature` e modela a característica comportamental de um aspecto. No metamodelo da UML, classes encapsulam características comportamentais e, desta forma, aspectos estão automaticamente associados aos adendos;
- ***Static CrossCutting Features***: o estereótipo que estende a metaclasse `Feature` (superclasse de `Property` e `Operation`) é denominado `<<StaticCrosscuttingFeatures>>`. Com esse estereótipo é possível representar a introdução de atributos e métodos a partir dos aspectos nas classes existentes;

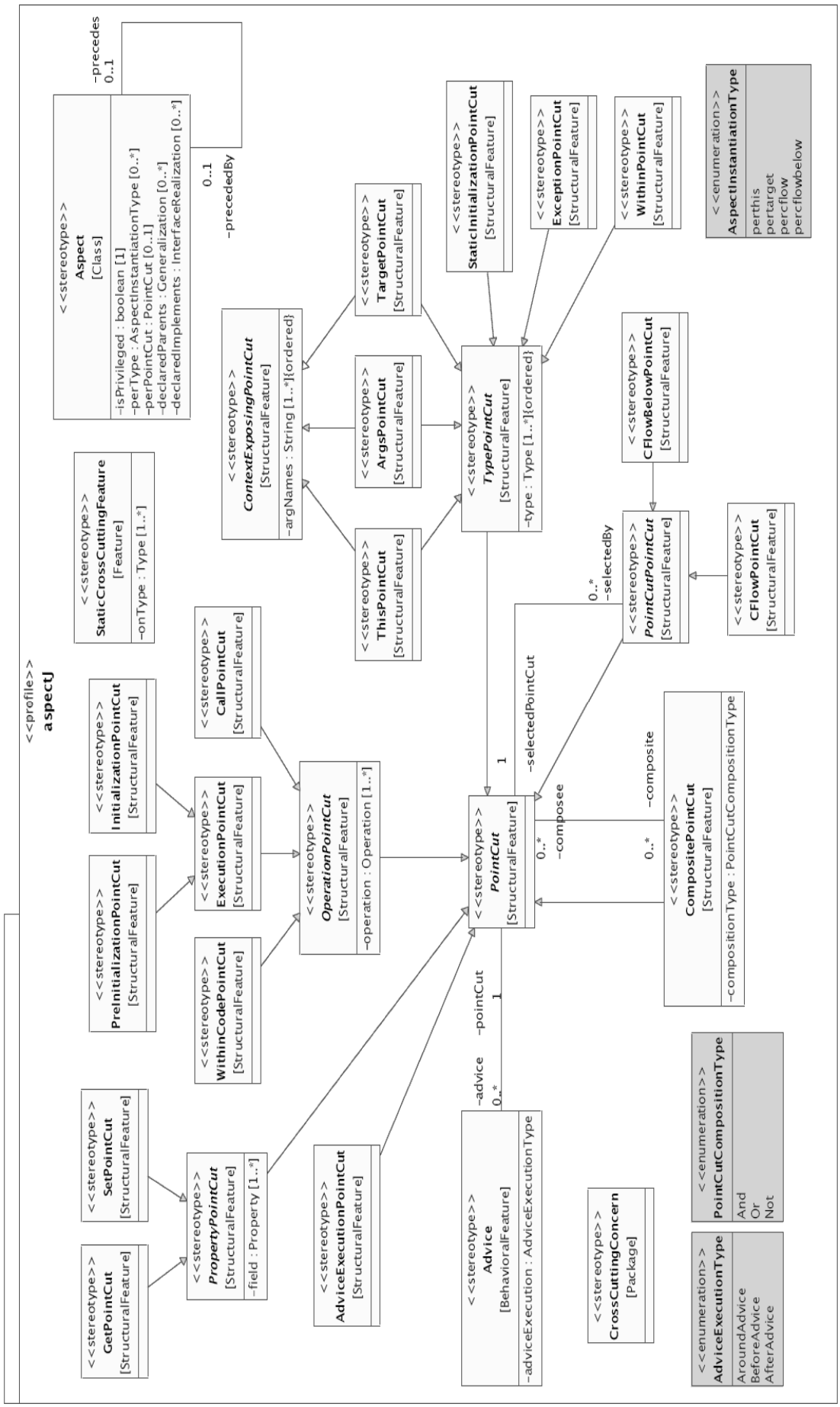

Figura 2.2 - Perfil UML para AspectJ (Evermann, 2007).

- ***PointCut***: o estereótipo `<<PointCut>>` estende a metaclasse `StructuralFeature`. O estereótipo `<<PointCut>>` é abstrato, assim ele não pode ser aplicado aos atributos de um aspecto no nível de modelo. Para isto, existem as subclasses `<<CallPointCut>>` e `<<ExecutionPointCut>>`. Ao invés de definir declarações textuais de conjuntos de junção, o autor criou várias subclasses, permitindo que diferentes atributos pudessem ser usados nos conjuntos de junção (`<<TypePointCut>>`, `<<OperationPointCut>>`, `<<AdviceExecutionPointCut>>` entre outras).

Na Figura 2.3 é apresentado um exemplo de aplicação do perfil ProAJ/UML. O interesse transversal, modelado como um pacote com o estereótipo `<<CrossCuttingConcern>>` é denominado `LoggerConcern` e seu aspecto é `Log`.

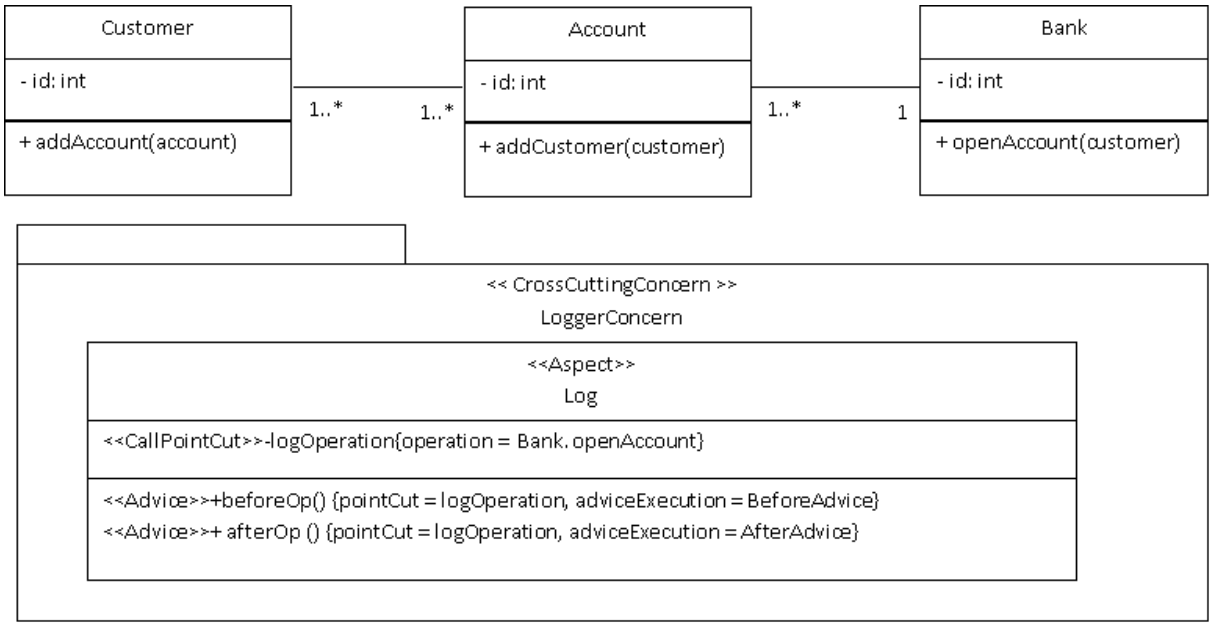

Figura 2.3 – Modelo Base e Interesse Transversal.

O conjunto de junção `logOperation()` do aspecto `Log` é estereotipado com `<<CallPointCut>>`. O atributo `operation` da metaclasse `CallPointcut` torna-se uma etiqueta que fornece a lista dos pontos de junção selecionados por este conjunto de junção. Neste caso, o conjunto de junção selecionado refere-se à chamada do método `openAccount()` da classe `Bank`.

Adendos são operações estereotipadas com `<<Advice>>`. A metaclasse `Advice` é associada com a metaclasse `Pointcut`. Desta forma, cada adendo no aspecto está associado a um conjunto de junção especificado com o valor da

etiqueta `pointCut`. No exemplo da Figura 2.3, os adendos `beforeOp()` e `afterOp()` inserem um novo comportamento antes e após alcançar o conjunto de junção `logOperation()`, que por sua vez, intercepta as chamadas realizadas ao método `openAccount()`. Outros exemplos da aplicação deste perfil podem ser obtidos em Evermann (2007).

O ProAJ/UML apresenta um metamodelo bem definido com base na UML e permite clara distinção entre aplicação base e seus interesses transversais. Além disso, ele pode ser utilizado em qualquer ferramenta CASE que seja compatível com UML 2.0. Porém, o perfil proposto não permite a utilização de curingas (*wildcards*) na declaração conjuntos de junção. Desta maneira, caso seja necessário declarar um conjunto de junção para interceptar todos os métodos de uma determinada classe, o desenvolvedor deverá selecionar cada ponto de junção desejado. Outros pontos fracos desta abordagem são: i) permite apenas a elaboração do modelo de classes; e ii) é específica para modelar conceitos da linguagem de programação OA AspectJ. Porém, para o contexto desse trabalho, tais limitações não apresentaram impactos negativos relevantes, uma vez que apenas modelos de classes são considerados e não há geração de código fonte.

2.5 Considerações Finais

Neste capítulo foram apresentados os principais conceitos sobre OA, abordando a sua origem, sua evolução e uma abordagem que pode ser utilizada para realizar modelagem de software orientado a aspectos. Como pode ser observado, OA tem como principal objetivo separar interesses, de modo a evitar o entrelaçamento de código com diferentes propósitos e o espalhamento de código com propósito específico em várias partes do software.

Apresentou-se ainda o Perfil UML para modelagem de software OA proposto por Evermann (2007). Não existe atualmente na comunidade científica, convenção ou abordagem padrão para modelagem de software OA, tal como a UML é para a Orientação a Objetos.

Embora não haja consenso sobre qual a melhor linguagem de modelagem OA, a abordagem proposta por Evermann (2007) se apresentou mais adequada

como base para a realização deste trabalho, pois: i) possui um metamodelo compreensível e robusto baseado na UML o que visa a facilitar a geração de modelos OA, uma vez que esses modelos serão obtidos a partir de modelos OO especificados em UML; ii) apresenta boa legibilidade e clareza, características estas que devem ser refletidas pelos modelos OA gerados; iii) é um perfil completo para AspectJ o que provê subsídio para geração de código OA a partir dos modelos de projeto elaborados.

Além disso, neste trabalho optou-se por abordagens que utilizam mecanismo de extensão leve, como é o caso da ProAJ/UML, pois a linguagem UML não é alterada e tona-se possível manipular os modelos OA com ferramentas CASE disponíveis atualmente.

Capítulo 3

REENGENHARIA DE SOFTWARE ORIENTADO A OBJETOS PARA SOFTWARE ORIENTADO A ASPECTOS

3.1 Considerações Iniciais

Para Sommerville (2006) a reengenharia de software é descrita como a reorganização e modificação de sistemas de software existentes, parcial ou totalmente, para torná-los mais manuteníveis. Aplicando-se reengenharia de software, um sistema pode ser redocumentado ou reestruturado, os programas podem ser implementados em uma linguagem de programação mais moderna, bem como seus dados podem ser migrados para uma base de dados diferente. A reengenharia de software OO para software OA possui como objetivo a conversão de um software originalmente desenvolvido segundo o paradigma OO em um software OA, preservando a funcionalidade do sistema original (OO).

Entretanto, aplicar técnicas orientadas a aspectos manualmente em sistemas legados é um processo difícil e propenso a erros. Para contornar esses problemas, muitos estudos têm sido conduzidos em emergentes áreas de pesquisa como mineração e refatoração de interesses transversais. Mineração de interesses transversais é a atividade de descoberta de potenciais interesses transversais que poderão ser modularizados em aspectos. Refatoração é a atividade que efetivamente transforma esses potenciais aspectos em aspectos reais no sistema.

Uma representação simplificada da migração de um sistema legado para um sistema orientado a aspectos é apresentada na Figura 3.1.

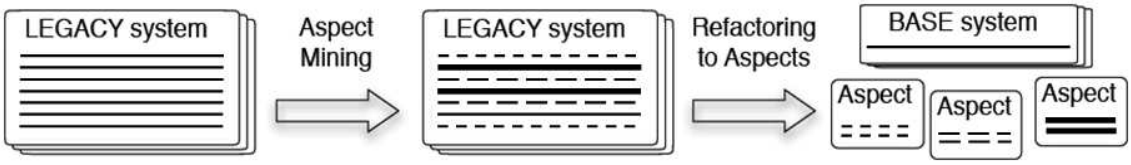

Figura 3.1 – Migração de um Sistema Legado para um Sistema Orientado a Aspectos (Kellens *et al.*, 2007).

Neste capítulo são comentados alguns assuntos relacionados à reengenharia de software orientado a objetos para software orientado a aspectos. Ele encontra-se organizado da seguinte forma. Na Seção 3.2, são apresentadas as técnicas para mineração de interesses transversais baseadas em análise de tipos e texto, análise exploratória e análise de fan-in. Além disso, as ferramentas FEAT e FINT que implementam, respectivamente, as técnicas de análise exploratória e análise por fan-in são discutidas. A Seção 3.3 apresenta as principais técnicas e ferramentas relacionadas à refatoração de interesses transversais para aspectos. As considerações finais estão apresentadas na Seção 3.4.

3.2 Técnicas e Ferramentas para Mineração de Aspectos

Nesta seção são apresentadas três diferentes técnicas para mineração de interesses transversais em código fonte: i) análise de tipos e texto; ii) análise por *fan-in*; e iii) análise exploratória. São apresentadas também duas ferramentas que automatizam a atividade de mineração: FINT (Marin *et al.*, 2004), que implementa a técnica de análise por *fan-in* e FEAT (Robillard e Murphy, 2007) que realiza análise exploratória. Além dessas técnicas há vários trabalhos cujo enfoque é mineração de aspectos (Kellens *et al.*, 2007), entretanto, as abordagens tratadas neste trabalho foram escolhidas porque: i) apresentam suporte ferramental; ii) têm recebido boas avaliações nos estudos comparativos realizados pela comunidade científica (Kellens *et al.*, 2007).

Abaixo, introduz-se alguns termos que são utilizados ao longo deste capítulo e adaptados de Marin *et al.* (2004):

- **Semente**: uma semente é um elemento ou uma coleção de elementos de código, tais como métodos, classes e atributos que indicam a existência de um interesse transversal. Uma semente oferece um ponto de partida para descoberta de outras sementes, bem como para entendimento de como o interesse transversal representado por essa semente encontra-se implementado no sistema. Por exemplo, um atributo `conn` do tipo `java.sql.Connection` pode representar uma semente do interesse transversal de persistência que pode ser utilizada para descoberta de outras sementes, como os métodos que utilizam o atributo `conn`;
- **Candidato à semente**: é um elemento de código identificado a partir da utilização de alguma técnica automática para mineração de aspectos, mas que ainda não foi confirmado se é uma semente ou não.

3.2.1 Análise de Tipos e Texto

A técnica de análise de tipos e texto permite pesquisar por sementes utilizando como base para a pesquisa um “tipo” ou um conjunto de caracteres (*string*). Por exemplo, sementes para o interesse de conexão com Banco de Dados podem ser identificadas utilizando como base o tipo `Connection` do pacote `java.sql` – todo ponto da aplicação que declarar ou utilizar um objeto desse tipo será identificado. Outra alternativa de busca é por “texto”; nesse caso, uma determinada semente será buscada utilizando uma *string* que será comparada com os elementos do código fonte tais como nomes de classes, interfaces, atributos, métodos ou valores associados a atributos do tipo `String`. Com esse tipo de busca é possível encontrar, por exemplo, todos os atributos do tipo *string* declarados no código fonte que recebam como valor uma cadeia de caracteres que contém a palavra “mysql”. Essa técnica é conveniente quando existem convenções consistentes de nomes de identificadores no código fonte do sistema. No Capítulo 5 será apresentado o ComSCId (*Computational Support for Concern Identification*) (Parreira Júnior *et al.*, 2010a; Parreira Júnior *et al.*, 2010b), uma das ferramentas computacionais que automatiza a técnica de análise de tipos e textos.

3.2.2 Análise Exploratória - FEAT

Análise exploratória é uma técnica para mineração de interesses transversais que permite explorar o código do software a partir de um conjunto de sementes de interesses transversais previamente identificadas. Essas sementes são identificadas manualmente pelo Engenheiro de Software a partir de uma análise inicial do código fonte do software e servem então para identificação de outras sementes de interesses transversais. A partir desse conjunto inicial de sementes é possível consultar informações relacionadas a elas, como por exemplo, locais onde essas sementes são declaradas e/ou utilizadas. FEAT (Robillard e Murphy, 2007) e JQuery (Janzen e Volder, 2003) são exemplos de ferramentas que implementam essa técnica. Na ferramenta FEAT, por exemplo, o conceito de Grafos de Interesses (*Concern Graphs*) é utilizado para representar o conjunto inicial de sementes e identificar de forma semiautomática os interesses existentes no software. Um Grafo de Interesse é um conjunto de interesses e subinteresses associados às sementes que é criado manualmente pelo Engenheiro de Software durante a análise do código fonte do software. Esse grafo é criado a partir da experiência do Engenheiro de Software, que ao analisar o código fonte, identifica elementos de código, como classes/interfaces, métodos e atributos que estão relacionados a um determinado interesse e os adiciona ao grafo de interesses.

Como exemplo, considere um software de uma agência bancária. Suponha que as classes `Account` e `Bank` sejam responsáveis pelas operações básicas de "Verificar Saldo", "Realizar Transferência", "Efetuar Depósito" e "Efetuar Saque". Na Figura 3.2 e Figura 3.3 são apresentados trechos de códigos dessas cujos destaques em cinza correspondem às sementes do interesse de persistência.

```
1  import java.sql.Connection;
2  import java.sql.DriverManager;
3  public class Account {
4   double balance = 0;
5   Connection conn;
6   public void withdraw(double value) {
7   Class.forName("com.mysql.jdbc.Driver");
8   conn = DriverManager.getConnection("url", "user", "pass");
9   balance -= value;
10  conn.close();
11  }
12 }
```

Figura 3.2 - Trecho de Código da Classe `Account`.

```
1  public class Bank {
2   public void transfer(double value, Account from, Account to) {
3   from.withdraw(value);
4   ...
5   }
6   public void withdraw(double value, Account from) {
7   from.withdraw(value);
8   ...
9   }
10 }
```

Figura 3.3 - Trecho de Código da Classe Bank.

As linhas 1 e 2 da Figura 3.2 são destacadas, pois são responsáveis pela importação das classes `Connection` e `DriverManager`, relacionadas ao interesse de persistência. Na linha 5 um atributo do tipo `Connection` é declarado e nas linhas 8 e 10 ele é utilizado, por isso esses trechos de código são classificados como sementes do interesse transversal de persistência. A linha 7 é destacada, pois é responsável pelo carregamento do *driver* utilizado para comunicação com o Banco de Dados.

Um grafo de interesses foi criado na ferramenta FEAT de acordo com código da Figura 3.2 e Figura 3.3 e o resultado é apresentado na Figura 3.4. Na visão "Concern Graph View" (1) é apresentado o nome de cada interesse na hierarquia do grafo de interesses, nesse caso, o interesse de persistência (*Persistence*). A partir dessa visão, desenvolvedores podem criar novos subinteresses, remover interesses existentes e mover interesses na hierarquia. Ao selecionar um interesse nessa visão, são apresentadas todas as sementes (métodos, atributos, classes e interfaces) correspondentes ao interesse selecionado que foram adicionadas manualmente pelo Engenheiro de Software no grafo de interesses.

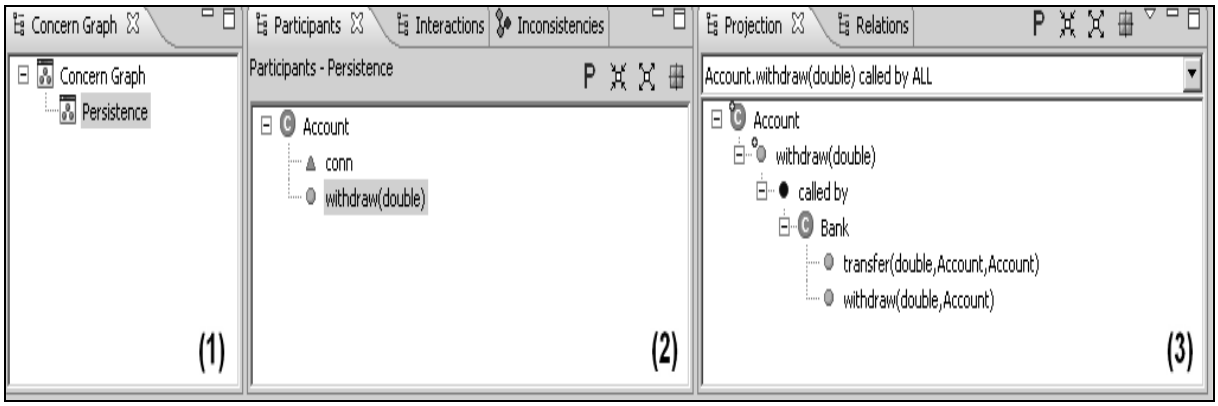

Figura 3.4 – Interface da ferramenta FEAT.

Por exemplo, ao selecionar o interesse de persistência nessa visão, todas as sementes identificadas pelo Engenheiro de Software são exibidas, nesse caso, o

atributo `conn` e o método `withdraw()` da classe `Account`. Essas sementes são apresentadas na visão “Participants View” (2).

Ao selecionar uma determinada semente, são apresentados os relacionamentos dessa semente com outros trechos de código do sistema na visão “Relation View” (3). Esses relacionamentos são obtidos por meio de consultas do tipo *Fan-in* e *Fan-out*. Por exemplo, a consulta “Declaring” do tipo *Fan-out* é aplicável a classes e interfaces e retorna todos os atributos e métodos declarados na classe/interface selecionada. A consulta “Called-by” do tipo *Fan-in*, por sua vez, é aplicável a métodos e retorna os elementos que invocam o método selecionado. No exemplo da Figura 3.4, a consulta “Called-by” foi aplicada sobre o método `withdraw()`da classe `Account`. Logo, todos os métodos que invocam o método `withdraw()` serão apresentado na visão “Relation View”, nesse caso, os métodos `withdraw()` `e transfer()` da classe `Bank`.

3.2.3 Análise por *Fan-in* - FINT

A técnica de análise por *fan-in* é baseada na ideia de que a quantidade de chamadas a um determinado método, isto é, seu *fan-in*, é uma boa medida para identificar candidatos à semente de interesses transversais (Ceccato *et al.*, 2006). O resultado da técnica de análise por *fan-in* é o cálculo do *fan-in* para todos os métodos existentes no sistema e quem decide se o método é uma semente ou não é o Engenheiro de Software, ou seja, com essa técnica, é possível obter um conjunto de candidatos à semente.

Assim como as ferramentas anteriores, a ferramenta FINT, que implementa a técnica de análise por *fan-in*, também tem como objetivo apoiar a identificação de sementes de interesses transversais em código fonte OO. Com FINT é possível verificar a quantidade de chamadas realizadas a todos os métodos existentes em um código fonte Java, além de apresentar quais métodos os invocam. FINT permite ainda filtrar métodos *getters* e *setters* e classes utilitárias, como coleções do tipo `Vector`, `List`, `ArrayList`, para que não sejam consideradas durante a análise do *fan-in*. Os resultados da análise do código fonte são apresentados em uma visão chamada “Fan-in view”. Essa visão consiste de uma estrutura de árvore de elementos chamados e chamadores (*callee-callers elements*) que podem ser ordenados pelo nome ou pelo valor da métrica *fan-in*. A partir dessa visão, o

Engenheiro de Software pode inspecionar o código fonte de cada elemento apresentado com o objetivo de descobrir sementes de um determinado interesse transversal.

Na Figura 3.5 é apresentada a visão geral da ferramenta FINT que foi executada sobre o código fonte da Figura 3.2 e Figura 3.3. Na parte (1) dessa figura é exibido o código fonte da classe `Account`. Na parte (2) - "Fan-in view"- é apresentado, após a assinatura do método `withdraw()`, o valor da métrica *fan-in* para esse método e, logo abaixo, é apresentada a listagem de todos os métodos que invocam o método `withdraw()` em algum ponto da aplicação.

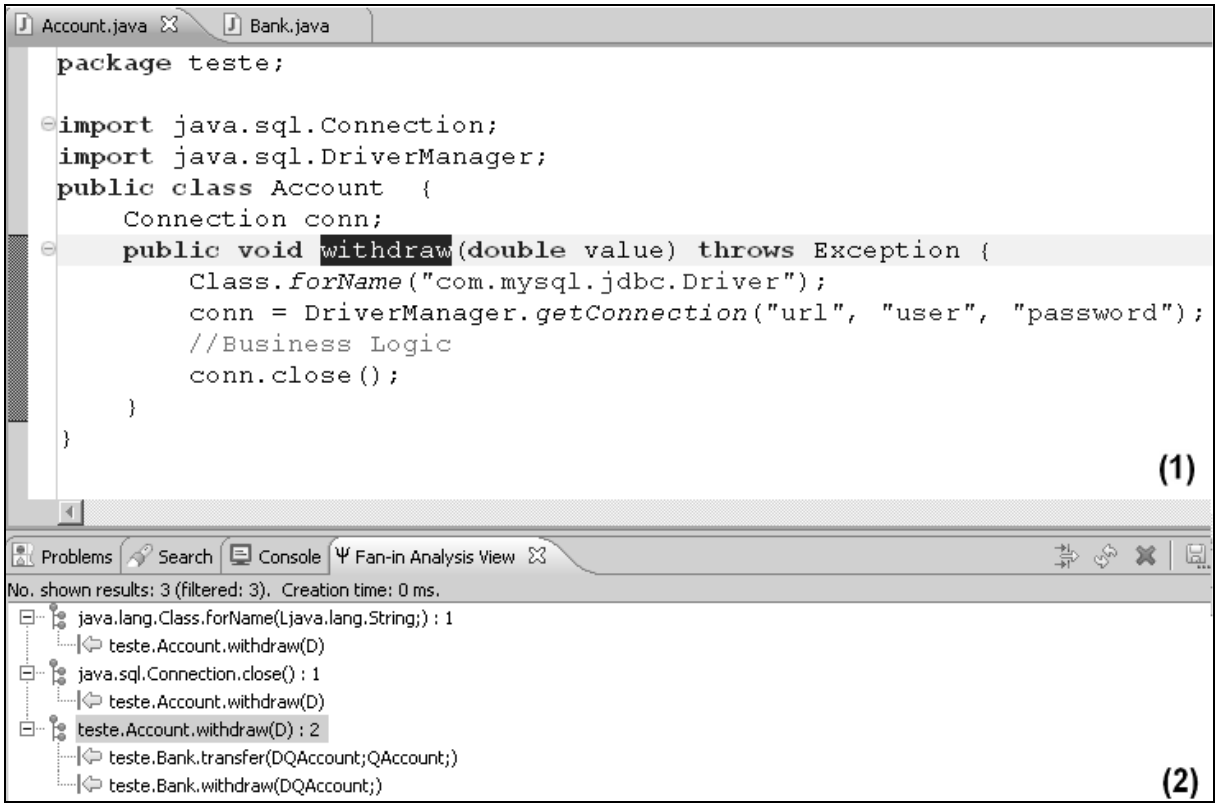

Figura 3.5 – Interface da ferramenta FINT.

3.2.4 Comparação das Técnicas para Mineração de Aspectos

A principal vantagem da técnica de mineração baseada em textos e tipos em relação às demais é que é de fácil implementação e é possível alcançar altas porcentagens de identificação de interesses em sistemas desde que as organizações mantenham nomenclatura consistente no desenvolvimento. Suas principais limitações são (Bounour *et al.*, 2006): i) depende de práticas de codificação, tais como convenções de nomes de variáveis e de métodos, que são

difíceis de garantir, especialmente em código legado; e ii) depende do cadastramento de boas regras (tipos ou textos) para identificação de sementes de interesses transversais. A formulação de uma regra que retorne bons resultados com relação à porcentagem de cobertura de identificação de interesses transversais não é uma tarefa trivial.

A técnica de análise exploratória permite realizar buscas no código fonte a partir de um conjunto de sementes pré-definidas. Como vantagem, tem-se que as buscas realizadas por essa técnica levam em consideração aspectos semânticos do código fonte, possibilitando a geração de resultados menos dependentes do estilo de codificação do software. Com relação a essa técnica, suas principais limitações são: i) de forma análoga à técnica de análise de tipos e texto, o desenvolvedor deve ter conhecimento sobre a estrutura e o funcionamento do software em análise. Além disso, necessita da utilização de sementes pré-cadastradas para identificação de interesses transversais; e ii) requer esforço elevado para identificar os interesses transversais, em consequência do alto grau de interação com o usuário exigido pela técnica.

A técnica de análise por *fan-in* tem como primeira vantagem que não é necessário cadastrar sementes e/ou regras antes da execução da técnica e como segunda vantagem que não é necessário que o sistema em análise siga padronizações e convenções de nomes para variáveis, métodos e classes. Sua principal limitação está na dificuldade de definição do valor *threshold*, que é um limitante para a métrica *fan-in*. Esse valor fornece um limite mínimo para essa métrica, de forma que apenas métodos com *fan-in* superiores a ele são considerados. Por exemplo, pode ser definido pelo Engenheiro de Software que apenas métodos com um *fan-in* acima de 5 serão considerados como candidatos à semente. O estabelecimento desse valor é dependente do sistema e do tipo de interesse transversal que está sendo analisado. Um valor muito alto para *fan-in* pode omitir métodos importantes, enquanto que um valor muito baixo pode tornar impraticável a análise dos métodos recuperados pela técnica. Por exemplo, para determinado interesse transversal em um determinado software, talvez o valor *X* seja interessante, mas o mesmo pode não ser a melhor opção para outro sistema com outro interesse. Dessa forma, desenvolvedores com menor experiência com relação ao software e aos Interesses transversais existentes nele podem ter dificuldade em definir um valor de *threshold* adequado.

As técnicas para mineração de Interesses Transversais discutidas nessa seção são comparadas na Tabela 3.1. A coluna “Depende de Convenções de Código” indica se as técnicas dependem de práticas de codificação existentes no código fonte do software em análise, como nomes de atributos, métodos, classes, entre outros. A coluna “Necessita de Informação Prévia” informa quais técnicas realizam busca por interesses transversais a partir de um conjunto de sementes e/ou regras informadas pelo usuário. A quarta coluna, “Oferece Suporte para Melhoria da Cobertura de Identificação”, indica quais técnicas oferecem apoio para definição de boas sementes/regras ou definição de melhores valores de *threshold* com o objetivo aumentar a cobertura de identificação de interesses transversais. Por último, a quinta coluna, “Facilidade de Implementação”, apresenta o nível de facilidade de implementação da técnica em questão. Esse valor foi determinado subjetivamente com base na experiência do autor dessa dissertação sobre mineração de ITs.

Tabela 3.1 – Comparativo de técnicas para mineração de Interesses Transversais.

Técnica	Depende de Convenções de Código	Necessita de Informação Prévia	Oferece Suporte para Melhoria da Cobertura de Identificação	Facilidade de Implementação
Baseada em Textos e Tipos	Sim	Sim	Não	Alto
Análise Exploratória	Sim	Sim	Não	Baixo
Baseada em *Fan-in*	Não	Não	Não	Médio

3.3 Refatorações de Interesses Transversais

Nesta seção são apresentados alguns trabalhos disponíveis na literatura sobre refatoração de interesses transversais para sistemas orientados a aspectos. Hannemann (2006) propôs uma classificação dos tipos de refatoração para software OA existentes que consistem em: i) refatorações OO convencionais adaptadas para funcionamento em sistemas orientados a aspectos (*Aspect-Aware OO Refactorings*); ii) refatorações que envolvem elementos da POA e; iii) refatorações que tratam interesses transversais. Essas três classes de refatorações são apresentadas nas Subseções 3.3.1, 3.3.2 e 3.3.3.

3.3.1 Refatorações OO Adaptadas para Sistemas Orientados a Aspectos

Refatorações OO adaptadas para sistemas orientados a aspectos consiste em refatorações convencionais aplicadas a sistemas OO, porém elas consideram que aspectos podem estar ligados aos elementos OO. Assim, se os elementos OO forem modificados, os aspectos relacionados a eles podem se tornar inconsistentes. Dessa forma, há a necessidade de conhecer os aspectos existentes no sistema para poder refatorá-los juntamente com os elementos OO. Refatorações desse tipo foram inicialmente identificadas e apresentadas por Iwamoto e Zhao (Iwamoto e Zhao, 2003) e Hanenberg *et al.* (2003).

Iwamoto e Zhao (2003) apresentaram um estudo sobre impacto que as refatorações OO propostas por Fowler *et al.* (1999) poderiam exercer sobre as construções orientadas a aspectos relacionadas aos elementos refatorados. Na Tabela 3.2 um subconjunto das refatorações de Fowler *et al.* é apresentado, indicando, em fundo cinza, as refatorações que podem ser aplicadas em um sistema OA sem que isso afete os aspectos existentes nesse sistema. As demais refatorações (em fundo branco), quando aplicadas, podem exercer impacto nos aspectos, ou seja, elas podem alterar a semântica dos aspectos relacionados aos elementos a serem refatorados.

Tabela 3.2 – Refatorações OO propostas por Fowler *et al.*, (1999).

Refatorações OO	
Add Parameter	Encapsulate Downcast
Extract Class	Extract Interface
Extract Method	Extract Superclass
Extract Subclass	Inline Class
Hide Method	Inline Temp
Inline Method	Move Field
Introduce Explaining Variable	Move Setting Method
Move Method	Pull Up Constructor
Parameterize Method	Pull Up Method
Pull Up Field	Push Down Method
Push Down Field	Replace Array with Object
Rename Method	Replace Exception with Test
Replace Conditional with Polymorphism	Replace Nested Conditional with Guarddd Clauses
Replace Magic Number with Symbolic Constant	Replace Temp with Query
Replace Parameter with Explicit Methods	Set Encapsulate Field
Remove Parameter	

De acordo com a Tabela 3.2, nota-se que apenas 3 das 31 refatorações OO listadas não possuem impacto sobre construções OA. Assim, há a necessidade de

estender as demais 28 refatorações para que elas possam ser aplicadas em sistemas OA. Por exemplo, na Figura 3.6 é apresentado o código do aspecto `LoggingAccountAspect`, responsável por capturar as chamadas ao método `withdraw()` da classe `Account` (Figura 3.2) e imprimir a mensagem "Withdrawing..." na tela, antes da execução desse método.

```
1  public aspect LoggingAccountAspect {
2   pointcut withdrawMethod(): call(void Account.withdraw(..));
3   before()withdrawMethod () {
4   System.out.println("Withdrawing...");
5   }
6  }
```

Figura 3.6 - Código fonte do Aspect `LoggingAccountAspect`.

Se a refatoração *Rename Method* (Fowler *et al.*, 1999) for aplicada ao código da classe `Account` para modificar o nome do método `withdraw()` para `getMoney()` o aspecto `LoggingAccountAspect` e seu conjunto de junção `withdrawMethod()` ficarão inconsistente e seu comportamento será alterado. Assim, quando há existência de aspectos no sistema, a aplicação de refatorações como *Rename Method* deve levar em consideração não somente as construções OO, mas também as possíveis construções OA que podem estar relacionadas aos elementos OO refatorados. Na Figura 3.7 é apresentado o código fonte do aspecto `LoggingAccountAspect` refatorado em decorrência da modificação do nome do método `withdraw()` da classe `Account` para `getMoney()`. Na linha 2 é apresentado o trecho de código refatorado após a aplicação da refatoração *Rename Method* adaptada para sistemas OA. Nesse caso, o conjunto de junção `withdrawMethod()`foi modificado para capturar corretamente o método renomeado `getMoney()` da classe `Account`.

```
1  public aspect LoggingAccountAspect {
2   pointcut withdrawMethod(): call(void Account.getMoney(..));
3   before()withdrawMethod () {
4   System.out.println("Withdrawing...");
5   }
6  }
```

Figura 3.7 - Código fonte do Aspect `LoggingAccountAspect`.

Segundo Hanenberg *et al.* (2003), a razão para a existência de conflitos entre refatorações OO e as características da POA é a própria natureza da especificação dos aspectos. Conforme foi explicado na Seção 2.2 sobre orientação a aspectos, aspectos interceptam pontos bem definidos no fluxo de execução de um programa (pontos de junção) que são especificados por conjuntos de junção em uma

linguagem OA. Como refatorações OO geralmente alteram a estrutura do código base (por exemplo, os identificadores de métodos, atributos, classes e interfaces), os conjuntos de junção podem se tornar inconsistentes.

Com base no problema descrito acima e no trabalho de Opdyke (1992), Hanenberg *et al.* (2003) propuseram três condições necessárias para se garantir a preservação do comportamento externo (ou comportamento observável) de sistemas OA após a aplicação de refatorações. As condições propostas por Opdyke foram apresentadas a fim de orientar a verificação da preservação do comportamento de um software OO após refatorações. Em Hanenberg *et al.*, o conjunto dessas condições foi estendido, acrescentando-se três novas condições para o contexto de software OA:

1) A quantidade de pontos de junção que são capturados por um dado conjunto de junção não deve ser modificado após a refatoração;
2) Os pontos de junção que são capturados por um dado conjunto de junção devem ter uma posição equivalente dentro do fluxo de controle do sistema se comparado ao estado anterior à refatoração.
3) A informação dos pontos de junção oferecida por cada conjunto de junção não deve ser diminuída.

A primeira condição é necessária para garantir que os pontos de junção capturados por aspectos antes e após a refatoração são os mesmos. A segunda condição, por sua vez, garante que os pontos de junção capturados pelos conjuntos de junção após a refatoração tenham uma posição equivalente no fluxo de controle do sistema antes da refatoração. Por fim, a terceira condição tem como objetivo assegurar que a informação relacionada aos pontos de junção de cada conjunto de junção não deve ser enfraquecida, isto é, diminuída, em relação à informação antes da refatoração, caso haja alguma alteração nestes conjuntos.

Tanto o trabalho de Iwamoto e Zhao (2003) como o de Hanenberg *et al.* (2003) não apresentam ferramentas que apoiem a aplicação das refatorações propostas o que pode tornar custosa a tarefa de refatoração para sistemas OA de média e larga escala.

3.3.2 Refatorações Orientadas a Aspectos

O segundo grupo da classificação apresentada por Hannemann (2006), “Refatorações Orientadas a Aspectos”, inclui refatorações que envolvem diretamente as construções orientadas a aspectos, podendo também modificar elementos OO que estão relacionados a essas construções.

Iwamoto e Zhao (2003) também apresentaram exemplos de refatorações OA, como *Extract Pointcut* e *Extract Advice*. Na Figura 3.8 e Figura 3.9 é apresentado um exemplo proposto por Silva (2009) da aplicação da refatoração *Extract Pointcut*. Na Figura 3.8 a classe `Impressao` possui o método `imprime()`, responsável por imprimir um determinado texto na tela, e o método `main()`, responsável pela execução da aplicação. Além disso, há um aspecto denominado `Registro` com dois adendos responsáveis por capturar a chamada ao método `imprime()` e adicionar um texto antes e após a execução desse método.

```
1  public class Impressao {
2   public void imprime (String texto) {
3    System.out.println(texto) ;
4   }
5   public static void main (String args[]) {
6    Impressao imp = new Impressao();
7    imp.imprime("Ola!") ;
8   }
9  }
10 public aspect Registro {
11  before(): call(void Impressao.imprime(String)) {
12   System.out.println("Imprimindo texto:");
13  }
14  after(): call(void Impressao.imprime(String)) {
15   System.out.println("Impressao concluída:");
16  }
17 }
```

Figura 3.8 - Classe e aspecto antes da refatoração OA *Extract Pointcut*.

A partir da Figura 3.8, nota-se que os adendos do aspecto `Registro` capturam o mesmo ponto de junção (chamada ao método `imprime()` da classe `Impressao`), com a única diferença de que um adendo é executado antes e o outro após a chamada do método `imprime()`. Nesse caso, a refatoração *Extract Pointcut* pode ser aplicada para extrair um conjunto de junção a partir do ponto de junção utilizado nos dois adendos. Como vantagem disso, tem-se que o conjunto de junção criado poderá ser reutilizado em demais adendos que possam vir a ser criados. A Figura 3.9 exemplifica o resultado da aplicação da refatoração *Extract Pointcut* sobre

o código Figura 3.8. Um conjunto de junção denominado `chamadaImpressao` foi extraído e é utilizado nos adendos `before` e `after` do aspecto `Registro`.

```
 1 public aspect Registro {
 2  pointcut chamadaImpressao(): call(void Impressao.imprime(Str
 3   ing)) ;
 4  before(): chamadaImpressao() {
 5   System.out.println("Imprimindo texto:");
 6  }
 7  after():chamadaImpressao() {
 8   System.out.println("Impressao concluída:");
 9  }
10 }
```

Figura 3.9 - Classe e aspecto após a refatoração OA *Extract Pointcut*.

Como é possível observar na Figura 3.8 e Figura 3.9, as refatorações OA geralmente modificam construções orientadas a aspectos, podendo ou não modificar elementos da programação orientada a objetos. Nesse exemplo, o código da classe OO `Impressao` não foi modificado, por isso, ele foi omitido na Figura 3.9.

Complementando o trabalho de Iwamoto e Zhao (2003), Hanenberg *et al.* (2003) descreveram mais três refatorações OA: *Extract Advice*, *Extract Introduction* e *Separate Pointcut*. Posteriormente, Garcia *et al.* (2004) apresentaram um conjunto de dez refatorações OA, baseadas nas refatorações propostas por Fowler *et al.* (1999): *Extract Field to Aspect, Extract Method to Aspect, Extract Code to Advice, Extract Pointcut Definition, Collapse Pointcut Definition, Rename Pointcut, Rename Aspect, Pullup Advice/Pointcut, Collapse Aspect Hierarchy, Inline Pointcut Definition*. Cada refatoração proposta por Garcia *et al.*, (2004) inclui a motivação para aplicação da refatoração, os passos para sua realização e um exemplo ilustrativo. Segundo Garcia *et al.*, (2004) uma ferramenta para apoiar a aplicação das refatorações foi desenvolvida, entretanto, o desenvolvimento de tal ferramenta foi descontinuado.

Binkley *et al.* (2006) apresentaram uma abordagem de refatorações apoiada por ferramenta. Essa proposta inclui a ferramenta AOP-Migrator e ainda seis refatorações OA: *Extract Beginning/End of Method/Handler*, *Extract Before/After Call*, *Extract Conditional*, *Pre Return*, *Extract Wrapper* e *Extract Exception Handling*. AOP-Migrator é uma ferramenta que suporta migração semiautomática de um sistema escrito em Java para um sistema em AspectJ. A coleção de refatorações desenvolvidas com AOP-Migrator foi integrada pelos autores ao ambiente Eclipse (Eclipse, 2011). Entretanto essa ferramenta foi descontinuada e sua execução

depende de versões antigas do ambiente Eclipse, da linguagem Java e do *plug-in* de aspecto AJDT.

Laddad (2006) propõe diversas refatorações OA para criação e reestruturação de aspectos. Essas refatorações encontram-se voltadas para situações em que se utilizam estruturas e técnicas recorrentes da programação orientada a objetos, como tratamento de exceções, implementação de interfaces, entre outras. As refatorações propostas por Laddad são: *Extract Method Calls*, *Extract Exception Handling*, *Extract Concurrency Control*, *Extract Worker Object Creation*, *Replace Argument Trickle by Wormhole*, *Extract Interface Implementation*, *Replace Override with Advice*, *Extract Lazy Initialization*, *Extract Contract Enforcement*. Também não foram encontradas propostas de ferramentas que apoiem a aplicação dessas refatorações.

Por fim, Monteiro e Fernandes (2006) apresentam um catálogo de refatorações de elementos OA subdividido em dois grupos: i) refatorações que extraem aspectos a partir do código de sistemas OO; e ii) refatorações oriundas de melhorias necessárias dentro das estruturas dos aspectos. Os autores descrevem essas refatorações de maneira similar à descrição feita para as refatorações OO por Fowler *et al.* (1999). Na Tabela 3.3 são listadas as 27 refatorações do catálogo organizadas em seus respectivos grupos. A tradução dos nomes dessas refatorações são apresentadas em Silva (2009). Cada refatoração proposta por Monteiro e Fernandes inclui:

- nome, identificando a refatoração;
- a situação típica, explicando as situações passíveis de aplicar a refatoração;
- a ação recomendada, resumindo a reestruturação realizada pela refatoração;
- a motivação, enfatizando e focando na necessidade de aplicação;
- um exemplo, para facilitar o entendimento e ilustrar uma possível aplicação;
- e, por fim, a mecânica, que reúne uma sequência de passos para guiar a realização da refatoração.

Embora não esteja baseado em definições formais, nem apresente propostas de apoios computacionais, a estrutura do catálogo de Monteiro e Fernandes (2006)

possui uma facilidade de compreensão maior que as demais refatorações discutidas nesse capítulo.

Tabela 3.3 – Refatorações do catálogo de Monteiro e Fernandes (2006).

Grupo	Nome da Refatoração	Tradução (Silva, 2009)
Refatorações para Extração de Aspectos	*Change abstract class to interface.*	Trocar classe abstrata por interface.
	Extract feature into aspect.	Extrair funcionalidade para aspecto.
	Extract fragment into advice.	Extrair fragmento para adendo.
	Extract inner class to stand-alone.	Extrair classe interna para autônoma.
	Inline class within aspect.	Internalizar classe em aspecto.
	Inline interface within aspect.	Internalizar interface em aspecto.
	Move field from class to intertype.	Mover atributo de classe para intertipo.
	Move method from class to intertype.	Mover método de classe para intertipo
	Replace implements with declare parents.	Trocar *implements* por *declare parents*.
	Split abstract class into aspect and interface.	Dividir classe abstrata em aspecto e interface.
Refatorações para Reestruturação Interna de Aspectos	*Extend marker interface with signature.*	Estender interface marcadora por assinatura.
	Generalise target type with marker interface.	Generalizar tipo alvo com interface marcadora.
	Introduce aspect protection.	Introduzir proteção de aspecto.
	Replace intertype field with aspect map.	Trocar campo intertipo por mapeamento no aspecto.
	Replace intertype method with aspect method.	Trocar método intertipo por método no aspecto.
	Tidy up internal aspect structure.	Limpar estrutura interna de aspectos.
	Extract superaspect.	Extrair superaspecto.
	Pull up advice.	Generalizar adendo.
	Pull up declare parents.	Generalizar *declare parents*.
	Pull up intertype declaration.	Generalizar declaração intertipo.
	Pull up marker interface.	Generalizar interface marcadora.
	Pull up pointcut.	Generalizar conjunto de junção.
	Push down advice.	Especializar adendo.
	Push down declare parents.	Especializar *declare parents*.
	Push down intertype declaration.	Especializar declaração intertipo.
	Push down marker interface.	Especializar interface marcadora.
	Push down pointcut.	Especializar conjunto de junção.

3.3.3 Refatorações para Interesses Transversais

O terceiro e último grupo de refatorações segundo a classificação proposta em Hannemann (2006) consiste na refatoração de interesses transversais por meio de construções OA. Na Figura 3.10 é apresentado o resultado da aplicação de uma refatoração para modularização de um interesse transversal hipotético (Silva, 2009).

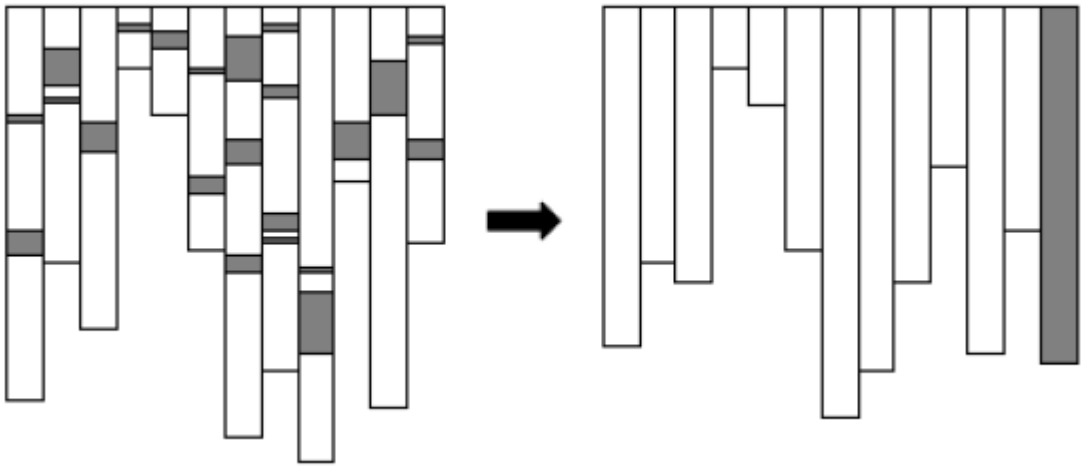

Figura 3.10 - Um interesse transversal não-modularizado extraído para um aspecto (Silva, 2009).

As barras verticais da figura são componentes de um sistema e as partes destacadas em cinza correspondem aos trechos de código responsáveis pela implementação de um interesse transversal hipotético. Percebe-se que uma implementação não-modularizada é representada pelas barras da esquerda da Figura 3.10, pois há trechos de código, relacionados ao interesse, espalhados em todo o sistema. Uma possível refatoração para extração e modularização deste interesse transversal hipotético, com a utilização de aspectos, poderia obter a implementação do sistema representada pelas barras da parte direita da Figura 3.10, onde a realização do interesse aparece completamente concentrada em um componente (um aspecto).

Uma refatoração para interesses transversais deve levar em consideração todos os elementos e relacionamentos que compõem e participam do interesse transversal, podendo assim envolver diversos componentes (classes, aspectos, interfaces, entre outros) e seus respectivos membros. Tais refatorações possuem um nível de granularidade maior, pois, em apenas uma refatoração, transformações são realizadas em um número maior de elementos. Isto deve-se ao fato de que um interesse transversal geralmente está manifestado em diversos componentes.

Esse é o tipo de refatoração tratado nesta dissertação. Os interesses transversais, representados por meio de modelos de classe OO anotados, são refatorados a fim de se obter um modelo OA no qual os interesses transversais são modularizados em aspecto. Para isso, diversos elementos e relacionamentos especificados nos modelos de classes OO anotados podem ser alterados após a aplicação das refatorações propostas.

Na literatura, há pelo menos três propostas que apresentam refatorações deste grupo, são elas: i) o trabalho de Hannemann *et al.* (2005), que apresenta refatorações baseadas em papéis; ii) o trabalho de Marin *et al.* (2005) que propõe refatorações baseadas em tipos de interesses transversais e iii) o trabalho de Silva *et al.*, (2009), que propõe refatorações dirigidas a sintomas de interesses transversais.

A proposta de Hannemann *et al.* (2005) descreve as refatorações em termos de papéis e seus elementos, independentemente da implementação do sistema. A ideia é que os passos da refatoração sejam os mesmos para qualquer elemento do sistema que realiza um determinado papel em um interesse transversal. Dessa forma, um mapeamento deve ocorrer entre os papéis e os elementos concretos da implementação do sistema e os passos de refatoração, descritos para os papéis, devem ser aplicados aos elementos do sistema que foram mapeados.

Hannemann *et al.* apresentaram um conjunto de quatro atividades que devem ser seguidas para a aplicação de sua abordagem:

1. **Seleção da refatoração para interesse transversal**: o desenvolvedor deve selecionar, a partir de uma lista, a refatoração apropriada para o interesse transversal a ser modularizado;
2. **Realização do mapeamento**: nesta atividade, uma ferramenta deve auxiliar o desenvolvedor no mapeamento dos papéis para elementos concretos do sistema;
3. **Planejamento da refatoração**: nesta etapa, o desenvolvedor deve verificar em quais situações deve-se ou não aplicar refatorações dependendo do impacto a ser obtido como resultado. Esta atividade também deve ser auxiliada por uma ferramenta. Por exemplo, a ferramenta pode alertar o desenvolvedor se um novo aspecto a ser criado possuirá um nome que colide com o de outra entidade do sistema;

4. **Execução**: após o planejamento da refatoração, uma ferramenta poderá auxiliar o desenvolvedor na transformação dos elementos do código para obter o resultado desejado, seguindo as definições realizadas nas etapas anteriores.

Ainda no contexto de refatorações para modularização de interesses transversais, Marin *et al.* (2005) apresentaram a proposta de "Refatoração Baseada em Tipos de Interesses Transversais". Segundo os autores, um tipo de interesse transversal é constituído de três propriedades: i) uma intenção; ii) um idioma de implementação; e iii) um mecanismo de linguagem orientada a aspectos para representá-lo concretamente. Interesses transversais concretos encontrados nos sistemas são instâncias dos tipos definidos pelos autores.

Por último, Figueiredo *et al.* (2009b) apresentam treze sintomas nomeados através de metáforas. Sintomas são configurações de entrelaçamento/espalhamento de um determinado interesse que podem classificá-lo como um interesse transversal ou não. Metáforas são nomes sugestivos dados a determinados sintomas que visam a facilitar o entendimento desses sintomas por parte do Engenheiro de Software. As metáforas são escolhidas com base no cenário de entrelaçamento/espalhamento de um determinado interesse. Segundo Figueiredo *et al.*, dos treze sintomas apresentados em seu trabalho, dois foram inicialmente propostos em Ducasse *et al.*, (2006), *Black Sheep* e *Octopus*. Os sintomas são agrupados em quatro categorias que foram reunidas de acordo com diferentes maneiras que interesses podem se organizar: sintomas com formas transversais, sintomas relativos à herança, sintomas relativos ao acoplamento e sintomas com outros problemas de modularidade.

Na Tabela 3.4 são apresentadas duas das metáforas propostas por Silva *et al.* e algumas heurísticas utilizadas para sua identificação. A metáfora *Black Sheep* consiste em uma categoria de interesses transversais que afetam poucos pontos do software. *Octopus* representa a categoria em que os interesses transversais se encontram parcialmente modularizados em uma ou mais classes, mas que também entrecortam outras classes do sistema.

As heurísticas propostas para identificação dessas metáforas baseiam-se nas métricas CA (*Concern Attributes*) e CO (*Concern Operations*) que contam, respectivamente, o número de atributos e métodos que são afetados por um determinado interesse (Figueiredo *et al.*, 2009a).

Tabela 3.4 –Metáforas de Interesses Transversais e Heurísticas para sua Identificação (Ducasse *et al*., 2006).

Classificação	Descrição da Heurística
Black Sheep	Um interesse encontra-se implementado por poucos atributos e métodos em todas as classes onde ele aparece.
Octopus	Um interesse encontra-se bem modularizado em, pelo menos, uma classe e afeta poucos atributos e métodos das outras classes.

A primeira heurística classifica um interesse como *Black Sheep* se em todas as classes nas quais esse interesse está implementado poucos atributos e métodos (menos de 33%) são afetados por ele. Segundo os autores desse trabalho, Lanza *et al.* (2006) sugerem o uso dos valores limites para heurísticas baseadas em métricas como 0.33 (1/3), 0.5 (1/2) e 0.67 (2/3). Além disso, esses valores tentam ser o mais significante possível pela definição de *Octopus* e *Black Sheep* (Ducasse *et al.*, 2006).

De modo análogo, a segunda heurística apresentada na Tabela 3.4 verifica se um interesse transversal não classificado como *Black Sheep* é um *Octopus*. Segundo esta heurística, um interesse é classificado como *Octopus* se de todas as classes que ele entrecorta (i) muitos atributos e métodos são afetados (corpo do *Octopus*); ou (ii) poucos atributos e métodos são afetados (tentáculos do *Octopus*). Uma classe pertence ao corpo do *Octopus* quando a porcentagem de membros (atributos e métodos) afetados pelo interesse é maior que 67%. Da mesma forma, uma classe pertence a um tentáculo quando a porcentagem de atributos e métodos afetados é menor que 33%.

Para exemplificar as heurísticas sugeridas por Silva *et al.* (2009), na Figura 3.11 é apresentado o código fonte Java da classe `PrinterSingleton`, que é um exemplo de um interesse *Black Sheep* que implementa o padrão *Singleton*. Essa instância específica do padrão *Singleton* foi classificada como *Black Sheep*, pois apenas um método, `instance()`, e um atributo, `single`, fazem parte da implementação do interesse.

```
public class PrinterSingleton {
  protected static int objects = 0;
  protected static PrinterSingleton single;
  protected int id;

  protected PrinterSingleton() {
     id = ++objects;
  }

  public static PrinterSingleton instance() {
   if(single==null) single=new PrinterSingleton();
   return single;
  }
  public void print() {
   System.out.println("My ID is "+id);
  }
}
```

Figura 3.11 - Padrão *Singleton*: um interesse do tipo *Black Sheep* (Silva *et al.*, 2009).

Na Figura 3.12 é apresentado um diagrama de classes que é uma instância do padrão *Observer*.

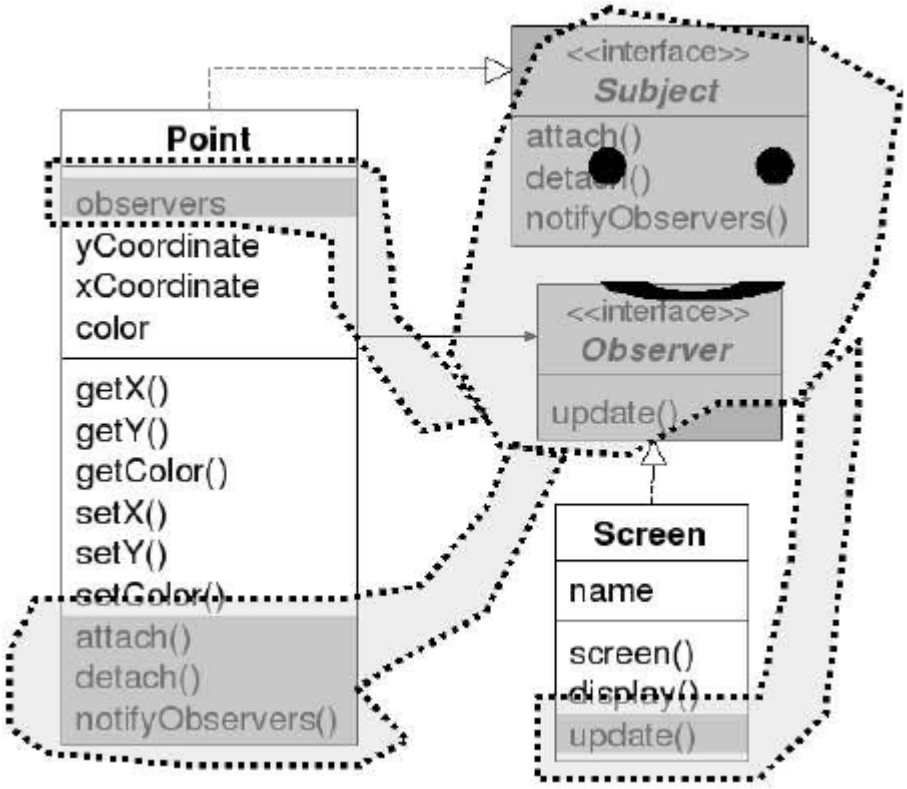

Figura 3.12 - Padrão *Observer*: um interesse do tipo *Octopus* (Silva *et al.*, 2009).

As duas interfaces, `Subject` e `Observer`, são completamente afetadas pelo interesse em questão. Por outro lado, as classes `Point` e `Screen` possuem poucos elementos afetados. De acordo as com as heurísticas apresentadas anteriormente

(Tabela 3.4), esse interesse é classificado como *Octopus*. Dessa forma, as interfaces `Subject` e `Observer` representam o corpo do *Octopus* e as classes `Point` e `Screen` representam seus tentáculos.

Além da classificação dos sintomas e das heurísticas para sua identificação, Silva *et al.* (2009) propuseram refatorações orientadas a aspectos propostas a partir da abordagem de identificação e classificação de interesses transversais baseados em sintomas. Tais refatorações possuem o objetivo de apoiar a modularização de interesses transversais e encontram-se definidas em um nível de granularidade mais elevado em relação a outras refatorações de granularidade baixa, como as apresentadas nas Subseções 3.3.1 e 3.3.2.

Por exemplo, a refatoração de interesses criado por Silva para a metáfora *Black Sheep*, apresenta o seguinte mecanismo:

1. Identificar as partes da *Black Sheep*: identificar as partes dos componentes do interesse dedicadas à realização do mesmo.
2. Passos de refatoração para separar as partes identificadas:
   a) Se houver atributos relacionados ao interesse, aplicar “Mover atributo de classe para intertipo”.
   b) Se houver operações relacionadas ao interesse, aplicar “Mover método de classe para intertipo”.
   c) Se houver trechos de código relacionados ao interesse aplicar “Extrair fragmento para adendo”.

Aplicando essa refatoração no exemplo da Figura 3.11, o aspecto apresentado na Figura 3.13 é obtido.

```
 1 public aspect PrinterAspect {
 2  protected static PrinterSingleton PrinterSingleton.single;
 3  public static PrinterSingleton PrinterSingleton.instance() {
 4   if (single == null) single = new PrinterSingleton();
 5   return single;
 6  }
10 }
```

Figura 3.13 – Aspecto Criado após a Aplicação da Refatoração para metáfora *Black Sheep*.

3.3.4 Discussão

Diante da variedade de trabalhos e, principalmente, de refatorações é possível notar que as refatorações são apresentadas utilizando terminologias

distintas com diferentes formas de descrição. Algumas apresentam apenas uma breve definição textual enquanto outras apresentam um passo a passo detalhado para aplicação incluindo exemplos. A falta de uma terminologia consistente e uma descrição padronizada pode prejudicar o entendimento e a aplicação das refatorações. Por exemplo, a refatoração "*Extract Pointcut*" proposta por Iwamoto e Zhao (2003) é equivalente a "*Extract Pointcut Definition*" (Garcia *et al.*, 2004); a refatoração "Extract Begining/End of Method/Handler" (Binkley *et al.*, 2006) é englobada pela definição de "*Extract Code to Advice*" (Garcia *et al.*, 2004).

Já em relação às propostas de refatorações para interesses transversais nota-se que ambas as propostas levantadas nos estudos conduzidos por Hannemann *et al.* (2005) e Marin *et al.* (2005) preocupam-se em agrupar interesses com estruturas similares, a fim de criar uma categorização. Entretanto, conforme observado por Silva *et al.* (2009), a proposta de Hannemann *et al.* (2005), mesmo envolvendo interesses transversais com papéis abstratos, continua tendo um nível de abstração pouco elevado. Por exemplo, o interesse relacionado ao padrão de projeto *Observer* (Gamma *et al.*, 1995) é abstrato no sentido de existirem diversas implementações concretas possíveis para ele. Entretanto, a estrutura do padrão *Observer* pode ser semelhante à estrutura de outros tipos de interesses que não são efetivamente instâncias desse padrão. Por exemplo, o padrão *Mediator* (Gamma *et al.*, 1995) pode ser estruturalmente igual ao *Observer*.

Quanto à segunda proposta para refatoração de interesses transversais (Marin *et al.*, 2005), a classificação baseada em tipos é dependente do idioma de implementação. Por exemplo, um tipo de interesse que é tratado por Marin *et al.* é denominado "Cláusula Declarativa Throws". Esse interesse é intrinsecamente dependente do idioma de implementação, pois a própria definição depende da existência de cláusulas *throws* na linguagem de programação OO. Portanto, nessa abordagem o nível de abstração encontra-se baixo, pois os tipos de interesses ainda dependem de idiomas de implementação. Além disso, de forma análoga ao que acontece na abordagem de Hannemann *et al.*, é possível encontrar tipos diferentes de interesses de acordo com a classificação de Marin *et al.* que possuem configuração estrutural igual.

Por fim, Silva *et al.* (2009) dá um passo importante no contexto de refatorações de interesses transversais com base nos cenários de entrelaçamento/espalhamento de interesses transversais (sintomas). A abordagem

proposta pelos autores consiste na aplicação de refatorações genéricas para qualquer tipo de interesse que apresente um determinado sintoma. Entretanto, essa estratégia pode produzir resultados inadequados dependendo do tipo de interesse implementado na aplicação. Por exemplo, o trecho de código apresentado na Figura 3.14 corresponde a uma aplicação afetada por dois diferentes tipos de interesses transversais, *logging* e o padrão de projeto *Singleton* (Gamma *et al.*, 1995). A aplicação em questão é responsável por registrar as vendas de itens de produtos em um arquivo de *log* do sistema e imprimir uma cópia desse item de venda na tela do computador.

```
 1 public class Item {
 2  private Logger log;
 3  private File flog;
 4  private Printer pr;
 5  ...
 6  public void saveItem() {
 7  log.entryLog("Item saved!");
 8  pr = Printer.instance();
 9  pr.print(this);
10  }
11  public void entryLog(String msg) {
12  flog.append(msg);
13  }
14  ...
15 }
16 public class Printer {
17  private static Printer pr;
18  private Printer() {;}
19  public static Printer instance() {
20  if (pr == null) return new Printer(); else return pr;
21  }
22  public void print(Item it) {
23   System.out.println(it);
24  }
25  ...
26 }
```

Figura 3.14 – Aplicação afetada pelos Interesses de *Logging* e pelo Padrão *Singleton*.

Na Figura 3.14, a parte destacada por linha tracejada representa os trechos de código correspondentes ao interesse de *logging* e a parte destacada em cinza correspondente à implementação do padrão *Singleton*. Para facilitar sua visualização, os demais métodos das classes `Item` e `Printer` foram omitidos da Figura 3.14. De acordo com as heurísticas propostas por Silva *et al.*, os dois interesses existentes na aplicação são classificados como *Black Sheep*, pois ambos interesses afetam poucos (menos de 33%) pontos da aplicação.

Ao aplicar a refatoração para a metáfora *Black Sheep* proposta por Silva *et al.* e apresentada na Subseção 3.3.3, o código da Figura 3.14 refatorado é apresentado na Figura 3.15.

```
 1 Public class Item {
 2  public void saveItem() {
 3  pr = Printer.instance();
 4  pr.print(this);
 5  }
 6  ...
 7 }
 8 public class Printer {
 9  private Printer() {;}
10  public void print(Item it) {
11   System.out.println(it);
12  }
13  ...
14 }
15 public aspect LoggingAspect {
16  private Logger Item.log;
17  private File Item.flog;
18  public void Item.entryLog(String msg) {
19   flog.append(msg);
20  }
21  pointcut log(): call(Printer.saveItem());
22  before(): log() {
23  log.entryLog("Item saved!");
24  }
25 }
26 public aspect SingletonAspect {
27  protected static Printer Printer.single;
28  public static Printer Printer.instance() {
29   if (single == null) single = new Printer ();
30   return single;
31  }
32 }
```

Figura 3.15 – Código obtido com a aplicação da refatoração para interesses do tipo Black Sheep.

Percebe-se que para o interesse de *logging*, o mecanismo da refatoração foi suficiente para modularização desse interesse. Porém, para modularização do padrão *Singleton*, os passos da refatoração proposta por Silva *et al.* não são suficientes e a solução obtida é parcial. Por exemplo, na linha 3 da Figura 3.15, uma instância da classe `Printer` é obtida por meio do método `instance()` e é utilizada no comando posterior (linha 4).

Nesse caso, uma solução adequada para modularização do padrão *Singleton* proposta por Hannemann e Kiczales (2002) é tornar o construtor da classe `Printer` público, modificar o trecho de código da linha 9 para que a instância dessa classe seja obtida pelo seu construtor. Finalmente, um conjunto de junção para capturar

chamadas ao construtor da classe `Printer` e um adendo do tipo *around* devem ser criados para que seja possível capturar os pontos da aplicação que necessitam de uma instância dessa classe e retorná-la. A refatoração proposta por Silva *et al.* não foi suficiente para modularização do padrão *Singleton* porque os sintomas e refatorações utilizados são independentes do tipo de interesse transversal e assim, as soluções geralmente precisam ser refinadas pelo Engenheiro de Software.

Como pode-se observar nas subseções anteriores, todos os trabalhos mencionados tratam da refatoração de software OO para OA em nível de código. Apesar de alguns deles utilizarem conceitos de mais alto nível para aplicação das refatorações como é o caso dos trabalhos de Hannemann *et al.* (2005) e Silva *et al.* (2009) ainda assim as refatorações são aplicadas sobre o código fonte legado OO e a saída é um código fonte OA.

Após pesquisa na literatura especializada, notou-se escassez de trabalhos relacionados à refatoração de modelos OA. Boger *et al.* (2002) desenvolveram um *plug-in* para a ferramenta CASE ArgoUML que apoia a refatoração de modelos UML. Com ele é possível refatorar diagramas de classe, de estados e de atividades permitindo ao usuário aplicar refatorações que não são simples de se aplicar em nível de código. Van Gorp *et al.* (2003) propuseram um perfil UML para expressar pré e pós-condições de refatoração de código fonte utilizando restrições OCL (OCL, 2010). O perfil proposto permite que uma ferramenta CASE: i) verifique pré e pós-condições para composição de sequências de refatorações; e ii) use o mecanismo de consulta OCL para detectar *bad smells*. Além desses, há trabalhos que tratam do uso de refatorações de modelos para introdução de Padrões de Projeto (*Design Patterns*) (Gamma *et al.*, 1995) em software orientado a objetos (Tokuda e Batory, 1995; Genssler *et al.*, 1998). Embora esses trabalhos tenham enfoque na refatoração de modelos, nenhum deles apresenta estudos que consideram, especificamente, modelos orientados a aspectos.

O diferencial do trabalho apresentado nesta dissertação em relação aos demais é a proposta de construção de um modelo OA, considerando modelos de classes OO anotado com estereótipos representando interesses transversais. Além disso, para evitar problemas como os que foram mencionados para os trabalhos de Marin *et al.* (2005), Hanneman *et al.* (2005) e Silva *et al.* (2009), dois conjuntos de refatorações são propostos. Um deles possui refatorações independentes do tipo de interesse implementado no software (refatorações genéricas) o outro consiste em

refatorações específicas para determinados tipos de interesses transversais, como persistência, *logging*, entre outros.

3.4 Considerações Finais

Este capítulo apresentou algumas abordagens para mineração e refatoração de interesses transversais. Como foi visto, mineração de interesses transversais consiste em descobrir interesses que apresentam características de espalhamento e entrelaçamento e que são potenciais candidatos para serem implementados em aspectos. Neste capítulo foram comentadas as abordagens de análise baseada em tipos e texto, análise exploratória e análise por fan-in, evidenciando suas vantagens e limitações. Além disso, para as técnicas de análise exploratória e análise por fan-in, ferramentas computacionais que implementam essas técnicas (FEAT e FINT) foram apresentadas sucintamente. Esse assunto é importante, pois a mineração de interesses transversais é uma atividade fundamental para aplicação das refatorações de interesses transversais.

As abordagens para refatoração de interesse transversais têm como objetivo obter um software com melhor modularização, encapsulando interesses transversais em aspectos, a partir de um código não-aspectual - no contexto deste trabalho, a partir de um código OO - sem, no entanto, alterar o comportamento original do software. Algumas abordagens são criadas para que a aplicação de refatorações OO não altere o comportamento dos aspectos existentes na aplicação; outras estendem as refatorações OO para serem aplicadas em construções OA; por último, há algumas abordagens que visam a refatoração de interesses transversais para aspectos como um todo, ou seja, elas envolvem passos mais complexos e podem lidar com diversos elementos de software, como classes, interfaces, entre outros.

Das abordagens apresentadas, todas elas lidam com refatorações de interesses transversais em nível de código, isto é, recebem como entrada um código OO e podem produzir como saída um código OA. Entretanto, essas abordagens, além de serem específicas de plataforma de implementação, podem levar a resultados indesejados. Por isso, nesta dissertação propõem-se a utilização de refatorações em nível de modelos. Não foi encontrado na literatura trabalhos que

proponham a aplicação de refatorações em modelos OO para obtenção de modelos OA. Esse assunto será melhor discutido no Capítulo 4.

Capítulo 4

REFATORAÇÃO DE MODELOS DE CLASSES ANOTADOS COM INDÍCIOS DE ITS

4.1 Considerações Iniciais

Refatorações são tipicamente definidas como técnicas para promover melhorias na estrutura interna de um software sem afetar seu comportamento (Fowler *et al.*, 1999) e, originalmente, são aplicadas ao código fonte. No contexto deste trabalho, essa definição é estendida para que modelos possam ser utilizados. Isso está sendo feito para que seja possível a obtenção de modelos de classes OA a partir de modelos de classes OO anotados com indícios de interesses transversais representados por estereótipos aplicados aos seus elementos (classes, atributos e métodos).

Na Seção 4.2 são discutidas as justificativas para criação e aplicação das refatorações em modelos de classes OO anotados. Na Seção 4.3 é apresentado um conjunto de refatorações genéricas, ou seja, refatorações que podem ser aplicadas a qualquer tipo de interesses transversal e alguns exemplos de aplicação dessas refatorações em aplicações simples. Na Seção 4.4 são apresentadas as refatorações desenvolvidas especificamente para modularização dos interesses transversais de persistência, *logging* e dos padrões *Singleton* e *Observer* (Gamma *et al.*, 1995) com exemplos para ilustrar sua aplicação. As considerações finais estão na Seção 4.5.

4.2 Justificativa para Aplicação de Refatorações em Modelos

Alguns dos principais motivos para criação e aplicação de refatorações em nível de modelos, uma vez que há vários trabalhos na literatura (Silva *et al.*, 2009; Murphy *et al.*, 2009; Monteiro e Fernandes, 2006; Hannemann *et al.*, 2005; Marin *et al.*, 2005; Iwamoto e Zhao, 2003) que apresentam refatorações de interesses transversais em nível de código, são:

- A utilização de modelos de classes anotados proporciona redução ou controle da complexidade do software, podendo melhorar a tomada de decisão por parte do Engenheiro de Software para modularização dos interesses transversais existentes nesse software.
- Os modelos de classes anotados podem ajudar a identificar e evitar determinados vícios (práticas não recomendadas) de programação.
- Os modelos gerados durante o processo de refatoração do software servem como documentação para o software OO e OA.
- As refatorações propostas são independentes de plataforma de implementação, o que não acontece com refatorações em nível de código.

O primeiro benefício é inerente à própria modelagem de software que proporciona a redução ou controle da complexidade do software. Por exemplo, ao olhar apenas para o trecho de código da classe `Account` (Figura 4.1) o Engenheiro de Software detecta que há pontos relacionados ao interesse de persistência (destaques em cinza). Com base nessa informação ele cria um aspecto para modularizar esse interesse, interceptando os métodos afetados e adicionando o comportamento transversal ao alcançar a execução desses métodos, como apresentado na Figura 4.2.

```
1  public class Account {
2   double balance = 0;
3   Connection conn;
4   public void withdraw(double value) throws Exception {
5   Class.forName(“com.mysql.jdbc.Driver”);
6   conn = DriverManager.getConnection(“url”, “user”, “pass”);
7   balance -= value;
8   conn.close();
9   }
10 }
```

Figura 4.1 - Trecho de Código da Classe `Account`.

```
1  public aspect Persistence {
2   Connection conn;
3   pointcut affectedMethods(): call(void Account.withdraw(..));
4   before() affectedMethods() {
5   Class.forName("com.mysql.jdbc.Driver");
6   conn = DriverManager.getConnection("url", "user", "pass");
7   }
8   after() affectedMethods() {
9   conn.close()
10  }
11 }
```

Figura 4.2 - Código do Aspecto Persistence Criado para Modularização do Interesse Transversal de Persistência.

Para criação desse aspecto foi considerado apenas o fato de que a classe Account encontra-se afetada pelo interesse de persistência, porém, ao analisar as demais classes do sistema (`Customer` e `Bank`), percebe-se que alguns métodos também são afetados por esse interesse. Assim, considerando que o Engenheiro de Software procura aplicar boas práticas de desenvolvimento de software OA, ele deverá modificar sua estratégia de modularização inicial e criar um aspecto abstrato, que possui um conjunto de junção abstrato para interceptar métodos afetados pelo interesse de persistência. Posteriormente, ele deve implementar um aspecto concreto que estende o criado anteriormente e especificar os pontos da aplicação que serão afetados pela persistência.

Observando, entretanto, o modelo de classes da Figura 4.3 é possível reconhecer as três classes `Account`, `Bank` e `Customer` dessa aplicação, responsáveis por realizar operações básicas em contas bancárias: saque, depósito, transferência, entre outros. Além disso, os estereótipos utilizados nesse modelo indicam a presença de determinados interesses transversais espalhados e/ou entrelaçados pelos demais módulos do software. As classes `Account`, `Bank` e `Customer` são estereotipadas com `<<Persistence>>`, pois são afetadas pelo interesse de persistência.

O segundo benefício diz respeito à utilização de modelos de classes OO anotados para auxiliar na identificação e remoção de determinados vícios (práticas não recomendadas) de programação. Por exemplo, com a utilização de modelos de classes OA não é possível criar conjuntos de junção anônimos, considerados uma prática não recomendada no desenvolvimento de software OA.

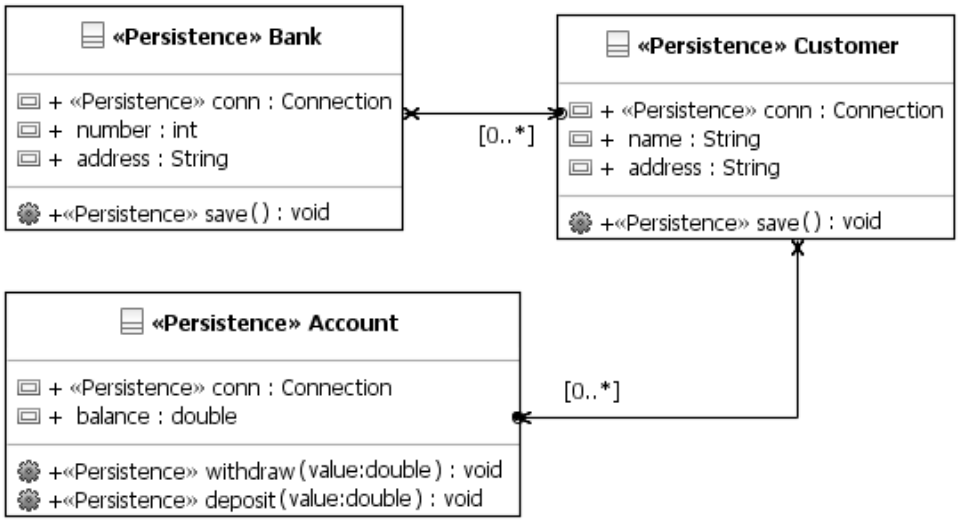

Figura 4.3 - Modelo de Classes Estereotipado.

Outro exemplo mais completo refere-se às classes apresentadas na Figura 4.3, que o desenvolvedor criou atributos do tipo `Connection` em todas elas. Esses atributos são utilizados dentro do corpo dos métodos que representam as regras de negócio para abertura e fechamento de conexões com o Banco de Dados (BD). Como já é bem difundido na literatura, essa não é uma boa prática de programação, pois: i) há um entrelaçamento/espalhamento significante de código de persistência dentro das classes de negócio do sistema; ii) a abertura de conexão com o BD é uma tarefa custosa e que consome recursos de tempo, processamento e memória. iii) a manutenção do software pode ser comprometida em consequência dos problemas de entrelaçamento/espalhamento de código apresentados anteriormente. Uma das boas prática para implementação do controle de conexões com BD sugere a utilização de um *pool* de conexões, ou nos casos mais simples, da aplicação do padrão *Singleton* (Gamma *et al.*, 1995) para que haja apenas uma classe responsável pela conexão com o BD. Esse problema poderia ser facilmente identificado a partir do modelo de classes da Figura 4.3, ao observar que todas as classes de negócio apresentam um atributo `Connection` para controle de conexão. Dessa forma, o Engenheiro de Software pode optar por refatorar o código OO antes de iniciar o processo de reengenharia do software, ou mesmo aplicar refatorações OA conscientes do problema a fim de evitar sua replicação no código do novo sistema.

Por último, tem-se a questão da documentação do software OO e OA. Em se tratando de código legado, sabe-se que muitas vezes o único artefato disponível

desses sistemas é o código fonte com todas as regras de negócio embutidas. Ao aplicar refatorações baseadas em código fonte, tanto o código legado quanto o código do novo sistema OA sofrerão do mesmo problema, ou seja, a falta de documentação. Com a aplicação de refatorações baseadas em modelos, além do apoio de redução de complexidade e melhoria do entendimento do software, o Engenheiro de Software terá como artefatos, modelos de classes OO anotados e OA que servirão de documentação do software legado e do novo software. Esses artefatos poderão facilitar a realização de uma nova reengenharia do software, caso necessário, além de servir como documentação do próprio processo de reengenharia do software.

4.3 Refatorações Genéricas

As refatorações genéricas desenvolvidas neste trabalho são responsáveis por transformar um modelo OO anotado com indícios de interesses transversais, como o apresentado na Figura 4.3, em um modelo OA parcial. O modelo gerado é denominado parcial, pois os interesses transversais presentes no software existente ainda não são completamente modularizados, ou seja, existem elementos de software (classes, atributos e métodos) que continuam sendo afetados por determinados interesses. Salienta-se que as refatorações genéricas são aplicáveis a qualquer tipo de interesse existente no software. Na verdade, o que é levado em consideração é o cenário (configuração) que esses interesses apresentam no modelo OO do software e não o tipo desse interesse (por exemplo, persistência, *logging*, entre outros).

4.3.1 Conceituação e Justificativas para Aplicação de Refatorações Genéricas

Para ilustrar a aplicabilidade das refatoração genéricas considere que duas classes A e B são afetadas por um interesse transversal denominado “IT” (Figura 4.4). Os conceitos de Interesses Primários e Secundários apresentados na Seção 4.3 são utilizados nesta subseção. Percebe-se que para a classe A, “IT” é um

interesse primário e para `B`, é um interesse secundário (prefixos “Pri_” e “Sec_” colocados antes do nome do estereótipo relacionado ao interesse “IT”). Ou seja, a classe `A` foi criada especificamente para implementação do interesse “IT” e a classe `B` foi desenvolvida para desempenhar outra responsabilidade, contudo é afetada por esse interesse. Esse é um cenário clássico de entrelaçamento de interesses e que pode gerar problema para manutenção, evolução e reutilização desse software. O elemento de software que provoca tal entrelaçamento é o atributo `it` do tipo `IT` existente nas classes `A` e `B`.

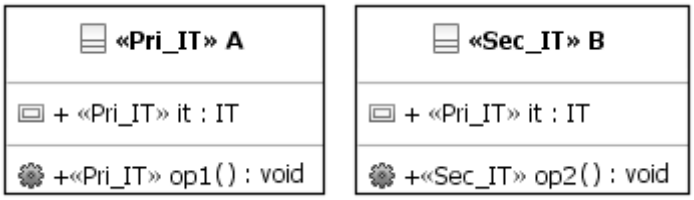

Figura 4.4 - Exemplo para Ilustrar a Aplicação de Refatorações Genéricas.

Independentemente do tipo de interesse que é “IT”, uma estratégia para sua modularização pode ser aplicada nesse momento. Por exemplo, a classe `A` encontra-se bem modularizada uma vez que o único interesse presente nela é o seu próprio interesse primário. Assim, ela é mantida como está.

No caso da classe `B` tem-se que o interesse “IT” é secundário, pois afeta alguns pontos dessa classe, como por exemplo, o método `op2()`. Esse método provavelmente invoca algum método pertencente à classe `IT` por meio do atributo `it`. Além disso, o atributo `it` não deveria existir na classe `B`, uma vez que essa classe não foi criada para implementar o interesse “IT”. Assim, para modularizar o interesse “IT” o atributo `it` da classe `B` poderia ser movido para um aspecto. Entretanto, quanto ao método `op2()` não é possível realizar qualquer alteração, pois não se sabe como o interesse “IT” está implementado no corpo desse método. Nesse ponto encontra-se o limite de atuação das refatorações genéricas e o porquê do modelo OA obtido por meio dessas refatorações ser denominado parcial.

A Figura 4.5 apresenta o modelo OA resultante após as modificações anteriormente comentadas. Percebe-se que a modularização do interesse “IT” não é completa, pois a classe B continua sendo afetada por esse interesse.

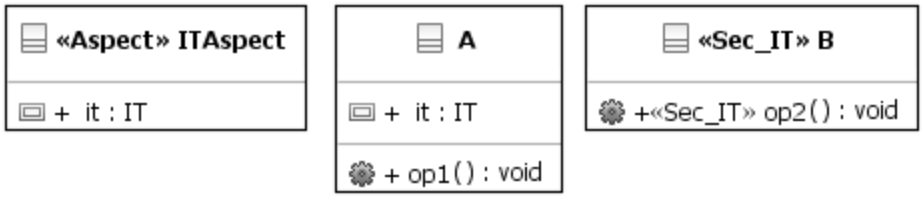

Figura 4.5 - Modelo OA Parcial.

Conhecendo-se o tipo do interesse “IT”, pode-se aplicar algum tipo de refatoração específica capaz de capaz de obter um modelo OA mais refinado. Por exemplo, se “IT” fosse um interesse responsável pelo gerenciamento de conexões com o Bando de Dados, seria possível verificar se os métodos invocados no corpo do método `op2()` são de abertura ou fechamento de conexão e assim, criar conjuntos de junção e adendos adequados para modularizar tal comportamento e remover esse interesse da classe `B`.

Algumas das principais justificativas para aplicação de refatorações genéricas sobre modelos OO anotados são:

1) A elaboração e aplicação de refatorações genéricas são importantes para a obtenção de um modelo OA de melhor qualidade. Decisões equivocadas tomadas pelos Engenheiros de Software, em conseqüência de sua inexperiência, podem comprometer a qualidade do projeto OA. Por exemplo, no cenário apresentado na Figura 4.4, a classe `A` possui como interesse primário o interesse transversal “IT”. Como todos os elementos da classe `A` são afetados por esse interesse, se o Engenheiro de Software transformar a classe `A` em um aspecto provocaria efeitos indesejados, pois outras classes que possuem algum tipo de relacionamento com a classe `A` ficariam inconsistentes.
2) Refatorações genéricas podem ser aplicadas sobre qualquer tipo de interesse, mesmo sobre aqueles que não são amplamente conhecidos como interesses transversais. Embora haja na literatura relatos sobre diversos tipos de interesses transversais, como persistência, *logging*, tratamento de exceções, entre outros (Kiczales *et al.*, 2001), nem sempre é fácil identificar se um determinado interesse é ou não um interesse transversal. Assim, com o auxílio das refatorações genéricas, pode-se identificar cenários que evidenciem ou deem indícios da existência de interesses transversais no software.

4.3.2 Refatorações Genéricas para Modelos de Classes Anotados com Indícios de Interesses Transversais

Para que um modelo OA seja obtido a partir de um OO, o Engenheiro de Software deve seguir alguns passos, uma vez que as diferenças entre abstrações oferecidas por essas tecnologias são consideráveis. Assim, nesta Subseção são apresentados os passos que devem ser realizados para três tipos de refatorações de modelos de classes OO anotados para modelos de classes OA parciais. O *template* utilizado para apresentá-los é o proposto por Fowler *et al.* (1999) e é descrito na Tabela 4.1.

Tabela 4.1 – Itens do *Template* Utilizado para Apresentação das Refatorações.

Item	Definição
Silga e Nome	Sigla e nomes dados a uma determinada refatoração. A sigla R-N, significa Refatoração número N.
Aplicação	Apresenta os cenários para os quais a refatoração proposta pode ser aplicada.
Motivação	Apresenta alguns problemas provocados pelo entrelaçamento e espalhamento de interesses transversais no software e como a aplicação da refatoração proposta pode amenizá-los.
Mecanismo Geral	Apresenta um conjunto de passos gerais para construção de um modelo de classes OA a partir de um modelo de classes OO anotado. Esses passos podem ser especificados para construção de um modelo de classes condizente com alguma abordagem de modelagem OA existente na literatura.
Mecanismo para ProAJ/UML	Apresenta um conjunto de passos específicos para construção de um modelo de classe OA em conformidade com a abordagem de modelagem OA ProAJ/UML (Evermann, 2007).
Exemplo	Apresenta fragmentos de modelos de classes de aplicações simples para mostrar a aplicabilidade da refatoração proposta.

R-1: Interesse transversal com entrelaçamento em decorrência da invocação de métodos

Aplicação

Quando um interesse transversal é Primário em algumas classes/interfaces e Secundário em outras, porém não há relacionamentos de generalização/especialização entre as classes/interfaces onde o interesse transversal é Primário e onde ele é Secundário.

Motivação

A utilização (invocação) de métodos de classes criadas para implementação de um determinado interesse transversal por classes cujo interesse primário é outro pode aumentar o espalhamento/entrelaçamento desse interesse, prejudicando a manutenibilidade do software.

Mecanismo Geral

Mover atributos/métodos bem modularizados (conceito apresentado na Seção 4.3) das classes em que esse interesse é Secundário para um aspecto e reintroduzir os métodos nas classes base por meio de declarações inter-tipo.

Mecanismo para ProAJ/UML

1. Criar um CrossCuttingConcern (CCC).
2. Criar um Aspect (As), e adicioná-lo ao CrossCuttingConcern “CCC”.
3. Para cada atributo bem modularizado com relação ao interesse transversal adicioná-lo ao aspecto “As”.
4. Para cada método bem modularizado com relação ao interesse transversal, criar um *IntroductionMethod* e adicioná-lo ao aspecto “As”.
5. Mover as classes nas quais o interesse em questão é um Interesse Primário e que se encontram bem modularizadas para o CrossCuttingConcern “CCC”.

Exemplo

Na Figura 4.6 são apresentadas duas classes: `Account` e `Database`. Persistência é o Interesse Primário da classe `Database`, que foi criada especificamente para persistir dados no Banco de Dados. A classe `Account`, que se relaciona com a classe `Database` por associação, possui os métodos

`withdraw()` e `deposit()`, que estão marcados com o estereótipo `<<Sec_Persistence>>`.

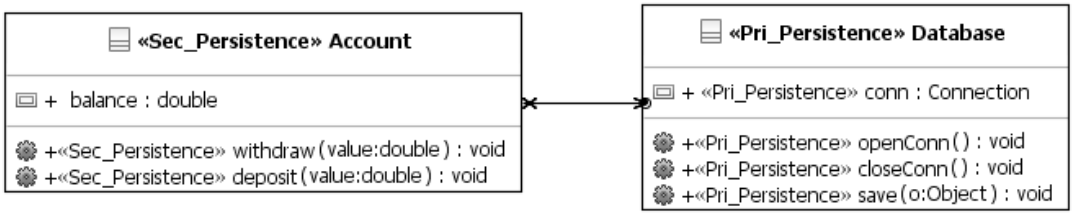

Figura 4.6 - Classes Afetadas pelo Interesse de Persistência.

Para que os métodos da classe `Account` tenham recebido o estereótipo `<<Sec_Persistence>>` o corpo desses métodos deve conter invocações aos métodos da classe `Database` cujo interesse de persistência é um Interesse Primário. Para obter um modelo OA adequado a partir desse cenário os passos descritos nessa refatoração devem ser aplicados e o modelo obtido é apresentado na Figura 4.7.

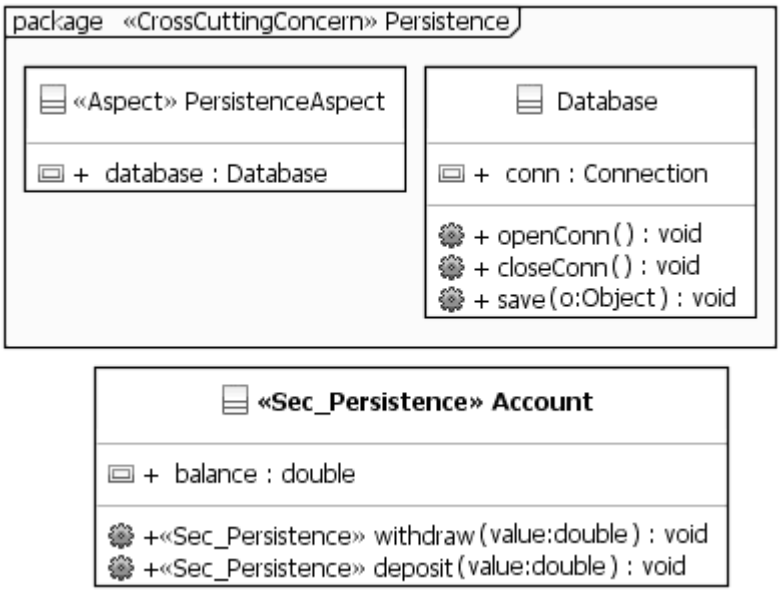

Figura 4.7 - Modelo OA Parcial Obtido após a Aplicação da R-1.

O relacionamento de associação da classe `Account` com a classe `Database` foi removido, sendo o atributo `database` que gerava esse relacionamento inserido no aspecto `PersistenceAspect`. Isso foi feito, pois esse atributo foi criado para implementação do interesse de persistência e não deve estar presente na classe `Account`, que é uma classe de negócio. A classe `Database` foi movida para o pacote `Persistence`, pois ela é totalmente dedicada à implementação do interesse de persistência. Percebe-se,

entretanto, que ainda há estereótipos `<<Sec_Persistence>>` na classe `Account`, sinalizando a presença do interesse de persistência nessa classe. Isso ocorre, pois, por meio das refatorações genéricas, não é possível saber qual interesse está sendo analisado, nem como ele deve ser modularizado. Esse problema será resolvido com a utilização de refatorações específicas para o interesse de persistência que serão discutidas mais adiante nesse trabalho.

R-2: Interesse transversal com entrelaçamento em decorrência da sobrecarga de métodos

Aplicação

Quando há classes/interfaces cujo Interesse Primário é um interesse transversal e há métodos dessas classes/interfaces sobrecarregados ou utilizados em suas subclasses nas quais o interesse em questão é Secundário.

Motivação

Ao sobrecarregar ou utilizar um método que foi criado especificamente para implementação de um interesse transversal, a classe que o sobrecarrega/utiliza passa a ser afetada por esse interesse. Esse cenário provoca redução da coesão e aumento do acoplamento dos módulos do sistema, prejudicando sua manutenibilidade e evolução.

Mecanismo Geral

Mover todos os atributos/métodos bem modularizados com relação ao interesse transversal de todas as classes/interfaces afetadas pelo interesse para um aspecto e reintroduzir os métodos nas classes/interfaces base por meio de declarações inter-tipo. Mover os relacionamentos de herança e realização de interface para um aspecto.

Mecanismo para ProAJ/UML

1. Criar um *CrossCuttingConcern* (CCC);
2. Criar um Aspect (As) e adicioná-lo ao CrossCuttingConcern “CCC”.
3. Para cada atributo bem modularizado com relação ao interesse transversal adicioná-lo ao aspecto “As”.
4. Para cada método bem modularizado com relação ao interesse

transversal, criar um *IntroductionMethod* e adicioná-lo ao aspecto “As”.

5. Mover as declarações de herança e realização de interfaces (*declare parents*) para o aspecto “As”.

Exemplo

Na Figura 4.8 é apresentado um exemplo da utilização do padrão *Observer* (Gamma *et al.*, 1995) em uma aplicação simples para desenho gráfico. Nessa aplicação, a classe `Point` é responsável por armazenar os dados relacionados a um ponto, como suas coordenadas *x* e *y* e a classe `Screen` apresenta os pontos criados pelo usuário na tela.

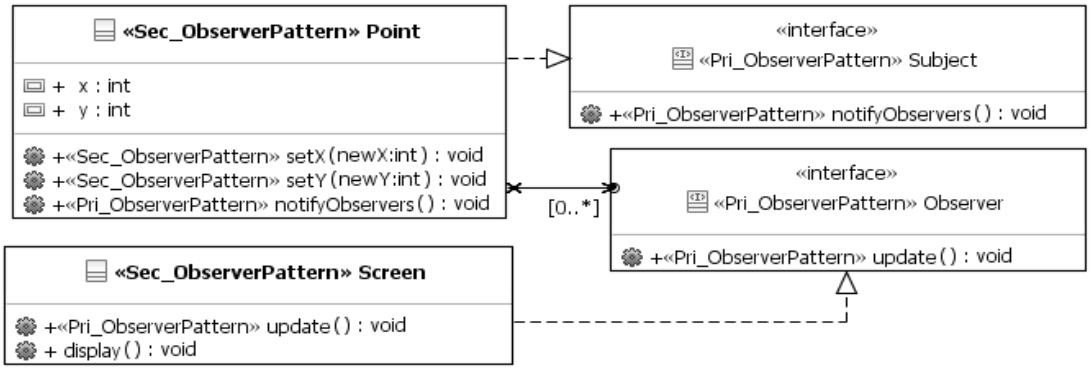

Figura 4.8 - Aplicação Implementada com o Padrão Observer.

As interfaces `Subject` e `Observer` e seus métodos encontram-se estereotipados com `<<Pri_ObserverPattern>>`, pois foram criados especificamente para implementação do padrão *Observer*. Pode-se observar que os métodos definidos nessas interfaces são sobrecarregados nas classes `Point` e `Screen`, e por isso, também receberam o estereótipo `<<Pri_ObserverPattern>>`. Os métodos estereotipados com `<<Sec_ObserverPattern>>` não foram criados para implementação do padrão *Observer*, mas são afetados por ele. Isso acontece, pois os métodos `setX()` e `setY()` da classe `Point` invocam o método `notifyObservers()`, que é específico da implementação do padrão, para sinalizar a modificação nas coordenadas do ponto. Aplicando os passos do mecanismo para ProAJ/UML, obtém-se o modelo apresentado na Figura 4.9.

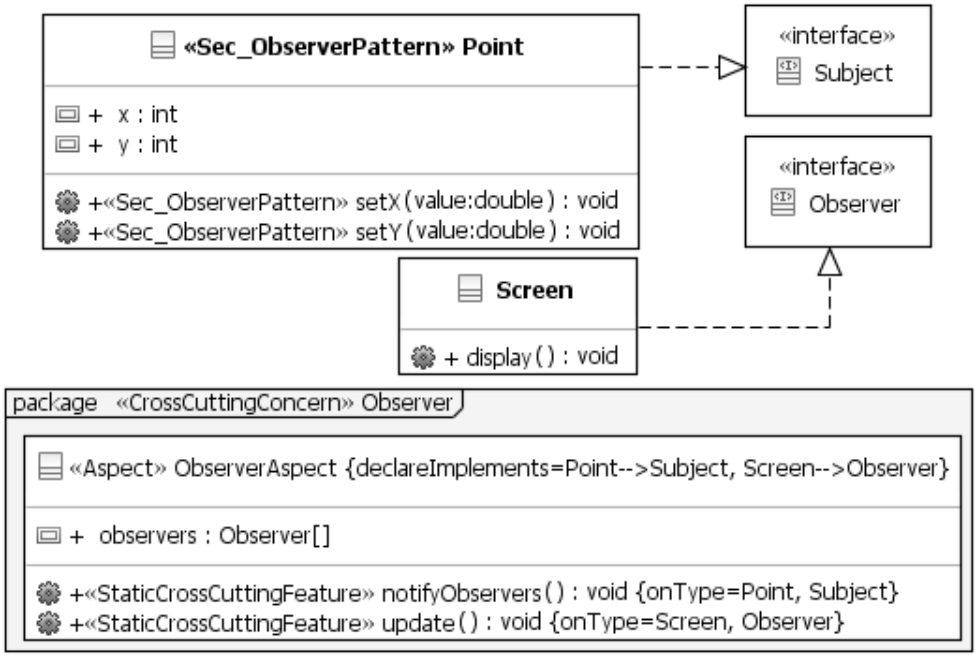

Figura 4.9 - Modelo OA obtido a partir da aplicação da R-2.

Observa-se que a classe `Screen` não é mais afetada pela implementação do padrão *Observer*, entretanto, a classe `Point` ainda possui métodos afetados. Isso acontece, pois não há como saber, nesse momento, que o interesse sendo analisado corresponde ao padrão *Observer*. As realizações de interface entre as classes `Point` e `Screen` e as interfaces `Subject` e `Observer` foram movidas para o aspecto `ObserverAspect`. Os demais elementos, como o atributo `observers`, responsável pelo relacionamento de associação entre a classe `Point` e a interface `Observer`, e os métodos `notifyObservers()` e `update()` foram transferidos para apenas um módulo, o aspecto `ObserverAspect`, melhorando assim a modularização dessa aplicação. Esses métodos são reintroduzidos pelo aspecto em suas respectivas classes/interfaces para que não haja problemas de dependência entre interesses transversais. Por exemplo, considerando que o interesse de *logging* também afetasse o método `update()` da classe `Screen` e fosse modularizado antes do padrão *Observer*. Ao utilizar a refatoração para o padrão *Observer*, se o método `update()` fosse removido para um aspecto isso iria comprometer a modularização do interesse de *logging*, uma vez que esse método não existiria mais na classe `Screen`.

R-3: O Interesse transversal não é Interesse Primário de nenhuma classe.

Aplicação

Quando há classes cujo Interesse Secundário é um interesse transversal que não é Interesse Primário de nenhuma classe.

Motivação

Alguns interesses transversais podem encontrar-se espalhados em diversas classes do sistema e não haver uma ou mais classes criadas especificamente para implementação desses interesses. Um interesse desse tipo não é interesse primário de nenhuma classe da aplicação. Esse cenário representa alto grau de espalhamento de interesses e consequentemente um baixo nível de modularização do software.

Mecanismo Geral

Aplicar o mesmo mecanismo da R-1. Apesar de apresentarem o mesmo mecanismo de refatoração, a R-1 e R-3 são destinadas a cenários de entrelaçamento/espalhamento distintos e, por isso, foram classificadas em refatorações separadas.

Mecanismo para ProAJ/UML

Aplicar o mesmo mecanismo da R-1, com exceção do passo 5.

Exemplo

Na Figura 4.10 é apresentada novamente a classe `Account`, entretanto, nesse exemplo, ela é afetada pelo interesse de *logging*, que não é Interesse Primário de alguma outra classe da aplicação.

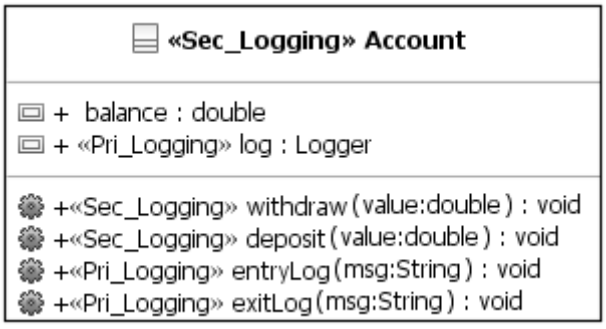

Figura 4.10 - Classe de uma Aplicação Bancária Simples Afetadas pelo Interesse de Logging.

O atributo `log`, do tipo `Logger`, pertencente à API de *logging* da plataforma Java e os métodos `entryLog()` e `exitLog()`, utilizados para gravar entradas e saídas no arquivo de *log* da aplicação, receberam o estereótipo

`<<Pri_Logging>>`. Isso porque esses elementos foram criados especificamente para implementação do interesse de *logging*. Os métodos de negócio `withdraw()` e `deposit()`, por sua vez, utilizam os métodos `entryLog()` e `exitLog()` para registrar operações de saque e depósito em uma conta, por isso, recebem o estereótipo `<<Sec_Logging>>`. Ao aplicar os passos descritos nessa refatoração, tem-se como resultado o modelo OA apresentado na Figura 4.11.

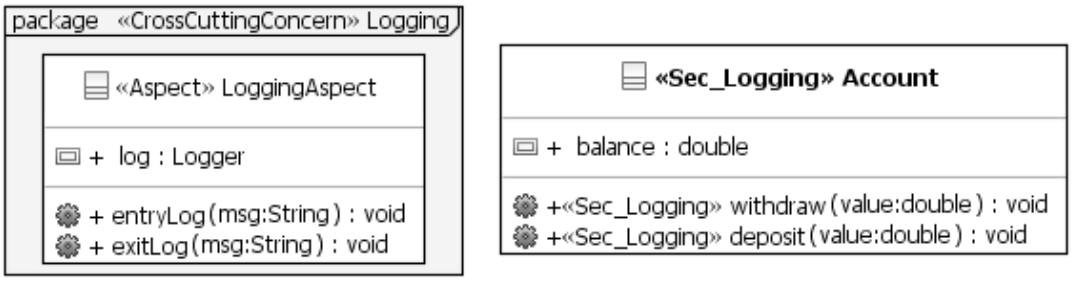

Figura 4.11 - Modelo OA obtido a partir da aplicação da R-3.

No exemplo anterior, o atributo `log` e os métodos `entryLog()` e `exitLog()` foram movidos para o aspecto `LoggingAspect`. Nos métodos `withdraw()` e `deposit()` *logging* é um interesse secundário e por isso eles permaneceram na classe `Account`.

4.3.3 Ordem para Aplicação das Refatorações Genéricas

De acordo com a descrição das refatorações apresentada nas seções anteriores, há indícios de que não existe uma sequência específica para execução das refatorações independentes do tipo de interesse quando é lavado em consideração o modelo resultante obtido a partir da aplicação dessas refatorações. Isso acontece, pois os passos criados para as refatorações só são aplicados quando um determinado elemento é bem modularizado, ou seja, quando não há interferência de outros interesses nesse elemento. As refatorações genéricas elaboradas são aplicadas em três cenários específicos de modularização de um software: i) quando uma classe possui um Interesse Primário e seus métodos são invocados por outras classes que passam a conter esse interesse como secundário (cenário I); ii) quando uma classe possui um Interesse Primário e seus métodos são sobrecarregados por outras classes que passam a conter esse interesse como secundário (cenário II); e

iii) quando uma classe possui um Interesse Secundário que não é Interesse Primário de qualquer classe da aplicação (cenário III). Salienta-se que é necessária a aplicação de técnicas formais para avaliar corretamente se a ordem de execução das refatorações interfere nos modelos de classes gerados, porém, para apresentar alguns indícios de que a ordem de aplicação das refatorações não interfere no modelo OA gerado, são analisadas quatro combinações possíveis de cenários: (I e II), (I e III), (II e III) e (I, II e III).

Para a combinação (I e II) é utilizado o exemplo da Figura 4.12. Observa-se a presença de um interesse transversal `IT` que é simultaneamente Interesse Primário da classe `A` e Interesse Secundário das classes `B` e `C`. A diferença é que na classe `C`, o interesse transversal `IT` aparece como Interesse Secundário porque o método `operationC()` dessa classe utiliza um objeto da classe `A`, cujo Interesse Primário é o interesse transversal em questão. Por outro lado, na classe `B`, o interesse transversal `IT` aparece como Interesse Secundário, pois essa classe sobrescreve o método `operationA()` da classe `A`, cujo Interesse Primário é o interesse `IT`.

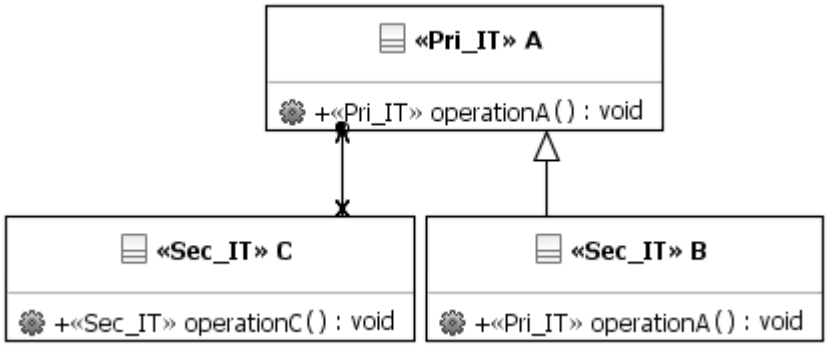

Figura 4.12 – Modelo OO Anotado para Combinação (I e II).

Para esse exemplo as refatorações passíveis de serem aplicadas são R-1 e R-2. Se a R-1 for executada primeiramente, um novo aspecto será criado e o atributo que representa o relacionamento entre as classes `C` e `A` será movido para esse aspecto. Ao executar a R-2, o método `operationA()` será removido das classes `A` e `B` e reintroduzido pelo aspecto criado anteriormente pela R-1, por meio de declarações inter-tipo. O relacionamento de herança entre as classes `B` e `A` também será movida para um aspecto. O resultado é apresentado na Figura 4.13.

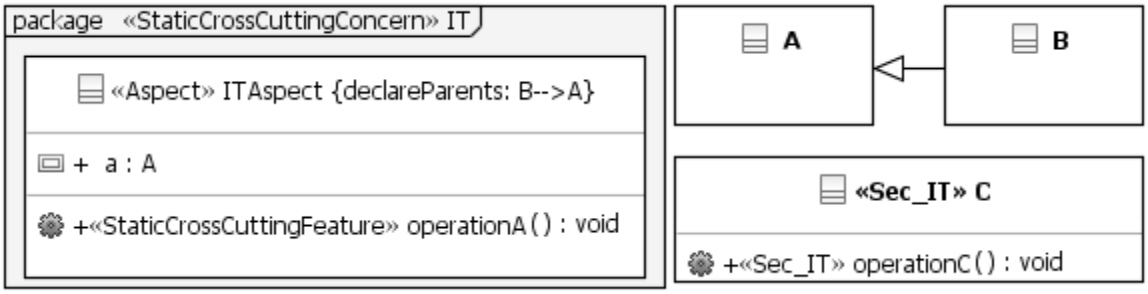

Figura 4.13 – Modelo OA Parcial para Combinação (I e II).

Caso R-2 fosse aplicada antes de R-1 o resultado seria o mesmo, apenas a ordem de execução dos passos seria invertida. Nesse exemplo, utilizou-se apenas um tipo de interesse transversal. Caso fossem utilizados dois interesses distintos, dois aspectos seriam criados, um deles conteria o atributo que relaciona as classes `C` e `A` e o outro, a declaração inter-tipo e *declare parents* para as classes `A` e `B`. Mesmo assim, a ordem de aplicação das refatorações não influenciaria no resultado final.

Para a combinação (I e III) é utilizado o exemplo da Figura 4.14. Nesse caso, observa-se a presença de dois interesses transversais `ITA` e `ITB`. `ITA` é simultaneamente Interesse Primário da classe `A` e Interesse Secundário da classe `B`. `ITB` é Interesse Secundário em `B` e não há uma classe específica na qual `ITB` seja um Interesse Primário. Na classe `B`, o interesse transversal `ITA` aparece como Interesse Secundário, pois o método `operationB1()` dessa classe utiliza um objeto da classe `A`, cujo Interesse Primário é o `ITA`. De modo análogo, o interesse transversal `ITB` aparece como Interesse Secundário na classe `B`, pois o método `operationB2()` dessa classe utiliza o atributo `it`, cujo interesse primário é o `ITB`.

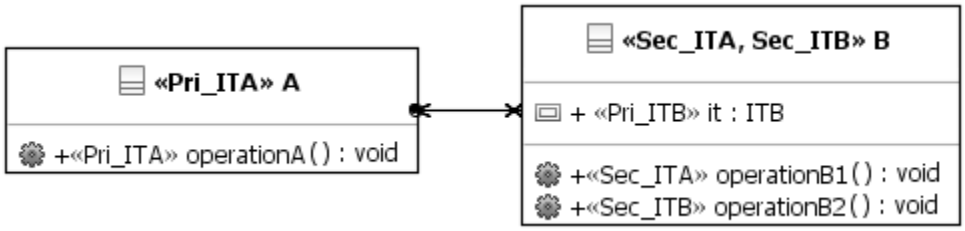

Figura 4.14 – Modelo OO Anotado para Combinação (I e III).

Para esse exemplo as refatorações passíveis de serem aplicadas são R-1 e R-3. Se a R-1 for executada primeiramente para modularização do interesse `ITA`, um novo aspecto será criado e o atributo que representa o relacionamento entre as classes `B` e `A` será movido para esse aspecto. Ao executar a R-3, outro aspecto será

criado, pois se trata de um outro tipo de interesse (`ITB`), e o atributo `it` da classe `B` será movido para esse aspecto. O resultado final é apresentado na Figura 4.15. Caso R-3 fosse executada antes de R-1 o resultado seria o mesmo, apenas a ordem de execução dos passos seria invertida.

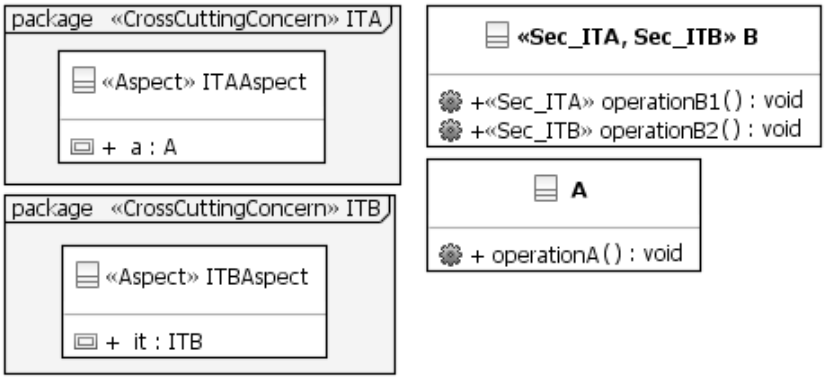

Figura 4.15 – Modelo OA Parcial para Combinação (I e III).

Para a combinação (II e III) é utilizado o exemplo da Figura 4.16. Nesse caso, as refatorações passíveis de aplicação são R-2 e R-3. De modo análogo ao que aconteceu nas demais combinações, a ordem de aplicação dessas refatorações não exercerá influência o resultado final. Observa-se que, se a R-2 for executada primeiramente, o método `operationA()` será removido das classes `A` e `B` e reintroduzido por um aspecto. Além disso, o relaciomento de herança entre as classes `B` e `A` será movido para um aspecto. Posteriormente, ao aplicar a R-3, o atributo `it` da classe `B` será movido para outro aspecto já que se trata de um interesse diferente (`ITB`). O resultado da aplicação das refatorações R-2 e R-3 sobre o modelo da Figura 4.16, em qualquer ordem, é apresentado na Figura 4.17.

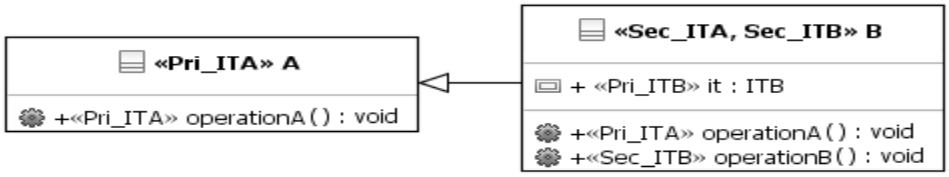

Figura 4.16 – Modelo OO Anotado para Combinação (II e III).

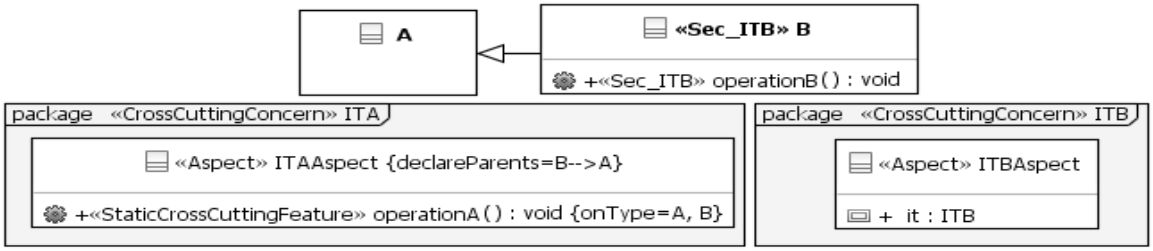

Figura 4.17 – Modelo OA Parcial para Combinação (II e III).

Por último, a combinação (I, II e III) é apresentada pela Figura 4.18. Nesse exemplo, os três cenários aparecem juntos e, portanto, qualquer uma das três refatorações pode ser aplicada. Como visto anteriormente, o mecanismo elaborado para as refatorações modificam somente elementos bem modularizados para um determinado interesse, justamente para prevenir que modificações realizadas sobre um interesse venham a interferir na implementação de outros interesses. Dessa forma, pode-se escolher qualquer ordem para refatoração do modelo da Figura 4.18, nesse caso, nesse exemplo, a sequência R-1, R-2 e R-3 foi escolhida. Ao aplicar R-1, apenas o atributo que gera relacionamento entre as classes `C` e `A` será movido para um aspecto. Esse aspecto é criado para modularização do interesse `ITB`.

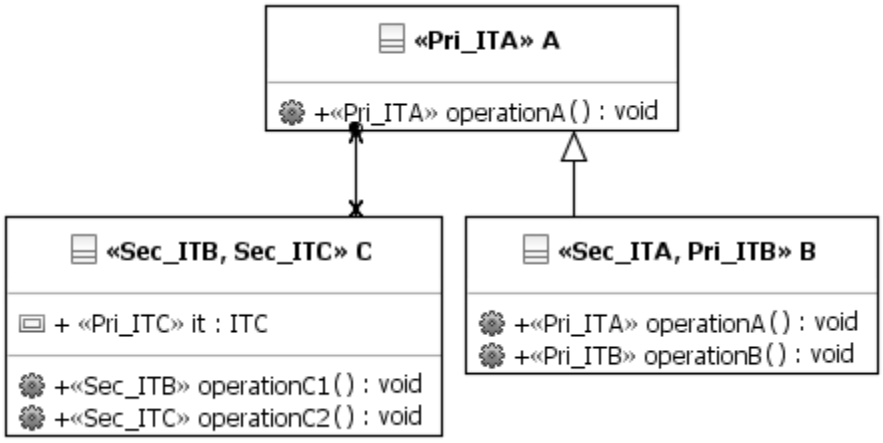

Figura 4.18 – Modelo OO Anotado para Combinação (I, II e III).

Posteriormente, ao aplicar R-2, o método `operationA()` será removido e das classes `A` e `B` e será reintroduzido por um outro aspecto criado para modularizar o interesse `ITA`. Por último, a R-3 irá criar um novo aspecto, usado para modularização do interesse `ITC`, e moverá o atributo `it` da classe `C` para esse novo aspecto. O modelo gerado após a aplicação das refatorações R-1, R-2 e R-3 é apresentado na Figura 4.19.

Como pode ser visto nas combinações anteriores, ao executar as refatorações nas ordens (R-1, R-3 e R-2), (R-2, R-1 e R-3), (R-2, R-3 e R-1), (R-3, R-1 e R-2) ou (R-2, R-3 e R-1) o resultado será o mesmo. Há situações, porém, em que a ordem de aplicação das refatorações pode implicar na reaplicação de uma determinada refatoração, sem, contudo, comprometer o modelo gerado.

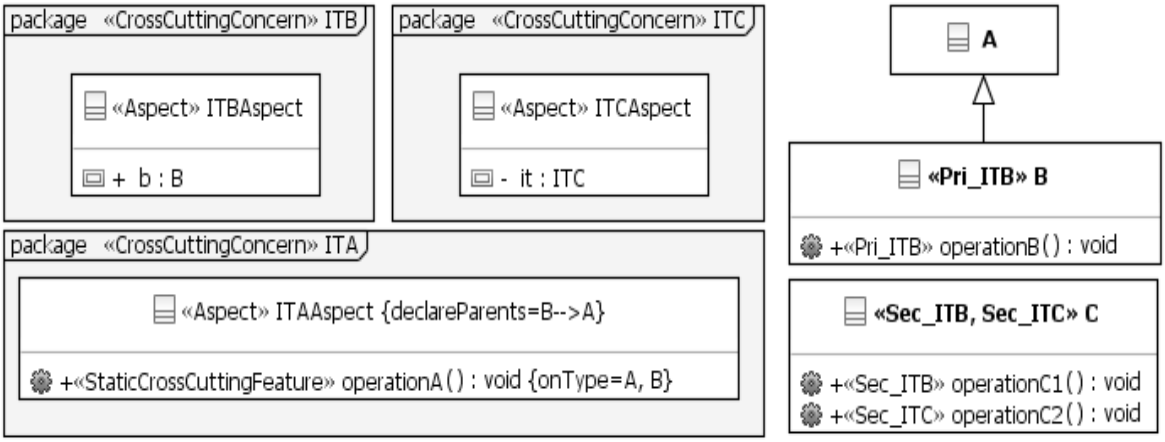

Figura 4.19 – Modelo OA Parcial para Combinação (I, II e III).

Por exemplo, dado o cenário de entrelaçamento/espalhamento da Figura 4.20, observa-se que o interesse `ITA` não pode ser completamente modularizado antes do interesse `ITB`, pois esse afeta elementos relacionado ao interesse `ITA`.

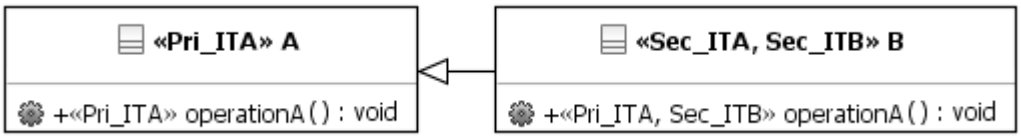

Figura 4.20 – Modelo OO Anotado com Indícios dos Interesses `ITA` e `ITB`.

A melhor ordem para aplicação das refatorações é: 1) aplicar uma refatoração genérica para o interesse `ITB`; 2) aplicar uma refatoração específica para o interesse `ITB` de forma que o método `operationA()`seja afetado apenas pelo interesse `ITA`; 3) aplicar as refatorações genéricas e específicas para o interesse `ITA`. Caso o Engenheiro de Software escolha a ordem inversa, ou seja, começar a modularização pelo interesse `ITA`, a refatoração genérica para esse interesse será executada apenas parcialmente, como pode ser visto na Figura 4.22.

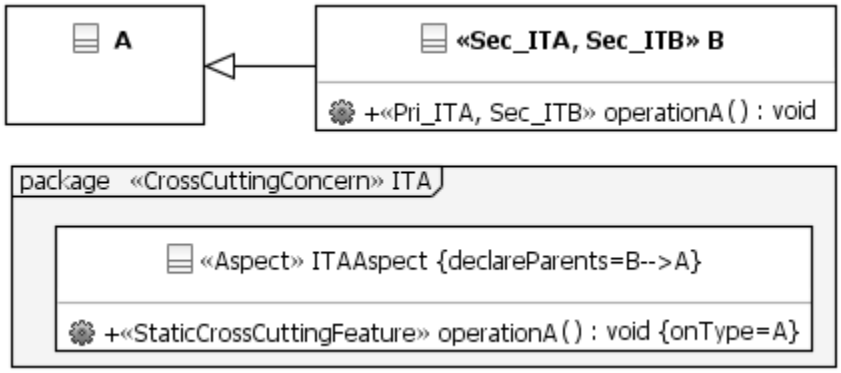

Figura 4.21 – Modelo OA Parcial para Modularização dos Interesses `ITA` e `ITB`.

Percebe-se que o método `operationA()` foi movido para um aspecto `ITAAspect` que irá reintroduzi-lo novamente apenas na classe `A`, ou seja, o método `operationA()` continua na classe `B`, apesar de ser um método cujo interesse `ITA` é primário. Isso acontece, pois o método `operationA()` não é bem modularizado para esse interesse, ou seja, há interferência do interesse `ITB` nesse método. Assim, será necessário refatorar o interesse `ITB` para que a refatoração R-2 seja reaplicada e a modularização do interesse `ITA` possa ser completada. Salienta-se que o modelo resultante será o mesmo nos dois casos, entretanto, um passo a mais será necessário para completar a modularização da aplicação.

Em todos os casos apresentados os elementos que possuem Interesses Secundários permaneceram inalterados. A ideia é que esses interesses sejam removidos por meio de refatorações específicas, pois não há como saber, nesse momento, que tipos de interesse estão sendo analisados e quais são as melhores maneiras de refatorá-los.

4.4 Refatorações Específicas para Modelos de Classes OO Anotados com Indícios de Interesses Transversais

As refatorações específicas desenvolvidas neste trabalho são responsáveis por transformar modelos de classes OA parciais em modelos de classes OA finais, nos quais os interesses transversais presentes no software foram completamente modularizados em aspectos. Para isso, cada refatoração possui um conjunto de passos específicos para modularização dos interesses de persistência, *logging* e dos padrões de projeto Singleton e Observer (Gamma *et al.*, 1995). As refatorações específicas também são apresentadas no *template* proposto por Fowler *et al.* (1999). Os exemplos de aplicação para essas refatorações são apresentados na Subseção 4.4.2.

4.4.1 Refatorações Específicas para o Interesse Transversal de Persistência

Nesta Subseção são apresentadas as refatorações criadas para o interesse transversal de persistência. A implementação de persistência em Bancos de Dados relacionais pode ser dividida em: gerenciamento de conexões, gerenciamento de transações e gerenciamento de sincronização de objetos. Dessa forma, foram criadas refatorações específicas para tratar cada um desses interesses.

R-*Connection*: O interesse transversal é responsável pelo gerenciamento de conexões com o Banco de Dados.

Aplicação

Quando há classes afetadas pelo interesse transversal de persistência e que são responsáveis pelo gerenciamento de conexões com o Banco de Dados.

Motivação

Para softwares que utilizam Banco de Dados relacionais, antes de se realizar alguma operação que exigirá acesso ao Banco de Dados, é necessário criar uma conexão com o Sistema Gerenciador de Banco de Dados (SGBD). De modo análogo, ao terminar a utilização do Banco de Dados, essa conexão deve ser fechada para que não consuma demasiados recursos da máquina onde a aplicação é executada. A responsabilidade de abertura e fechamento de conexões com o Banco de Dados é um interesse que, quando implementado de maneira inadequada, pode provocar alto entrelaçamento e espalhamento de código pelos módulos funcionais do software, comprometendo sua manutenibilidade e reusabilidade.

Mecanismo Geral

1. Identificar o aspecto correspondente ao interesse de controle de conexão.
2. Verificar se o aspecto identificado é abstrato. Se não for, transformá-lo em abstrato.
3. Criar conjuntos de junção abstratos para abertura e fechamento de conexões. Criar adendos que efetivamente abrem e fecham uma conexão com o Banco de Dados e relacioná-los com os conjuntos de junção criados anteriormente.
4. Criar um aspecto concreto que estende o aspecto abstrato identificado

anteriormente para as classes afetadas pelo interesse de gerenciamento de conexão. Neste momento o Engenheiro de Software pode decidir por criar um aspecto concreto para cada classe afetada ou para um conjunto delas de acordo com as características de implementação da aplicação. Por exemplo, se uma aplicação possui muitas classes afetadas pelo interesse de gerenciamento de conexões, o Engenheiro de Software pode optar por criar um aspecto para cada pacote que contém classes afetadas por esse interesse.

5. Localizar os pontos nessas classes onde a conexão é aberta e fechada e criar conjuntos de junção adequados em cada aspectos para capturar esses pontos.

Mecanismo para ProAJ/UML

1. Identificar o aspecto “As” correspondente ao interesse de controle de conexão.
2. Verificar se o aspecto é abstrato. Se não for, transformá-lo em abstrato.
3. Definir os conjuntos de junção abstratos `abstract pointcut openConnection()` e `abstract pointcut closeConnection()` e adendos do tipo `before` e `after` relacionados a eles.
4. Criar um aspecto “AsN” derivado do aspecto “As” para as classes afetadas pelo interesse de gerenciamento de conexão de acordo com as características de implementação aplicação.
5. Localizar os pontos nessas classes onde a conexão é aberta e fechada e criar conjuntos de junção `pointcut openConnection()` e `pointcut closeConnection()` adequados para cada aspecto.

R-*Transaction*: O interesse transversal é responsável pelo gerenciamento de transações com o Banco de Dados.

Aplicação

Quando há classes afetadas pelo interesse transversal de persistência e que são responsáveis pelo gerenciamento de transações com o Banco de Dados.

Motivação

Semelhante ao que acontece para o interesse de gerenciamento de conexões, aplicações geralmente possuem métodos que são transacionais, isto é,

métodos cuja execução deve garantir que os dados armazenados no Banco de Dados permanecerão consistentes após ocorrer uma atualização no estado de objetos persistentes. Basicamente, antes de iniciar a atualização do estado, tais métodos devem comunicar ao SGBD o início da transação e caso a atualização seja efetuada com sucesso, o comando (*commit*) deve ser executado. Caso contrário, deve-se comunicar o insucesso da atualização (*rollback*). Esse procedimento se repete para todos os métodos transacionais, o que provoca alto índice de entrelaçamento/espalhamento do interesse de gerenciamento de transações. A modularização desse tipo de interesse pode facilitar o entendimento, a manutenção e a reusabilidade do software.

Mecanismo Geral

1. Identificar o aspecto correspondente ao interesse de gerenciamento de transações.
2. Verificar se o aspecto identificado é abstrato. Se não for, transformá-lo em abstrato.
3. Criar um conjunto de junção abstrato que interceptará operações transacionais da aplicação. Criar um adendo responsável por executar os procedimentos necessários para realização de uma transação no Banco de Dados.
4. Criar um aspecto concreto que estende o aspecto abstrato identificado anteriormente para as classes afetadas pelo interesse de gerenciamento de transações.
5. Localizar os métodos transacionais dessas classes e criar um conjunto de junção adequado para cada aspecto.

Mecanismo para ProAJ/UML

1. Identificar o aspecto “As” correspondente ao interesse de controle de transação.
2. Verificar se o aspecto é abstrato. Se não for, transformá-lo em abstrato.
3. Definir o conjunto de junção abstrato `abstract pointcut transactionalMethods()` e um adendo do tipo `around` relacionados a esse conjunto de junção.
4. Criar um aspecto “AsN” derivado do aspecto “As” para as classes afetadas pelo interesse de gerenciamento de transações de acordo com as

características de implementação aplicação.

5. Localizar os métodos transacionais dessas classes e criar um conjunto de junção concreto `pointcut transactionalMethods()` adequado para cada aspecto.

R-Sync: O interesse transversal é responsável pelo gerenciamento de sincronização de objetos em memória com o Banco de Dados.

Aplicação

Quando há classes afetadas pelo interesse transversal de persistência e que são responsáveis pelo gerenciamento de sincronização de objetos em memória com o Banco de Dados.

Motivação

Os objetos criados/alterados em memória devem ser persistidos no Banco de Dados. A responsabilidade de manter a sincronização dos dados do Banco de Dados e da memória principal do computador é um exemplo de interesse transversal e é importante para que não haja inconsistência entre os dados exibidos ao usuário e os dados que efetivamente estão gravados no Banco de Dados. A modularização desse interesse consiste em remover das classes de negócio os elementos relacionados à sincronização, simplificando sua lógica de execução e facilitando a manutenção do software.

Mecanismo Geral

1. Identificar o aspecto correspondente ao interesse de controle de conexão.
2. Verificar se o aspecto identificado é abstrato. Se não for, transformá-lo em abstrato.
3. Criar uma interface vazia.
4. Para cada classe que deve ser persistida no Banco de Dados, declarar um relacionamento de realização de interface entre essa classe e a interface criada no passo anterior.
5. Definir conjuntos de junção abstratos para inserção, atualização e remoção de dados no Banco de Dados. Criar adendos correspondentes para realizar as operações de criação, atualização e remoção de um registro no Banco de Dados.
6. Criar um aspecto concreto que estende o aspecto abstrato identificado

anteriormente para as classes afetadas pelo interesse de sincronização de objetos em memória com o Banco de Dados.

7. Localizar os pontos nessas classes onde a sincronização deve ser realizada e criar conjuntos de junção adequados para cada aspecto.

Mecanismo para ProAJ/UML

1. Identificar o aspecto “As” correspondente ao interesse de controle de transação.
2. Verificar se o aspecto é abstrato. Se não for, transformá-lo em abstrato.
3. Criar uma interface `Persistent` vazia.
4. Para cada classe que deve ser persistida no Banco de Dados, declarar um relacionamento de realização de interface `declare implements` entre essa classe e a interface `Persistent`.
5. Definir os conjuntos de junção abstratos `abstract pointcut insert(Persistent p)`, `abstract pointcut update(Persistent p)` e `abstract pointcut delete(Persistent p)`, o conjunto de junção concreto `pointcut target(Persistent p): target(Persistent)` e adendos do tipo `after` relacionados aos conjuntos de junção `insert`, `update` e `delete` em composição com o conjunto de junção `target`. Por exemplo: `insert(Persistent p) && target(Persistent)`.
6. Criar um aspecto “AsN” derivado do aspecto “As” p para as classes afetadas pelo interesse de sincronização de objetos em memória com o Banco de Dados de acordo com as características de implementação aplicação.
7. Localizar os pontos nessas classes onde a sincronização deve ser realizada e criar conjuntos de junção concretos `pointcut insert(Persistent p)`, `pointcut update(Persistent p)` e `pointcut delete(Persistent p)` adequados para cada aspecto.

4.4.2 Refatorações Específicas para o Interesse Transversal de *Logging*

Nesta subseção são apresentadas as refatorações criadas para o interesse transversal de *logging*.

R-*Logging*: O interesse transversal é responsável pelo controle de registro

***logging* da aplicação.**
Aplicação
Quando há classes afetadas pelo interesse transversal de *logging* que são responsáveis pelo registro de informações de contexto da aplicação.
Motivação
O interesse de *logging* é bem conhecido na literatura como um interesse transversal (Kiczales *et al.*, 2001). Isso ocorre, pois em todos os pontos da aplicação nos quais se deseja registrar o contexto de execução da aplicação, será necessário realizar uma chamada ao mecanismo de *logging* para que esse registro seja gravado. Essas chamadas espalhadas geram alto entrelaçamento de interesses na aplicação, aumentando o acoplamento e diminuindo a coesão dos módulos do software.
Mecanismo Geral
1. Identificar o aspecto correspondente ao interesse de *logging*. 2. Verificar se o aspecto identificado é abstrato. Se não for, transformá-lo em abstrato. 3. Criar conjuntos de junção abstratos para registro de *logs*. Criar adendos que efetivamente gravam os dados de contexto da aplicação utilizando o mecanismo de *logging*. 4. Criar um aspecto concreto que estende o aspecto abstrato identificado anteriormente para as classes afetadas pelo interesse de *logging*. 5. Localizar os pontos nessas classes onde ocorrem os registros de *logs* e criar conjuntos de junção adequados para cada aspecto.
Mecanismo para ProAJ/UML
1. Identificar o aspecto “As” correspondente ao interesse de *logging*. 2. Verificar se o aspecto é abstrato. Se não for, transformá-lo em abstrato. 3. Definir os conjuntos de junção abstratos `abstract pointcut entryLog()` e `abstract pointcut exitLog()` e adendos do tipo `before` e `after` relacionados a eles. 4. Criar um aspecto “AsN” derivado do aspecto “As” para as classes afetadas pelo interesse de *logging* de acordo com as características de implementação da aplicação. 5. Localizar os pontos nessas classes onde registros de *logs* são realizados e

criar conjuntos de junção `pointcut entryLog()` e `pointcut exitLog()` adequados para cada aspecto. Se há informações de contexto como valores de argumentos sendo utilizados nos métodos que realizam o registro de *logs* da aplicação, conjuntos de junção `target` e `args` devem ser criados para que essas informações possam ser capturadas.

4.4.3 Refatorações Específicas para os Padrões de Projeto *Singleton* e *Observer*

Nesta subseção são apresentadas as refatorações criadas para os Padrões de Projeto *Singleton* e *Observer* (Gamma *et al.*, 1995). Essas refatorações foram criadas com base no trabalho de Hannemam *et al.* (2002), adaptando as refatorações originalmente proposta por Hanneman *et al*. ao contexto de modelos de classes anotados.

R-*Singleton*: O interesse transversal corresponde ao padrão de projeto *Singleton*.

Aplicação

Quando há classes dedicadas à implementação do padrão *Singleton*.

Motivação

Hanneman *et al.* (2002) perceberam que determinados padrões de projeto, quando implementados em aplicações OO, geram problemas de entrelaçamento e espalhamento de interesses, e assim, poderiam ser modularizados com a utilização de OA. O padrão *Singleton* é um desses padrões, sendo criada uma refatoração específica para modularizá-lo.

Mecanismo Geral

1. Identificar o aspecto relacionado à implementação do padrão *Singleton*.
2. Verificar se o aspecto identificado é abstrato. Se não for, transformá-lo em abstrato.
3. Criar uma interface vazia.
4. Definir um conjunto de junção que intercepte as chamadas ao construtor das classes *Singleton*.
5. Identificar as classes que implementam o padrão *Singleton*, ou seja, as classes cuja instância deve ser única na aplicação (classe *Singleton*). Caso

seu construtor seja privado, transforme-o em público.

6. Substituir as chamadas ao método que retorna uma instância de cada classe *Singleton* para uma chamada ao construtor dessa classe.
7. Para cada classe *Singleton,* criar um aspecto concreto que estende o aspecto abstrato identificado anteriormente e declarar um relacionamento de realização de interface entre essa classe e a interface criada no passo 3.
8. Mover atributos e métodos relacionados a cada classe *Singleton* para seu respectivo aspecto.
9. Criar um adendo que implementa a lógica do padrão *Singleton*, ou seja, verifica se já há uma instância *Singleton* criada e a retorna ou cria uma nova instância. Relacionar esse adendo ao conjunto de junção criado anteriormente.

Mecanismo para ProAJ/UML

1. Identificar o aspecto “As” correspondente relacionado à implementação do padrão *Singleton*.
2. Verificar se o aspecto é abstrato. Se não for, transformá-lo em abstrato.
3. Criar uma interface vazia `Singleton`.
4. Definir o conjunto de junção `instance()` que intercepte as chamadas ao construtor das classes *Singleton*.
5. Identificar as classes “Cls” que implementam o padrão *Singleton*, ou seja, as classes cuja instância deve ser única na aplicação. Caso o construtor dessas classes seja privado (`private`), transforme-o em público (`public`).
6. Substituir as chamadas ao método que retorna uma instância de cada classe “Cl” para uma chamada ao construtor dessa classe.
7. Para cada classe “Cl” pertencente à “Cls”, criar um aspecto “AsN” derivado do aspecto “As” e declarar um relacionamento de realização de interface entre essa classe e a interface `Singleton` utilizando `declare implements`.
8. Mover atributos e métodos relacionados a cada classe “Cl” para seu respectivo aspecto “AsN”.
9. Criar um adendo do tipo `around` que retorna um objeto do tipo `Singleton`. Esse adendo implementa a lógica do padrão *Singleton*, ou seja, verifica se já uma instância criada e a retorna ou cria uma nova instância. Relacionar o adendo `around` ao conjunto de junção `instance()`.

R-*Observer*: O interesse transversal corresponde ao padrão de projeto *Observer*.

Aplicação

Quando há classes dedicadas à implementação do padrão *Observer*.

Motivação

A motivação para criação desta refatoração é análoga à motivação R-*Singleton*.

Mecanismo Geral

1. Identificar o aspecto relacionado à implementação do padrão *Observer*.
2. Criar um conjunto de junção que intercepte as operações nas quais os observadores devem ser notificados pelo elemento observado e capture a instância desse objeto observado.
3. Criar um conjunto de junção que intercepte a execução dos construtores dos elementos observados.
4. Criar um adendo que notifique os observadores após a execução das operações interceptadas pelo conjunto de junção criado no passo 2.
5. Criar um adendo que inicialize o conjunto de observadores após a execução dos construtores interceptados pelo conjunto de junção criado no passo 3.

Mecanismo para ProAJ/UML

1. Identificar o aspecto “As” correspondente relacionado à implementação do padrão *Observer*.
2. Criar os conjuntos de junção `pointcut hasChange()` e `pointcut targetSubject(Subject s)` que, respectivamente, intercepte as operações nas quais os observadores devem ser notificados pelo elemento observado e capture a instância desse objeto observado.
3. Criar um adendo `after` que notifica os observadores após a execução das operações interceptadas pelo conjunto de junção `hasChange() && targetSubject(Subject s)`.
4. Criar os conjuntos de junção `pointcut initializeObservers()`que intercepte a execução dos construtores dos elementos observados.
5. Criar um adendo `after` que inicializa o conjunto (lista, vetor) de

observadores após a execução dos construtores interceptados pelo conjunto de junção `initializeObservers()`.

4.4.4 Ordem para Aplicação das Refatorações Específicas

De modo análogo ao que acontece com as refatorações genéricas, a ordem de aplicação das refatorações específicas não interfere no produto final das refatorações, ou seja, no modelo de classes OA final. Isso ocorre, pois cada refatoração atua apenas sobre um determinado interesse de cada vez, não comprometendo elementos relacionados a outros interesses.

4.4.5 Considerações sobre a Aplicação das Refatorações Específicas

As estratégias de modularização utilizadas pelas refatorações elaboradas neste trabalho são condizentes com boas práticas de projeto OA, como: i) definição de um interesse por aspecto (evita o *bad smell God Aspect* (Piveta *et al.,* 2007)); ii) utilização de conjuntos de junção nomeados (melhora a legibilidade e manutenibilidade do software) e iii) especificação de conjuntos de junção semânticos (conjuntos de junção semânticos são aqueles baseados em estruturas abstratas de classes e interfaces, como métodos abstratos (Piveta *et al.,* 2007)).

Em alguns passos das refatorações específicas apresentadas anteriormente, há necessidade de análise do código fonte da aplicação. Por exemplo, no passo 2 da refatoração para o padrão *Observer*, deve ser criado um conjunto de junção `hasChange()` capaz de interceptar as operações da aplicação nas quais os observadores devem ser notificados. Nesse caso, o modelo de classes anotado contribui para aplicação da refatoração ao sinalizar, por meio de estereótipos, quais métodos possuem trechos de código relacionados com o padrão *Observer*. Entretanto, para saber qual(is) desse(s) métodos estereotipados realmente solicita(m) chamada(s) ao método que notifica os observadores, é preciso analisar o código fonte da aplicação. Isso ocorre em consequência da própria característica dos modelos de classes que é de não apresentar detalhes da implementação dos métodos. Dessa forma, se as refatorações descritas neste trabalho fossem aplicadas utilizando apenas o modelo de classes da UML, seria inevitável recorrer à análise do código fonte ou a outro tipo de modelo com característica comportamental, como o

modelo de sequência. Neste trabalho esse problema é solucionado com a utilização do metamodelo para instanciação de modelos de classes OO anotados proposto por Costa *et al.* (2009). Esse metamodelo foi criado com base no da UML e possui um nível de detalhamento maior, permitindo, por exemplo a descoberta de quais métodos invocam outros da aplicação (relacionamento `invokedMethods` da metaclasse `Operation` do metamodelo do DMAsp, Seção 4.3). Apesar de não serem visualizadas no modelo de classes da UML, essas informações estão presentes no arquivo XML que representa o modelo de classes anotado e podem ser consultadas a qualquer momento de forma manual ou automática. Com esse metamodelo, é possível aplicar as refatorações diretamente sobre os modelos de classes anotados, sem a necessidade de consulta ao código fonte da aplicação.

4.4.6 Exemplo da Aplicação das Refatorações Específicas para o Interesse Transversal de Persistência

Como comentado anteriormente, o metamodelo utilizado para criação dos modelos de classes anotados possui informações sobre os métodos invocados por outros métodos somente em arquivo XML. Assim, para facilitar o entendimento dos exemplos apresentados a seguir, notas da UML (*notes*) com trechos de código dos métodos são inseridas estrategicamente nos modelos.

Na Figura 4.22 é apresentada a aplicação de domínio bancário semelhante à da Figura 4.3. Entretanto, nesse exemplo, observa-se que a classe `Account` está estereotipadas com `<<Sec_Conn>>`, `<<Sec_Trans>>` e `<<Sec_Syn>>`, por ser afetada pelos interesses de gerenciamento de conexões, transações e sincronização, respectivamente.

Ainda com relação à Figura 4.22, observa-se que os interesses de gerenciamento de conexões, transações e sincronização que afetam a classe `Account` são secundários. Isso significa que essa classe não foi desenvolvida originalmente para implementar qualquer um desses interesses. Além disso, as classes `Account` e `Customer` possuem o estereótipo `<<Pri_Entity>>`, indicando que essas classes são classes persistentes.

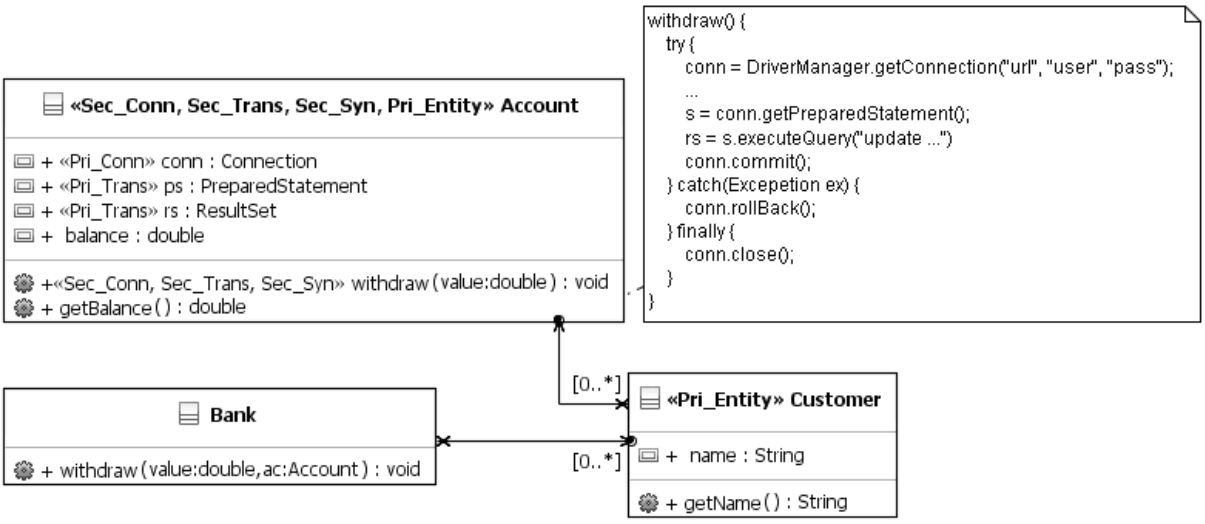

Figura 4.22 – Modelo OO Estereotipado com Interesses de Conexão, Transação e Sincronização.

4.4.6.1 Aplicação das Refatorações Genéricas

Com o objetivo de modularizar os interesses transversais existentes na aplicação apresentada na Figura 4.22, o conjunto de refatorações genéricas foi aplicado obtendo-se o modelo apresentado na Figura 4.23.

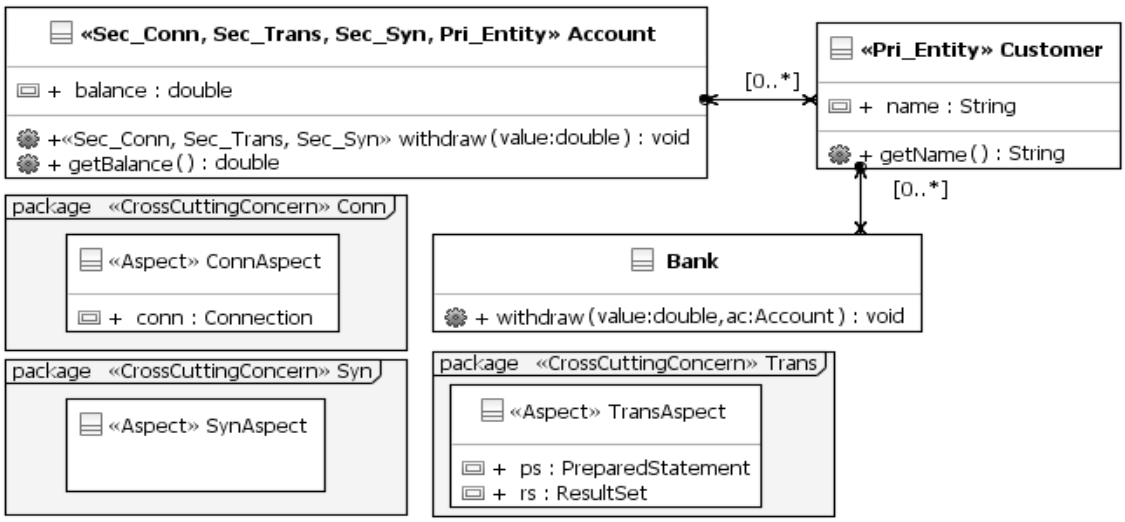

Figura 4.23 – Modelo OA Parcial Após a Aplicação das Refatorações Genéricas.

As modificações realizadas foram: i) criaram-se três aspectos `ConnAspect`, `TransAspect` e `SynAspect`; ii) os atributos `PreparedStatement ps` e `ResultSet rs` da classe `Account` foram movidos para o aspecto `TransAspect`; e iii) o atributo `Connection conn` foi movido para o aspecto `ConnAspect`. A refatoração aplicada para os três interesses foi a R-3 (Subseção 4.3.2). No exemplo apresentado na Figura 4.22 os interesses de gerenciamento de conexões, transações e

sincronização apresentam essa configuração. Salienta-se que a classe `Account` continua com os três estereótipos `<<Sec_Trans>>`, `<<Sec_Conn>>` e `<<Sec_Syn>>`, pois ainda há métodos nessa classe afetados pelos interesses correspondentes a esses estereótipos.

4.4.6.2 Aplicação da Refatoração Específica para Gerenciamento de Conexões (R-*Connection*)

Nesse ponto, há necessidade da aplicação de refatorações específicas para modularização dos interesses de gerenciamento de conexões, transações e sincronização. Inicialmente, será refatorado o interesse de gerenciamento de conexões. Ao aplicar a refatoração para esse interesse, tem-se o modelo OA apresentado na Figura 4.24. Para facilitar a visualização, apenas os elementos que sofreram alterações foram apresentados.

Observa-se que os estereótipos `<<Sec_Conn>>` foram removidos da classe `Account`, pois o interesse de gerenciamento de conexões foi modularizado nos aspectos `ConnAspect` e `AccountConnectionAspect`. O aspecto `ConnAspect` é um aspecto abstrato que possui os conjuntos de junção também abstratos `openConnection` e `closeConnection`. Esses conjuntos de junção são concretizados no aspecto `AccountConnectionAspect` e são responsáveis por especificar os pontos da aplicação nos quais é necessário que haja controle de abertura e fechamento da conexão com o Banco de Dados. Nesse exemplo, esse ponto é o método `withdraw()`, o que pode ser percebido na nota colocada no modelo da Figura 4.22. O método `getConnection()` da classe `DriverManager` retorna uma conexão aberta com o SGBD e o método `close()` da classe `Connection`, fecha essa conexão.

Ainda existem alguns estereótipos no modelo apresentado na Figura 4.24. A intenção é removê-los, assim como foi feito com o estereótipo `<<Sec_Conn>>`. Para isso, serão aplicadas as refatorações para gerenciamento de transações e sincronização.

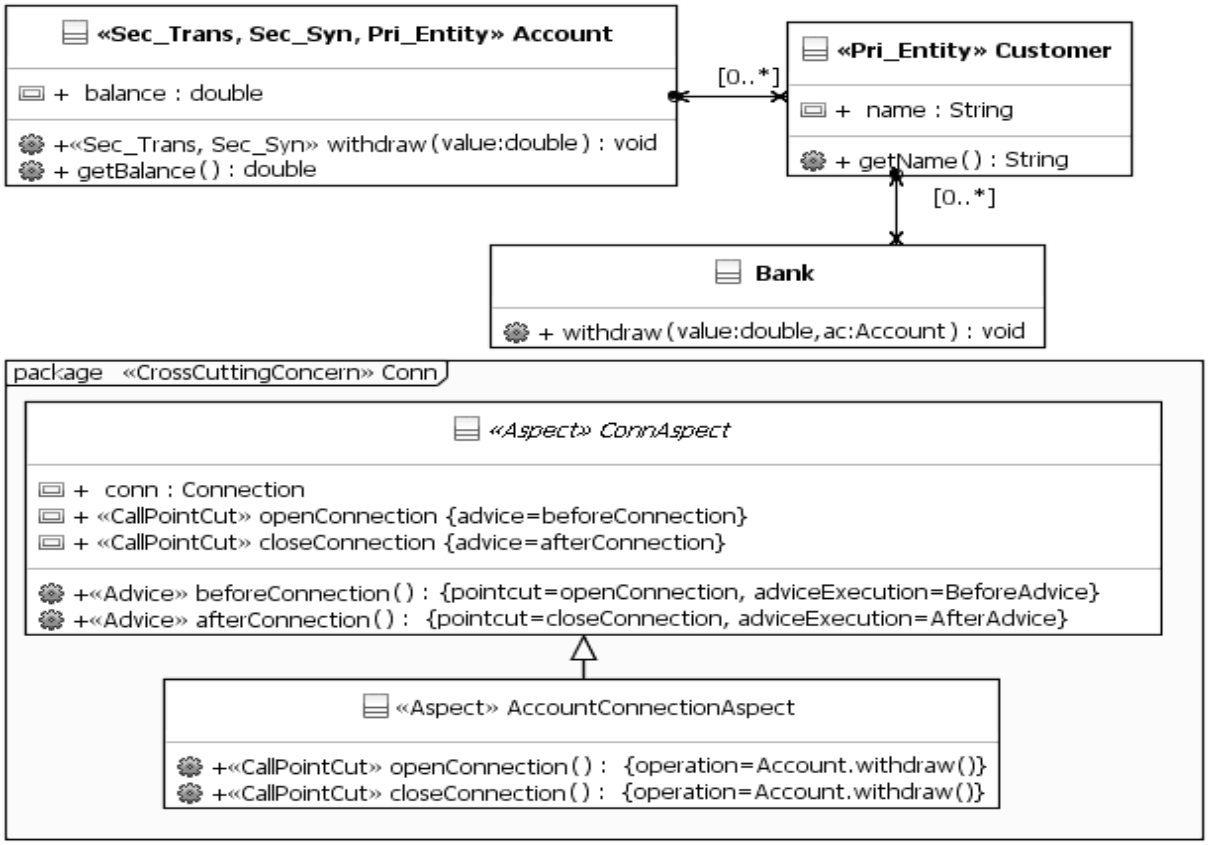

Figura 4.24 – Modelo OA Após a Aplicação da Refatoração Específica para Controle de Conexão.

4.4.6.3 Aplicação da Refatoração Específica para Gerenciamento de Transações (R-*Transaction*)

O procedimento para aplicação da refatoração específica para gerenciamento de transações é similar ao apresentado anteriormente para conexão. Após a execução dos passos correspondentes a essa refatoração, o resultado obtido é o modelo OA apresentado na Figura 4.25.

Os estereótipos `<<Sec_Trans>>` foram removidos da classe `Account`, pois o interesse de gerenciamento de transações foi modularizado nos aspectos `TransAspect` e `AccountTransactionAspect`. O aspecto abstrato `TransAspect` possui um conjunto de junção abstrato `transactionalMethods` que é concretizado no aspecto `AccountTransAspect`. Esse conjunto de junção concreto especifica os pontos da aplicação nos quais é necessário que haja gerenciamento de transações.

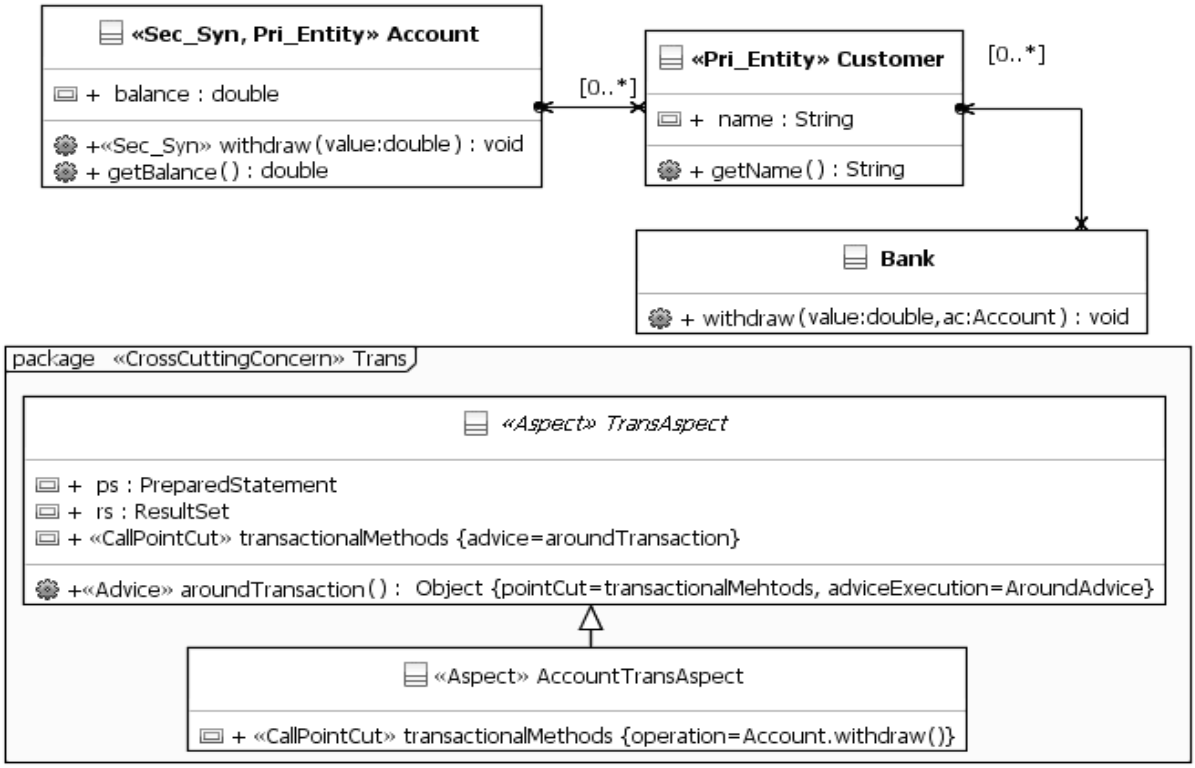

Figura 4.25 – Modelo OA Após a Aplicação da Refatoração Específica para Controle de Transação.

4.4.6.4 Aplicação da Refatoração Específica para Sincronização de Objetos em Memória com o Banco de Dados (*R-Sync*)

O último interesse a ser modularizado é o de sincronização, representado pelo estereótipo <<Sec_Syn>>. Para modularizá-lo deve-se aplicar a refatoração *R-Sync* e como pode ser visto na descrição dessa refatoração (Subseção 4.4.1), há um passo intermediário que consiste na criação de uma interface `Persistent` e na realização dessa interface por todas as classes persistentes. Nesse caso, o estereótipo utilizado para determinar as classes persistentes é <<Pri_Entity>>.

Após aplicar os passos da refatoração para sincronização gera-se o modelo OA apresentado na Figura 4.26.

Todos os estereótipos foram removidos das classes `Account` e `Customer`, pois os interesses transversais existentes na aplicação original (OO) foram completamente modularizados em aspectos.

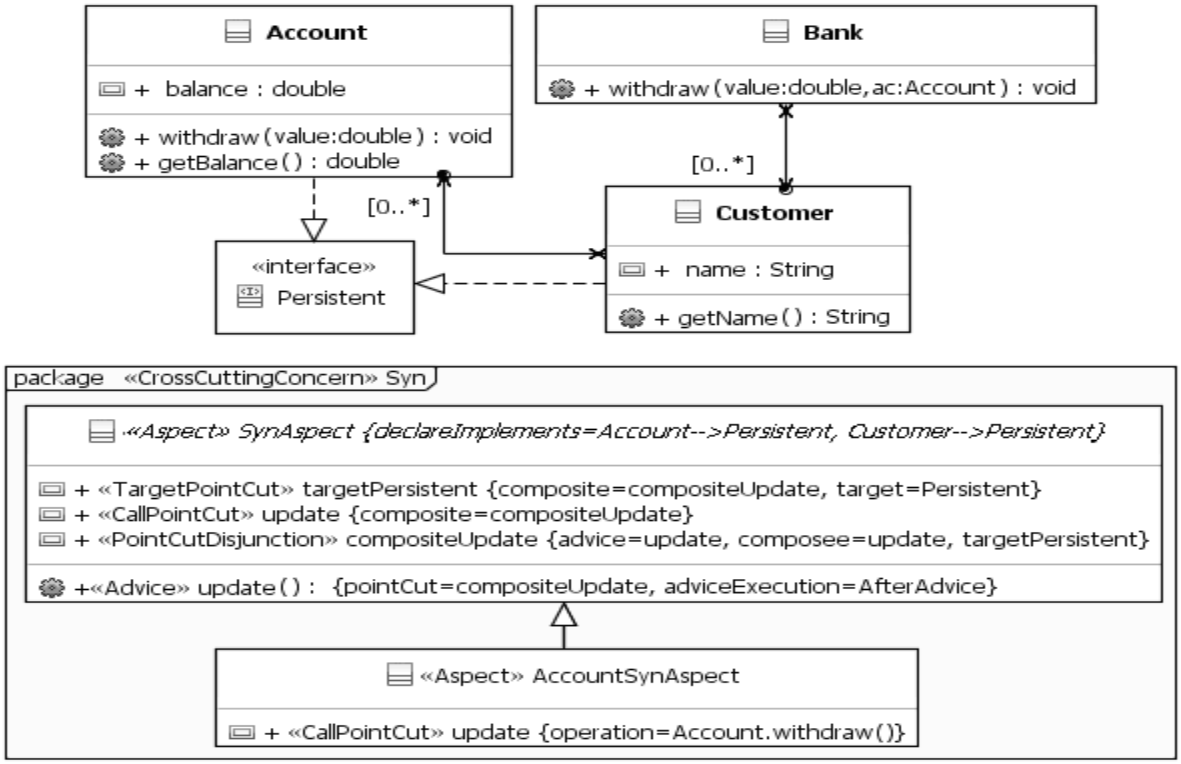

Figura 4.26 – Modelo OA Após a Aplicação da Refatoração Específica para Controle de Sincronização.

Os aspectos responsáveis pelo interesse de sincronização são `SynAspect` e `AccountSynAspect`. O aspecto `SynAspect` possui o conjunto de junção abstrato `update` que é especializado no subaspecto `AccountSynAspect`. Esse conjunto de junção especifica as operações que realizam alteração de dados no Banco de Dados da aplicação, nesse caso, o método `withdraw()` da classe `Account`. Além disso, foi criada uma interface `Persistent` que é realizada pelas classes `Account` e `Customer`. Essa realização de interface é declarada no aspecto `SynAspect` por meio de construções `declare implements`. De acordo com a nota da Figura 4.22, que exibe um trecho do código fonte do método `withdraw()`, apenas a atualização de dados é necessária (o método `executeQuery()` executa o comando SQL “update”), por isso, o único conjunto de junção criado foi `update`. Já na classe `Customer`, não há utilização de métodos para sincronização, por isso, não foi criado um aspecto específico para essa classe.

4.4.7 Exemplo da Aplicação das Refatorações Específicas para o Interesse Transversal de *Logging* e para os Padrões de Projeto *Singleton* e *Observer*

Na Figura 4.27 é apresentada uma aplicação simples de desenho que imprime pontos 2D na tela do computador e em uma impressora.

A classe `Point` é responsável por armazenar os dados relacionados a um ponto, como suas coordenadas *x* e *y*. A classe `Screen` é responsável por exibir pontos na tela e em uma impressora implementada pela classe `Printer`. Para garantir que a tela seja atualizada todas as vezes em que a coordenada de um ponto for alterada, utilizou-se o padrão *Observer* (Gamma *et al.*, 1995). Assim sendo, as classes `Point` e `Screen` fazem parte da implementação desse padrão juntamente com as interfaces `Observer` e `Subject`. A classe `Printer`, que define uma impressora do sistema, deve possuir apenas uma instância na aplicação e, por isso, foi implementada com o padrão *Singleton* (Gamma *et al.*, 1995).

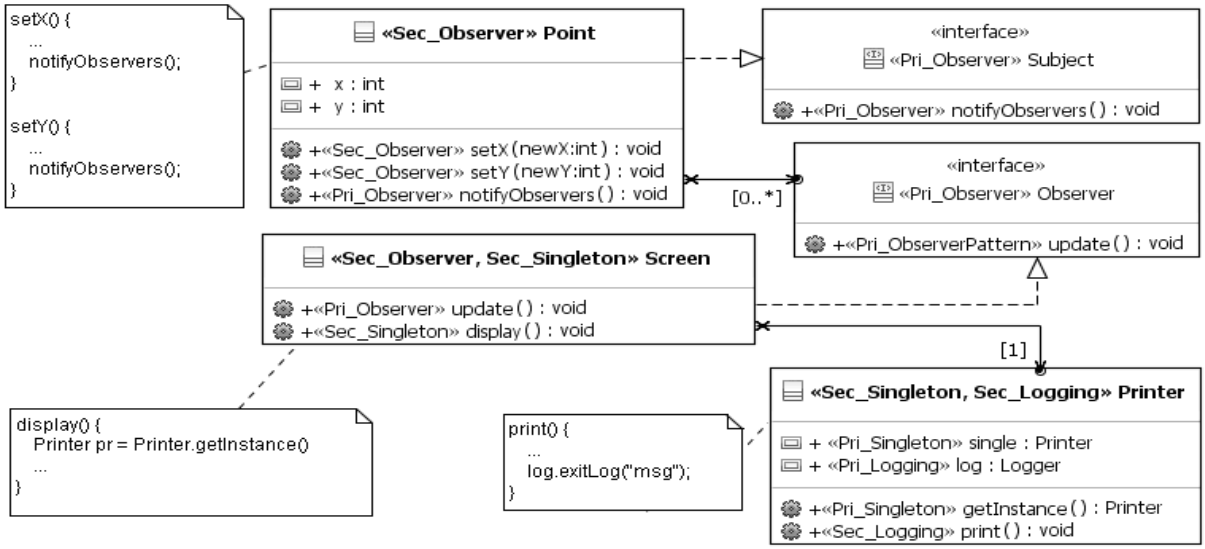

Figura 4.27 – Modelo OO Estereotipado com Interesses de *Logging*, *Singleton* e *Observer*.

Além de imprimir os dados do ponto, a classe `Printer` grava um registro do contexto de execução da aplicação toda vez que seu método `print()` é invocado. Isso é feito por meio do atributo `Logger log` dessa classe.

Os interesses presentes nessa aplicação são: *logging* e os padrões de projeto *Observer* e *Singleton*, o que pode ser percebido pelos estereótipos presentes no diagrama de classes da Figura 4.27. A modularização parcial desses interesses

transversais foi realizada por meio das refatorações genéricas, obtendo como saída o modelo apresentado na Figura 4.28.

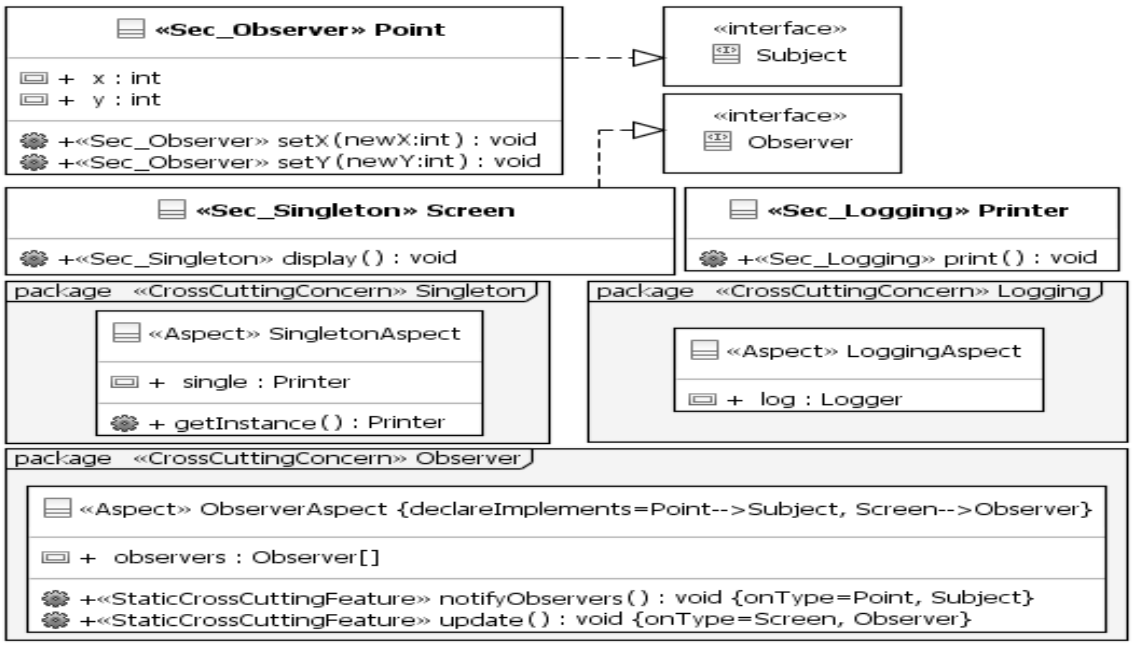

Figura 4.28 – Modelo OA Parcial Após a Aplicação das Refatorações Genéricas.

4.4.7.1 Aplicação da Refatoração Específica para *Logging* (R-*Logging*)

Ao aplicar a refatoração para o interesse de *logging* tem-se o modelo OA apresentado na Figura 4.29.

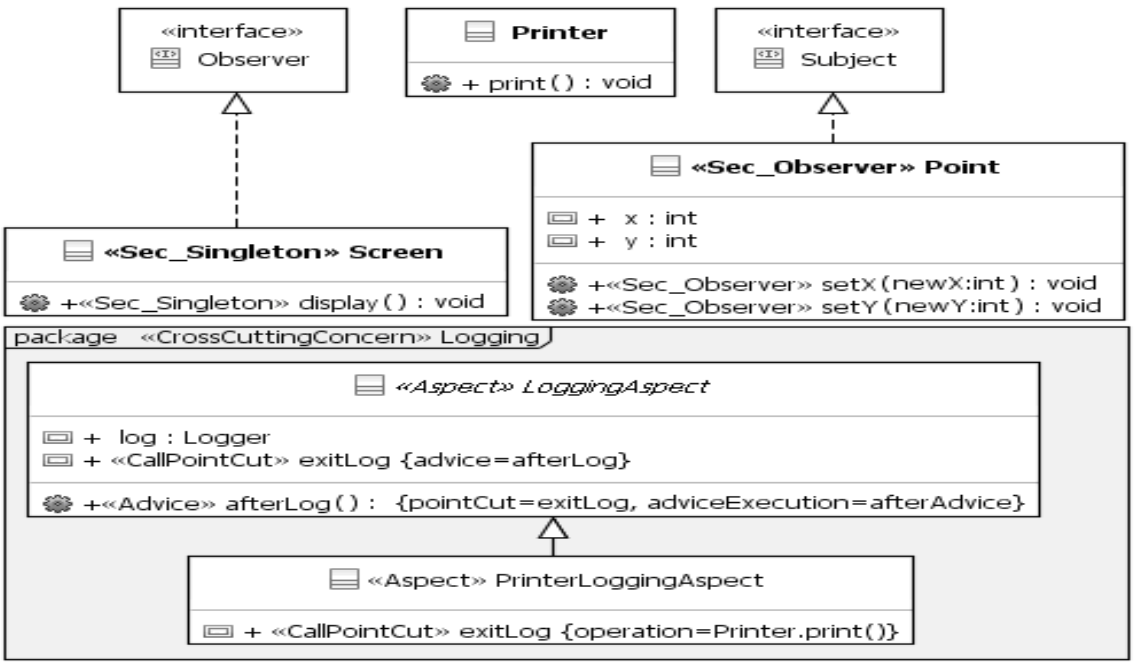

Figura 4.29 – Modelo OA Após a Aplicação das Refatorações Específica para *Logging*.

Observa-se que o estereótipo `<<Sec_Logging>>` foi removido da classe `Printer`. Nesse exemplo, apenas um conjunto de junção `exitLog` foi criado, pois o registro de `log` é feito apenas ao final do método `print()` da classe `Printer`, como pode ser observado na nota colocada no modelo da Figura 4.27. Esse conjunto de junção é concretizado no aspecto `PrinterLoggingAspect` e é responsável por especificar os pontos da aplicação nos quais é necessário que haja um registro de *log*.

4.4.7.2 Aplicação da Refatoração Específica para o Padrão de Projeto Singleton (R-*Singleton*)

Após aplicar a refatoração para padrão *Singleton* é gerado o modelo OA apresentado na Figura 4.30.

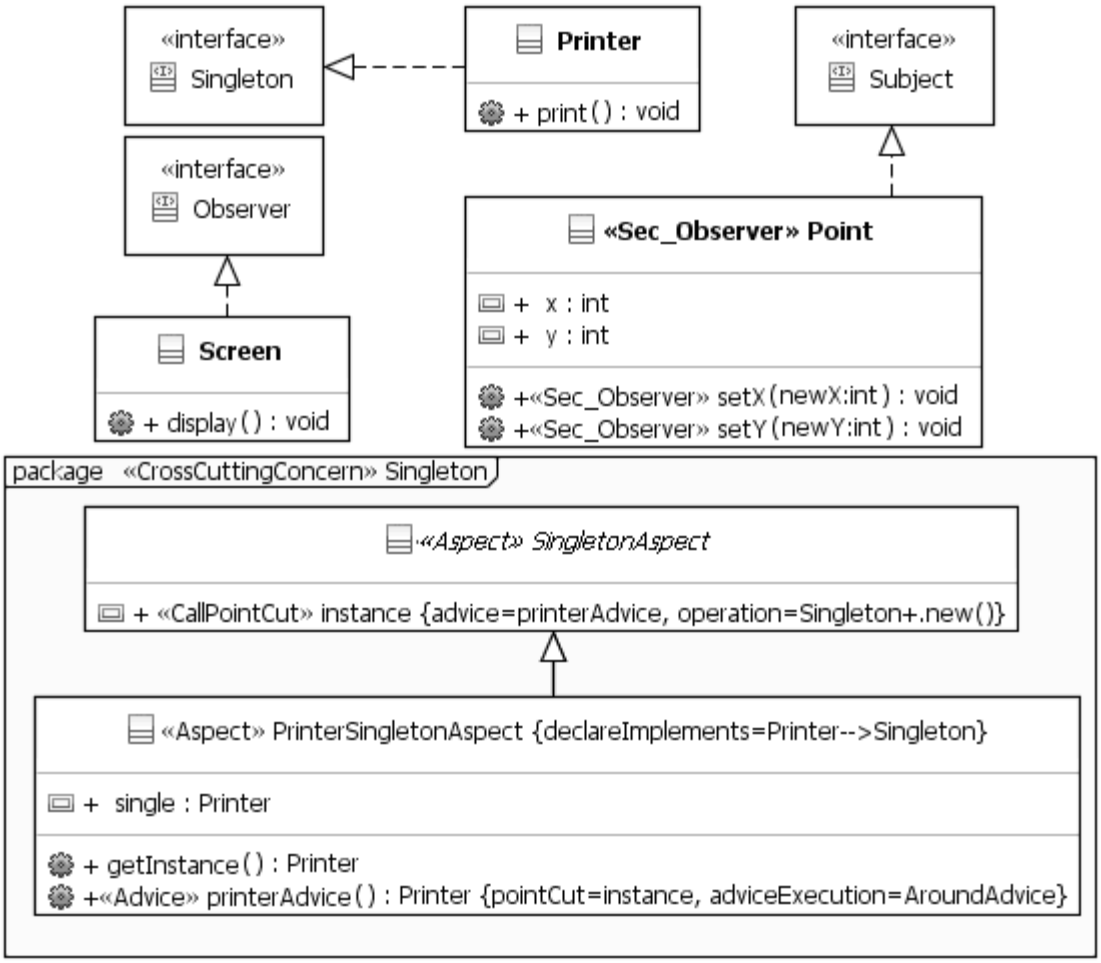

Figura 4.30 – Modelo OA Após a Aplicação das Refatorações Específica para o Padrão *Singleton*.

As modificações realizadas foram: i) criou-se uma interface denominada `Singleton` e estabeleceu o relacionamento de realização de interface entre a classe

`Printer` e `Singleton`; ii) criou-se um conjunto de junção `instance` que captura as chamadas aos construtores das classes quem implementam a inteerface `Singleton`; e iii) criou-se um adendo responsável por criar uma nova instância ou retornar uma instância já existente da classe `Printer` toda que você o conjunto de junção `instance` for alcançado. Com isso, eliminou-se a interferência do Padrão *Singleton* nas classes `Printer` e `Screen` e a responsabilidade pela implementação do Padrão foi modularizada nos aspectos `SingletonAspect` e `PrinterSingletonAspect`.

A modularização obtida após a aplicação dessa refatoração é similar à solução proposta por Hannemann e Kiczales (2002) para implementação do padrão *Singleton*. A diferença observada é que na solução apresentada por Hannemann e Kiczales (2002), é criado um conjunto de junção que captura todas as chamadas ao construtor da classe *Singleton* e é deixado para que o usuário defina pontos do código que não devem ser capturados por esse conjunto de junção. A solução escolhida neste trabalho visou à adequação ao contexto da refatoração de modelos.

4.4.7.3 Aplicação da Refatoração Específica para o Padrão de Projeto *Observer* (R-*Observer*)

Após aplicar a refatoração para padrão *Observer* é gerado o modelo OA apresentado na Figura 4.31.

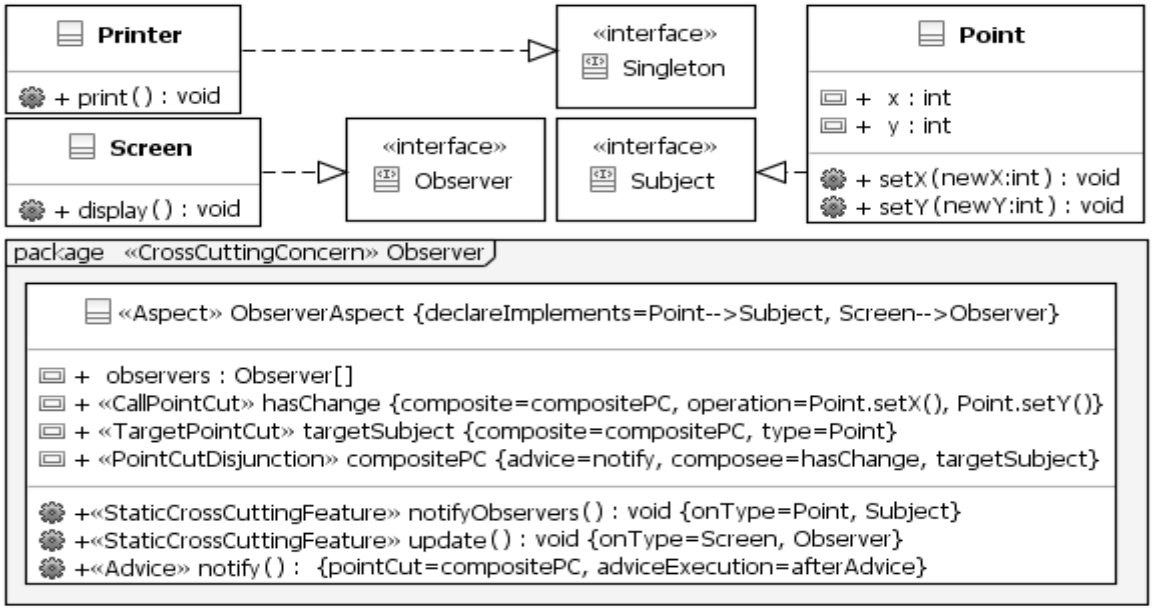

Figura 4.31 – Modelo OA Após a Aplicação das Refatorações Específica para o Padrão *Observer*.

Nesse exemplo, criou-se o conjunto de junção `hasChange`, que é responsável por determinar os pontos da aplicação onde pode ocorrer alguma modificação nos elementos observados (*Subjects*). Isso ocorre nos métodos `setX()` e `setY()` da classe `Point` (nota da Figura 4.27). O adendo `notify` é responsável por notificar os observados (*Observers*) toda vez em que o conjunto de junção `hasChange` for alcançado.

4.5 Considerações Finais

Neste capítulo discutiu-se as principais vantagens da elaboração e utilização de refatorações em nível de modelos. Um conjunto de refatorações que podem ser aplicadas a qualquer tipo de interesse transversal (refatorações genéricas) foram apresentadas. Essas refatorações são importantes, pois oferecem uma estratégia inicial para modularização de interesses transversais com base nos níveis de entrelaçamento/espalhamento desses interesses na aplicação. Como essas refatorações são aplicáveis a qualquer tipo de interesse, a solução dada por elas geralmente é parcial, ou seja, podem existir interesses que ainda não foram completamente modularizados na aplicação. Para completar a modularização desses interesses, refatorações específicas devem ser aplicadas.

Refatorações específicas são dependentes do tipo do interesse, com passos adequados para refatoração desse tipo de interesse. Essas refatorações são responsáveis por refinar o modelo de classes OA parcial obtido com as refatorações genéricas. Neste capítulo, foi apresentado ainda um breve estudo sobre a influência da ordem de aplicação das refatorações em uma aplicação OO. Para cada refatoração especificada, exemplos foram utilizados para ilustrar sua aplicabilidade em softwares OO.

Capítulo 5

MoBRe: Um Apoio Computacional para Refatoração de Modelos de Classes OO Anotados

5.1 Considerações Iniciais

Os modelos de classes OO anotados podem ser melhorados para refletir uma modularização mais adequada dos interesses transversais existentes no software que está sendo aprimorado. O uso de refatorações é uma das técnicas para melhorar a qualidade desses modelos.

Como apresentado no Capítulo 4, neste trabalho foram definidas nove refatorações em nível de modelo (3 refatorações genéricas e 6 específicas). Para aplicação das refatorações, um conjunto de passos deve ser executado a fim de produzir como saída um modelo OA parcial ou completo, no qual os interesses transversais existentes no software são modularizados. Entretanto, a execução manual desses passos utilizando modelos de classes de softwares de média e grande escala pode ser custoso e propenso a erros. Para amenizar esse esforço foi desenvolvido um *plug-in* denominado MoBRe que é capaz de realizar de modo semiautomático as tarefas relacionadas à refatoração de interesses transversais.

Este Capítulo está organizado em sete seções, esta que é a de Considerações Iniciais. Na Seção 5.2 são apresentados os apoios computacionais

ComSCId e DMAsp que são responsáveis, respectivamente, pela identificação de interesses transversais e pela recuperação de modelos de classes OO anotado a partir do código fonte.

Na Seção 5.3 são descritas as principais características e a interface do MoBRe. Na Seção 5.4 um exemplo da utilização do MoBRe para refatoração de um interesse transversal é apresentado. Na Seção 5.5 descreve-se como o *plug-in* pode ser estendido com o acréscimo de novos tipos de refatorações. Na Seção 5.6 é apresentado esquematicamente como foi realizada a integração entre as ferramentas ComSCId, DMAsp e MoBRe e como elas podem ser utilizadas em conjunto para modularização de interesses transversais em um software OO. As considerações finais são apresentadas na Seção 5.7.

5.2 Os Apoios Computacionais ComSCId (*Computational Support for Concern Identification*) e DMAsp (*Design Model to Aspect*)

O apoio computacional ComSCId foi desenvolvido como um *plug-in* do Eclipse e possibilita a identificação de indícios de interesses transversais em um software legado OO implementado em Java por meio da técnica de mineração baseada em análise de tipos e texto.

As cadeias de caracteres e tipos de dados utilizados pelo ComSCId para identificar um determinado candidato à semente são denominados “regras”. Os conceitos relacionados às técnicas de mineração de interesses transversais, como sementes e candidatos a sementes foram apresentados na Seção 3.2. ComSCId possui um repositório de regras e um módulo que permite o gerenciamento desse repositório, tornando possível o armazenamento de novas regras, remoção e atualização das regras existentes para quaisquer interesses.

Os trechos de código identificados pelo ComSCId são denominados candidatos à semente, pois a utilização de convenções de nomenclatura para mineração de interesses transversais pode gerar falsos positivos, ou seja, trechos de código que não representam sementes, mas que foram identificados. Assim, é necessário que o Engenheiro de Software faça uma análise dos candidatos à semente, verificando se eles são sementes verdadeiras ou são falsos positivos.

Caso sejam classificados pelo Engenheiro de Software como falsos positivos, o repositório de regras cadastrado no ComSCId deve ser atualizado para evitar a identificação desses falsos positivos na próxima vez em que a ferramenta foi executada.

O repositório de regras do ComSCId é específico para cada sistema analisado. Assim, cada sistema pode ter um conjunto específico de regras cadastrado pelo Engenheiro de Software. Há também a possibilidade de importar o conjunto de regras de um determinado sistema para ser utilizado em outros projetos. A existência do repositório de regras é o principal diferencial do ComSCId em relação às demais ferramentas para mineração de interesses transversais que implementam a técnica de análise de tipos e texto (Zhang e Jacobsen, 2003; Hannemann e Kiczales, 2001; William *et al.*, 1999).

Os indícios de interesses transversais originalmente contemplados pelo ComSCId são: i) persistência em banco de dados (*database persistence*); ii) controle de registro de informações (*logging*); e iii) persistência em memória temporária (*buffering*). Para cada projeto Eclipse, um diretório “indications” é criado contendo um arquivo chamado “indications.xml” que armazena as regras criadas para identificação de sementes de Interesses transversais.

DMAsp consiste de um *plug-in* para Eclipse que estende a funcionalidade do ComSCId, representando as sementes de Interesses transversais em modelos de classe da UML. Seu principal objetivo é elaborar um modelo OO intermediário antes da obtenção de um modelo OA, visando a melhorar a qualidade do processo de reengenharia de software OO para OA. Esse modelo intermediário é anotado com sementes de Interesses transversais, utilizando estereótipos nas classes, nos atributos e nos métodos. Esses estereótipos representam os Interesses transversais que entrecortam esses elementos do sistema. Para que essa representação seja possível, o ComSCId passou por modificações no sentido de realizar chamadas a métodos do DMAsp sempre que indícios de interesses transversais forem identificados.

Antes de apresentar os modelos de classes anotados construídos com o auxílio do DMAsp, alguns conceitos são definidos:

- **Componentes afetados por um interesse** são elementos de software, como atributos, métodos, classes e interfaces, que possuem sementes de interesses transversais. Nos modelos de classes anotados esses

elementos são representados com estereótipos dos interesses que os afetam. Por exemplo, uma classe afetada pelo interesse de Persistência receberá o estereótipo `<<Persistence>>`, onde *Persistence* é um nome arbitrário escolhido para representar esse tipo de interesse;

- **Interesse primário** é o interesse principal de um elemento de software e está relacionado com a razão pela qual ele foi criado. Por exemplo, se um determinado método foi criado para realizar conexão com o banco de dados, então "conexão" é o interesse primário desse método. No modelo de classes anotado os estereótipos relacionados aos interesses primários são precedidos do prefixo "Pri_" (por exemplo, `<<Pri_Connection>>`, `<<Pri_Logging>>`). Para que um elemento seja considerado primário, ele precisa estar cadastrado no conjunto de regras do ComSCId para identificação de um determinado interesse transversal. Por exemplo, se uma classe BD foi criada para persistência de objetos da aplicação, ele precisa ser cadastrada no repositório de regras do ComSCId e estar associada ao interesse de persistência. Dessa forma, ela receberá o estereótipo `<<Pri_Peristence>>`, onde `Persistence` corresponde ao nome dado ao interesse relacionado à persistência de dados. Salienta-se que não apenas classes, mas interfaces, atributo e métodos podem ser cadastrados no conjunto de regras do ComSCId;
- **Interesse secundário** de um elemento de software corresponde a determinadas funções que esse componente desempenha, mas que não estão diretamente relacionadas com a razão pela qual ele foi criado. Por exemplo, um método criado para realizar conexão com o banco de dados pode possuir um interesse secundário de *logging*. No modelo de classes anotado os estereótipos relacionados aos interesses secundários são precedidos do prefixo "Sec_" (por exemplo, `<<Sec_Connection>>`, `<<Sec_Logging>>`). Para que um elemento receba um estereótipo do tipo "Sec_" relacionado a algum interesse, ele não pode estar cadastrado no conjunto de regras para identificação desse interesse. Ao invés disso, ele precisa estar relacionado a algum elemento que possua um estereótipo do tipo "Pri_", ou seja, que tenha como interesse primário o interesse em questão. Por exemplo, se uma classe Conta se relaciona por meio de um

associação com a classe BD que possui como interesse primário persistência, então a classe Conta receberá um estereótipo `<<Sec_Persistence>>`;

- **Componentes bem modularizados** são elementos de software compostos apenas pelo Interesse Primário para o qual foram desenvolvidos. Por exemplo, um método `abrirConexao()` é considerado bem modularizado se o único tipo de estereótipo desse método é relacionado ao interesse de abertura de conexões com o Banco de Dados. Ou seja, nesse método não há presença de outros interesses primários, nem de interesses secundários.
- **Interesses bem modularizados** são interesses localizados apenas nos componentes que foram criados para implementação desses interesses. Por exemplo, um interesse `Peristence` é considerado bem modularizado se os componentes onde ele se encontra implementado o têm como interesse primário. Ou seja, não há componentes no sistema que têm `Persistence` como um Interesse Secundário (`<<Sec_Persistence>>`).

Na Figura 5.1 é apresentado o modelo de classe recuperado a partir do código fonte da classe `Account` da Figura 3.2.

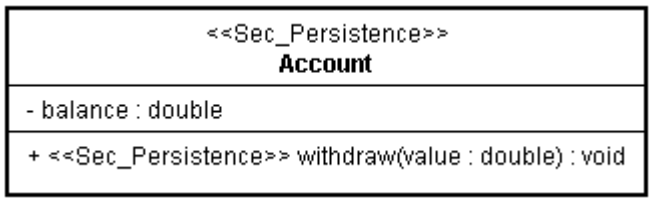

Figura 5.1 – Classes Afetadas pelo Interesse de Persistência.

É possível observar que o atributo `connection` e o método `withdraw()`apresentam estereótipos do tipo `<<Sec_Persistence>>` indicando que eles são afetados pelo interesse de persistência, que é um interesse secundário nessa classe. Salienta-se que atributos, métodos e classes/interfaces podem receber mais de um estereótipo, por exemplo, `<<Sec_Persistence>>` e `<<Sec_Logging>>`.

As informações necessárias para gerar o modelo de classes OO anotado, como classes, associações, operações, atributos e interesses, são representadas por meio de um metamodelo apresentado na Figura 5.2 (Parreira Júnior *et al.*, 2010b).

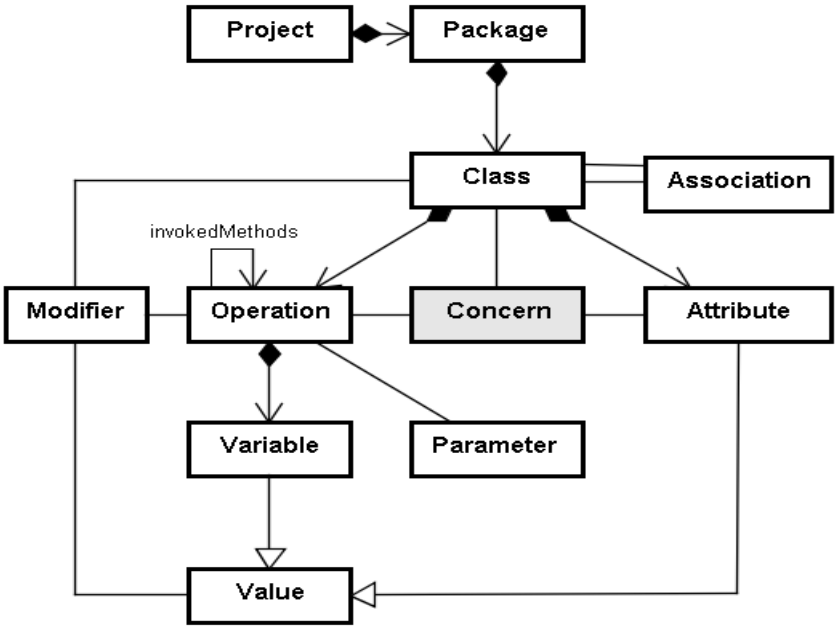

Figura 5.2 – Metamodelo para definição de modelos de classe OO anotados com indícios de interesses transversais (Parreira Junior *et al.*, 2010b).

Nesse metamodelo, a metaclasse `Project` representa um projeto do Eclipse e armazena o conjunto de pacotes com o código fonte do software em análise. A metaclasse `Concern` representa um interesse transversal que pode estar associado a uma ou mais classes, métodos e atributos. As demais metaclasses correspondem a elementos comuns de OO e da linguagem Java.

Instâncias geradas a partir desse metamodelo são representadas em um arquivo XML (*eXtensible Markup Language*), no qual os nomes das *tags* correspondem às respectivas metaclasses desse metamodelo. Além desse arquivo, um arquivo XMI (*XML Metadata Interchange*) é gerado utilizando as informações do arquivo XML. Para isso, foi utilizado um arquivo XSL (*eXtensible Stylesheet Language for Transformation*) que gera um documento com *tags* específicas para ser importado no ambiente Elipse (2011). Eclipse foi escolhido por ser *open source*, apresentar compatibilidade com a UML 2.X e ser o ambiente no qual os *plug-ins* ComSCId e DMAsp foram implementados, mantendo assim, um ambiente comum para mineração de visualização de interesses transversais.

As informações necessárias para gerar o modelo de classes são armazenadas em um arquivo XML (“XMLModel.xml”). Esse arquivo é obtido após a execução do ComSCId e é independente de qualquer ferramenta CASE. Além do arquivo XML, um arquivo XMI é criado usando as informações presentes no arquivo

"XMLModel.xml". No arquivo XMI as informações encontram-se organizadas de modo que esse arquivo possa ser importado no ambiente Eclipse.

Para transformar o arquivo "XMLModel.xml" no arquivo XMI foi utilizado um arquivo XSL (*eXtensible Stylesheet Language for Transformation*). Esse arquivo permite a reorganização das informações existentes no arquivo XML para atender as especificidades do formato reconhecido pelo Eclipse. Para gerar modelos de classes que possam ser lidos por outras ferramentas, deve-se criar apenas o arquivo XSL que possibilite reorganizar as informações existentes no arquivo "XMLModel.xml" segundo o formato exigido pela ferramenta CASE desejada.

Na Figura 5.3 é apresentado esquematicamente como o modelo de classes OO anotado é recuperado (Costa *et al.*, 2009).

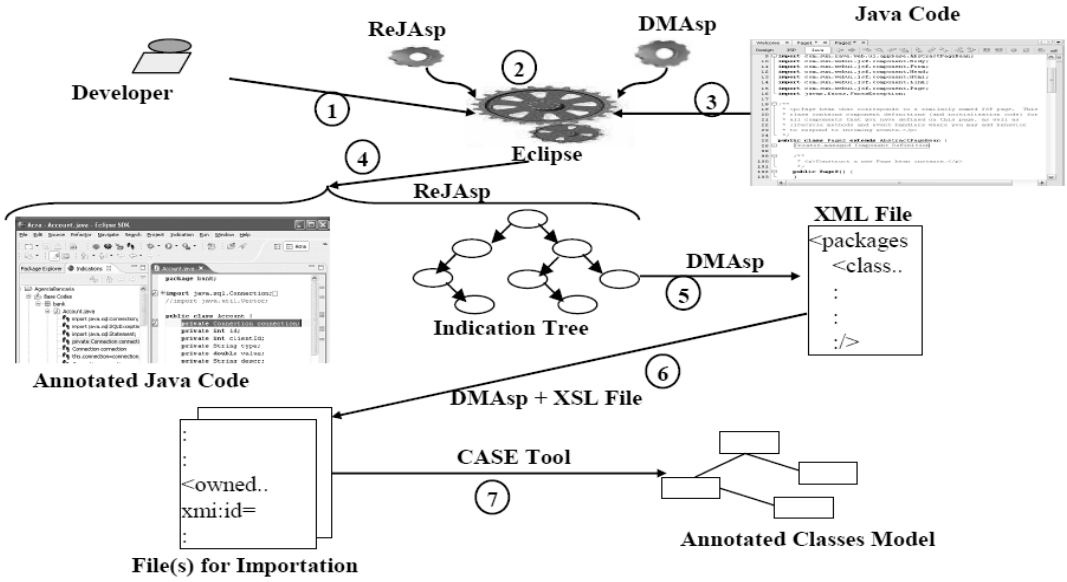

Figura 5.3 – Obtenção do modelo de classes OO anotado (Costa *et al.*, 2009).

O desenvolvedor usa a IDE Eclipse (1) juntamente com o *plug-in* ComSCId e DMAsp (2) para abrir o projeto cujo código será analisado (3). ComSCId encontra os trechos do código com indícios de interesses transversais e constrói uma árvore de indicações (4). DMAsp, por sua vez, serializa os objetos contidos na árvore de indicações e cria um arquivo XML (5). Esse arquivo XML juntamente com um arquivo XSL são usados pelo DMAsp para construção de um outro arquivo com extensão XMI que poderá ser importado por uma ferramenta CASE específica. Para isso, o arquivo XSL deve conter regras de transformação específicas para converter o arquivo XML, que é independente de plataforma, para um arquivo que possa ser utilizado por uma ferramenta específica, por exemplo, Astah*, Magic Draw, Borland

Together, entre outras. Após esses passos, o arquivo gerado pelo DMAsp pode ser importado em uma ferramenta CASE que irá representar o modelo de classes OO anotado (7).

De posse do modelo de classes OO anotado, o Engenheiro de Software pode ter melhor visão do software existente em nível de projeto e propor um documento de projeto mais consistente com os conceitos e propriedades da orientação a aspectos. Por exemplo, pode optar por encapsular parte do software em um ou vários aspectos, usar um *framework* de persistência, usar padrões de projeto, usar idiomas, usar uma combinação dessas alternativas ou usar outra alternativa. Esse modelo intermediário possibilita a construção de um modelo de classes OA bem como a documentação do software existente com mais informações, indícios de interesses transversais, do que tradicionalmente as/os ferramentas/apoios computacionais apresentam. Porém, com o metamodelo do DMAsp é possível especificar apenas modelos orientados a objetos, uma vez que não possui elementos de modelagem específicos para a tecnologia OA. Nessa dissertação, o metamodelo do DMASp foi estendido para que se torne adequado ao contexto da Modelagem de Software Orientado a Aspectos (MOA).

5.3 MoBRe (*MOdel-Based Refactorings*)

As tarefas de identificação dos cenários para aplicação das refatorações e de sua execução podem demandar grande esforço caso sejam realizadas sem apoio computacional. Um *plug-in* Eclipse denominado MoBRe foi desenvolvido para identificar cenários propícios para aplicação das refatorações em modelos de classes OO anotados com indícios de interesses transversais.

MoBRe foi desenvolvido com base no *plug-in* DMAsp (Costa *et al.*, 2009) e permite transformar um modelo OO anotado em um modelo OA parcial, quando as refatorações genéricas são aplicadas, ou em um modelo OA final, quando as refatorações específicas são aplicadas. Como pode ser visto na Figura 5.4, esse *plug-in* trabalha em conjunto com outros dois: ComSCId (Parreira Júnior *et al.*, 2010a; Parreira Júnior *et al.*, 2010b) e DMAsp, e permite: i) identificar indícios de interesses transversais em código fonte Java (ComSCId); ii) recuperar o modelo de

classes OO anotado (com a utilização de estereótipos) com os indícios identificados (DMAsp); e iii) aplicar refatorações genéricas e específicas sobre os modelos recuperados pelo DMAsp para construir modelos de classes OA (MoBRe). Além disso, MoBRe é um *plug-in* extensível, ou seja, novas refatorações podem ser adicionadas pelo Engenheiro de Software para atender as suas necessidades; essa característica é melhor detalhada na Seção 5.5.

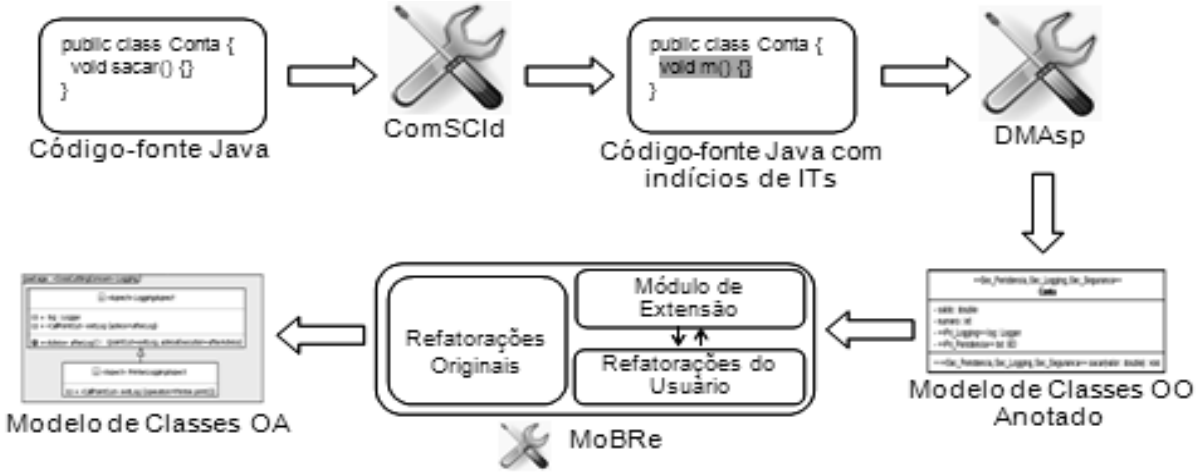

Figura 5.4 – Arquitetura do *plug-in* MoBRe.

Na Figura 5.5 é apresentada a interface padrão do MoBRe.

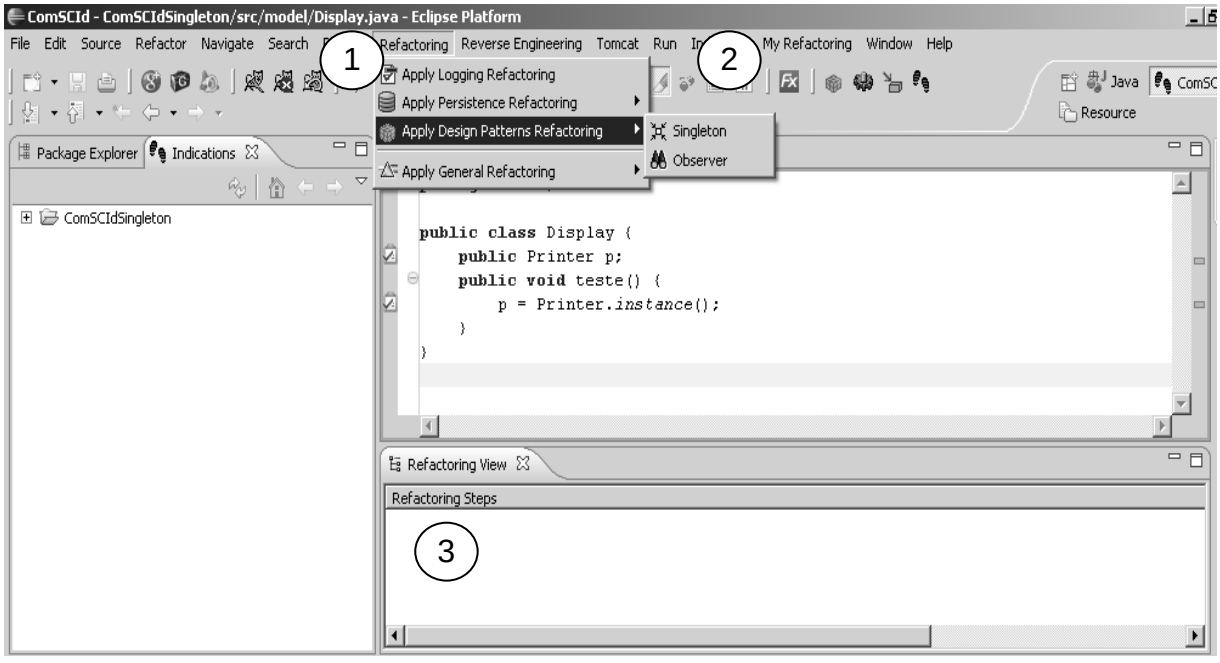

Figura 5.5 – Interface Padrão do MoBRe.

A interface do MoBRe é composta por dois *menus* localizados na barra de *menu* principal do Eclipse, *Refactoring* (1) e *My Refactoring* (2), e uma visão denominada *Refactoring View* (3).

O *menu Refactoring* disponibiliza os itens necessários para aplicação das refatorações genéricas e específicas sobre os modelos de classes OO anotados. Por exemplo, se o usuário deseja aplicar uma refatoração específica para o padrão de projeto *Observer*, ele deve clicar no *menu Refactoring*, depois em *Apply Design Patterns Refactoring* e por fim em *Observer*.

MoBRe fornece apoio a três tipos de refatorações genéricas, que foram apresentadas na Seção 5.3 como *R-1*, *R-2* e *R-3*, e seis específicas, que foram apresentadas na Seção 5.4 como R-*Connection*, R-*Transaction*, *R-Sync*, R-*Logging*, R-*Singleton* e R-*Observer.* As refatorações específicas foram elaboradas para modularização dos seguintes interesses transversais: *logging*, persistência (que é subdivido em gerenciamento de conexões, de transações e sincronização) e os padrões de projeto *Observer* e *Singleton* (Gamma *et al.*, 1995). Essas são as refatorações presentes no *menu* “Refactoring” e disponibilizadas originalmente com *plug-in*.

O *menu My Refactoring*, também localizado na barra de *menu* principal do Eclipse, contém refatorações implementadas por usuários.

Além dos *menus*, MoBRe possui uma visão, *Refactoring View*, responsável por exibir os passos aplicados sobre o modelo de classes OO anotado quando uma refatoração, genérica ou específica, é aplicada. Esses passos descrevem as modificações realizadas no modelo de classes OO para transformá-lo em um modelo OA. Essa visão traz as seguintes contribuições: i) quando vários interesses estão sendo refatorados de uma única vez, o usuário pode consultá-la para acompanhar todos os passos aplicados para cada interesse; e ii) enquanto estiver desenvolvendo suas próprias refatorações, o usuário pode utilizá-la para depurar possíveis erros existentes nos passos de sua refatoração.

5.4 Executando Refatorações no MoBRe

O conjunto de passos para utilizar o *plug-in* MoBRe é apresentado utilizando um exemplo de aplicação de refatorações genéricas ou específicas sobre um modelo de classes OO anotado.

1. **Executar o *plug-in* ComSCId**. Caso o ComSCId não seja executado, será emitida a mensagem "*Project is not generated. Please, run the ComSCId plug-in*.".

O código Java da Figura 5.6 é o utilizado como entrada para execução do *plug-in* ComSCId. Nesse exemplo, serão cadastradas regras no ComSCId para que indícios do interesse transversal de persistência sejam identificados. Assim, a classe `Connection` do pacote `java.sql` foi cadastrada como indício do interesse de persistência. Os trechos destacados em cinza na Figura 5.6 representam os indícios identificados pelo ComSCId.

```
1  public class Person {
2   private Connection conn;
3   public void save() throws Exception {
4    conn = DriverManager.getConnection("url", "user", "pass");
5         // Business Logic
6         conn.close();
7   }
8  }
```

Figura 5.6 – Código Afetado pelo Interesse Transversal de Persistência.

As refatorações já podem ser aplicadas, entretanto, as refatorações específicas implementadas no MoBRe foram desenvolvidas para serem executadas após a aplicação das refatorações genéricas.

2. **Gerar modelo de classes OO anotado**. Para saber qual refatoração genérica deve ser aplicada, o usuário precisa observar o modelo de classes OO anotado com indícios de interesses transversais, que é obtido com o *plug-in* DMAsp, clicando no *menu Reverse Engineering*.

O resultado é o modelo apresentado na Figura 5.7. Esse modelo apresenta um cenário de entrelaçamento/espalhamento do interesse transversal de persistência que é adequado para execução da refatoração *R-3* (O Interesse Transversal que não é Interesse Primário de nenhuma classe).

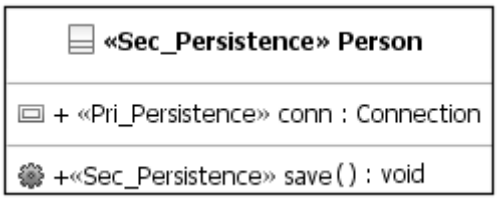

Figura 5.7 – Modelo OO Anotado Gerado pelo DMAsp.

Caso o usuário não saiba identificar os cenários relacionados a cada refatoração genérica, ele pode clicar no *menu Refactoring -> Apply Generic*

Refactoring e selecionar a opção *Analyze OO Class Model*. Uma mensagem descrevendo as refatoração passíveis de serem aplicadas para cada tipo de interesse é fornecida. Para o modelo apresentado na Figura 5.7, a mensagem exibida ao usuário é a apresentada na Figura 5.8..

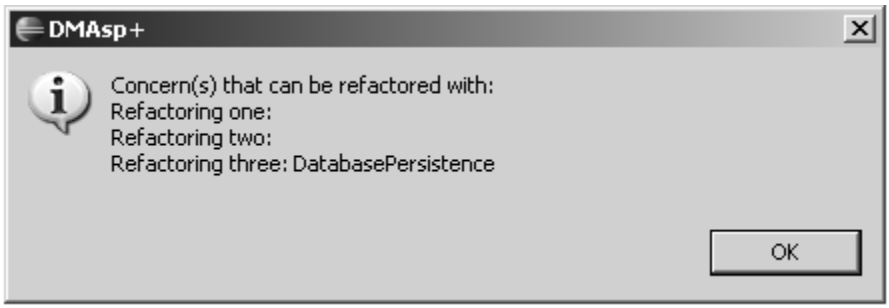

Figura 5.8 – Refatorações Passíveis de serem Aplicadas.

3. **Aplicar um tipo de refatoração genérica**. O usuário deve, em seguida, clicar sobre o *menu Refactoring -> Apply Generic Refactoring* e selecionar a opção correspondente à refatoração desejada. Para o caso do exemplo em questão, ao clicar na opção *Apply Refactoring 3*, a caixa de diálogo da Figura 5.9 será exibida.

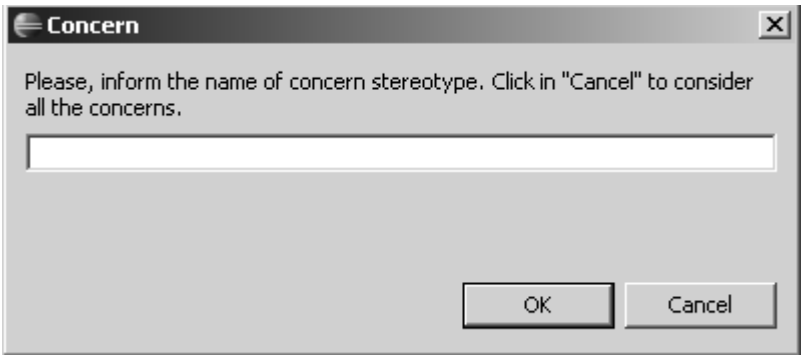

Figura 5.9 – Caixa de Diálogo para Informar o Interesse a ser Analisado.

O usuário deverá informar para qual tipo de interesse ele deseja aplicar a refatoração escolhida. Essa característica do MoBRe é importante quando há vários interesses no modelo OO anotado que podem ser refatorados com uma determinada refatoração, assim, o usuário pode escolher aplicá-la a cada interesse individualmente. Para que a refatoração seja aplicada a um determinado interesse o usuário deve informar o estereótipo que representa esse interesse. Para aplicar uma refatoração a todos interesses existentes no modelo, basta clicar em “Cancel” na caixa de diálogo da Figura 5.9 ou deixar o campo em branco e clicar em “Ok”.

Nesse exemplo, como existe apenas um interesse sobre o qual deseja-se aplicar a refatoração *R-3*, basta clicar em “Cancel”. Após a execução da refatoração, um novo arquivo XMI é criado para ser importado no Eclipse. O modelo resultante da aplicação dessa refatoração é apresentado na Figura 5.10. Além do arquivo XMI, a visão de refatorações (Figura 5.11) é atualizada com os passos realizados pela refatoração *R-3*.

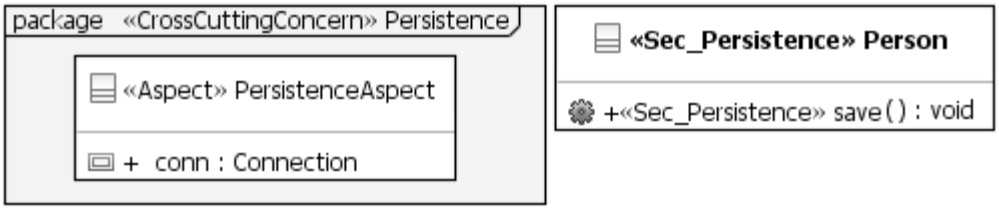

Figura 5.10 – Modelo OA Parcial Obtido Após a Aplicação da Refatoração 3.

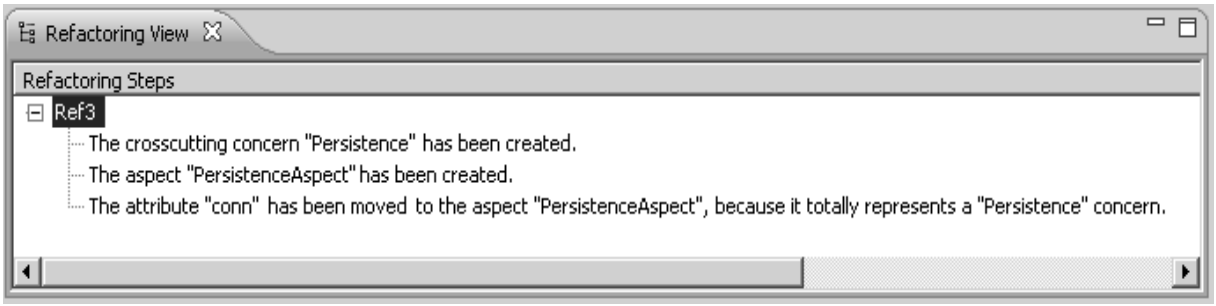

Figura 5.11 – Passos Executados pela Refatoração 3.

4. **Aplicar um tipo de refatoração específica**. A partir das informações da visão de refatorações, observa-se as modificações realizadas no modelo OO original para construção do modelo OA parcial.

A refatoração específica para o interesse de persistência pode ser aplicada sobre o modelo OA parcial. O interesse que está sendo analisado, *Persistence*, está relacionado ao gerenciamento de conexões com o Banco de Dados (Figura 5.6). O atributo `conn`, do tipo `Connection` (linha 2), é utilizado para conectar-se ao SGBD e no método `save()` (linhas 4 e 6) há os comandos utilizados para abertura e fechamento da conexão com o Banco de Dados. Dessa forma, pode-se aplicar a refatoração específica para gerenciamento de conexões. O usuário deve clicar no *menu Refactoring → Apply Persistence Refatoring* e escolher a opção *Connection Management*, sendo exibida a caixa de diálogo da Figura 5.12.

Figura 5.12 – Caixa de Diálogo para Informar o Estereótipo Utilizado para um Determinado Interesse.

O nome do estereótipo utilizado para o interesse de gerenciamento de conexão deve ser fornecido, pois o nome de cada interesse é escolhido pelo usuário no momento em que ele cadastra as regras para identificação de indícios de interesses transversais no ComSCid. Sendo assim, não há como saber qual estereótipo está relacionado ao interesse de gerenciamento de conexões. Entretanto, há a possibilidade do usuário informar, no momento do cadastramento das regras no ComSCId, a que tipo de interesse o estereótipo definido por ele se refere usando a interface apresentada na Figura 5.13.

Figura 5.13 – Interface para Associar o Nome de um Interesse ao seu Tipo.

O usuário pode, por exemplo, escolher o nome “ABC” para um estereótipo e informar, por meio da caixa de combinação “Indication Type”, que se refere ao interesse de gerenciamento de conexões. Assim, quando o *menu* de acesso à

refatoração para gerenciamento de conexões for acionado, a caixa de diálogo da Figura 5.12 não será exibida e o sistema identificará automaticamente o estereótipo relacionado a esse interesse no arquivo de regras mantido pelo ComSCId. O mesmo procedimento é válido para os demais tipos de refatorações específicas, como *Logging*, Gerenciamento de Transações, Sincronização, *Singleton* e *Observer*.

Considera-se, nesse caso, que o usuário irá informar o nome do estereótipo utilizado para o interesse de gerenciamento de conexões, “Persistence”. Após informar o nome do estereótipo, outras duas caixas de diálogo serão exibidas sequencialmente: são solicitados o nome dos métodos utilizado para abertura da conexão com o Banco de Dados (Figura 5.14 - esquerda) e para o fechamento da conexão (Figura 5.14 - direita).

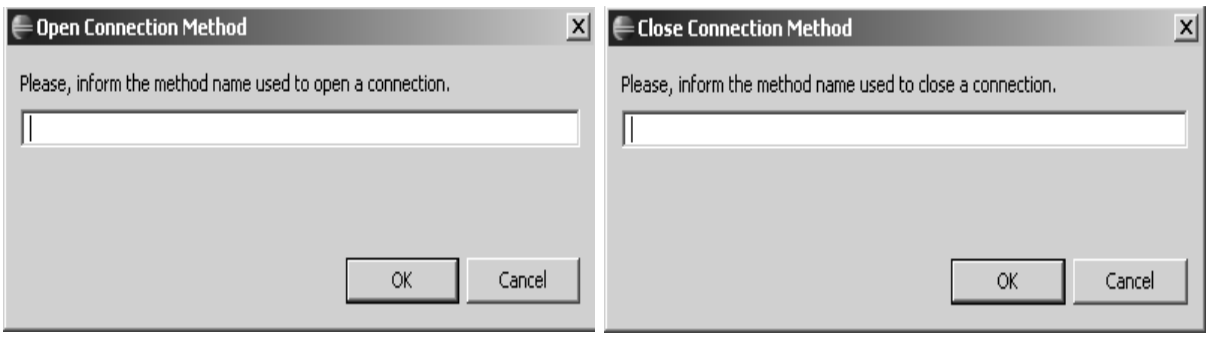

Figura 5.14 – Caixas de Diálogo para Informar os Nomes dos Métodos de Abertura e Fechamento da Conexão com o Banco de Dados.

Analogamente ao que ocorre com o nome do estereótipo, os nomes utilizados para os métodos para abertura e fechamento são atribuídos pelo usuário. Esses nomes são importantes, pois serão utilizados para encontrar os pontos onde o aspecto deve interceptar no código base. Essa busca será realizada no metamodelo DMAsp que armazena os métodos que são invocados dentro de outros métodos.

O nome do método de abertura de conexão é `getConnection()` da classe `DriverManager` e o nome do método de fechamento é `close()` da classe `Connection` (Figura 5.6). Essas informações também podem ser cadastradas no ComSCId durante o cadastramento das regras para identificação de indícios de interesses transversais. Assim, ao executar a refatoração, não serão exibidas as caixas de diálogo da Figura 5.14. A interface para cadastramento dessas informações é semelhante à da Figura 5.13. Para as demais refatorações específicas, o usuário também deve inserir informações adicionais como, por exemplo:

a) para o interesse de *logging* o(s) nome(s) do(s) método(s) utilizado(s) para gravar registros no arquivo de *log*;
b) para o padrão *Singleton* ele deve informar o nome do método utilizado para obter a instância da classe *Singleton*; e
c) para o padrão *Observer*, o usuário deve informar o nome do método utilizado para notificar os observadores.

Por último, o usuário pode selecionar uma das opções para refatoração do interesse de gerenciamento de conexões implementado pelo MoBRe: i) criar um aspecto concreto para cada classe afetada pelo interesse; ou ii) criar um aspecto concreto para cada pacote que contém pelo menos uma classe afetada pelo interesse (Figura 5.15). Essa funcionalidade visa a atender o passo 4 da refatoração R-*Connection* (Seção 5.4) no qual o Engenheiro de Software deve escolher uma estratégia para criação de aspectos de acordo com as características de implementação da aplicação. Nesse exemplo, como apenas uma classe encontra-se afetada pelo interesse de gerenciamento de conexões, a opção “Criar um aspecto concreto para cada classe afetada pelo interesse” foi selecionada.

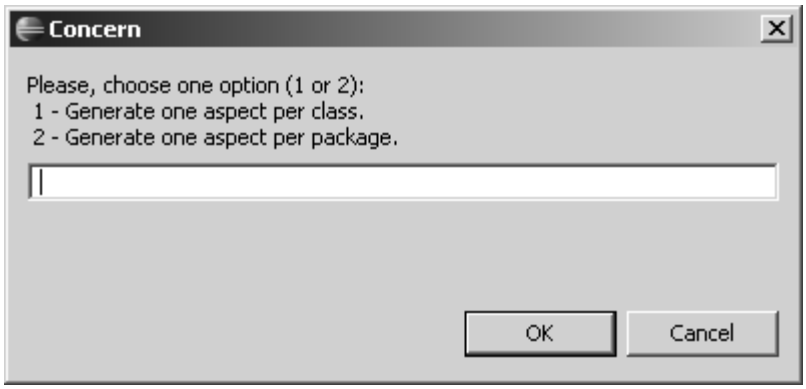

Figura 5.15 – Estratégias para Geração de Aspectos durante a Aplicação de uma Refatoração.

Após a execução da refatoração para gerenciamento de conexões, a visão de refatorações foi atualizada com os passos realizados por essa refatoração. O resultado pode ser visualizado na Figura 5.16.

Nota-se que os passos executados pela refatoração R-3 permaneceram na visão, garantindo assim, uma melhor visualização da rastreabilidade dos elementos criados e modificados pelas refatorações ao longo do tempo. O modelo OA final, obtido a partir da aplicação da refatoração para gerenciamento de conexões é apresentado na Figura 5.17.

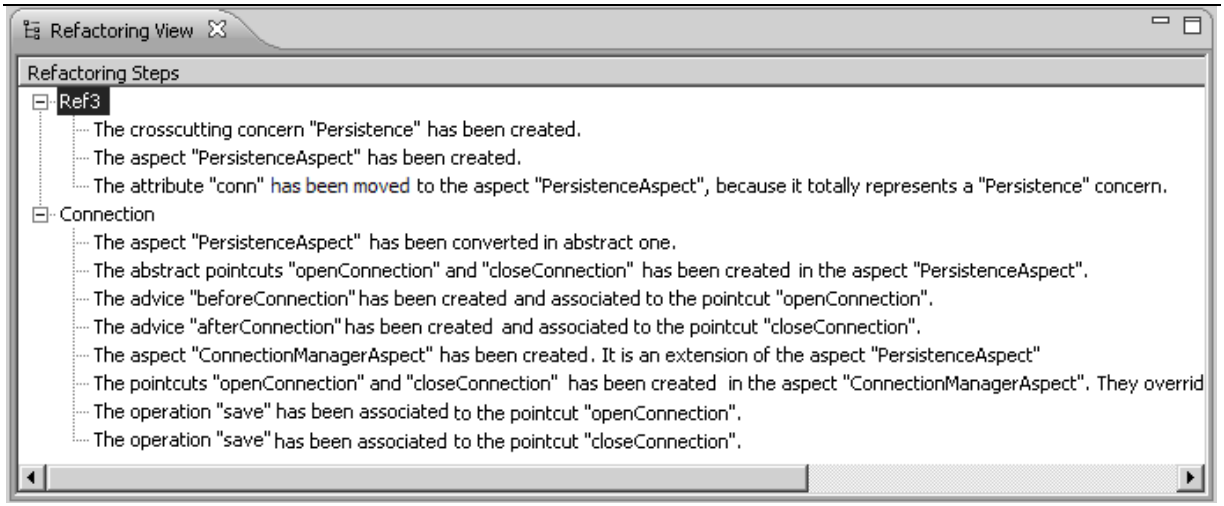

Figura 5.16 – Passos Executados pela Refatoração para Controle de Conexão.

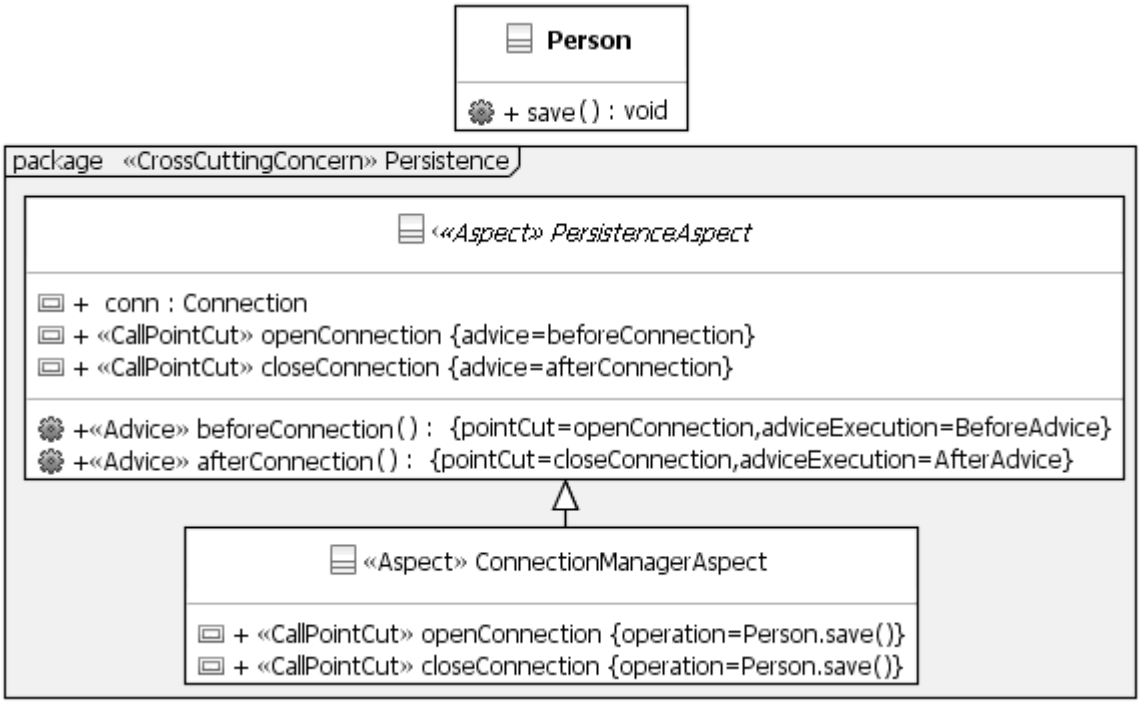

Figura 5.17 – Modelo OA Final Obtido Após a Aplicação da Refatoração para Controle de Conexão.

Salienta-se que a tarefa de obtenção de um modelo OA final deve ser realizada sob uma perspectiva iterativa e incremental. Dessa forma, usuários podem aplicar refatorações sobre cada interesse existente no software individualmente, observando, por meio do modelo de classes e da visão de refatorações, as modificações realizadas e progredindo até refatorar todos os interesses. Além disso, caso seja necessário, pode-se reiniciar o processo por um interesse diferente ou aplicando outros tipos de refatorações.

5.5 Estendendo o MoBRe

MoBRe disponibiliza originalmente dois tipos de refatorações: as genéricas e as específicas. Refatorações genéricas podem ser aplicadas sobre qualquer tipo de interesse e são responsáveis por criar uma solução de modularização de acordo com o cenário de entrelaçamento/espalhamento desse interesse no sistema. As refatorações específicas são criadas para determinados tipos de interesses e são responsáveis por refinar o modelo OA parcial, incorporando novos detalhes que não puderam ser obtidos por meios das refatorações genéricas. MoBRe implementa refatorações para tipos de interesses transversais como Persistência, *Logging* e os padrões de projeto *Singleton* e *Observer*.

Evidentemente, não seria possível implementar refatorações específicas para todos os tipos de interesses transversais por diversas razões, dentre elas: i) não há um catálogo fechado para os tipos de interesses transversais existentes e suas possíveis refatorações; ii) para alguns interesses há dificuldade em saber se eles são ou não interesses transversais; e iii) dependendo de como um determinado interesse encontra-se implementado, ele pode assumir características de interesses transversais, como alto entrelaçamento e espalhamento nos módulos funcionais da aplicação.

Durante o desenvolvimento do MoBRe uma das preocupações foi em construir um módulo que permita ao usuário acrescentar à sua funcionalidade novos tipos de refatorações genéricas e/ou específicas. Por exemplo, um Engenheiro de Software que trabalha com refatorações para interesse transversais de segurança não será completamente beneficiado com o MoBRe, pois ele não oferece apoio a esse tipo de refatoração. Entretanto, o Engenheiro de Software pode fazer o *download* do MoBRe e implementar sua própria refatoração para esse tipo de interesse utilizando como suporte a API do *plug-in*. Essa API dará suporte para implementação de consultas, alterações, remoções e/ou adições de elementos em um determinado modelo de classes OO utilizando a linguagem de programação Java.

Após implementar a refatoração desejada, basta gerar uma nova versão do *plug-in* MoBRe, no ambiente Eclipse, que as modificações realizadas pelo usuário serão incorporadas automaticamente e organizadas no *menu My Refactoring*

(destaque 2 na Figura 2.1). Além disso, com MoBRe o usuário pode implementar novos tipos de refatorações para um interesse já contemplado no *plug-in* (por exemplo, persistência), podendo assim, comparar duas ou mais soluções de modularização para um mesmo tipo de interesse.

Para explicar como MoBRe pode ser estendido, dois componentes importantes merecem destaque: i) a API MoBRe; e ii) o módulo de extensão de *plug-in* do MoBRe.

5.5.1 A API e o Módulo de Extensão do *Plug-in* MoBRe

A API do *plug-in* MoBRe consiste em um metamodelo para construção de modelos de classes OA. Ela possui um conjunto de classes relacionadas que permite armazenar informações sobre o código fonte de um sistema, como: classes e seus relacionamentos, atributos, métodos, aspectos e seus relacionamentos, conjuntos de junção, adendos e declarações inter-tipo que serão utilizadas posteriormente para geração de modelos de classes OA. O metamodelo do MoBRe é um extensão do metamodelo do DMAsp (Seção 4.3) e é apresentado na Figura 5.18 (as classes em cinza representam classes do metamodelo do DMAsp).

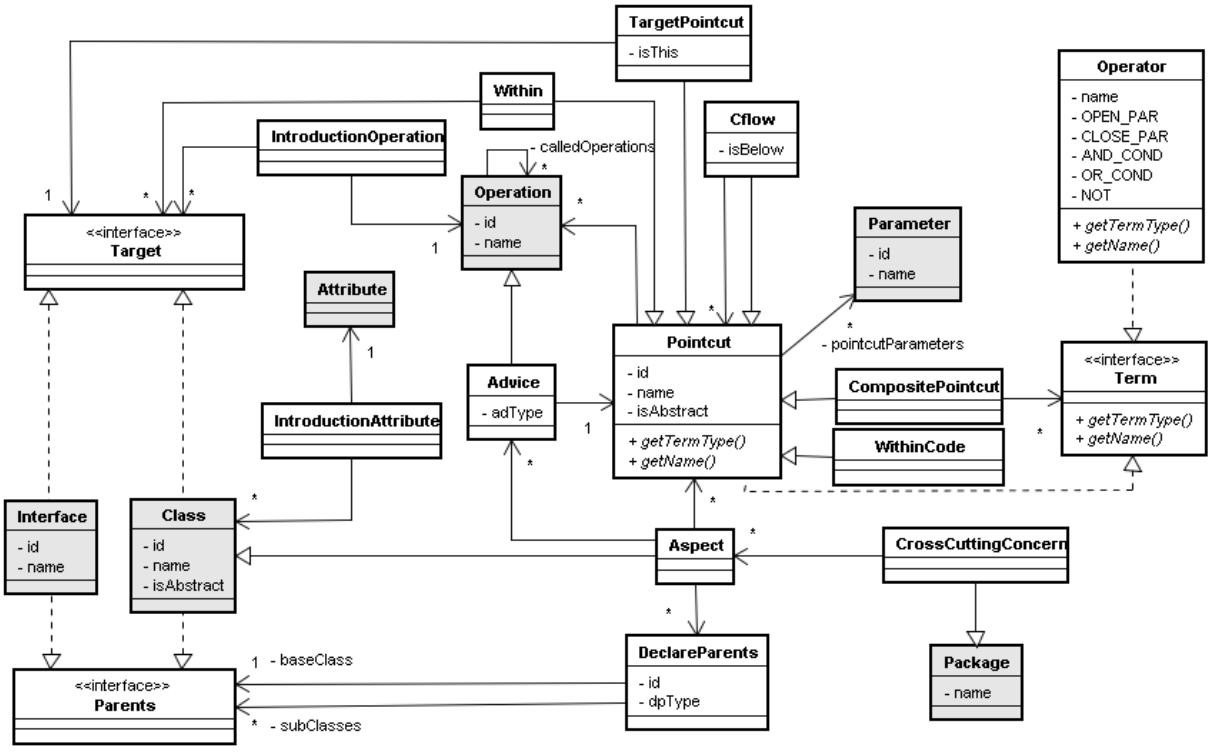

Figura 5.18 – Metamodelo do MoBRe.

A classe **`Advice`** representa um adendo, possui um atributo `adType` que armazena o tipo desse adendo: *before*, *after* ou *around*. Além disso, ela pode se relacionar com apenas um conjunto de junção (classe `Pointcut`). Apesar das linguagens de programação OA permitirem a associação de um adendo a mais de um conjunto de junção, essa decisão foi tomada para "forçar" a utilização de conjuntos de junção compostos, sendo essa uma boa prática para implementação de programas OA.

A classe **`Pointcut`** representa conjuntos de junção, possui o atributo `isAbstract` que determina se o conjunto de junção é ou não abstrato. Além disso, ela implementa a interface `Term` que será importante para construção de conjuntos de junção compostos. Um conjunto de junção pode estar relacionado a várias operações (classe `Operation`) e conter vários parâmetros (classe `Parameter`).

`Aspect` é a classe que representa um aspecto. Além de adendos (`Advice`) e conjuntos de junção (`Pointcut`), um aspecto pode conter declarações do tipo *declare parents* (`DeclareParents`). Essas construções especificam relacionamentos de herança e realização de interfaces entre classes da aplicação.

A classe **`DeclareParents`** representa uma construção do tipo *declare parents*. Além do identificador (`id`) e do tipo (`dpType`), que especifica se é uma construção do tipo *declare implements* ou do tipo *declare extends*, essa classe possui uma classe ou interface base (`baseClass`) e várias subclasses ou interfaces (`subClasses`). *Declare implements* declaram relacionamentos de realização de interfaces e *declare extends*, relacionamentos de herança.

`CrossCuttingConcern` é uma classe que representa um interesse transversal e pode conter vários aspectos (`Aspect`). Essa classe herda características e comportamento da classe `Package` e por isso pode conter várias classes também.

`CompositePointcut` é um tipo de conjunto de junção e possui uma lista de termos (`Term`). Esses termos podem ser outros conjuntos de junção ou operadores (`Operator`). O objetivo dessa classe é armazenar conjuntos de junção compostos, como por exemplo, `pc1() && pc2()`, onde `pc1` e `pc2` são conjuntos de junção e `&&` é o operador lógico “e”.

`Operator` é uma classe utilizada para criação de conjuntos de junção compostos. O atributo `name` armazena o símbolo correspondente ao operador. Os tipos de operadores disponíveis são especificados como constantes estáticas da

classe `Operator`. São eles: `OPEN_PAR` e `CLOSE_PAR`, que correspondem aos operadores de abertura e fechamento de parênteses, “(” e “)”, respectivamente; e `AND_COND`, `OR_COND` e `NOT`, que correspondem aos operadores lógicos "e", "ou" e "não".

Em Orientação a Aspectos os conjuntos de junção *Target* e *This* permitem capturar a instância de um objeto que executou ou onde está ocorrendo uma determinada operação. A diferença entre eles é o contexto no qual são executados. O *This* captura o objeto onde está o ponto de junção, já o *Target* captura o objeto que é o alvo da chamada do método. Para modelar esse comportamento, criou-se a classe **`TargetPointcut`** que representa um conjunto de junção que permite capturar a instância de um objeto. Essa classe está relacionada a um alvo (interface `Target`) que pode ser uma classe ou interface. O atributo `isThis` é utilizado para determinar o contexto no qual a instância da classe alvo deve ser capturada. Por exemplo, se `isThis` receber o valor falso, o conjunto de junção representa uma construção *Target*, caso contrário, representará uma construção *This*.

`Within` e **`WithinCode`** são classes que representam os conjuntos de junção responsáveis, respectivamentre, por capturar todos os pontos de junção que ocorrem no interior de uma classe ou método.

A classe **`Cflow`** representa o conjunto de junção que intercepta todos os pontos de junção que ocorrem no fluxo de execução de um determinado conjunto de junção, incluindo a chamada desse conjunto de junção. Há uma variação desse conjunto de junção, o *cflowbelow*, que desconsidera a chamada do conjunto de junção declarado no *cflowbelow*. Para representar essa diferença no metamodelo o atributo `isBelow` é utilizado. Quando `isBelow` é *true*, o conjunto de junção `Cflow` é do tipo *cflowbelow*.

`IntroductionAttribute` e **`IntroductionOperation`** são classes que representam conjuntos de junção para declarações inter-tipo e são utilizados para modificar a estrutura de classes e interfaces, inserindo novos atributos e métodos nelas. Dessa forma, `IntroductionAttribute` está relacionado com um atributo e com uma ou mais classes que receberão esses atributos. `IntroductionOperation`, por sua vez, está relacionado com uma operação e com um ou mais alvos (`Target`) que podem ser classes ou interfaces.

De modo análogo ao que acontece no DMAsp, toda instância criada a partir do metamodelo do MoBRe será armazenada em um arquivo XML, cujo conteúdo consiste das informações necessárias para geração do modelo de classes OA. Esse arquivo possui uma sintaxe própria que não está em conformidade com qualquer ferramenta CASE. Por isso, para poder visualizar o modelo de classe OA é necessário transformar o arquivo XML em um arquivo XMI compatível com alguma ferramenta CASE. No caso do MoBRe, a ferramenta utilizada é a mesma do DMAsp, o ambiente Eclipse.

O trecho de código da Figura 5.19 exemplifica a utilização das classes da API do MoBRe apresentadas anteriormente para criação da classe UML `Account` apresentada na Figura 5.20.

```
1  public static void main(String[] args) {
2   Project prj = new Project("Class Model", "c:\test");
3   Package pck1 = new Package("model");
4   Class cl = new Class("model.Account", "Account");
5   Attribute attr = new
6    Attribute("model.Account.connection","connection",
7    new Type("java.sql.Connection", "Connection"));
8   Concern concern = new Concern("DatabasePersistence",
9    "DatabasePersistence");
10  attr.addConcern(concern);
11  cl.addAttribute(attr);
12  pck1.addClass(cl);
13  prj.addPackage(pck1);
14 }
```

Figura 5.19 – Exemplo de Utilização da API do MoBRe.

Figura 5.20 – Classe `Account` afetada pelo Interesse de Persistência.

A linha 2 é responsável por criar um projeto Java, `prj`. Para esse projeto são armazenados seu nome e o caminho onde ele encontra-se localizado. Na linha 3 um pacote Java é criado (`pck1`), passando como parâmetro seu nome, que servirá também como identificador desse pacote. Na linha 4 uma classe, `cl`, é criada, passando como parâmetro seu identificador e seu nome. O identificador é utilizado para encontrar uma determinada classe no projeto e, por isso, deve ser único. O nome, por sua vez, é o que será apresentado no diagrama de classes. Isso serve também para outros elementos, como atributos e métodos. As linhas 5-7 são responsáveis pela criação de um atributo denominado `connection` do tipo

`Connection`. Nas linhas 8-10 é criado um interesse (`Concern`) denominado "DatabasePersistence" que é adicionado ao atributo `connection`. Ao adicionar um determinado interesse a um atributo ou método, ele é automaticamente adicionado à classe da qual esse atributo ou método pertence. Por fim, a linha 11 adiciona o atributo `connection` à classe `cl`, a linha 12 adiciona a classe `cl` ao pacote `pck1` e a linha 13 adiciona o pacote `pck1` ao projeto `prj`.

O módulo de extensão de *plug-in* MoBRe é um recurso que permite ao usuário implementar, sem muito esforço, suas próprias refatorações de interesses transversais. Entende-se por sem muito esforço o fato de que o MoBRe abstrai detalhes da plataforma de implementação de *plug-ins* de Eclipse, deixando com que o Engenheiro de Software se preocupe apenas com a resolução de seu problema, nesse caso, com a elaboração de refatorações para modularização de interesses transversais. Um exemplo de implementação de refatorações utilizando o módulo de extensão do MoBRe é apresentado nas próximas Subseções.

5.5.2 Cenário para Refatoração Específica

Nesta Subseção será apresentado um cenário específico de entrelaçamento e espalhamento de interesses como motivação para implementação de uma refatoração.

> *Em uma determinada aplicação, a classe `ClassA` foi criada para implementação do interesse "ConcernX" por meio dos métodos `method1()` e `method2()`. O modelo de classes correspondente a essa aplicação é apresentado na Figura 5.21.*

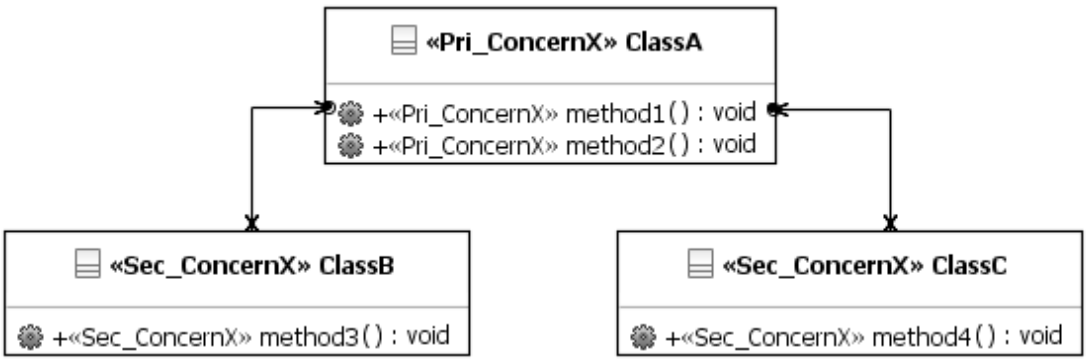

Figura 5.21 – Modelo de Classe que Especifica o Cenário para Implementação de uma Refatoração Específica.

A implementação dessa refatoração tem por objetivo modularizar o interesse “ConcernX” em um aspecto. Para isso, é necessário capturar todos os pontos onde esse interesse afeta, interceptá-los e adicionar o comportamento relacionado ao interesse via aspecto. Assim, as classes `ClassB` e `ClassC` ficariam livres de código relacionado ao interesse “ConcernX”. Para criação dos adendos ao implementar a refatoração é importante ter conhecimento de como o interesse afeta as classes envolvidas. Nesse caso, as classes `ClassB` e `ClassC` são afetadas por “CorcernX” pela chamada de `method1()`, que é realizada após o último comando desses métodos. Com base nessas informações e na configuração apresentada no modelo da Figura 5.21, foi criado um pseudocódigo (Figura 5.22) para orientar a implementação de uma refatoração específica para modularização do interesse “ConcernX”.

```
0. INÍCIO
1. Captura a classe “cl” cujo interesse “ConcernX” é primário.
2. Criar um interesse denominado “ConcernX”.
3. Criar um aspecto denominado “clAspect”.
4. Cria um conjunto de junção chamado “affectedMethods”.
5. Para cada classe “clAux” cujo interesse secundário seja “ConcernX”.
 5.1. Remove a referência à classe “cl” da classe “clAux”.
 5.2. Para cada método “mAux” que possui chamada ao método “method1”
 5.2.1 Adiciona “mAux” ao conjunto de operações do conjunto de
  junção “affectedMethods”.
 5.2.2 Remove o estereótipo relacionado ao interesse “ConcernX” do
  método “mAux”
 5.3. Atualiza o conjunto de estereótipos da classe “clAux”
6. Se o conjunto de operações do conjunto de junção “affectedMethods”
 é maior que zero, faça:
 6.1 Cria um adendo “adAffectedMethods” do tipo “after” e relacione-o
 ao conjunto de junção “affectedMethods”.
 6.2 Adicione o conjunto de junção “affectedMethods” e o adendo
 “adAffectedMethods” ao aspecto “clAspect”
7. Adicione o aspecto criado ao projeto ao interesse “ConcernX”.
8. Adicione o interesse “ConcernX” ao projeto.
9. FIM
```

Figura 5.22 – Pseudocódigo para Implementação da Refatoração Específica.

5.5.3 Implementando uma Refatoração no *Plug-in* MoBRe

Nesta Subseção será apresentado o procedimento necessário para implementar o pseudocódigo da Figura 5.22 no *plug-in* MoBRe. Inicialmente, o usuário deve criar uma classe que estende a classe `RefactoringHandler` e sobrescrever seu método `widgetSelected()`. Esse método será invocado toda vez que o usuário clicar sobre o item do *menu* correspondente à refatoração a ser criada.

No corpo do método `widgetSelected()` o usuário deve implementar os passos relacionados à sua refatoração ou criar uma nova classe que faça isso e acessá-la a partir desse método. Para acessar os elementos do modelo de classes OO anotado o usuário deve utilizar o trecho de código da API do MoBRe apresentado na Figura 5.23.

A linha 2 desse código é responsável por obter uma instância da classe `DmaspController` que possui os métodos necessários para preparação do ambiente para aplicação das refatorações (método `initializeProject()`) e para geração arquivo XMI que será importado posteriormente no Eclipse (método `ObjectToXMI()`).

```
1  try {
2   DmaspController dmasp = DmaspControllerFactory.getInstance();
3   dmasp.initializeProject();
4   Project prj = dmasp.getProject();
5   // Procedimento da refatoração
6   dmasp.ObjectToXMI(window);
7  } catch (ProjectNotInitializedException prjEx) {
8   MessageDialog.openInformation(window.getShell(), "MoBRe",
9   "Project is not generated. Please, run the ComSCId
10   Plug-in.");
11 }
```

Figura 5.23 – Código da API MoBRe para Acessar os Elementos Modelo de Classes OO Anotado.

Nas linhas 4 e 5 o usuário poderá implementar os passos correspondentes à sua refatoração. Para isso, o usuário deverá utilizar uma instância da classe `Project` que contém todas as informações recuperadas a partir do código fonte OO pelo ComSCId. Essa instância pode ser obtida por meio do método `getProject()` (linha 4) da classe `DmaspController`.

Salienta-se que o método `initializeProject()` pode lançar uma exceção do tipo `ProjectNotInitializedException` e, por isso, deve ser invocado dentro de um bloco `try-catch` ou no corpo de um método capaz de relançar uma exceção desse tipo. Uma exceção `ProjectNotInitializedException` é lançada toda vez que o usuário tentar inicializar uma refatoração antes do *plug-in* ComSCId ter sido executado.

Com base nas informações apresentadas anteriormente, criou-se uma classe chamada `ConcernXRefactoring` que herda da classe `RefactoringHandler` e sobrescreve o método `widgetSelected()`. No corpo desse método será adicionado o trecho de código da Figura 5.23, sendo que na linha 5 desse código é

implementada a refatoração específica para o interesse “ConcernX” com base no pseudocódigo apresentado na Figura 5.22. O código fonte Java referente a essa refatoração é apresentado no Apêndice A.

Uma vez terminada a implementação da refatoração, faz-se necessário informar ao MoBRe que uma nova refatoração foi criada para que ele possa criar um *menu* específico para acessá-la. Para isso, deve-se criar (ou modificar) o arquivo “myrefs.xml” no diretório raiz do *plug-in* MoBRe. Esse arquivo é utilizado para fornecer informações para criação do *menu* correspondente à refatoração elaborada. O arquivo “myrefs.xml” segue o formato apresentado na Figura 5.24.

```
1  <userrefactorings>
2   <myrefactorings>
3    <refactoring>
4    <name>ConcernX Refactoring</name>
5    <handler>dmasp.actions.ConcernXRefactoring</handler>
6    </refactoring>
7   </myrefactorings>
8  </userrefactorings>
```

Figura 5.24 – Arquivo de Criação do *Menu* das Refatorações de Usuário.

Na *tag* raiz desse arquivo (`<userrefactorings>`) são especificadas as refatorações criadas pelo usuário. Todo arquivo “myrefs.xml” deve conter apenas uma *tag* `<userrefactorings>` que, por sua vez, conterá uma *tag* denominada `<myrefactorings>`. Essa *tag* armazena a lista de refatorações criadas pelo usuário. A *tag* `<myrefactorings>` pode conter zero ou mais *tags* do tipo `<refactoring>`. *Tags* `<refactoring>` armazenam as informações necessárias para definição de uma nova refatoração no *plug-in* MoBRe. Essas informações são: i) o nome da refatoração, representado pela *tag* `<name>`; e ii) o nome da classe responsável pela execução da refatoração, representado pela *tag* `<handler>`. O nome da refatoração aparecerá como um item no *menu My Refactoring* do *plug-in* MoBRe. A *tag* `<handler>` armazena o nome da classe que trata do evento de clique sobre o *menu* correspondente à refatoração. Quando esse clique ocorre o método `widgetSelected()`, da classe especificada na *tag* `<handler>`, é invocado. Salienta-se que o usuário deve informar o nome completo da classe (nome do pacote + nome da classe) responsável pela refatoração para que o MoBRe possa localizar essa classe e criar o *menu* corretamente.

Uma vez criado o arquivo “myrefs.xml”, como apresentado na Figura 5.24, basta gerar uma nova versão do *plug-in* utilizando a ferramenta Eclipse. Detalhes da

implementação e geração de *plug-ins* na plataforma Eclipse não fazem parte do escopo deste trabalho e podem ser encontradas em Eclipse (2011). Na Figura 5.25 é apresentada uma nova versão do *plug-in* MoBRe a partir da configuração realizada.

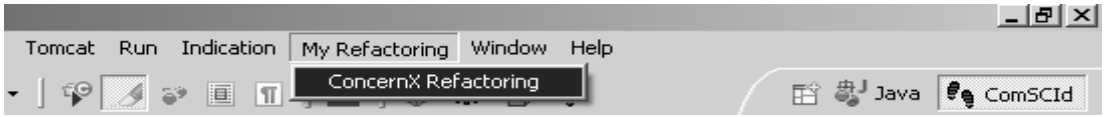

Figura 5.25 – Nova Versão do *Plug-in* MoBRe.

Após a implementação da refatoração, pode-se aplicá-la sobre o modelo apresentado na Figura 5.21. Salienta-se que, nesse caso, a maneira como a refatoração "ConcernX Refactoring" foi implementada permite que ela seja aplicada diretamente sobre o modelo OO anotado, sem a necessidade da aplicação refatorações genéricas. O resultado da aplicação da refatoração "ConcernX Refactoring" é apresentado na Figura 5.26. `ClassA` e `ClassB` não possuem mais estereótipos correspondentes ao interesse transversal "ConcernX", ou seja, tais classes não são mais afetadas por esse interesse. Para isso, criou-se um aspecto denominado `ClassAAspect` responsável por interceptar os pontos necessários no código da aplicação base e inserir o comportamento adicional, que é a execução do método `method1()` após a execução dos métodos `method2()` e `method3()`.

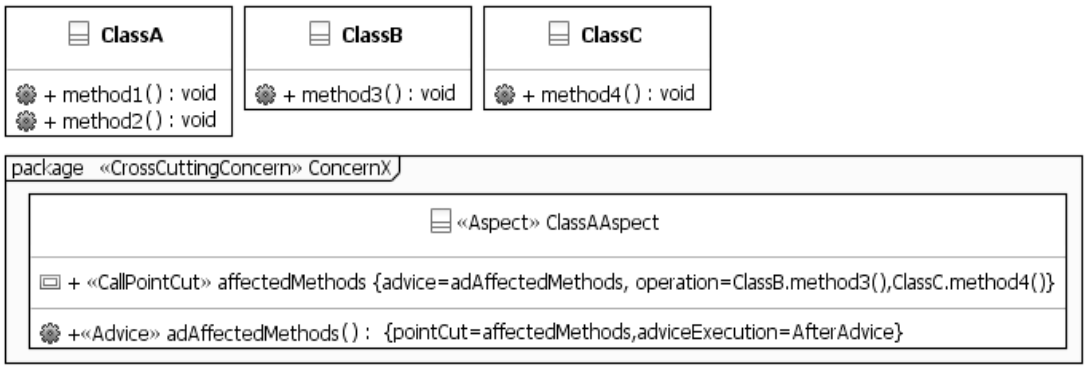

Figura 5.26 – Modelo OA Obtido a partir da Aplicação da Refatoração "ConcernX Refactoring".

Outra característica do MoBRe que merece destaque é que todas as refatorações são executadas em linhas de execução (*threads*) independentes da linha de execução principal do *plug-in.* Isso garante um controle mais apurado de *feedback* para o usuário, com exibição de barra de progresso da execução das refatorações e evita que um *bug* existente na implementação de uma refatoração comprometa a execução do ambiente Eclipse e de outros *plug-ins*. Tudo isso é feito

automaticamente pelo MoBRe, sem necessidade de intervenção do usuário no código fonte do *plug-in*.

5.6 Abordagem Iterativa para Refatoração de Interesses Transversais Apoiada por Ferramentas

Como comentado anteriormente, MoBRe trabalha em conjunto com as ferramentas ComSCId e DMAsp, que permitem ao Engenheiro de Software identificar, visualizar e refatorar interesses transversais existentes em sistema OO Java com base em modelos de classes da UML. A identificação dos interesses transversais é realizada em nível de código fonte pela ferramenta ComSCId, a visualização desses interesses é dada por meio de uma árvore de indícios (ComSCId) ou por diagramas de classes da UML anotados (DMAsp). A refatoração dos interesses é realizada em nível de modelos com o auxílio da ferramenta MoBRe. Com base nesse arcabouço de ferramentas, este trabalho teve como preocupação a elaboração de uma abordagem semiautomática para refatoração de interesses transversais apoiada por ferramentas que possam auxiliar o Engenheiro de Software na tarefa de modularização dos interesses transversais de um software OO. Essa abordagem é ilustrada esquematicamente na Figura 5.27 e é composta de quatro atividades representadas por retângulos com a extremidade direita triangular e artefatos consumidos e produzidos por essas atividades.

A atividade “Atualizar o Conjunto de Regras para Identificação de Interesses Transversais” refina o conjunto de regras do ComSCId para melhorar sua precisão. Para isso o gerenciador de regras do ComSCId deve ser utilizado, tendo como resultado o arquivo que armazena as regras de identificação de interesses é atualizado (*Regras para Identificação de Interesses Transversais*).

A atividade “Identificar Indícios de Interesses Transversais” encontra automaticamente indícios de interesses transversais no código fonte OO. Recebe como entrada um código fonte OO implementado em Java, usa o *plug-in* ComSCId com seu conjunto de regras para identificação de interesses transversais e produz como saída um código fonte anotado com indícios de interesses transversais.

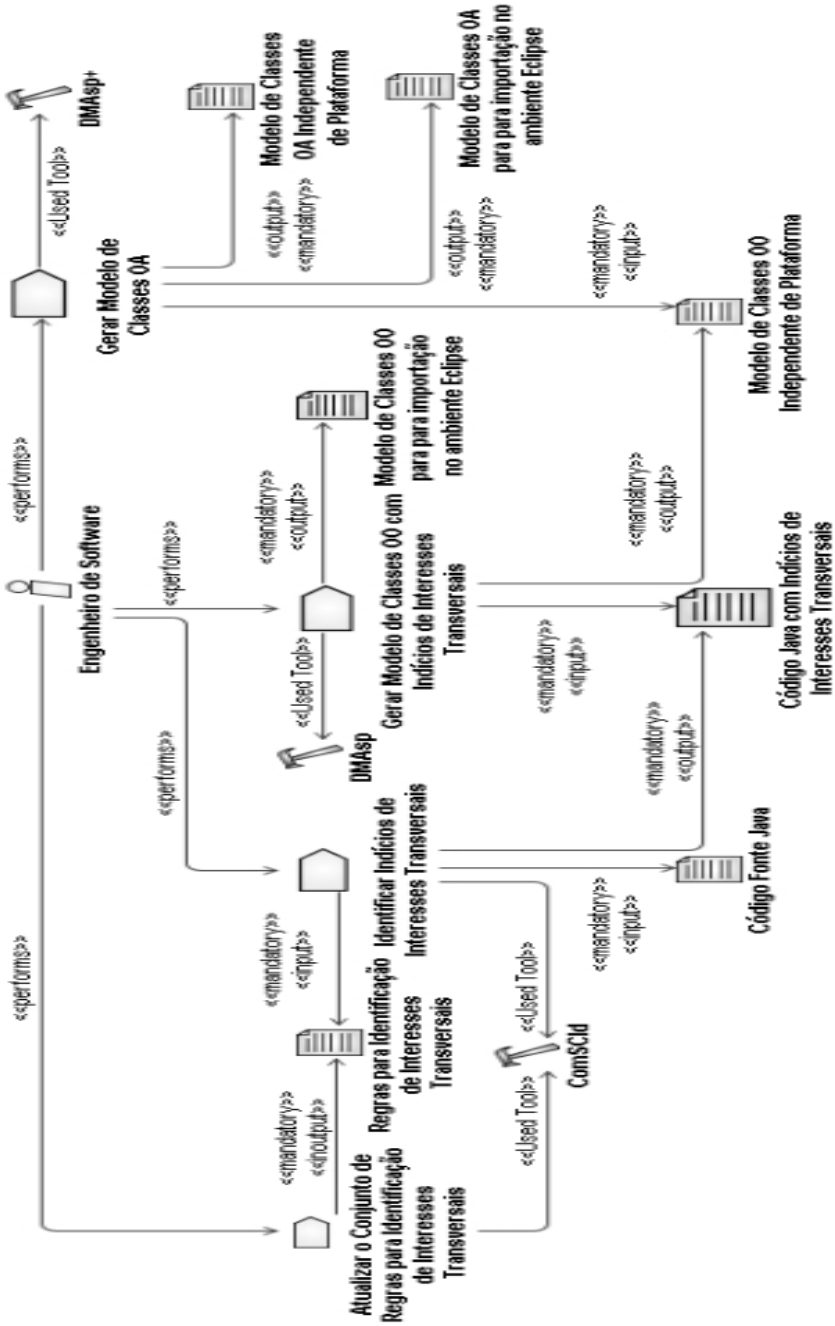

Figura 5.27 – Abordagem Iterativa para Refatoração de Interesses Transversais Apoiada por Ferramentas.

A atividade “Gerar Modelo de Classes OO com Indícios de Interesses Transversais” gera o modelo de classes orientado a objetos anotado recuperado na atividade anterior. Recebe o arquivo código OO anotado com indícios de interesse transversais, utiliza a ferramenta DMAsp e produz dois arquivos: i) um arquivo contendo informações do modelo de classes OO anotado que serão utilizadas para aplicação das refatorações OA (*Modelo de Classes OO Independente de Plataforma*); e ii) um arquivo que contém o diagrama de classes OO obtido a partir do código fonte Java e que pode ser visualizado no ambiente Eclipse. O arquivo *Modelo de Classes OO Independente de Plataforma* contém todas as informações

necessárias para geração do modelo de classes OO anotado, porém sua estrutura não condiz com o padrão exigido por qualquer ferramenta CASE.

Por último, a atividade “Gerar Modelos de Classes OA” recebe como entrada o arquivo com as informações do modelo de classes OO anotado, utiliza a ferramenta MoBRe e gera como saída dois arquivos: i) um arquivo contendo informações do modelo de classes OA (*Modelo de Classes OA Independente de Plataforma*); e ii) um arquivo que contém o diagrama de classes OA obtido a partir da aplicação das refatorações implementadas no MoBRe sobre o modelo de classes OO anotado. Esse arquivo é criado com base na abordagem para modelagem de software OA proposta por Evermann (2007) e também pode ser visualizado no ambiente Eclipse.

Salienta-se que embora a Figura 5.27 exiba a representação de um modelo sequencial, após a execução das atividades “Identificar Indícios de Interesses Transversais”, “Gerar Modelo de Classes OO” e “Gerar Modelo de Classes OA” o Engenheiro de Software, com base nas informações apresentadas, poderá retornar à atividade “Atualizar o Conjunto de Regras para Identificação de Interesses Transversais”. Na Figura 5.28 é apresentado o diagrama de atividades da abordagem evidenciando suas atividades e a ordem de execução de cada uma.

De acordo com o diagrama da Figura 5.28 o Engenheiro de Software pode:

- criar ou refinar o conjunto de regras para identificação de interesses transversais. Essa decisão leva em consideração que a atividade “Identificar Indícios de Interesses Transversais” utiliza a ferramenta ComSCId que possui um módulo para gerenciamento das regras de identificação de interesses transversais. Caso o Engenheiro de Software escolha por “Atualizar o Conjunto de Regras para Identificação de Interesses Transversais” a próxima atividade a ser executada deve ser obrigatoriamente a “Identificar Indícios de Interesses Transversais”, pois uma vez que o conjunto de regras foi alterado, o conjunto de indícios detectados pode ter se tornado desatualizado.
- realizar a atividade “Identificar Indícios de Interesses Transversais”, caso o conjunto de regras seja adequado para as necessidades do Engenheiro de Software.

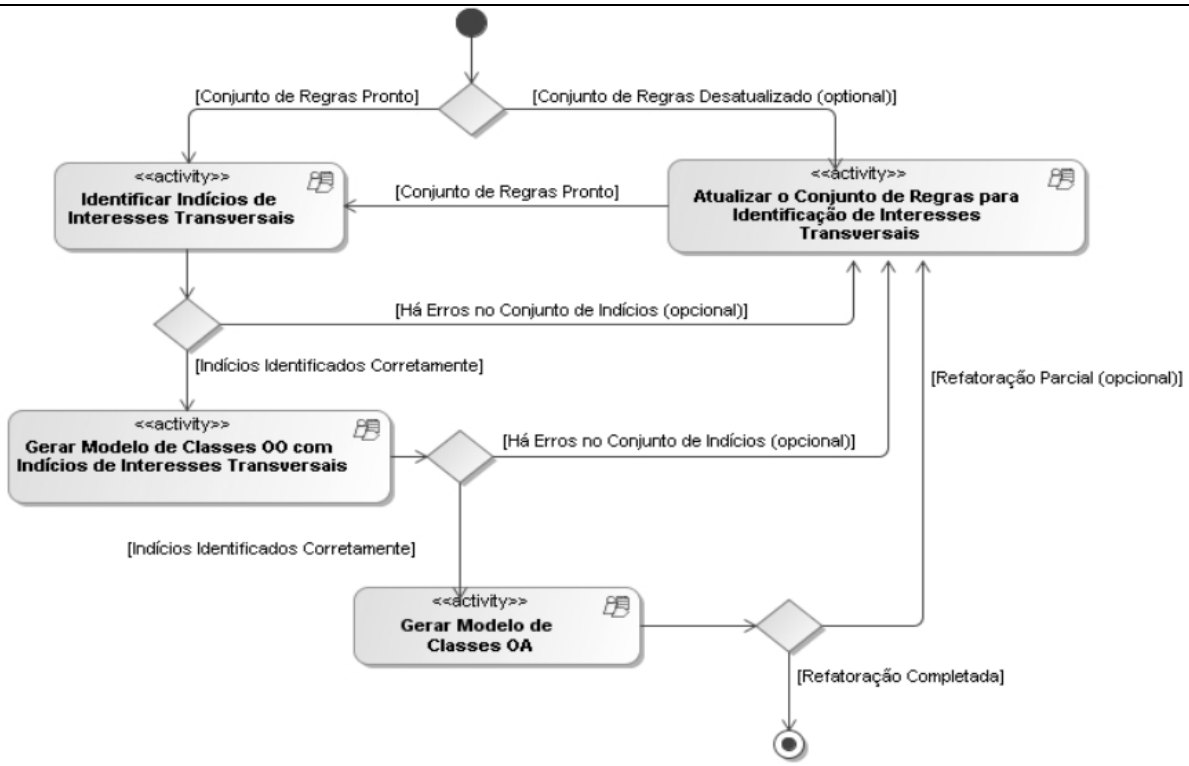

Figura 5.28 –Diagrama de Atividades da Abordagem para Refatoração de Interesses Transversais.

Uma vez identificados os indícios de interesses com base no conjunto de regras cadastrado, o Engenheiro de Software possui novamente duas escolhas para próxima atividade a ser executada:

- com base nos resultados obtidos a partir da execução da atividade “Identificar Indícios de Interesses Transversais” pode-se detectar alguns problemas, como falsos positivos e falsos negativos. Falsos positivos são trechos código identificados pelo ComSCId mas que não correspondem a indícios de interesses transversais e falsos negativos consistem em indícios que não foram identificados. Para ambas as situações, o conjunto de regras para identificação de interesses transversais deve ser adaptado e nesse caso, a próxima atividade a ser executada é “Atualizar o Conjunto de Regras para Identificação de Interesses Transversais”.
- caso os indícios de interesses transversais tenham sido identificados corretamente pelo ComSCId o Engenheiro de Software pode ir para a próxima atividade “Gerar Modelo de Classes OO com Indícios de Interesses Transversais”.

Após a geração do modelo de classes OO anotado com indícios de interesses transversais as possibilidades são:

- executar novamente a atividade “Atualizar o Conjunto de Regras para Identificação de Interesses Transversais”. É possível que o Engenheiro de Software detecte problemas na identificação dos indícios de interesses transversais nessa atividade que não foram identificados na atividade anterior. Isso pode ocorrer devido ao novo tipo de visualização de indícios de interesses transversais propiciado pela utilização de modelos de classes anotados. Ao olhar para o modelo de classes anotado o usuário pode notar situações de entrelaçamento/espalhamento de código que são mais difíceis de serem detectadas em nível de código. Por exemplo, dada uma classe A que possui como interesse primário *logging* e que é utilizada por outra classe B. Caso o Engenheiro de Software não cadastre a classe A no conjunto de regras para identificação de indícios de interesses transversais, os trechos de código relacionados à classe A e portanto ao interesse de *logging* podem não ser detectadas pelo ComSCId. Esse cenário é melhor visualizado com recursos de maior nível de abstração como é o caso do modelo de classes.
- executar a atividade “Gerar Modelo de Classes OA”. Se nenhum erro (falso positivo/negativo) foi detectado o Engenheiro de Software pode aplicar as refatorações OA para geração de modelos de OA a partir dos modelos de classes OO anotados.

Por último, após a obtenção do modelo de classes OA pode-se:

- encerrar a execução da abordagem.
- voltar para a atividade “Atualizar o Conjunto de Regras para Identificação de Interesses Transversais”. Essa alternativa é importante, pois como foi apresentado anteriormente nesse capítulo, os interesses transversais de uma aplicação podem ser refatorados incrementalmente na ferramenta MoBRe. Por exemplo, o Engenheiro de Software pode começar com um conjunto de regras para identificação de indícios do interesse transversal de *logging*, identificar os indícios desse interesse na aplicação, construir o modelo de classes anotado, aplicar a refatoração específica para *logging* e gerar o modelo de classes OA. Posteriormente, ele pode cadastrar novas regras para identificação de indícios do interesse de persistência e repetir todas as atividades da abordagem. Assim, de forma iterativa e

incremental, o Engenheiro de Software pode construir o modelo de classes OA a partir de uma aplicação OO afetada por interesses transversais.

Além de prover apoio para criação de novos tipos de refatorações, outra vantagem do MoBRe é que ele pode ser utilizado juntamente com alguma ferramenta para mineração de aspectos de forma a automatizar a tarefa de detecção de indícios de interesses transversais no código fonte. Neste trabalho isso foi feito com a ferramenta ComSCId, entretanto, outras ferramentas para mineração de aspectos podem ser propostas, utilizando o MoBRe para recuperar modelos OO anotados e aplicar refatorações sobre esses modelos.

5.7 Considerações Finais

Neste capítulo foi apresentado o *plug-in* MoBRe, uma ferramenta implementada para semi-automatizar a tarefa de refatoração de interesses transversais com base nas refatorações propostas neste trabalho. Além da aplicação semi-automática dos passos definidos pelas refatorações, MoBRe possui um conjunto de características quem trazem as seguintes contribuições: i) permite ao Engenheiro de Software rastrear as modificações provocadas pelas refatorações nos modelos de classes por meio da visão “Refactoring View”; ii) permite refatorar os interesses transversais existentes na aplicação de forma individual, compondo o modelo OA final de forma incremental; e iii) é flexível de modo que é possível ampliar o conjunto de refatorações implementadas pelo MoBRe por meio de seu módulo de extensão. O maior diferencial do MoBRe com relação às ferramentas comentadas no Capítulo 3 é o módulo de extensão que permite ao Engenheiro de Software elaborar suas próprias refatorações, ampliando a funcionalidade do *plug-in*.

Apresentou-se também os *plug-ins* ComSCId e DMAsp, para identificação de interesses transversais e geração de modelos de classes OO anotados, bem como uma abordagem para refatoração de interesses transversais baseada em modelos com o auxílio desses três *plug-ins*.

Capítulo 6

ESTUDO DE CASO

6.1 Considerações Iniciais

Um estudo de caso contendo três sistemas, desenvolvidos por outros pesquisadores, denominados Health Watcher (Greenwood et al., 2007), JSpider (JSpider, 2011) e JAccounting (JAccounting, 2011) foi conduzido para exemplificar a aplicação das reafatorações propostas.

Esses sistemas foram escolhidos, pois: i) possuem uma versão OO Java e uma versão OA AspectJ; ii) são conhecidos e já foram estudados pela comunidade científica; iii) foram modularizados por engenheiros de software experientes com a utilização de boas práticas de implementação OA e diferentes das aqui propostas. Os interesses transversais existentes nessas aplicações e modularizados neste estudo de caso são os de: persistência, *logging* e a implementação dos padrões *Observer* e *Singleton*.

Na Seção 6.2 são apresentadas refatorações para os interesses relacionados à persistência, ao *logging* e aos padrões de projeto *Singleton* e *Observer* da aplicação Health Watcher. Posteriormente; nas Seções 6.3 e 6.4, as refatorações para os interesses de persistência e *logging* são executadas no código fonte das aplicações JSpider e JAccounting para avaliar a aplicabilidade das refatorações propostas nesta dissertação. Na Seção 6.5 são apresentadas as considerações finais.

6.2 Estudo de Caso: *Health Watcher*

Trata-se um sistema de informação para registro de reclamações na área da saúde, foi modularizado em AspectJ por *Greenwood et al. (2007)* e será objeto de estudo para a modularização dos interesses relacionados à persistência e aos padrão de projeto *Singleton* e *Observer* (Gamma *et al.*, 1995).

6.2.1 Modularização dos Interesses Relacionados à Persistência e ao Padrão *Singleton*

Na Figura 6.1 é apresentada uma parte do modelo de classes do sistema Health Watcher (Greenwood et al., 2007) cujas classes são responsáveis pela manutenção do cadastro de reclamações de pacientes. A classe `HealthWatcherFacade` disponibiliza os métodos necessários para execução da lógica de negócios da aplicação como cadastramento de reclamações, doenças, sintomas, entre outros.

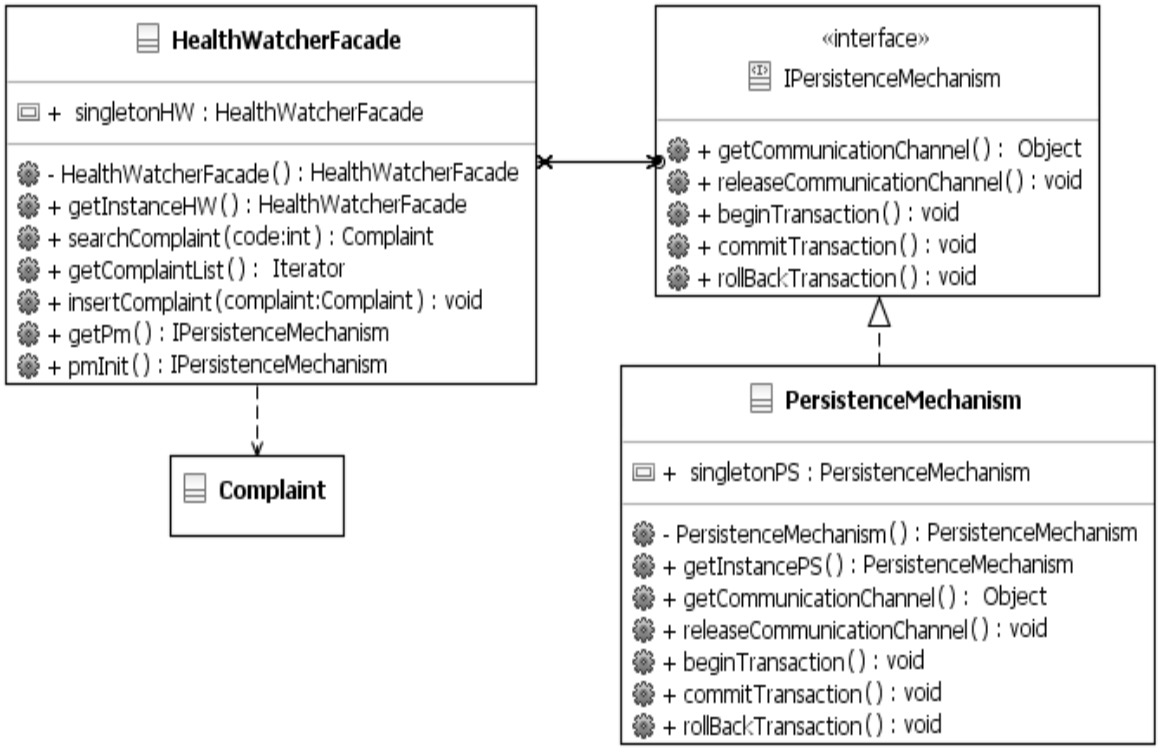

Figura 6.1 – Parte do Modelo de Classes OO do Health Watcher cujas Classes são Responsáveis pela Manutenção do Cadastro de Reclamações na Aplicação.

Nesse exemplo são apresentados os métodos responsáveis por realizar consultas por reclamações (`searchComplaint()`), recuperar a lista de reclamações cadastradas na aplicação (`getComplaintList()`) e cadastrar uma nova reclamação (`insertComplaint()`). Além desses, a classe `HealthWatcherFacade` possui os métodos: i) `getPm()` e `pmInit()`, responsáveis pela instanciação e inicialização do mecanismo de persistência implementado pela classe `PersistenceMechanism`; e ii) `getInstanceHW()` que junto com o atributo `singletonHW` são responsáveis por manter apenas uma instância da classe `HealthWatcherFacade` na aplicação (Padrão de Projeto *Singleton* (Gamma *et al.*, 1995)).

A classe `HealthWatcherFacade`, Figura 6.1, está relacionada à interface `IPersistenceMechanism`, que possui os métodos necessários para abertura e fechamento de conexões com o Banco de Dados (`getCommunicationChannel()` e `releaseCommunicationChannel()`) e para iniciação (`beginTransaction()`), confirmação (`commitTransaction()`) e recuperação de transações (`rollBackTransaction()`). Além disso, assim como acontece na classe `HealthWatcherFacade`, o padrão *Singleton* foi implementado para garantir a existência de apenas uma instância da classe `PersistenceMechanism` na aplicação. Os elementos dedicados a implementação desse padrão são o atributo `singletonPS` e o método `getInstancePS()`.

A maneira como o Health Watcher encontra-se implementado pode ser prejudicial à sua manutenibilidade, uma vez que há entrelaçamento entre os interesses correspondentes: i) à lógica de negócio da aplicação e à persistência (relacionamento entre a classe `HealthWatcherFacade` e a interface `IPersistenceMechanism`); ii) à lógica de negócio da aplicação e ao padrão *Singleton* (atributo `singletonHW` e método `getInstanceHW()` da classe `HealthWatcherFacade`) e iii) à persistência e ao padrão *Singleton* (atributo `singletonPS` e método `getInstancePS()` da classe `PersistenceMechanism`).

Para resolver o problema de entrelaçamento de interesses pode-se aplicar as refatorações desenvolvidas neste trabalho para modularização dos interesses de persistência e do padrão *Singleton*. O primeiro passo para aplicação das refatorações genéricas e específicas é utilizar o *plug-in* ComSCId para identificar os indícios dos interesses existentes no Health Watcher. Para isso é necessário cadastrar as regras para identificação desses indícios.

Como visto anteriormente, a interface `IPersistenceMechanism` e a classe `PersistenceMechanism` são os principais componentes responsáveis pela persistência dos dados da aplicação e possuem métodos dedicados à implementação dos interesses de gerenciamento de conexões e de transações. Dessa forma, são cadastrados no ComSCId os estereótipos `Conn` e `Trans` correspondentes a esses interesses (Figura 6.2).

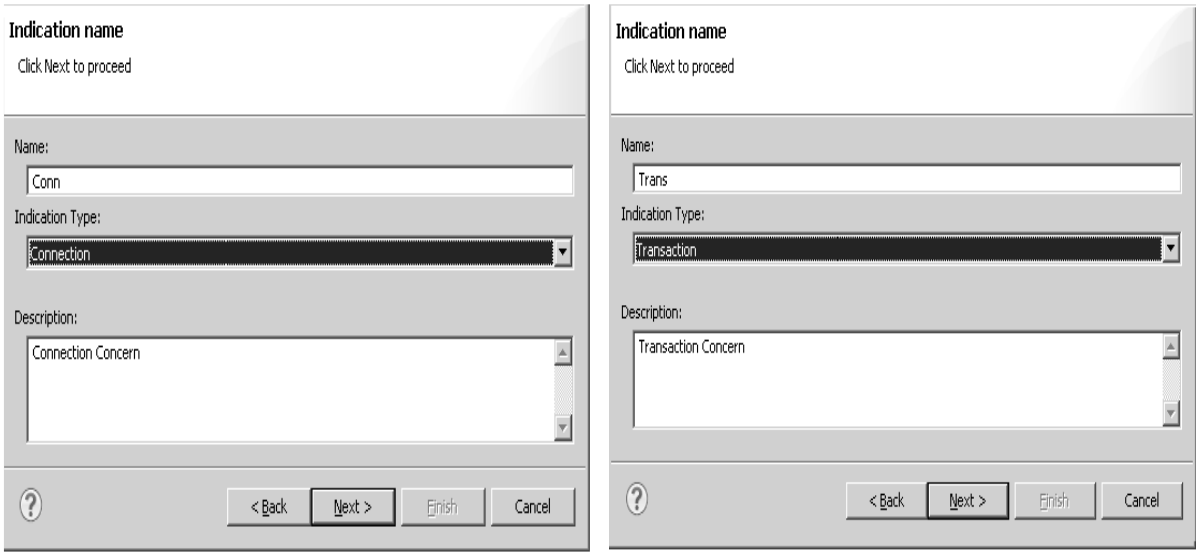

Figura 6.2 – Cadastramento dos estereótipos `Conn` e `Trans` Relacionados aos Interesses de Gerenciamento de Conexões e Transações.

Nesse caso, conforme pode ser visto na Figura 6.2, os estereótipos `Conn` e `Trans` foram relacionados aos interesses `Connection` e `Transaction` por meio do campo “Indication Type” da interface do ComSCId.

O próximo passo é relacionar a interface `IPersistenceMechanism` e a classe `PersistenceMechanism` aos interesses de gerenciamento de conexões e de transações. Por fim, deve-se cadastrar os métodos `getCommunicationChannel()` e `releaseCommunicationChannel()`, criados para implementação do interesse de gerenciamento de conexão e os métodos `beginTransaction()`, `rollBackTransaction()`, `commitTransaction()`, `pmInit()` e `getPm()`, para o interesse de gerenciamento de transações. Apesar de pertencerem à classe `HealthWatcherFacade`, os métodos `pmInit()` e `getPm()` são cadastrados como regras para identificação de indícios do interesse de gerenciamento de transações, pois são responsáveis pela obtenção de instâncias da interface `IPersistenceMechanism` utilizadas para execução dos métodos relacionado a esse interesse.

De modo análogo ao que foi feito no cadastramento dos estereótipos, o usuário pode informar a qual tipo de interesse se refere o método que ele está cadastrando. Por exemplo, ao cadastrar o método `getCommunicationChannel()`, o usuário pode escolher a opção “Open connection” no campo “Indication Type” da interface do ComSCId (Figura 6.3). Isso permitirá que a refatoração específica para esse interesse identifique automaticamente o método utilizado para abertura de conexão com o Banco de Dados.

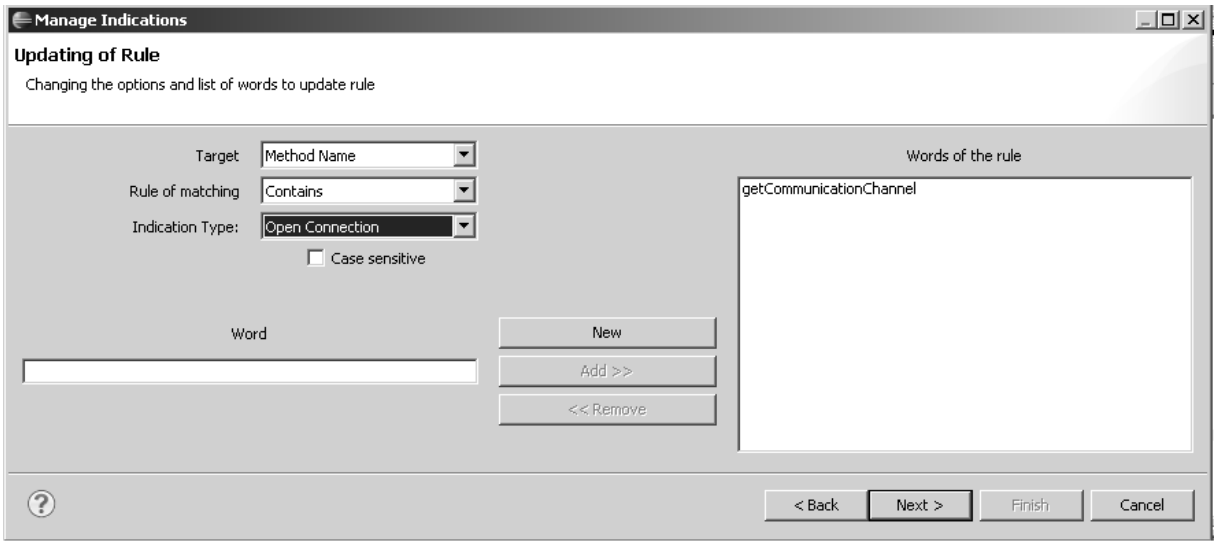

Figura 6.3 – Cadastramento do método `getCommunicationChannel()` no ComSCId.

Para o interesse relacionado ao padrão *Singleton* é criado o estereótipo `Singleton` e os seguintes elementos são utilizados para identificação dos indícios relacionados a esse interesse: o atributo `singletonHW` e o método `getInstanceHW()` da classe `HealthWatcherFacade` e o atributo `singletonPS` e o método `getInstancePS()` da classe `PersistenceMechanism`.

O conjunto completo de regras cadastradas no ComSCId para identificação dos indícios dos interesses de gerenciamento de conexões e transações e do padrão *Singleton* é apresentado na Tabela 6.1. Na primeira coluna dessa tabela estão as palavras-chaves utilizadas para identificação dos indícios, na segunda são descritos os tipos de elementos aos quais se referem as palavras-chaves, na terceira e na quarta colunas, são apresentados, respectivamente, os tipos de interesses a serem analisados e seus estereótipos.

Após o cadastramento das regras no ComSCId, pode-se executar o *plug-in* ComSCId que recupera o modelo de classes OO anotado com indícios de interesse

transversais com o auxílio do *plug-in* MoBRe, resultando no modelo apresentado na Figura 6.4.

Tabela 6.1 – Palavras-chaves Utilizadas para Identificação dos Interesses de Gerenciamento de Conexões, de Transações e do Padrão *Singleton*.

Palavra-chave	Tipo	Tipo de Interesse	Estereótipo
`IPersistenceMechanism`	Interface	Gerenciamento de Conexões	`<<Conn>>`
`PersistenceMechanism`	Classe		
`getCommunicationChannel`	Método		
`releaseCommunicationChannel`	Método		
`IPersistenceMechanism`	Interface	Gerenciamento de Transações	`<<Trans>>`
`PersistenceMechanism`	Classe		
`beginTransaction`	Método		
`commitTransaction`	Método		
`rollBackTransaction`	Método		
`pmInit`	Método		
`getPm`	Método		
`getInstanceHW`	Método	Padrão *Singleton*	`<<Singleton>>`
`singletonHW`	Atributo		
`getInstancePS`	Método		
`singletonPS`	Atributo		

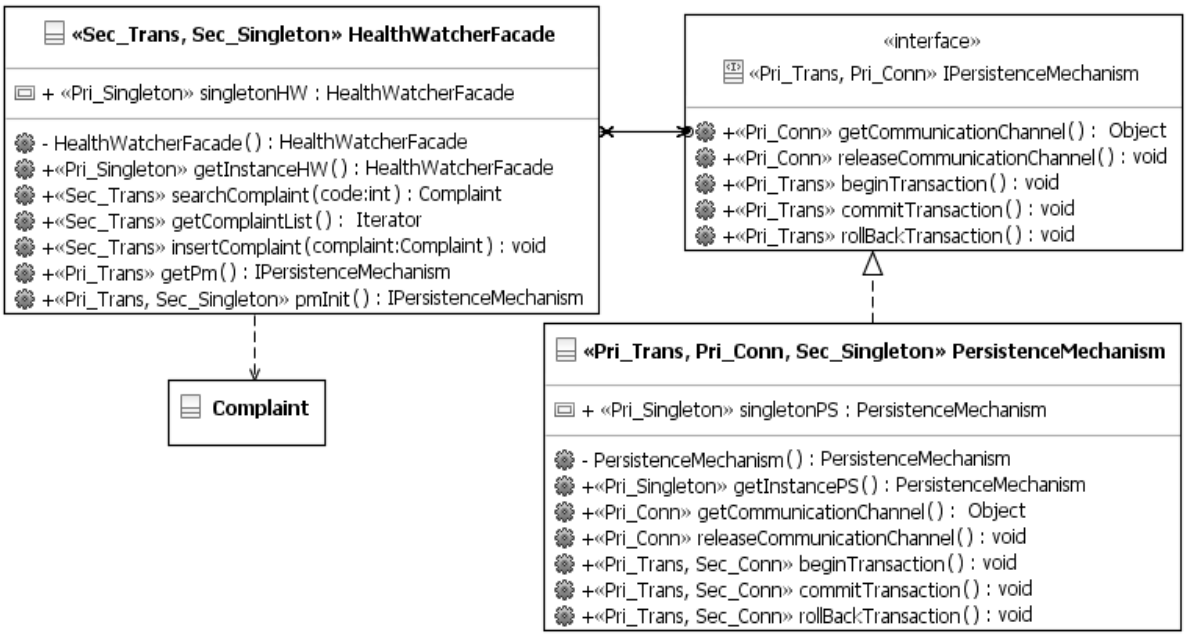

Figura 6.4 – Parte do Modelo de Classes OO do Health Watcher Anotado com Indícios dos Interesses `Conn` e `Trans` e `Singleton`.

Os atributos `singletonHW` e `singletonPS` e os métodos `getInstanceHW()` e `getInstancePS()` possuem o interesse `Singleton` como Primário, pois foram criados especificamente para implementação do padrão *Singleton*. Da mesma forma, os interesses `Conn` e `Trans` são Interesses Primários de `IPersistenceMechanism` e `PersistenceMechanism`, o que está correto uma vez que essas classes foram criadas para implementação desses interesses.

A classe `HealthWatcherFacade` possui os interesses `Trans` e `Singleton` como Interesses Secundários. Isso acontece, pois essa classe não foi cadastrada no conjunto de regras para identificação desses interesses, mas é afetada por eles, como pode ser observado nos trechos de código da classe `HealthWatcherFacade` destacados em cinza na Figura 6.5.

```
1  public Complaint searchComplaint(int code) throws RepositoryException,
2   ObjectNotFoundException, TransactionException {
3   Complaint q = null;
4   try {
5      getPm().beginTransaction();
6         q = this.complaintRecord.search(code);
7         getPm().commitTransaction();
8   } catch (RepositoryException e) {
9         getPm().rollbackTransaction();
10        throw e;
11  }
12  ...
13 }
14
15 public IPersistenceMechanism pmInit() {
16  IPersistenceMechanism returnValue = null;
17  if (Constants.isPersistent()) {
18   try {
19         returnValue = PersistenceMechanism.getInstancePS();
20         ...
21        }
22  ...
23 }
```

Figura 6.5 – Trechos de Código da Classe `HealthWatcherFacade` Afetados pelo Interesses `Trans` e `Singleton`.

Os métodos para gerenciamento de transações da interface `IPersistentMechanism` são invocados no corpo do método `searchComplaint()` (linhas 5, 7 e 9) e o método estático `getInstancePS()` da classe `PersistentMechanism` é invocado no corpo do método `getPm()`(linha 19).

Nesse momento, pode-se aplicar as refatorações genéricas para modelos de classes OO anotados. Utilizando a função “Analyze OO Class Model” do *plug-in* MoBRe, é possível descobrir em qual cenário cada interesse do modelo da Figura

6.4 se encontra e qual refatoração genérica deve ser aplicada para sua modularização. O retorno da função “Analyze OO Class Model” do MoBRe é a mensagem exibida na Figura 6.6, sendo que o MoBRe indicou a aplicação da refatoração *R-1* para o interesse de gerenciamento de transações e da refatoração *R-3* para o interesse relacionado ao padrão *Singleton* (a descrição das refatorações *R-1* e *R-3* são apresentadas na Seção 5.3).

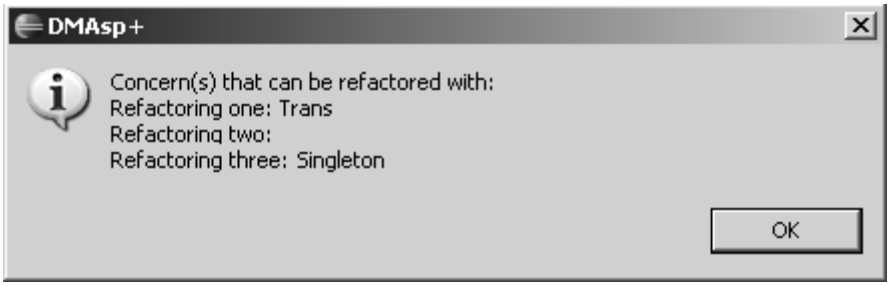

Figura 6.6 – Refatorações Genéricas Passíveis de Aplicação Segundo o *Plug-in* MoBRe.

Nota-se que o interesse `Conn` encontra-se implementado apenas na classe `PersistentMechanism`, isto é, no módulo criado especificamente para persistência de dados da aplicação. Nesse caso, o interesse de gerenciamento de conexões é considerado bem modularizado e, portanto, nenhuma refatoração foi associada a ele. O interesse `Conn` é descartado deste estudo de caso e seus estereótipos são removidos do modelo da Figura 6.4.

Apesar de a interface `IPersistenceMechanism` possuir métodos relacionados aos interesses de gerenciamento de conexões e de transações que são sobrecarregados pela classe `PersistenceMechanism`, os interesses `Conn` e `Trans` não se encontram no cenário adequado para aplicação da refatoração R-2. Isso ocorre, porque tanto a interface quanto a classe possuem esses interesses como Primários que não é o caso em que *R-2* deve ser aplicado (Seção 5.3).

Como visto na Seção 5.2, a ordem com que as refatorações genéricas são aplicadas não influencia no modelo OA resultante. Nesse caso, optou-se por aplicar inicialmente a refatoração R-3 para o interesse `Singleton`, obtendo como resultado o modelo OA da Figura 6.9. Isso foi feito, porque o método `initPm()` da classe `HealthWactherFacade` possui como Interesse Primário, `Trans` e como Secundário, `Singleton`. Assim, caso a refatoração para o interesse `Trans` fosse executada primeiramente, a modularização desse interesse seria incompleta e um reaplicação dessa refatoração seria necessária após a modularização do interesse `Singleton`.

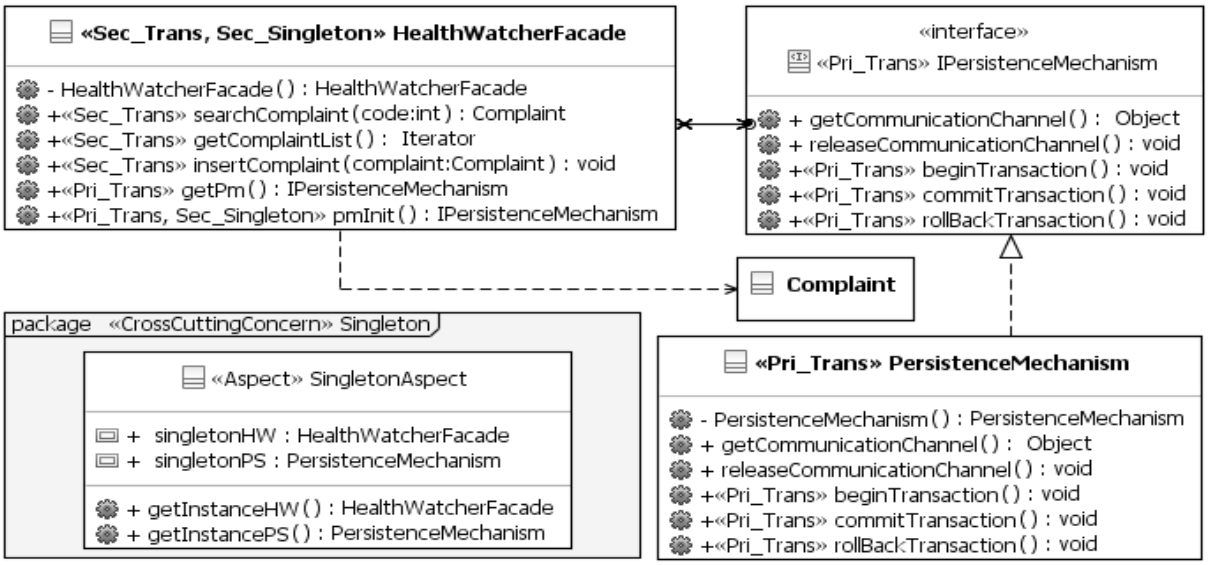

Figura 6.7 – Modelo OA Parcial Obtido a partir da Aplicação da Refatoração R-3 para o Interesse `Singleton`.

As modificações realizadas pela refatoração R-3 sobre o modelo de classes da Figura 6.4 foram: i) criar o aspecto `SingletonAspect`; e ii) mover os atributos `singletonHW` e `singletonPS` e os métodos `getInstanceHW()` e `getInstancePS()` para o aspecto `SingletonAspect`. Nota-se que o método `pmInit()` da classe `HealthWatcherFacade` continua dependendo pelo interesse `Singleton`. Para eliminar essa dependência, aplicou-se a refatoração específica para o padrão `Singleton`, R-*Singleton*, sobre o modelo da Figura 6.8. O modelo resultante é apresentado na Figura 6.9.

As modificações realizadas foram:

- o aspecto `SingletonAspect` se tornou abstrato e um conjunto de junção denominado `instance` foi adicionado a ele. Esse conjunto de junção intercepta as chamadas ao construtor das classes cujas instâncias devem ser únicas, ou seja, as classes que implementam a interface `Singleton`.
- dois novos aspectos foram criados, `HealthWatcherFacadeSingleton-Aspect` e `PersistenceMechanismSingletonAspect`, os quais estendem o aspecto abstrato `SingletonAspect`.

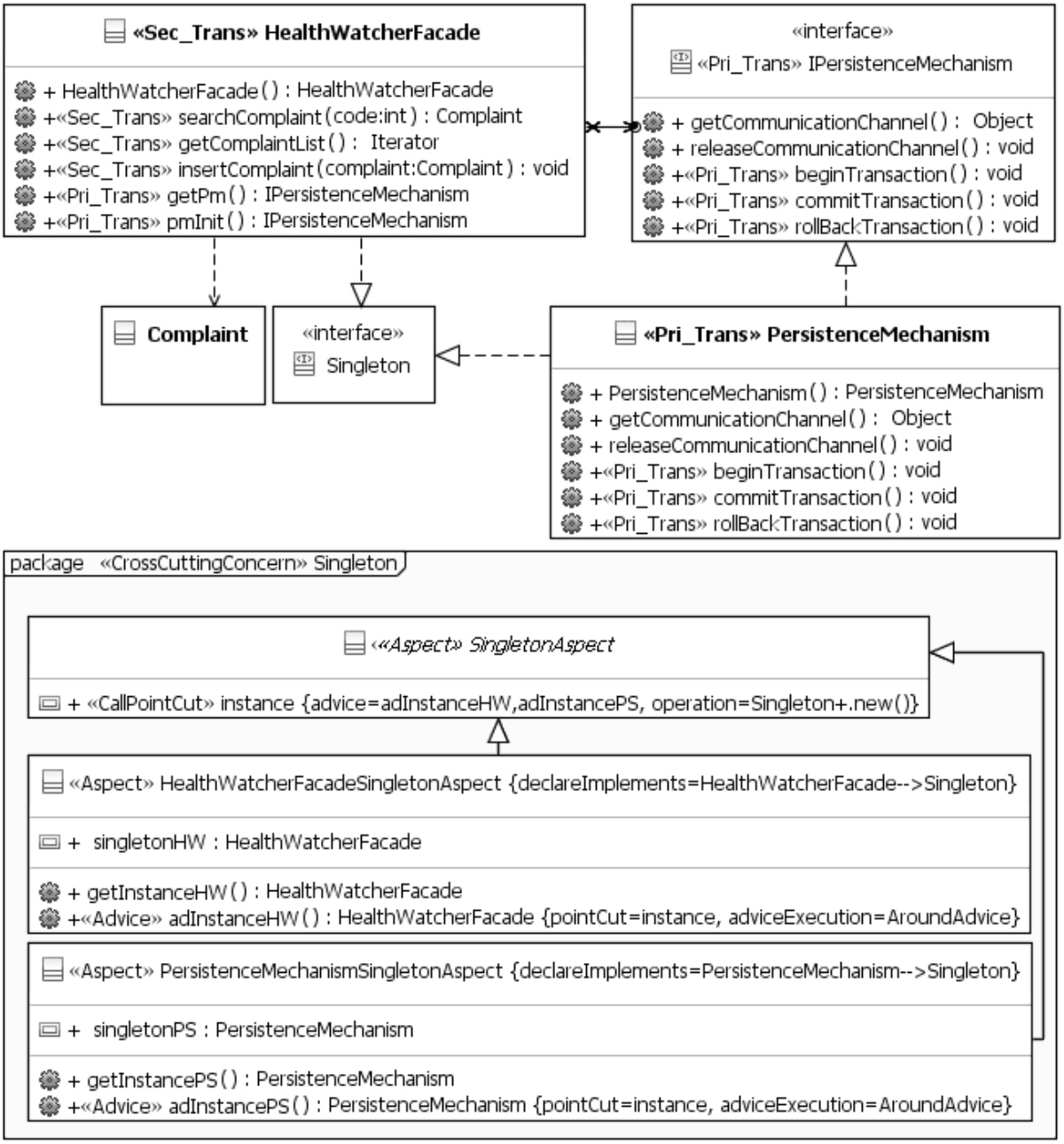

Figura 6.8 – Modelo OA Obtido a partir da Aplicação da Refatoração R-*Singleton*, Específica para o Interesse `Singleton`.

- para cada aspecto criado foram: i) movidos os atributos e métodos correspondentes a cada classe *Singleton*; e ii) criados adendos responsáveis por retornar uma instância da classe *Singleton* quando o conjunto de junção `instance` for alcançado. Por exemplo, o adendo `adInstance()` do aspecto `HealthWatcherFacadeSingletonAspect` retorna uma instância da classe `HealthWatcherFacade` e o adendo `adInstance()` do aspecto `PersistenceMechanismSingletonAspect` retorna uma instância de `PersistenceMechanism`.

- os modificadores dos construtores das classes `HealthWatcherFacade` e `PersistenceMechanism` foram modificados de `private` para `public`. Isso foi feito para que a instância de uma classe *Singleton* seja obtida por meio de seu construtor e não mais pelo método `getInstance()`.

De acordo com a Figura 6.8, nota-se que o único interesse restante é o `Trans` que será modularizado pela refatorações R-1 e R-*Transaction*. A partir da aplicação da refatoração R-1, tem-se o modelo da Figura 6.9.

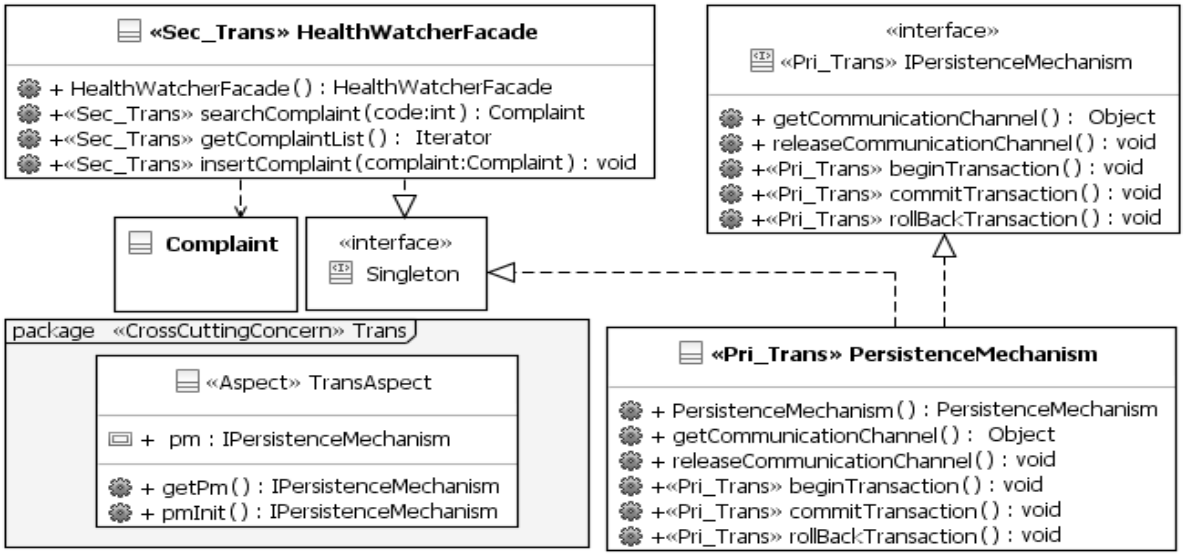

Figura 6.9 – Modelo OA Parcial Obtido a partir da Aplicação da Refatoração R-1 para o Interesse `Trans`.

O atributo correspondente ao relacionamento entre a classe `HealthWatcherFacade` e a interface `IPersistenceMechanism` e os métodos `getPm()` e `pmInit()` foram movido para o aspecto `TransAspect`. A interface `IPersistenceMechanism` e a classe `PersistenceMechanism` não sofrem modificações, pois o interesse `Trans` encontra-se bem modularizado nesses elementos.

Para completar a modularização do interesse `Trans`, aplicou-se a refatoração específica para interesse de gerenciamento de transações (R-*Transaction*, Seção 5.4). Como os tipos de interesses foram associados aos nomes dos estereótipos durante o cadastramento das regras no ComSCId, a refatoração é executada automaticamente sobre o modelo OA parcial da Figura 6.9 e o resultado é exibido na Figura 6.10.

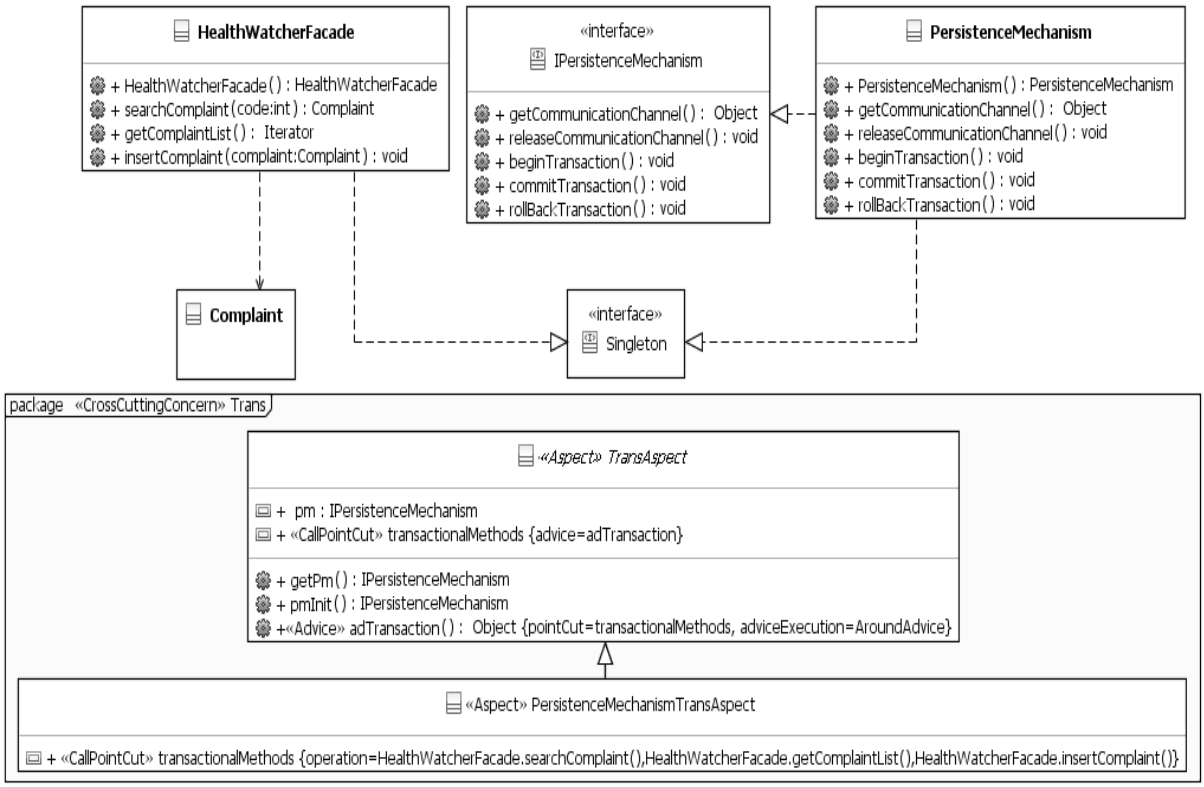

Figura 6.10 – Modelo OA Obtido a partir da Aplicação da Refatoração R-*Transaction*, Específica para o Interesse de Gerenciamento de Transações.

Percebe-se que os estereótipos `<<Sec_Trans>>` da classe `HealthWatcherFacade` foram removidos, o que significa que essa classe não se encontra mais afetada pelo interesse de gerenciamento de transações. Esse interesse foi modularizado nos aspecto `TransAspect` e `PersistenceMechanismAspect`. As modificações realizadas foram:

- o aspecto `TransApect` se tornou abstrato e um conjunto de junção também abstrato denominado `transactionalMethods` foi adicionado a ele. Esse conjunto de junção, quando concretizado em outro aspecto, interceptará as chamadas aos métodos afetados pelo interesse de gerenciamento de transações.
- criou-se o adendo `adTransaction()`, responsável por implementar a lógica de gerenciamento de transações da aplicação, e o associou ao conjunto de junção `transactionalMethods`.
- criou-se o aspecto `PersistenceMechanismTransAspect` que estende o aspecto abstrato `TransAspect` e especificou-se o conjunto de junção `transactionalMethods` com a definição dos métodos afetados pelo interesse `Trans`.

Os estereótipos `<<Pri_Conn>>` e `<<Pri_Trans>>` da interface `IPersistenceMechanism` e da classe `PersistenceMechanism` foram removidos, pois os interesses relacionados a esses estereótipos encontram-se bem modularizados nesses elementos.

6.2.2 Modularização do Interesse de *Logging*

Na Figura 6.11 é apresentada outra parte do modelo de classes OO da aplicação Health Watcher, enfatizando as classes afetadas pelo interesse de *logging*.

As classes `LogMechanism` e `ThreadLogging` com seus atributos e métodos são responsáveis pela implementação do mecanismo de *logging* da aplicação, que são cadastrados como regras para identificação desse interesse no ComSCId. Além disso, há o atributo `singleton` e o método `getInstance()`, responsáveis pela implementação do padrão *Singleton* (Gamma *et al*., 1995), são cadastrados como indícios do interesse relacionado a esse padrão. O conjunto de regras cadastradas no ComSCId para identificação dos indícios dos interesses de *logging* e do padrão *Singleton* é apresentado na Tabela 6.2.

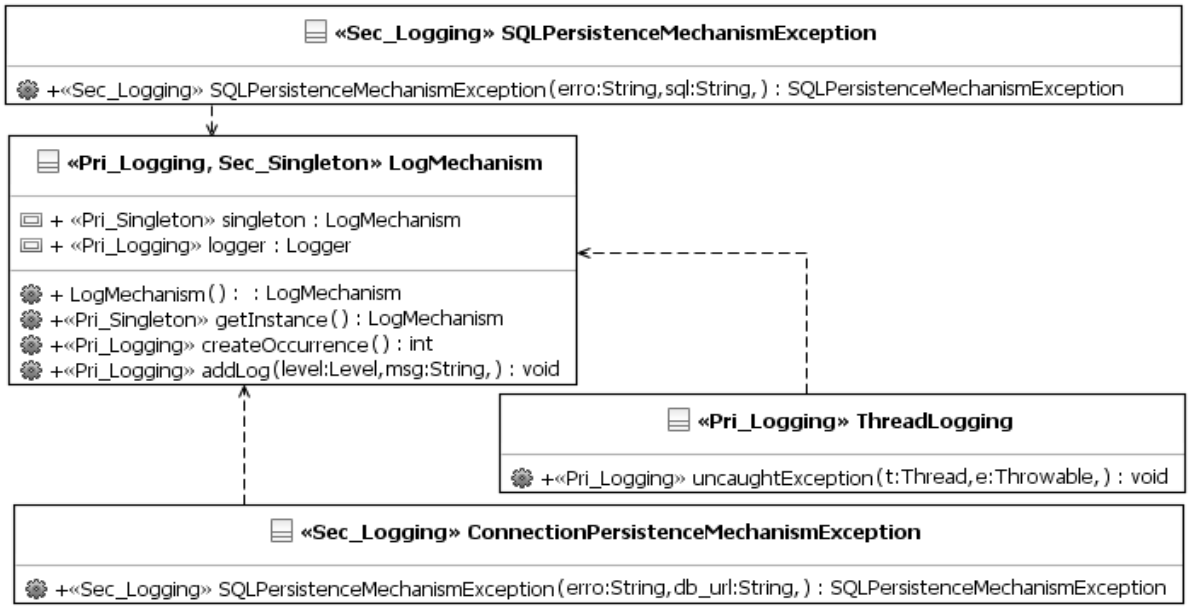

Figura 6.11 – Parte do Modelo de Classes OO do Health Watcher cujas Classes são Afetadas pelo Interesse de *Logging*.

Tabela 6.2 – Palavras-chaves Utilizadas para Identificação dos Interesses de *Logging* e do Padrão *Singleton*.

Palavra-chave	Tipo	Tipo de Interesse	Estereótipo
`LogMechanism`	Classe	*Logging*	`<<Logging>>`
`ThreadLogging`	Classe		
`createOccurrence`	Método		
`addLog`	Método		
`getInstance`	Método	Padrão *Singleton*	`<<Singleton>>`
`singleton`	Atributo		

Após a execução do *plug-in* ComSCId o modelo OO anotado foi recuperado e é o apresentado na Figura 6.12.

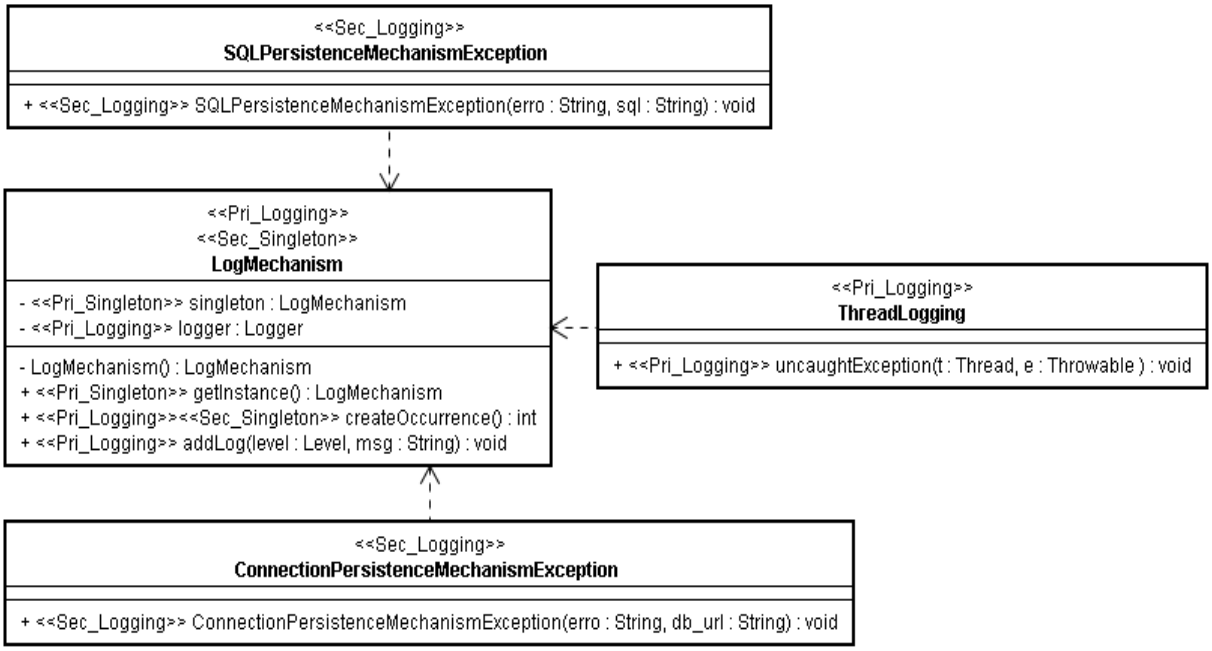

Figura 6.12 – Parte do Modelo de Classes OO do Health Watcher Anotado com Indícios dos Interesses `Logging` *e* `Singleton`.

Como pode ser visto as classes `ConnectionPersistenceMechanism-Exception` e `SQLPersistenceMechanismException`, relacionadas à classe `LogMechanism`, são afetadas pelo interesse de *logging*, e por isso receberam o estereótipo `<<Sec_Logging>>`. As classes responsáveis pela implementação desse interesse, por sua vez, receberam o estereótipo `<<Pri_Logging>>`.

Salienta-se que o padrão *Singleton*, implementado pela classe `LogMechanism`, é utilizado apenas dentro dessa classe. Isso pode ser observado pelo fato de que as demais classes não possuem o estereótipo `<<Singleton>>` e somente o método `createOccurrence()` da classe `LogMechanism`, além da própria

classe, o possui. Assim, nesse caso, o padrão *Singleton* é considerado bem modularizado e será desconsiderado durante a aplicação das refatorações. Essa é uma contribuição importante da utilização de modelos de classes anotados para aplicação de refatorações, pois é possível detectar os cenários em que é necessário ou não modularizar um determinado tipo de interesse e tomar decisões mais conscientes.

Para modularização do interesse de *logging* foram aplicadas as refatorações R-3 e R-*Logging* obtendo como resultado o modelo de classes OA da Figura 6.13.

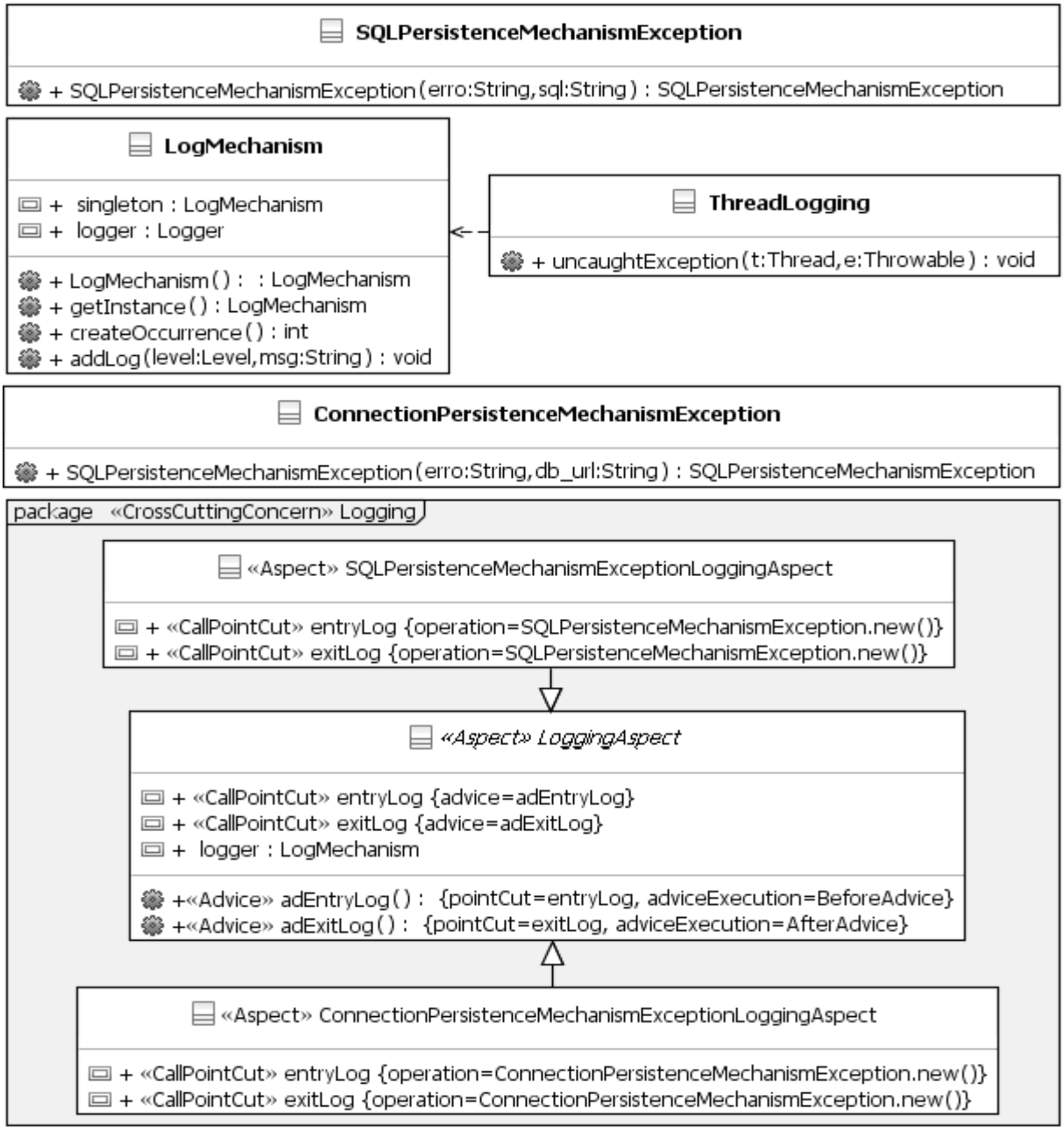

Figura 6.13 – Modelo OA Obtido a partir da Aplicação da Refatoração R-*Logging* Específica para o Interesse de *Logging*.

Observa-se que os relacionamentos entre as classes `SQLPersistenceMechanismException` e `ConnectionPersistenceMechanismException` com `LogMechanism` foram removidos e o comportamento relacionado ao interesse de *logging* foi movido para os aspectos `LoggingAspect`, `SQLPersistenceMechanismExceptionLoggingAspect` e `ConnectionPersistenceMechanismExceptionLoggingAspect`. Nesse caso, como apenas duas classes são afetadas pelo *logging* na aplicação, criou-se um aspecto concreto para cada classe. Como mencionado no passo 4 da refatoração R-*Logging* (Seção 5.4), essa decisão deve ser tomada pelo Engenheiro de Software durante a aplicação das refatorações, com base nas características de implementação da aplicação em análise. Os conjuntos de junção `entryLog` e `exitLog` especificam os pontos da aplicação que precisam ser registrados pelo interesse de *logging*. Para esse exemplo, o método `createOccurrence()` foi utilizado para identificar os pontos relacionados ao conjunto de junção `entryLog`. Isso significa que os métodos da aplicação que invocam o método `createOccurrence()` são interceptados e a criação de uma instância do mecanismo de *logging* é realizada pelo adendo `adEntryLog()`. O mesmo ocorre com o conjunto de junção `exitLog`, porém, nesse caso, o método utilizado para localizar os pontos de junção desse conjunto de junção é o método `addLog()` da classe `LogMechanism`.

Salienta-se que nesse caso, o Engenheiro de Software poderia optar por colocar os conjuntos de junção concretos `entryLog` e `exitLog` em um único aspecto para minimizar a quantidade de novos componentes criados. Para isso, ele poderia optar pela criação de um aspecto por pacote de classes afetadas. Dessa forma, apenas um aspecto seria criado, uma vez que as classes `SQLPersistenceMechanismException` e `ConnectionPersistenceMechanismException` estão localizadas no mesmo pacote.

6.2.3 Modularização do Padrão *Observer*

No HealthWatcher, o padrão *Observer* (Gamma *et al.*, 1995) encontra-se implementado como é apresentado no modelo de classes da Figura 6.14.

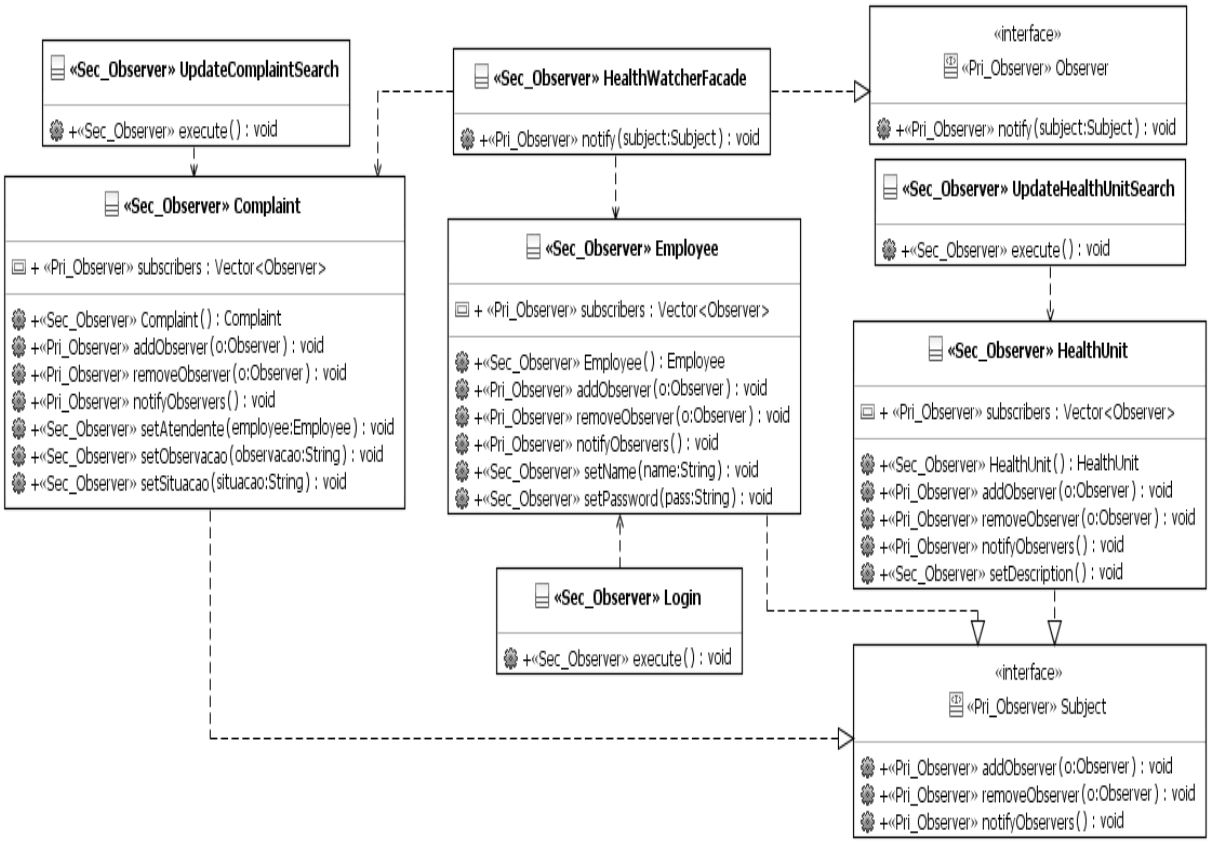

Figura 6.14 – Parte do Modelo de Classes OO do Health Watcher Anotado com Indícios do Interesse `Observer`.

Os elementos cadastrados no ComSCId para obtenção desse modelo anotado foram: i) as interfaces `Subject` e `Observer`; ii) o atributo `subscribers`; e iii) os métodos `addObserver()`, `removeObserver()`, `notifyObservers()` e `notify()`. Nesse exemplo, o relacionamento entre os objetos observados (`Subject`) e seus observadores (`Observer`) foi representado por meio do atributo `subscribers` ao invés do relacionamento de associação, para poder evidenciar o entrelaçamento/espalhamento da implementação do padrão *Observer* pelas classes da aplicação.

O observador é a classe `HealthWatcherFacade` e os elementos observados são as classes `Complaint`, `Employee` e `HealthUnit`. Os métodos estereotipados com `<<Sec_Observer>>` são afetados pelo padrão *Observer*, pois utilizam elementos (atributos e/ou métodos) dedicados à implementação desse padrão. Por exemplo, no código da Figura 6.15 observa-se que toda vez que valor do atributo `name` da classe `Employee` é alterado, o método `notifyObservers()` herdado da interface `Subject` é invocado (linha 5). Já os construtores das classes `Complaint`, `Employee` e `HealthUnit` recebem esse estereótipo, pois são responsáveis por

inicializar o conjunto de observadores dessa classe que é representado pelo atributo `subscribers`.

```
1  public class Employee {
2   ...
3   public void setName(String name){
4    this.name = name;
5    notifyObservers();
6   }
7   ...
8  }
```

Figura 6.15 – Trecho de Código da Classe `Employee`.

O Engenheiro de Software não precisa conhecer os pontos nos quais uma chamada ao método `notifyObservers()` é realizada, pois isso é feito automaticamente pelo *plug-in* MoBRe que consulta o modelo de classes OO anotado a procura de chamadas a esse método. Como esse procedimento depende de convenções de nomenclatura o nome do método responsável por notificar os observadores deve ser informado durante a aplicação da refatoração R-*Observer* ou durante o cadastramento das regras para identificação do interesse relacionado ao padrão *Observer* no ComSCId.

Como pode ser observado na Figura 6.14, a implementação do padrão *Observer* encontra-se espalhada pelas demais classes da aplicação, entrelaçando-se com a implementação de outros interesses como a lógica de negócios. Para modularização desse interesse é necessário executar a refatoração genérica R-2 e a refatoração específica R-*Observer*. O resultado da modularização é apresentado na Figura 6.16.

Observa-se que os métodos e atributos responsáveis pela implementação do padrão *Observer* foram movidos para o aspecto `ObserverAspect` que os reintroduz nas classes e interfaces correspondentes.

Nesse exemplo os principais conjuntos de junção são `initializeObservers` e `notify`. O primeiro é responsável por inicializar o conjunto de observadores de cada elemento observado. Para isso, o conjunto de junção `initializeObservers` intercepta a execução dos construtores das classes que herdam da interface `Subject` e o adendo com mesmo nome, `intializeObservers()`, inicializará o conjunto de o atributo `subscribers`.

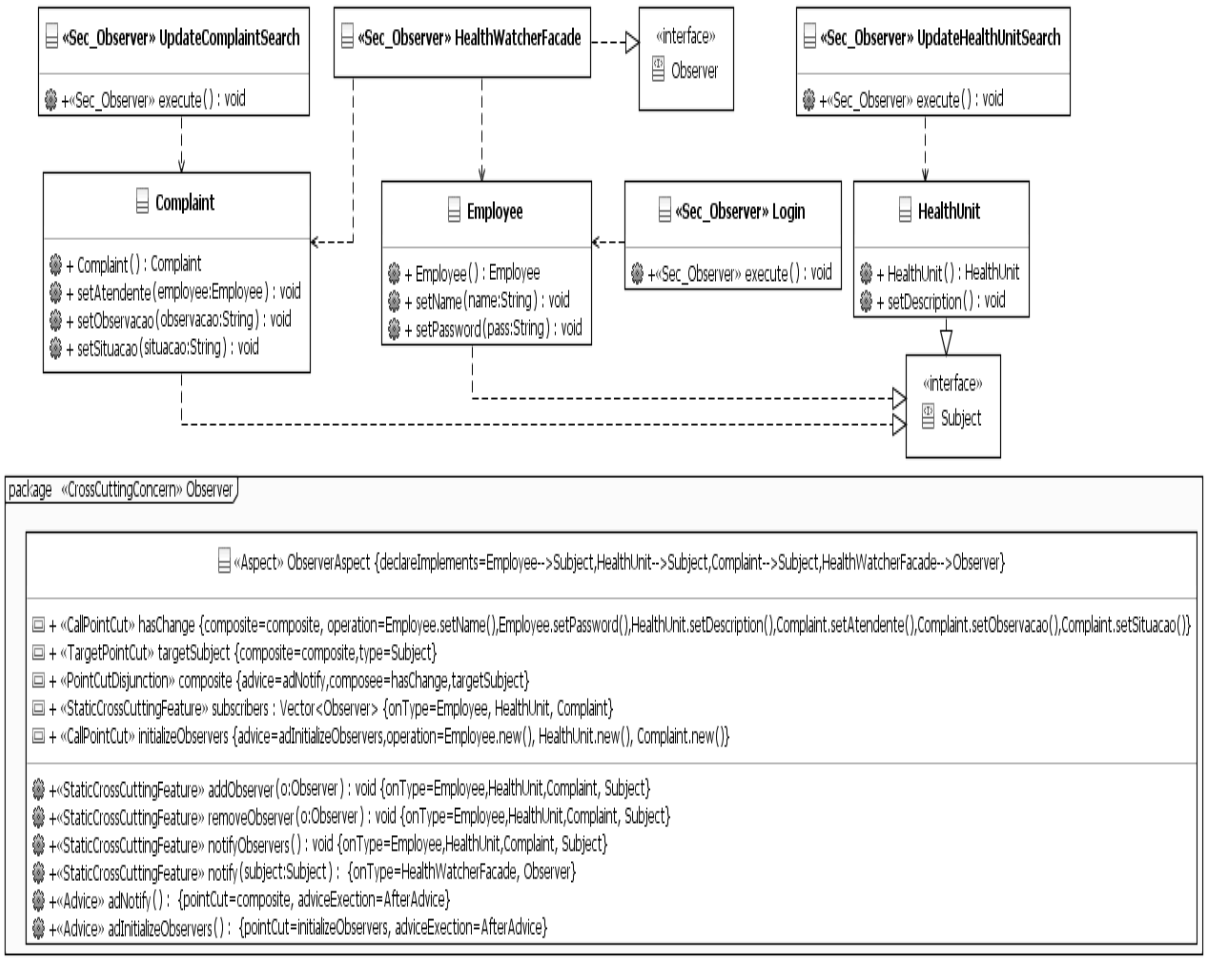

Figura 6.16 – Modelo OA Obtido a partir da Aplicação da Refatoração R-*Observer* Específica para o Padrão *Observer*.

Já o conjunto de junção `notify`, juntamente com o adendo `notify()` relacionado a ele, será responsável por notificar os observadores de um determinado elemento observado que alguma modificação ocorreu nesse elemento. Para que o método `notifyObservers()` do elemento observado seja corretamente invocado, é necessário capturar uma instância desse elemento com o conjunto de junção `targetSubject`. Os atributos e métodos relacionados ao padrão *Observer* são reintroduzidos em suas classes e interfaces correspondentes por meio de declarações intertipo.

Nota-se ainda que as classes `Login`, `UpdateComplaintSearch` e `UpdateHealthUnitSearch` continuam com o estereótipo `<<Sec_Observer>>`. Isso acontece, pois no método `execute()` dessas classes, os elementos observados são associados aos observadores por meio do método `addObserver()`. Essa responsabilidade geralmente não é completamente modularizada com o auxílio de aspectos, pois, muitas vezes é difícil obter as instâncias do elemento observado e do

observador para poder associá-los. Hanneman *et al.* (2002) ao modularizarem esse padrão, moveram os elementos responsáveis pela associação entre observadores e observados para um aspecto, entretanto, a associação entre eles continuou sendo realizada pela aplicação base (OO) por meio de chamadas aos métodos presentes nos aspectos criados.

6.3 Estudo de Caso: *JSpider*

Neste estudo de caso será aplicada a refatoração específica para o interesse de *logging* na aplicação *JSpider*. JSpider é um robô para *download* e análise de páginas *Web*. Na Figura 6.17 é apresentada uma parte do modelo de classes da aplicação *JSpider* cujas classes são afetadas pelos interesses transversais relacionados ao *logging* e ao padrão *Singleton*.

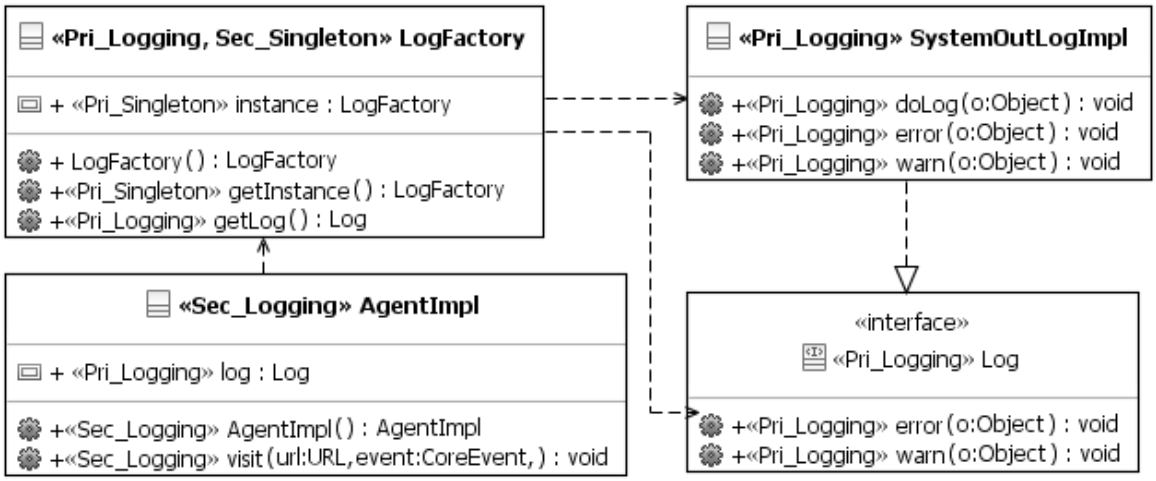

Figura 6.17 – Parte do Modelo de Classes OO do *JSpider* Anotado com Indícios dos Interesses `Logging` e `Singleton`.

As classes `LogFactory` e `SystemOutLogImpl` e a interface `Log` são responsáveis pela implementação do interesse de *logging* da aplicação. A classe `LogFactory` cria instâncias da classe `SystemOutLogImpl` por meio do método `getLog()`. As classes do tipo `Log`, como é o caso de `SystemOutLogImpl`, possuem métodos que permitem registrar informações do contexto de execução da aplicação, por exemplo, `error()` e `warn()`. Além disso, a classe `LogFactory` possui o atributo `instance` e o método `getInstance()` que são responsáveis pela implementação do padrão *Singleton*. Com base nessas informações, as regras apresentadas na Tabela

6.3 foram cadastradas para identificação dos interesses transversais de *logging* e do padrão *Singleton* na aplicação *JSpider*.

Tabela 6.3 – Palavras-chaves Utilizadas para Identificação do Interesse de *Logging* e do Padrão *Singleton*.

Palavra-chave	Tipo	Interesse	Estereótipo
`LogFactory`	Classe	*Logging*	`<<Logging>>`
`SystemOutLogImpl`	Classe		
`Log`	Interface		
`getLog`	Método		
`log`	Atributo		
`doLog`	Método		
`error`	Método		
`warn`	Método		
`instance`	Atributo	Padrão *Singleton*	`<<Singleton>>`
`getInstance`	Método		

A classe `AgentImpl` possui o atributo `log` que foi implementado especificamente para registrar as informações de *log* da aplicação, por isso, ele recebe o estereótipo `<<Pri_Logging>>`. No exemplo da Figura 6.17, além do relacionamento de associação, o atributo `log` foi colocado propositadamente no diagrama para evidenciar o tipo de estereótipo utilizado para esse atributo. Como `AgentImpl` é uma classe de negócio que não foi criada para implementação do interesse de *logging*, ela recebe o estereótipo `<<Sec_Logging>>`. De modo análogo ao que aconteceu na implementação do interesse de *logging* da aplicação *Health Watcher*, o interesse relacionado ao padrão *Singleton* encontra-se bem modularizado na classe `LogFactory` e por isso não será refatorado nesse estudo de caso.

O resultado da refatoração do interesse de *logging* da aplicação *JSpider* é apresentado na Figura 6.18. Nota-se que foi criado um aspecto denominado `Net_Javacoding_Jspider_Core_Aspect`, que corresponde ao nome do pacote no qual a classe `AgentImpl` está contida. Nesse exemplo, escolheu-se a estratégia de criar um aspecto concreto para cada pacote que possui classes afetadas pelo *logging*, pois a quantidade de classes afetadas é grande. O método `getLog()` da classe `LogFactory` foi considerado como método de entrada para registro de *log* e

os métodos `error()` e `warn()` da interface `Log` foram considerados métodos para registro de saída para o registro de *logs*.

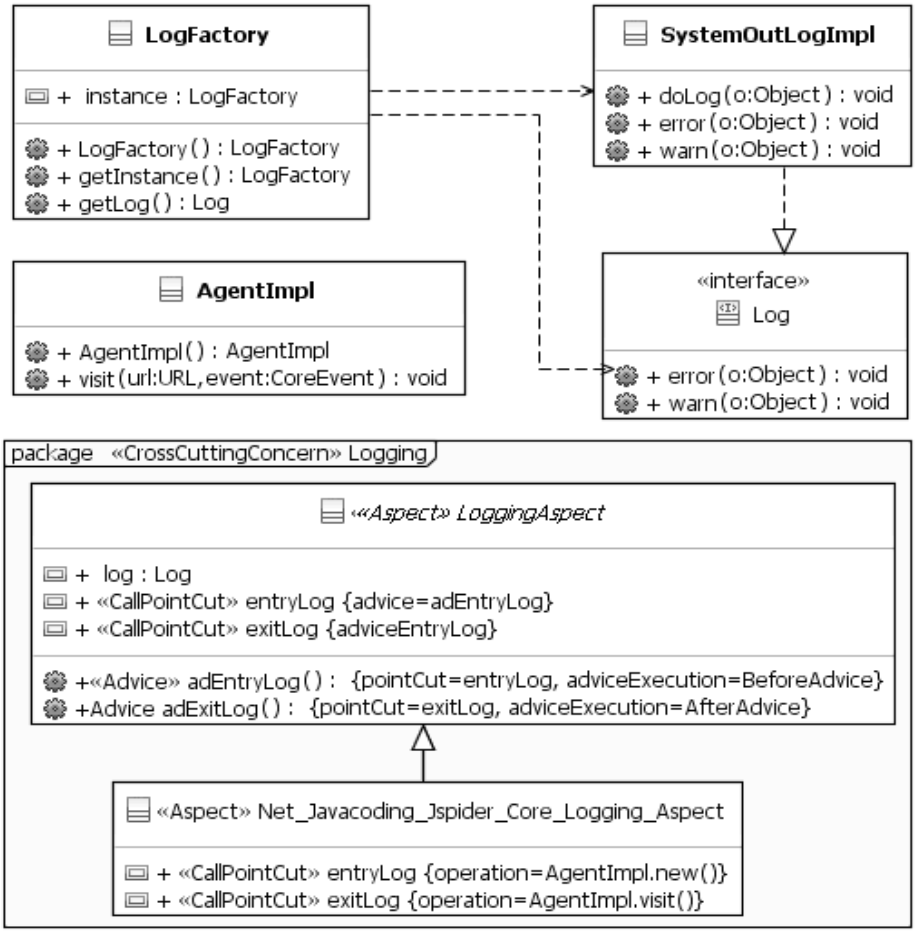

Figura 6.18 – Modelo OA Obtido a partir da Aplicação da Refatoração R-*Transaction.*

Percebe-se que o conjunto de junção `entryLog()` está associado ao construtor da classe `AgentImpl`. Isso ocorreu, pois o *plug-in* MoBRe detectou durante a aplicação das refatorações, que esse construtor invoca o método `getLog()` da classe `Log`. Da mesma forma, o método `visit()` dessa classe `AgentImpl` está relacionado ao conjunto de junção `exitLogError`, pois no corpo desse método há chamadas ao método `error()` da interface `Log`. Foi criado apenas o conjunto de junção `exitLogError`, pois o método `warn()` não foi utilizado. Esses detalhes não estão presentes no modelo de classes anotado, mas podem ser encontrados no metamodelo do MoBRe que é responsável pela geração do modelo de classes da Figura 6.17. Salienta-se que por meio das refatorações aqui propostas, foi possível obter modelos OA que modularizam o interesse de *logging* para duas aplicações distintas, implementadas por terceiros: *Health Watcher* e *JSpider*.

6.4 Estudo de Caso: *JAccounting*

Neste estudo de caso é aplicada a refatoração para o interesse de persistência, em particular, para gerenciamento de transações na aplicação *JAccounting*, um sistema *Web* para controle de pedidos e pagamentos. Na Figura 6.19 é apresentada uma parte do modelo de classes da aplicação *JAccounting* cujas classes são afetadas pelo interesse transversal de gerenciamento de transações. O conjunto de regras cadastradas no ComSCId para identificação dos indícios dos interesses de gerenciamento de transações é apresentado na Tabela 6.4.

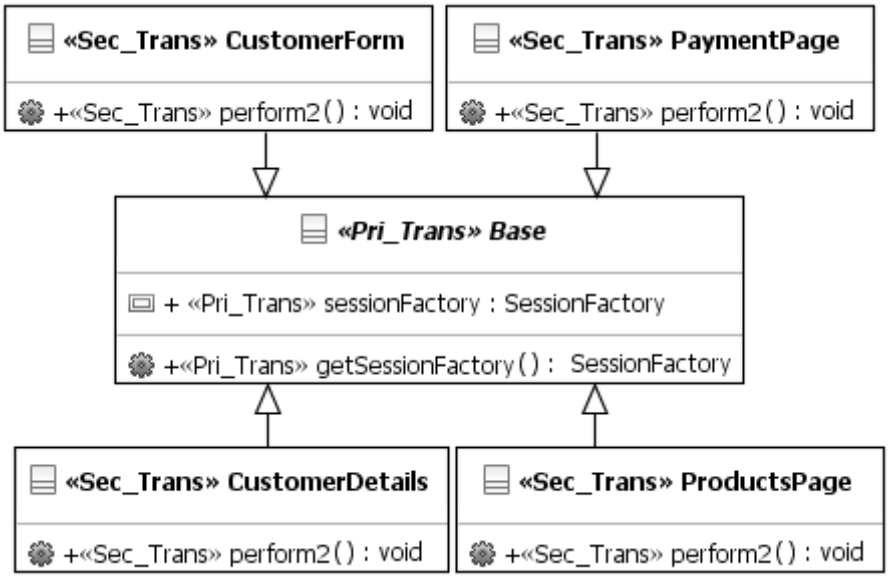

Figura 6.19 – Parte do Modelo de Classes OO do JAccounting Anotado com Indícios do Interesse `Trans`.

Tabela 6.4 – Palavras-chaves Utilizadas para Identificação do Interesse de Gerenciamento de Transações.

Palavra-chave	Tipo	Interesse	Estereótipo
`Base`	Classe	Gerenciamento de Transações	`<<Trans>>`
`sessionFactory`	Atributo		
`getSessionFactory`	Método		

As classes `CustomerForm`, `CustomerDetails`, `ProductPage` e `PaymentPage` pertencem à camada de visão da aplicação e são responsáveis pela apresentação e pelo cadastramento de informações do negócio, como pagamentos, clientes e produtos. A classe `Base` possui o método `getSessionFactory()` e o atributo

`sessionFactory`, responsáveis pela criação de sessões com o Banco de Dados que são utilizadas para inicialização de transações. A classe `Base` foi criada especificamente para implementação da persistência na aplicação *JAccounting*, por isso, o interesse de `Trans` é primário nessa classe.

Como se pode observar na Figura 6.19, o cenário de entrelaçamento é adequado para aplicação da refatoração R-2, uma vez que as classes `CustomerForm`, `CustomerDetails`, `ProductPage` e `PaymentPage` são subclasses de `Base` e utilizam métodos dessa classe para implementação do interesse de gerenciamento de transações. Como refatoração específica, o Engenheiro de Software pode aplicar a refatoração *R-Transaction* para o interesse de gerenciamento de transações, obtendo-se o modelo da Figura 6.20.

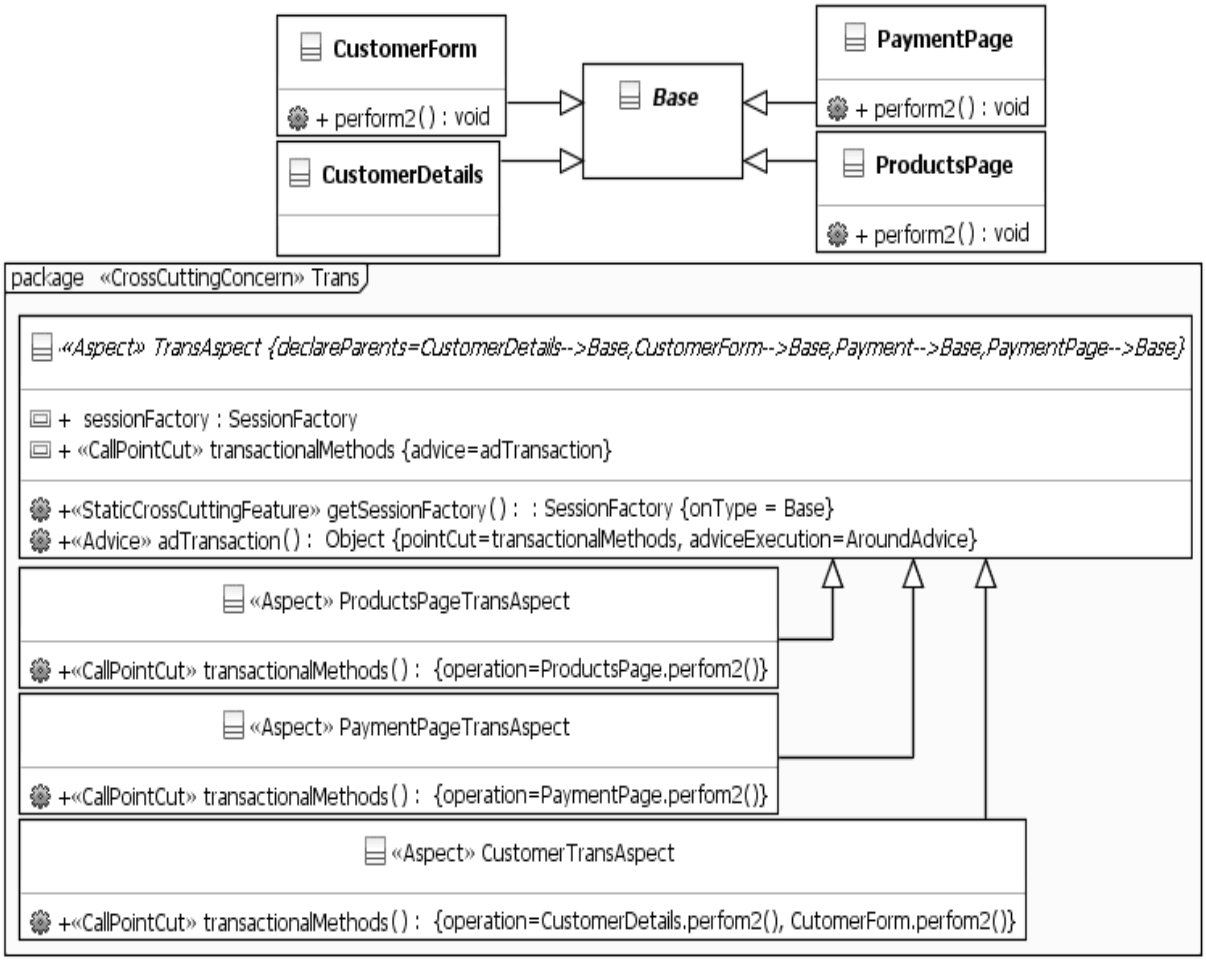

Figura 6.20 – Modelo OA Obtido a partir da Aplicação da Refatoração R-*Transaction.*

Os estereótipos `<<Sec_Trans>>` das classes `CustomerForm`, `CustomerDetails`, `ProductPage` e `PaymentPage` foram removidos, pois o comportamento relacionado ao gerenciamento de transações foi modularizado pelos

aspectos `TransAspect`, `ProductsPageTransAspect`, `CustomerTransAspect`, e `PaymentPageTransAspect`. As classes que possuem responsabilidade relacionadas com cliente, `CustomerDetails` e `CustomerForm`, são modularizadas apenas pelo aspecto `CustomerTransAspect`, isso foi feito pelo Engenheiro de Software com o auxílio da ferramenta MoBRe com o objetivo de minimizar a quantidade de aspectos criados na aplicação. As modificações realizadas nesse exemplo são semelhantes às modificações apresentadas na Subseção 6.2, para a aplicação Health Watcher. Para a aplicação *JAccounting* um número maior de aspectos foi criado, pois mais classes são afetadas pelo interesse de gerenciamento de transações.

6.5 Considerações Finais

Neste capítulo apresentou-se a aplicação das refatorações elaboradas sobre os modelos de classes anotados de três aplicações distintas implementadas em Java. Essas aplicações foram desenvolvidas para contextos diferentes (Health Watcher, sistema de informação para área da saúde; JSpider, um robô para análise de páginas Web; e JAccounting, um sistema de informação para controle de venda de produtos) e por desenvolvedores distintos. Todas as nove refatorações elaboradas foram aplicadas e pode-se observar que os resultados obtidos oferecem indícios da viabilidade de aplicação das refatorações elaboradas. No próximo capítulo será realizada uma avaliação mais objetiva dos resultados com base na utilização de métricas orientadas a aspectos.

Capítulo 7

AVALIAÇÃO

7.1 Considerações Iniciais

Com base no estudo de caso realizado no Capítulo 6, observou-se indícios da viabilidade de aplicação das refatorações propostas em aplicações OO. Entretanto, não foi realizado um estudo mais significativo dos resultados da aplicação dessas refatorações com relação à modularização e à qualidade do modelo OA gerado.

Neste capítulo é apresentada a comparação entre os modelos OA obtidos com o auxílio das refatorações desenvolvidas neste trabalho e os modelos existentes na literatura com base em métricas OA. Para isso, realizou-se a engenharia reversa das versões OA das aplicações Health Watcher (Greenwood et al., 2007), JSpider (JSpider, 2011) e JAccounting (JAccounting, 2011) com objetivo de obter seus diagramas de classes para depois compará-los aos diagramas obtidos no estudo de casos aqui realizados. A comparação entre os modelos OA obtidos a partir das refatorações e os modelos obtidos a partir da engenharia reversa do código fonte é feita de por meio da utilização de métricas OA adaptadas ao contexto de modelos de classes anotados.

A organização do capítulo é a seguinte: na Seção 7.2 são apresentados os conjuntos de métricas criados e/ou adaptados ao contexto de modelos anotado para comparação entre as soluções de modularização das aplicações Health Watcher, JSpider e JAccounting. Na Seção 7.3 e Seção 7.4 são apresentadas as avaliações das soluções para modularização das aplicações Health Watcher, JSpider e

JAccounting, respectivamente. As considerações finais são apresentadas na Seção 7.5.

7.2 Métricas para Verificação da Qualidade dos Modelos de Classes OA Obtidos a Partir da Aplicação das Refatorações

A avaliação da qualidade dos modelos desenvolvidos no estudo de casos do Capítulo 6 é realizada de duas maneiras: i) comparando o modelo de classes OO com o modelo de classes OA obtido a partir da aplicação das refatorações desenvolvidas neste trabalho; e ii) comparando o modelo de classes OA obtido a partir da aplicação das refatorações com outra versão desse modelo de classes OA obtido a partir da engenharia reversa do código OA desenvolvido por terceiros e disponibilizados na literatura. Neste trabalho, realizou-se a engenharia reversa da versão OA das aplicações Health Watcher (Greenwood et al., 2007), JSpider (2011) e JAccounting (2011) com objetivo de obter seu diagrama de classes e depois compará-lo com os diagramas obtidos a partir da aplicação das refatorações.

A comparação entre os modelos é realizada subjetivamente por meio da avaliação conceitual entre esses modelos e objetivamente por meio da utilização de métricas OA adaptadas ao contexto de modelos de classes anotados. As métricas utilizadas para comparação objetiva são apresentadas na Tabela 7.1 e Tabela 7.2.

A Tabela 7.1 apresenta um conjunto de métricas OA extraído da literatura (Figueiredo *et al.*, 2008; Sant'anna *et al.*, 2007; Ceccato e Tonella, 2004; Ducasse *et al.*, 2006) e adaptado ao contexto dos modelos de classes anotados. Isso foi feito, pois tais métricas foram elaboradas para serem aplicadas em códigos fonte OO e OA e com essa adaptação puderam ser obtidas diretamente a partir do modelo de classes anotado. Na primeira e segunda colunas da Tabela 7.1 são apresentados o nome da métrica e seu significado original. Na terceira coluna é apresentada a adaptação do significado dessa métrica para o contexto dos modelos de classes anotados. Caso o significado adaptado seja idêntico ao significado original, o símbolo "-" é colocado.

Tabela 7.1 – Métricas OA Adaptadas para Comparação entre Modelos de Classes OO Anotados e Modelos de Classes OA.

Métrica	Significado Original dada pelo Autor da Métrica	Adaptação ao Contexto de Modelos de Classe Anotados
NOA (*Numbers of Attributes*)	Conta o número de variáveis e atributos de classes, interfaces ou aspectos.	Conta o número de atributos de classes, interfaces ou aspectos, pois no modelo de classes não é possível identificar variáveis existente no corpo de um método. Além disso, como são utilizados modelos de classes da UML, relacionamentos de associação, agregação e composição entre classes são contadas como atributos.
NOO (*Number of Operations*)	Conta o número de métodos construtores e adendos de classes, interfaces e aspectos.	-
CA (*Concern Attributes*)	Conta o número de atributos que contribuem para implementação de um determinado interesse.	Conta o número de atributos que são afetados por interesses classificados como Secundários no componente (classe, interface ou aspecto) ao qual ele pertence.
CO (*Concern Operations*)	Conta o número de operações que participam da implementação de um interesse.	Conta o número de operações que são afetadas por interesses classificados como Secundários no componente (classe, interface ou aspecto) ao qual ele pertence.
CA% ((CA/NOA)*100)	Porcentagem entre o número de atributos relacionados à implementação de um interesse e todos os atributos existentes em um módulo.	Porcentagem entre o número de atributos que são afetados por interesses classificados como Secundários e todos os atributos existentes em um componente.
CO% ((CO/NOO)*100)	Porcentagem entre o número de operações relacionadas à implementação de um interesse e todas as operações existentes em um módulo.	Porcentagem entre o número de operações que são afetadas por interesses classificados como Secundários e todas as operações existentes em um componente.
VS (*Vocabulary Size*)	Conta o número de componentes do software.	-

Na Tabela 7.2 o conjunto de métricas elaborado neste trabalho é apresentado, sendo que na primeira coluna tem-se o nome da métrica e na segunda, a sua descrição.

Tabela 7.2 – Métricas OA Elaboradas para Comparação entre Modelos de Classes OO Anotados e Modelos de Classes OA.

Métrica	Descrição da Métrica
NOSC (*Number of Secundary Concerns*)	Conta o número de Interesses Secundários que afetam um determinado componente (classe, interface ou aspecto).
NOP (*Number of Pointcuts*)	Conta o número de conjuntos de junção existentes em um aspecto.
NOAP (*Number of Anonymous Pointcuts*)	Conta o número de conjuntos de junção anônimos existentes em um aspecto.
NOAP% ((NOAP/NOP)*100)	Porcentagem entre o número de conjuntos de junção anônimos e todos os conjuntos de junção existentes em um aspecto.
NORP (*Number of Operations Related with the Pointcut*)	Conta o número de operações relacionadas a um determinado conjunto de junção.
NORP/ (NORP/NOP)	Proporção entre o número de operações relacionadas aos conjuntos de junção de um aspecto e o total de conjuntos de junção existentes neste aspecto.
NOAC (*Number of Affected Components*)	Conta o número de componentes (classes, interfaces e aspectos) que são afetados por Interesses Secundários.
NOAC% ((NOAC/VS)*100)	Porcentagem entre o número de componentes que são afetados por Interesses Secundários e todos os componentes do sistema.

Esse conjunto de métricas é classificado em dois grupos:

- **Métricas de Modularização**: são as utilizadas para comparação entre as duas versões OO do modelo de classes de um software: o modelo original da aplicação, ou seja, antes da modularização dos interesses transversais existentes no software e o outro corresponde ao modelo OO após a modularização dos interesses transversais. O objetivo é determinar o grau de modularização dos interesses existentes em um software. Para isso são utilizadas as métricas NOA, NOO, CA, CO, CA%, CO%, NOAC, NOAC%, VS e NOSC.
- **Métricas de Adequação às Boas Práticas de Projeto OA**: são utilizadas para comparação entre as duas versões OA do modelo de classes de um software: o modelo obtido com o auxílio das refatorações elaboradas neste

trabalho e o outro corresponde a uma solução criada por outro pesquisador, sem utilizar as refatorações. O objetivo é determinar a correspondência desses modelos de classes OA às práticas de projeto OA preconizadas pela comunidade científica. As métricas NOC, NOP, NOAP, NOAP%, NORP e NORP/ são utilizadas.

A métrica NOC é classificada no grupo de métricas de adequação às boas práticas de projeto OA, pois pode indicar a existência de aspectos responsáveis pela modularização de mais de um tipo de interesse, o que corresponde a uma má prática de projeto OA e pode levar à implementação de *bad smells* como o *God Aspect* (Piveta *et al.*, 2007).

7.3 Avaliação das Soluções para Modularização da Aplicação Health Watcher

Nesta seção as métricas de modularização e de adequação às boas práticas de projeto OA foram aplicadas à aplicação Health Watcher (Greenwood et al., 2007) apresentadas no Capítulo 6 e comparadas com as soluções propostas pelos seus desenvolvedores. No Capítulo 6, com o objetivo de facilitar a visualização dos modelos de classes, alguns elementos desses modelos foram omitidos propositadamente. Assim, os valores aqui apresentados podem diferir dos apresentados no Capitulo 7. O modelo de classes foi analisado por completo (com todos os atributos, métodos e relacionamentos entre classes).

Na Tabela 7.3, Tabela 7.4 e Tabela 7.5 são apresentados os valores das métricas de modularização para os modelos de classes OO afetados pelos interesses de Persistência, *Logging* e os Padrões de Projeto *Singleton* e *Observer*, antes da aplicação das refatorações desenvolvidas nesse trabalho.

Tabela 7.3 – Valores das Métricas de Modularização para o Modelo de Classes OO Anotado com Indícios do Interesse de Persistência e do Padrão *Singleton*.

Componentes	Métricas						
	NOA	NOO	CA	CO	NOA%	NOO%	NOSC
HealthWatcherFacade	9	30	3	28	33,33	93,33	2
IPersistenceMechanism	0	7	0	0	0,00	0,00	0
PersistenceMechanism	10	14	1	1	10,00	7,14	1
TOTAL	**19**	**51**	**4**	**29**	**21,05**	**56,86**	**-**

Métricas		
VS	**NOAC**	**NOAC%**
135	3	2,22

Tabela 7.4 – Valores das Métricas de Modularização para o Modelo de Classes OO Anotado com Indícios do Interesse de *Logging*.

Componentes	Métricas						
	NOA	**NOO**	**CA**	**CO**	**NOA%**	**NOO%**	**NOSC**
`SQLPersistenceMechanismException`	1	1	0	1	0,00	100,00	1
`LogMechanism`	4	7	0	0	0,00	0,00	0
`ThreadLogging`	0	1	0	0	0,00	0,00	0
`ConnectionPersistenceMechanism-Exception`	1	1	0	1	0,00	100,00	1
TOTAL	**6**	**10**	**0**	**2**	**0,00**	**20,00**	**-**

Métricas		
VS	**NOAC**	**NOAC%**
135	2	1,48

Tabela 7.5 – Valores das Métricas de Modularização para o Modelo de Classes OO Anotado com Indícios do Interesse Relacionado ao Padrão *Observer*.

Componentes	Métricas						
	NOA	**NOO**	**CA**	**CO**	**NOA%**	**NOO%**	**NOSC**
`UpdateComplaintSearch`	1	2	0	1	0,00	50,00	1
`UpdateHealthUnitSearch`	1	2	0	1	0,00	50,00	1
`HealthUnit`	4	12	1	5	25,00	41,67	1
`Complaint`	5	29	1	7	20,00	24,14	1
`HealthWatcherFacade`	9	30	0	1	0,00	3,33	1
`Observer`	0	1	0	0	0,00	0,00	0
`Subject`	0	3	0	0	0,00	0,00	0
`Employee`	4	11	1	6	25,00	54,55	1
`Login`	3	2	0	1	0,00	50,00	1
TOTAL	**27**	**92**	**3**	**22**	**11,11**	**23,91**	**-**

Métricas		
VS	**NOAC**	**NOAC%**
135	7	5,19

Em todos os casos, nota-se que a implementação dos interesses transversais existentes no Health Watcher encontra-se bem entrelaçada e espalhada pelos demais módulos da aplicação. Apesar dos valores da métrica NOAC% serem relativamente baixos. Essa afirmação pode ser justificada pelos altos valores das métricas NOA% e NOO%, aproximadamente 33% dos atributos e 93% dos métodos da classe `HealthWatcherFacade` são afetados pelo interesse de persistência e pelo padrão *Singleton*. Outro exemplo é a classe `Employee`, uma classe de negócio que possui 25% de seus atributos e aproximadamente 55% de seus métodos afetados pelo padrão *Observer*. Assim, o uso de uma solução de modularização deve ser capaz de remover todo entrelaçamento existente no software, ou seja, zerar os valores dessas métricas.

7.3.1 Soluções para Modularização do Interesse de Persistência

Os valores das métricas de modularização do interesse de persistência e do padrão *Singleton*, para duas versões do modelo de classes OA da aplicação Health Watcher, são apresentados: i) o primeiro foi obtido com auxílio das refatorações para o interesse de persistência implementadas no *plug-in* MoBRe (Tabela 7.6); ii) o segundo foi obtido pela realização da engenharia reversa do código fonte OA disponibilizado pelos desenvolvedores do Health Watcher (Greenwood et al., 2007) (Tabela 7.7).

Tabela 7.6 – Valores das Métricas para o Modelo de Classes OA Obtido com o Auxílio das Refatorações para os Interesses de Persistência e do Padrão *Singleton*.

Componentes	Métricas											
	NOA	NOO	CA	CO	NOA%	NOO%	NOSC	NOP	NOAP	NORP	NOAP%	NORP/
HealthWatcher-Facade	9	30	0	0	0,00	0,00	0					
IPersistence-Mechanism	0	7	0	0	0,00	0,00	0					
Persistence-Mechanism	10	14	0	0	0,00	0,00	0					
TransAspect	2	3	0	0	0,00	0,00	0	1	0	0	0,00	0
PersistenceMecha-nismTransAspect	0	0	0	0	0,00	0,00	0	1	0	25	0,00	25
SingletonAspect	0	0	0	0	0,00	0,00	0	1	0	1	0,00	1
HealthWatcher-FacadeSingleton-Aspect	1	2	0	0	0,00	0,00	0	0	0	0	0,00	0
PersistenceMecha-nismSingletonAs-pect	1	2	0	0	0,00	0,00	0	0	0	0	0,00	0
Singleton	0	0	0	0	0,00	0,00	0	0	0	0	0,00	0
TOTAL	**23**	**58**	**0**	**0**	**0,00**	**0,00**	**-**	**3**	**0**	**26**	**0,00**	**8,67**

Métricas		
VS	NOAC	NOAC%
141	0	0,00

A solução para modularização proposta por Greenwood *et al.*, (2002) apresentou similaridades à solução obtida com o auxílio das refatorações propostas neste trabalho. A diferença é que na solução de Greenwood, dois aspectos foram criados para modularização do interesse de gerenciamento de transações. O aspecto `HWPersistence` é responsável por criar instâncias da classe `PeristenceMechanism` e o aspecto `HWTransactionManagement` intercepta os pontos da aplicação que realizam chamadas aos métodos para gerenciamento de transações. Como pontos fracos da solução de Greenwood, destacam-se:

Tabela 7.7 – Valores das Métricas para o Modelo de Classes OA Obtido sem o Auxílio das Refatorações para os Interesses de Persistência e do Padrão *Singleton*.

Componentes	Métricas											
	NOA	NOO	CA	CO	NOA%	NOO%	NOSC	NOP	NOAP	NORP	NOAP%	NORP/
HealthWatcher-Facade	9	30	1	1	11,11	3,33	1					
IPersistence-Mechanism	0	7	0	0	0,00	0,00	0					
Persistence-Mechanism	10	14	0	0	0,00	0,00	0					
HWPersistence	2	3	0	1	0,00	33,33	1	1	1	1	100,00	1
HWTransaction-Management	0	0	0	0	0,00	0,00	0	1	1	26	100,00	25
TOTAL	**21**	**54**	**1**	**2**	**4,76**	**3,70**	**-**	**2**	**2**	**27**	**100,00**	**13,50**

Métricas		
VS	NOAC	NOAC%
135	2	1,48

- O interesse relacionado ao padrão *Singleton* não foi modularizado: isso pode ter ocorrido pelo fato de que o Engenheiro de Software não possuía um recurso de visualização de interesses transversais adequado que o permitisse detectar as características transversais do padrão *Singleton*;
- para modularização do interesse de gerenciamento de transações, apenas aspectos concretos são utilizados: essa estratégia pode tornar o projeto OA pouco flexível e de difícil manutenção. Pouco flexível, pois sem a abstração proporcionada pelo mecanismo de generalização/especialização de aspectos, é mais custoso criar/reutilizar aspectos especializados para tratamento do gerenciamento de transações em outros módulos do sistema. De difícil manutenção, pois muitas responsabilidades podem estar presentes em um único aspecto dificultando seu entendimento e consequentemente sua manutenção.
- Foram utilizados conjuntos de junção anônimos na modularização do interesse de gerenciamento de transações: a utilização de conjuntos de junção anônimos é não considerada boa prática de projeto OA (Piveta *et al.*, 2007),pois pode diminuir a legibilidade do software OA e aumentar o esforço necessário para sua manutenção.

7.3.2 Soluções para Modularização do Interesse de *Logging*

De modo análogo ao que foi feito para os interesses de persistência e do padrão *Singleton*, na Tabela 7.8 e Tabela 7.9 são apresentados os valores das

métricas de modularização e adequação às boas práticas de projeto OA para as soluções de modularização do interesse de *logging*.

Para melhorar a visualização dessas tabelas os identificadores das classes, interfaces e aspectos foram substituídos pelas seguintes siglas: SQLPME para a classe `SQLPersistenceMechanismException`, ConnPME para a classe `ConnectionPersistenceMechanismException`, SQLPMEAs para o aspecto `SQLPersistenceMechanismExceptionLoggingAspect` e ConnPMEAs para o aspecto `ConnectionPersistenceMechanismExceptionLoggingAspect`.

Tabela 7.8 – Valores das Métricas para o Modelo de Classes OA Obtido com o Auxílio das Refatorações para o Interesse de *Logging*.

Componentes	Métricas											
	NOA	NOO	CA	CO	NOA%	NOO%	NOSC	NOP	NOAP	NORP	NOAP%	NORP/
SQLPME	1	1	0	0	0,00	0,00	0					
LogMechanism	4	7	0	0	0,00	0,00	0					
ThreadLogging	0	1	0	0	0,00	0,00	0					
ConnPME	1	1	0	0	0,00	0,00	0					
LoggingAspect	1	2	0	0	0,00	0,00	0	2	0	0	0,00	0
SQLPMEAs	0	0	0	0	0,00	0,00	0	2	0	1	0,00	1
ConnPMEAs	0	0	0	0	0,00	0,00	0	2	0	1	0,00	1
TOTAL	**7**	**12**	**0**	**0**	**0,00**	**0,00**	**-**	**6**	**0**	**2**	**0,00**	**0,33**

Métricas		
VS	NOAC	NOAC%
138	0	0,00

Tabela 7.9 – Valores das Métricas para o Modelo de Classes OA Obtido com o Auxílio das Refatorações para o Interesse de *Logging*.

Componentes	Métricas											
	NOA	NOO	CA	CO	NOA%	NOO%	NOSC	NOP	NOAP	NORP	NOAP%	NORP/
SQLPME	1	1	0	1	0,00	100,00	1					
LogMechanism	4	7	0	0	0,00	0,00	0					
ThreadLogging	0	1	0	0	0,00	0,00	0					
ConnPME	1	1	0	1	0,00	100,00	1					
HWLogging	0	1	0	0	0,00	0,00	0	1	1	1	100,00	1
TOTAL	**6**	**11**	**0**	**2**	**0,00**	**18,18**	**-**	**1**	**1**	**1**	**100,00**	**1**

Métricas		
VS	NOAC	NOAC%
136	2	1,47

Nesse exemplo, apenas o modelo OA obtido com o auxílio das refatorações genéricas e específicas representa uma boa solução para modularização do interesse de *logging*. Isso pode ser observado pelos valores obtidos para as métricas de modularização da Tabela 7.9: i) a quantidade de componentes afetados pelo *logging* continua igual; ii) todos os métodos das classes `SQLPersistenceMechanismException` e `ConnectionPersistenceMechanism-`

`Exception` são afetados pelo interesse de *logging*. Conforme o trecho de código da aplicação Health Watcher apresentado na Figura 7.1, a modularização realizada por *Greenwood et al. (2007)* permite que apenas o mecanismo de *logging* seja configurado durante a inicialização da aplicação.

```
1  public aspect HWLogging {
3   /**
4   * Before initializing a facade or a servlet, config the logger
5   */
6   before() : call(HealthWatcherFacade.new(..))
7    || staticinitialization(HWServlet) {
8         LogMechanism.configure(Constants.LOG_PATH);
9         System.out.println("Configuring logger");
10  }
11 }
```

Figura 7.1 – Trecho de Código da Aplicação Health Watcher Responsável pela Modularização do Interesse de *Logging*.

A solução obtida com a aplicação das refatorações desenvolvidas neste trabalho possibilitou a remoção da interferência do interesse de *logging* sobre as classes afetadas por ele. Porém, para isso foram criados três novos aspectos, um abstrato e dois concretos, aumentando assim a quantidade de componentes a serem mantidos na aplicação. Essa estratégia foi escolhida, pois pode garantir maior flexibilidade ao software, permitindo a redução de esforços para reutilização do mecanismo de *logging* nos demais módulos da aplicação ou em outros softwares.

7.3.3 Soluções para Modularização do Interesse Relacionado ao Padrão *Observer*

Nas Tabela 7.10 e Tabela 7.11 são apresentados os valores das métricas para o interesse relacionado ao padrão de projeto *Observer*.

Ao contrário do que ocorreu para o interesse de *logging*, a única solução que apresenta a modularização correta do padrão *Observer* é a proposta por *Greenwood et al. (2007)*. Nota-se que, para a solução obtida com o auxílio das refatorações genéricas e específicas, as classes `Login`, `UpdateComplaintSearch` e `UpdateHealthUnitSearch` continuam sendo afetadas pelo padrão. Isso acontece, pois no método `execute()` dessas classes, os elementos observados são associados aos observadores por meio do método `addObserver()`.

Tabela 7.10 – Valores das Métricas para o Modelo de Classes OA Obtido com o Auxílio das Refatorações para os Interesse de Persistência e do Padrão Singleton.

Componentes	Métricas											
	NOA	NOO	CA	CO	NOA%	NOO%	NOSC	NOP	NOAP	NORP	NOAP%	NORP/
UpdateComplaint-Search	1	2	0	1	0,00	50,00	1					
UpdateHealth-UnitSearch	1	2	0	1	0,00	50,00	1					
HealthUnit	4	12	0	0	0,00	0,00	0					
Complaint	5	29	0	0	0,00	0,00	0					
HealthWatcher-Facade	9	30	0	0	0,00	0,00	0					
Observer	0	1	0	0	0,00	0,00	0					
Subject	0	3	0	0	0,00	0,00	0					
Employee	4	11	0	0	0,00	0,00	0					
Login	3	2	0	1	0,00	50,00	1					
ObserverAspect	1	6	0	0	0,00	0,00	0	4	0	7	0,00	1,75
TOTAL	**28**	**98**	**0**	**3**	**0,00**	**3,06**	**-**	**4**	**0**	**7**	**0,00**	**1,75**

Métricas		
VS	NOAC	NOAC%
136	3	2,20

Tabela 7.11 – Valores das Métricas para o Modelo de Classes OA Obtido sem o Auxílio das Refatorações para os Interesse de Persistência e do Padrão Singleton.

Componentes	Métricas											
	NOA	NOO	CA	CO	NOA%	NOO%	NOSC	NOP	NOAP	NORP	NOAP%	NORP/
UpdateComplaint-Search	1	2	0	0	0,00	0,00	0					
UpdateHealth-UnitSearch	1	2	0	0	0,00	0,00	0					
HealthUnit	4	12	0	0	0,00	0,00	0					
Complaint	5	29	0	0	0,00	0,00	0					
HealthWatcher-Facade	9	30	0	0	0,00	0,00	0					
Observer	0	1	0	0	0,00	0,00	0					
Subject	0	3	0	0	0,00	0,00	0					
Employee	4	11	0	0	0,00	0,00	0					
Login	3	2	0	0	0,00	0,00	0					
ObserverProtocol	1	5	0	0	0,00	0,00	0	1	0	0	0,00	0,00
UpdateState-Observer	0	1	0	0	0,00	0,00	0	2	1	7	50,00	3,50
TOTAL	**28**	**98**	**0**	**0**	**0,00**	**0,00**	**-**	**3**	**1**	**7**	**33,33**	**2,33**

Métricas		
VS	NOAC	NOAC%
137	0	0,00

Essa responsabilidade geralmente não é completamente modularizada com o auxílio de aspectos, pois, muitas vezes é difícil obter as instâncias do elemento observado e do observador para poder associá-lo. Hanneman *et al.* (2002) ao modularizar esse padrão, move os elementos responsáveis pela associação entre observadores e observados para um aspecto, entretanto, a associação entre eles continua sendo realizada pela aplicação base (OO) por meio de chamadas aos

métodos presentes nos aspectos criados. A solução desenvolvida por Greenwood baseou-se no amplo conhecimento do Engenheiro de Software com relação ao software do Health Wathcer, o que facilitou a criação de um conjunto de junção específico para associação entre observadores e elementos observados.

Salienta-se, entretanto, que a solução obtida com o auxílio das refatorações obteve bons resultados para os demais módulos da aplicação. Por exemplo, houve uma redução de 57% no valor da métrica NOAC% que representada a quantidade de componentes afetados pelo interesse relacionado ao padrão *Observer*. Com relação às métricas que indicam adequação às boas práticas de projeto OA, NOAP% e NORP/, a solução de Greenwood apresentou valores inferiores aos valores da solução obtida com uso das refatorações. Por exemplo, o valor da métrica NOAP%, que representa a porcentagem de conjuntos de junção anônimos existentes na aplicação, foi 33,33% para a solução de Greenwood e 0% para a outra solução. Além disso, o valor da métrica NORP/, que mede a complexidade dos conjuntos de junção da aplicação, também foi maior para a solução de Greenwood.

7.4 Avaliação das Soluções para Modularização das Aplicações *JSpider* e *JAccounting*

Nesta seção as métricas de modularização e de adequação às boas práticas de projeto OA foram aplicadas às soluções de modularização das aplicações JSpider e JAccounting apresentadas no estudo de casos do Capítulo 6 e comparadas às soluções propostas por Binkley *et al.* (2006).

Na Tabela 7.12 e Tabela 7.13 são apresentados os valores das métricas de modularização para os modelos de classes OO das aplicações *JSpider* e *JAccounting*, respectivamente, que são afetados pelos interesses de *Logging* e Persistência.

De acordo com os dados das Tabela 7.12 e Tabela 7.13, nota-se que a implementação dos interesses de *logging* e de gerenciamento de transações, nas aplicações *JSpider* e *JAccounting,* encontra-se entrelaçada e espalhada pelos demais módulos dessas aplicações. Aproximadamente 13% de todos os componentes da aplicação *JSpider* e 10% da *JAccounting* possuem *logging* e

gerenciamento de transações como interesses secundários. Em alguns casos, um interesse pode afetar até 100% dos elementos de um mesmo componente, como é o caso da classe `PluginInstantiator`, afetada pelo interesse de *logging*.

Tabela 7.12 – Valores das Métricas de Modularização para o Modelo de Classes OO Anotado com Indícios do Interesse de *Logging* da Aplicação *JSpider*.

Componentes	Métricas						
	NOA	NOO	CA	CO	NOA%	NOO%	NOSC
EventDispatcherImpl	7	4	1	3	14,29	75,00	1
AgentImpl	5	17	1	2	20,00	11,76	1
PluginFactory	1	2	0	1	0,00	50,00	1
PluginInstantiator	1	5	1	5	100,00	100,00	1
SpiderContextImpl	17	21	1	3	5,88	14,29	1
SpiderImpl	3	2	0	2	0,00	100,00	1
RuleFactory	2	5	0	1	0,00	20,00	1
StorageFactory	0	1	0	1	0,00	100,00	1
CookieDAOImpl	3	3	1	1	33,33	33,33	1
DecisionDAOImpl	3	5	1	1	33,33	20,00	1
EMailAddressDAOImpl	3	5	1	1	33,33	20,00	1
ResourceDAOImpl	5	21	1	2	20,00	9,52	1
SiteDAOImpl	4	5	1	1	25,00	20,00	1
ContentDAOImpl	3	3	1	3	33,33	100,00	1
CookieDAOImpl	4	4	1	3	25,00	75,00	1
DBUtil	5	9	0	3	0,00	33,33	1
DecisionDAOImpl	11	7	1	3	9,09	42,86	1
EMailAddressDAOImpl	3	7	1	5	33,33	71,43	1
FolderDAOImpl	7	8	1	5	14,29	62,50	1
ResourceDAOImpl	12	23	1	16	8,33	69,57	1
SiteDAOImpl	14	7	1	6	7,14	85,71	1
SchedulerFactory	1	1	0	1	0,00	100,00	1
BaseWorkerTaskImpl	3	4	1	2	33,33	50,00	1
WorkerThread	8	7	0	1	0,00	14,29	1
ThrottleFactory	0	1	0	1	0,00	100,00	1
DistributedLoadThrottleProvider	3	1	0	1	0,00	100,00	1
SimultaneousUsersThrottleProvider	4	1	0	1	0,00	100,00	1
ConsolePlugin	10	7	1	2	10,00	28,57	1
DiskWriterPlugin	11	12	1	3	9,09	25,00	1
FileWriterPlugin	9	8	1	3	11,11	37,50	1
StatusBasedFileWriterPlugin	6	35	1	2	16,67	5,71	1
VelocityPlugin	23	9	1	5	4,35	55,56	1
MaxNumberOfURLParamsRule	2	2	0	1	0,00	50,00	1
MaxResourcesPerSiteRule	2	2	0	1	0,00	50,00	1
LogFactory	4	6	0	0	0,00	0,00	0
SystemOutLogImpl	0	21	0	0	0,00	0,00	0
Log	0	18	0	0	0,00	0,00	0
TOTAL	**199**	**299**	**22**	**92**	**11,05**	**30,76**	**-**

Métricas		
VS	NOAC	NOAC%
251	34	13,54

Tabela 7.13 – Valores das Métricas de Modularização para o Modelo de Classes OO Anotado com Indícios do Interesse de Gerenciando de Transações da Aplicação *JAccounting*.

Componentes	Métricas						
	NOA	NOO	CA	CO	NOA%	NOO%	NOSC
`ChargePage`	11	23	0	1	0,00	4,35	1
`CompanySelect`	0	1	0	1	0,00	100,00	1
`CustomerDetails`	2	5	0	1	0,00	20,00	1
`CustomerForm`	3	5	0	1	0,00	20,00	1
`InvoicePage`	4	9	0	1	0,00	11,11	1
`PaymentPage`	4	10	0	2	0,00	20,00	1
`ProductsPage`	2	5	0	1	0,00	20,00	1
`RecurrencePage`	4	9	0	1	0,00	11,11	1
`Statement`	2	5	0	1	0,00	20,00	1
`Base`	14	13	0	0	0,00	0,00	0
TOTAL	**46**	**85**	**0**	**10**	**0,00**	**11,76**	**-**

Métricas		
VS	NOAC	NOAC%
85	9	10,58

Na Tabela 7.14 e Tabela 7.15 são apresentados valores das métricas de modularização e adequação às boas práticas de projeto OA para duas soluções de modularização do interesse de *logging*: i) a primeira foi proposta por Binkley *et al.* (2006); e ii) a segunda foi obtida com o auxílio das refatorações elaborados neste trabalho.

Na Tabela 7.14 pode-se observar que a solução proposta por Binkley (Binkley *et al.*, 2006) modularizou parcialmente o interesse de *logging* da aplicação *JSpider*. O valor da métrica NOAC% continua o mesmo, ou seja, todos os componentes afetados pelo *logging* antes modularização continuam sendo afetados após esse interesse ter sido modularizado. Ainda de acordo com o dados da Tabela 7.14 pode-se notar que a quantidade de métodos afetados pelo *logging* na aplicação zerou, porém a quantidade de atributos afetados aumentou de 11% para 16%. Isso ocorreu por dois motivos: i) os atributos do tipo `Log` localizados nos componentes afetados pelo *logging* não foram movidos para um aspecto; ii) as classes que não possuíam um atributo do tipo `Log`, mas que utilizavam o mecanismo dentro de seus métodos receberam um atributo desse tipo. Dessa forma, a quantidade de atributos afetados pelo *logging* na aplicação *JSpider* aumentou. Uma solução possível para esse problema é criar um atributo do tipo `Log` no aspecto `LogAspect` e utilizá-lo para gravar as informações de *logging* da aplicação.

Tabela 7.14 – Valores das Métricas para Modelo de Classes OA da Aplicação JSpider Obtido a partir da Solução Proposta por Binkley *et al.* (2006).

Componentes	Métricas											
	NOA	NOO	CA	CO	NOA%	NOO%	NOSC	NOP	NOAP	NORP	NOAP%	NORP/
EventDispa-tcherImpl	7	4	1	0	14,29	0,00	1					
AgentImpl	5	17	1	0	20,00	0,00	1					
PluginFactory	2	2	1	0	50,00	0,00	1					
PluginIns-tantiator	1	5	1	0	100,00	0,00	1					
SpiderContext-Impl	17	21	1	0	5,88	0,00	1					
SpiderImpl	4	2	1	0	25,00	0,00	1					
RuleFactory	3	5	1	0	33,33	0,00	1					
StorageFactory	1	1	1	0	100,00	0,00	1					
CookieDAOImpl	3	3	1	0	33,33	0,00	1					
DecisionDAOImpl	3	5	1	0	33,33	0,00	1					
EMailAddress-DAOImpl	3	5	1	0	33,33	0,00	1					
ResourceDAOImpl	5	21	1	0	20,00	0,00	1					
SiteDAOImpl	4	5	1	0	25,00	0,00	1					
ContentDAOImpl	3	3	1	0	33,33	0,00	1					
CookieDAOImpl	4	4	1	0	25,00	0,00	1					
DBUtil	6	9	1	0	16,67	0,00	1					
DecisionDAOImpl	11	7	1	0	9,09	0,00	1					
EMailAddress-DAOImpl	3	7	1	0	33,33	0,00	1					
FolderDAOImpl	7	8	1	0	14,29	0,00	1					
ResourceDAOImpl	12	23	1	0	8,33	0,00	1					
SiteDAOImpl	14	7	1	0	7,14	0,00	1					
SchedulerFactory	2	1	1	0	0,00	0,00	1					
BaseWorker-TaskImpl	3	4	1	0	33,33	0,00	1					
WorkerThread	9	7	1	0	11,11	0,00	1					
ThrottleFactory	1	1	1	0	100,00	0,00	1					
DistributedLoad-ThrottleProvider	4	1	1	0	25,00	0,00	1					
SimultaneousUsers-ThrottleProvider	5	1	1	0	20,00	0,00	1					
ConsolePlugin	10	7	1	0	10,00	0,00	1					
DiskWriterPlugin	11	12	1	0	9,09	0,00	1					
FileWriterPlugin	9	8	1	0	11,11	0,00	1					
StatusBasedFile-WriterPlugin	6	35	1	0	16,67	0,00	1					
VelocityPlugin	23	9	1	0	4,35	0,00	1					
MaxNumberOfURL-ParamsRule	3	2	1	0	33,33	0,00	1					
MaxResourcesPer-SiteRule	3	2	1	0	33,33	0,00	1					
LogFactory	4	6	0	0	0,00	0,00	0					
SystemOutLogImpl	0	21	0	0	0,00	0,00	0					
Log	0	18	0	0	0,00	0,00	0					
LogAspect	0	236	0	0	0,00	0,00	0	236	0	236	0,00	1,00
TOTAL	**211**	**535**	**34**	**0**	**16,11**	**0,00**	**-**	**236**	**0**	**236**	**0,00**	**1,00**

Métricas		
VS	NOAC	NOAC%
252	34	13,49

Tabela 7.15 – Valores das Métricas para o Modelo de Classes OA da Aplicação *JSpider* Obtido com o Auxílio das Refatorações Prospotas.

Componentes	Métricas											
	NOA	NOO	CA	CO	NOA%	NOO%	NOSC	NOP	NOAP	NORP	NOAP%	NORP/
EventDispa-tcherImpl	6	4	0	0	0,00	0,00	0					
AgentImpl	4	17	0	0	0,00	0,00	0					
PluginFactory	1	2	0	0	0,00	0,00	0					
PluginIns-tantiator	0	5	0	0	0,00	0,00	0					
SpiderContext-Impl	16	21	0	0	0,00	0,00	0					
SpiderImpl	3	2	0	0	0,00	0,00	0					
RuleFactory	2	5	0	0	0,00	0,00	0					
StorageFactory	0	1	0	0	0,00	0,00	0					
CookieDAOImpl	2	3	0	0	0,00	0,00	0					
DecisionDAOImpl	2	5	0	0	0,00	0,00	0					
EMailAddress-DAOImpl	2	5	0	0	0,00	0,00	0					
ResourceDAOImpl	4	21	0	0	0,00	0,00	0					
SiteDAOImpl	3	5	0	0	0,00	0,00	0					
ContentDAOImpl	2	3	0	0	0,00	0,00	0					
CookieDAOImpl	3	4	0	0	0,00	0,00	0					
DBUtil	5	9	0	0	0,00	0,00	0					
DecisionDAOImpl	10	7	0	0	0,00	0,00	0					
EMailAddress-DAOImpl	2	7	0	0	0,00	0,00	0					
FolderDAOImpl	6	8	0	0	0,00	0,00	0					
ResourceDAOImpl	11	23	0	0	0,00	0,00	0					
SiteDAOImpl	13	7	0	0	0,00	0,00	0					
SchedulerFactory	1	1	0	0	0,00	0,00	0					
BaseWorker-TaskImpl	2	4	0	0	0,00	0,00	0					
WorkerThread	8	7	0	0	0,00	0,00	0					
ThrottleFactory	0	1	0	0	0,00	0,00	0					
DistributedLoad-ThrottleProvider	3	1	0	0	0,00	0,00	0					
SimultaneousUsers-ThrottleProvider	4	1	0	0	0,00	0,00	0					
ConsolePlugin	9	7	0	0	0,00	0,00	0					
DiskWriterPlugin	10	12	0	0	0,00	0,00	0					
FileWriterPlugin	8	8	0	0	0,00	0,00	0					
StatusBasedFile-WriterPlugin	5	35	0	0	0,00	0,00	0					
VelocityPlugin	22	9	0	0	0,00	0,00	0					
MaxNumberOfURL-ParamsRule	2	2	0	0	0,00	0,00	0					
MaxResourcesPer-SiteRule	2	2	0	0	0,00	0,00	0					
LogFactory	3	6	0	0	0,00	0,00	0					
SystemOutLogImpl	0	21	0	0	0,00	0,00	0					
Log	0	18	0	0	0,00	0,00	0					
LoggingAspect	1	2	0	0	0,00	0,00	0	2	0	0	0,00	0,00
Net_Javacoding_-Jspider_Api_-Event_Aspect	0	2	0	0	0,00	0,00	0	2	3	0	0,00	1,5
Net_Javacoding_-Jspider_Api_-Core_Aspect	0	2	0	0	0,00	0,00	0	2	72	0	0,00	36
Net_Javacoding_-Jspider_Mod_-Plugin_Aspect	0	2	0	0	0,00	0,00	0	2	15	0	0,00	7,5
Net_Javacoding_-Jspider_Mod_-Rule_Aspect	0	2	0	0	0,00	0,00	0	2	2	0	0,00	1
TOTAL	**177**	**309**	**0**	**0**	**15,71**	**0,00**	**-**	**10**	**92**	**0**	**0,00**	**1,00**

Métricas		
VS	NOAC	NOAC%
255	0	0,00

De acordo com a estratégia escolhida para modularização do interesse de *logging* da aplicação *JSpider* criou-se um aspecto concreto para cada conjunto de pacotes selecionado pelo Engenheiro de Software. Por exemplo, para todas as classes embutidas nos pacotes pertencentes à hierarquia `net.javacoding.jspider.api.core` criou-se um único aspecto denominado `Net_Javacoding_Jspider_Api_Core_Aspect`. Essa estratégia foi utilizada, pois há grande quantidade de classes afetadas pelo interesse de *logging* nessa aplicação e, além disso, essa estratégia possui como vantagem a redução da quantidade de novos componentes (aspectos) introduzidos na aplicação legada. Nesse caso, apenas 5 novos componentes foram criados para modularização do interesse de *logging*. Por outro lado, esse tipo de estratégia traz como limitação o fato de que em alguns aspectos, a proporção do número de métodos relacionados a um determinado conjunto de junção pode ser alta, como acontece com o aspecto `Net_Javacoding_Jspider_Api_Core_Aspect`. Nesse aspecto a proporção de métodos selecionados por conjuntos de junção existentes é de 36 para 1.

A solução obtida com o auxílio das refatorações propostas neste trabalho foi capaz de modularizar completamente o interesse de *logging*, tornando os módulos não relacionados a esse interesse livres de sua interferência. Além disso, tal solução utiliza recursos de generalização/especialização por meio de aspectos abstratos o que pode facilitar a reutilização, manutenção e evolução dos módulos de negócio da aplicação, bem como de seu mecanismo de *logging*.

Na Tabela 7.16 e Tabela 7.17 são apresentados, respectivamente, os valores das métricas de modularização e adequação às boas práticas de projeto OA para modularização do interesse de gerenciamento de transações da aplicação *JAccounting*, para a solução proposta por Binkley *et al.* (2006) e para a solução obtida a partir da aplicação das refatorações propostas neste trabalho.

De acordo com os valores das métricas apresentadas na Tabela 7.16 e Tabela 7.17, as duas soluções para modularização do interesse de gerenciamento de transações da aplicação *JAccounting* foram capazes de remover a interferência desse interesse dos demais módulos da aplicação. A principal diferença está na quantidade de novos componentes criados em cada solução: i) a solução proposta por Binkley *et al.* Possui apenas um aspecto denominado `Transaction` que encapsula a lógica de gerenciamento de transações para toda aplicação; e ii) a solução obtida com o auxílio das refatorações propostas neste trabalho criou 8

novos aspectos. Isso ocorreu, pois na solução (ii) utilizou-se conceitos de generalização/especialização de aspectos que podem tornar a aplicação OA mais flexível no sentido de ser mais fácil de reutilizá-la e evoluí-la.

Tabela 7.16 – Valores das Métricas para Modelo de Classes OA da Aplicação *JAccounting* Obtido a partir da Solução Proposta por Binkley *et al.* (2006).

Componentes	Métricas											
	NOA	NOO	CA	CO	NOA%	NOO%	NOSC	NOP	NOAP	NORP	NOAP%	NORP/
ChargePage	11	23	0	0	0,00	0,00	0					
CompanySelect	0	1	0	0	0,00	0,00	0					
CustomerDetails	2	5	0	0	0,00	0,00	0					
CustomerForm	3	5	0	0	0,00	0,00	0					
InvoicePage	4	9	0	0	0,00	0,00	0					
PaymentPage	4	10	0	0	0,00	0,00	0					
ProductsPage	2	5	0	0	0,00	0,00	0					
RecurrencePage	4	9	0	0	0,00	0,00	0					
Statement	2	5	0	0	0,00	0,00	0					
Base	14	13	0	0	0,00	0,00	0					
Transaction	1	4	0	0	0,00	0,00	0	4	0	8	0,00	2,00
TOTAL	**47**	**89**	**0**	**0**	**0,00**	**0,00**	**-**	**4**	**0**	**8**	**0,00**	**2,00**

Métricas		
VS	NOAC	NOAC%
86	0	0,00

Tabela 7.17 – Valores das Métricas para o Modelo de Classes OA da Aplicação *JAccounting* Obtido com o Auxílio das Refatorações Propostas.

Componentes	Métricas											
	NOA	NOO	CA	CO	NOA%	NOO%	NOSC	NOP	NOAP	NORP	NOAP%	NORP/
ChargePage	11	23	0	0	0,00	0,00	0					
CompanySelect	0	1	0	0	0,00	0,00	0					
CustomerDetails	2	5	0	0	0,00	0,00	0					
CustomerForm	3	5	0	0	0,00	0,00	0					
InvoicePage	4	9	0	0	0,00	0,00	0					
PaymentPage	4	10	0	0	0,00	0,00	0					
ProductsPage	2	5	0	0	0,00	0,00	0					
RecurrencePage	4	9	0	0	0,00	0,00	0					
Statement	2	5	0	0	0,00	0,00	0					
Base	13	12	0	0	0,00	0,00	0					
TransAspect	1	2	0	0	0,00	0,00	0	1	0	0	0,00	0,00
ProductsPage-TransAspect	0	0	0	0	0,00	0,00	0	1	0	1	0,00	1,00
CustomerTransAspect	0	0	0	0	0,00	0,00	0	1	0	2	0,00	2,00
ChargePage-TransApect	0	0	0	0	0,00	0,00	0	1	0	1	0,00	1,00
CompanySelect-TransApect	0	0	0	0	0,00	0,00	0	1	0	1	0,00	1,00
PaymentTransAspect	0	0	0	0	0,00	0,00	0	1	0	2	0,00	2,00
RecurrencePage-TransAspect	0	0	0	0	0,00	0,00	0	1	0	1	0,00	1,00
StatementTransAspect	0	0	0	0	0,00	0,00	0	1	0	1	0,00	1,00
TOTAL	**46**	**86**	**0**	**0**	**0,00**	**0,00**	**-**	**8**	**0**	**9**	**0,00**	**1,12**

Métricas		
VS	NOAC	NOAC%
93	0	0,00

Por exemplo, para modularizar o interesse de gerenciamento de transações de uma nova classe `A`, bastaria que o Engenheiro de Software criasse um novo aspecto que herdasse do aspecto `TransAspect` e concretizasse o conjunto de junção abstrato `transactionMethods` de acordo com os pontos da classe `A` que fossem necessários obter. Com essa estratégia, o Engenheiro de Software não precisa criar um novo conjunto de junção e associá-lo aos adendos necessários, como ocorreria na solução proposta por Binkley *et al.*.

7.5 Considerações Finais

Neste capítulo realizou-se a avaliação das soluções de modularização obtidas com o auxílio das refatorações elaboradas neste trabalho. Para isso, comparou-se os modelos de classes OO e OA apresentados no Capítulo 6 com os modelos obtidos a partir de soluções criadas por terceiros.

A partir das avaliações, observou-se que em alguns casos as soluções obtidas com o auxílio das refatorações que foram propostas neste trabalho apresentaram melhores resultados no quesito modularização. Isso ocorreu para o interesse relacionado ao padrão Singleton da aplicação Health Watcher e para o interesse de logging da aplicação JSpider. Notou-se ainda que, para alguns casos, as soluções propostas por outros pesquisadores apresentaram uma quantidade menor de novos componentes inseridos na aplicação legada. Esse fato ocorre, pois como o processo de criação de aspectos é automático, muitas vezes a estratégia para criação desses aspecto gera vários novos elementos. Por exemplo, no MoBRe o Engenheiro de Software pode escolher criar um aspecto por classe afetada ou um aspecto por pacote de classes afetadas. Sendo assim, dependendo do porte da aplicação e da quantidade de elementos afetados, o número de aspectos criados pode ser significativo. Essa é uma das limitações do MoBRe que precisam ser melhoradas para uma próxima versão da ferramentas.

Entre todas as refatorações realizadas com a proposta aqui apresentada, apenas uma (refatoração do interesse relacionado ao padrão Observer da aplicação Health Watcher) não obteve bons resultados. Dessa forma, há indícios de que a aplicação de refatorações com base em modelos de classes anotados é uma

alternativa a ser considerada pelos engenheiros da aplicação para modularização de interesses transversais em software OO.

Capítulo 8

CONSIDERAÇÕES FINAIS

8.1 Introdução

Reengenharia de software não é um processo trivial, especialmente quando se deve realizar a transformação de um software OO para um software OA. Isso ocorre, dentre outros fatores, em consequência da existência de novos conceitos e abstrações, como aspectos, conjuntos de junção, entre outros.

Com o intuito de minimizar tal problema, apresentou-se um conjunto de nove refatorações para obtenção de modelos de classe OA a partir de modelos de classe OO anotados que visam diminuir o *gap* semântico entre um sistema OO e os conceitos da POA. Três dessas refatorações (*R-1*, *R-2* e *R-3*) são denominadas genéricas, pois podem ser aplicadas a qualquer tipo de interesse. Desta forma, elas são aplicadas de acordo com o cenário de entrelaçamento/espalhamento dos interesses existentes na aplicação. As outras seis refatorações (*R-Transaction*, *R-Sync*, *R-Connection*, *R-Logging*, *R-Observer* e *R-Singleton*) são responsáveis pela refatorações dos interesses transversais de persistência, *logging* e dos padrões de projeto *Observer* e *Singleton*. Essas refatorações são denominadas específicas, pois foram elaboradas para refatoração de um determinado tipo de interesse transversal. Isso é necessário, pois somente com a aplicação das refatorações genéricas, pode ocorrer que alguns interesses não sejam modularizados completamente, necessitando assim de uma refatoração com passos específicos para tratar esse interesse.

A ideia de se utilizar modelos de classes OO anotados para construção de modelos OA foi adotada, primeiro para reduzir o *gap* semântico comentado anteriormente e pelos benefícios que esses modelos anotados podem trazer, tais como: i) ajudam a visualizar possibilidades de modularização sem necessidade de utilizar OA; ii) provêm maior nível de abstração auxiliando no entendimento do software; iii) servem de documentação para o software OA e para o software legado; iv) permitem a geração de código de maior qualidade, uma vez que os vícios de codificação podem ser repensados em nível de modelo; e v) são independentes de linguagem de programação.

Para dar apoio à aplicação automatizada das refatorações genéricas e específicas, além das refatorações propostas, desenvolveu-se um *plug-in do Eclipse,* denominado MoBRe. Esse *plug-in* recebe como entrada um modelo de classes OO anotado com indícios de interesses transversais e permite a construção, de forma iterativa e incremental, de um modelo OA no qual os interesses da aplicação encontram-se modularizados. As nove refatorações propostas neste trabalho foram implementadas no MoBRe, porém, somente elas não são suficientes para tratar qualquer tipo de interesse transversal. Por isso, desenvolveu-se um módulo de extensão no MoBRe para garantir-lhe a capacidade de ser evoluído com o acréscimo de novos tipos de refatorações implementadas pelo usuário. Ou seja, um pesquisador que trabalha com modularização do interesse de segurança pode implementar sua própria refatoração para esse interesse no MoBRe, com base em sua experiência, e gerar uma nova versão do *plug-in* para seu uso e/ou distribuição. Além disso, com MoBRe o usuário pode implementar novos tipos de refatorações para um interesse já contemplado no *plug-in* (por exemplo, persistência), podendo assim, comparar duas ou mais soluções de modularização para um mesmo tipo de interesse.

Com base nas refatorações propostas e no *plug-in* MoBRe, realizou-se um estudo de caso com três aplicações OO desenvolvidas por outros pesquisadores: *Health Watcher* (Greenwood *et al*., 2007), *JSpider* (JSpider, 2011) e *JAccounting* (JAccounting, 2011). No estudo de caso apresentado neste trabalho, as refatorações desenvolvidas foram aplicadas aos modelos de classes anotados dessas aplicações para alguns tipos de interesses (*logging*, persistência e padrões *Observer* e *Singleton*). Os modelos de classes anotados foram obtidos com o auxílio dos *plug-ins* ComSCId e DMAsp, que foram apresentados na Seção 5.2, e de informações

disponibilizadas na literatura pelos autores dessas aplicações. Por exemplo, para determinar os interesses existentes nas aplicações Health Watcher, JSpider e JAccounting, utilizou-se a versão OA dessas aplicações desenvolvidas pelos pesquisadores que desenvolveram também a versão OO.

A partir do estudo de caso conduzido, os resultados obtidos foram avaliados com o auxílio de métricas OA. Percebeu-se que a utilização das refatorações propostas, neste trabalho, pode permitir a obtenção de modelos OA de qualidade, pois: i) fornecem um guia passo a passo para modularização de determinados CCC. Isso pode evitar que engenheiros de software escolham estratégias inadequadas para modularização de interesses transversais; e ii) as diretrizes aqui propostas foram elaboradas considerando boas práticas de projeto OA. Assim, utilização dessas diretrizes pode levar a geração de modelos OA, e consequentemente, de códigos de melhor qualidade, por exemplo, livre da presença de *bad smells*.

8.2 Limitações da Proposta

Uma das limitações deste trabalho e do apoio computacional desenvolvido é permitir a modularização de apenas quatro tipos de interesses distintos: persistência, *logging* e os padrões de projeto *Observer* e *Singleton*. Entretanto, essa limitação é minimizada pela existência do módulo de extensão do MoBRe que permite o desenvolvimento de refatorações para outros tipos de interesses.

O processo de aplicação das refatorações em uma aplicação de médio e grande porte pode consumir quantidade de tempo considerável, dependendo do número de interesses transversais existentes na aplicação e da experiência do Engenheiro de Software. Para as aplicações utilizadas no estudo de caso, o tempo não foi significativo em conseqüência de alguns fatores: i) o estudo de caso foi conduzido pelo desenvolvedor das refatorações e do *plug-in* MoBRe, o que pode ter influenciado na diminuição do tempo de execução do estudo de caso; ii) as aplicações utilizadas no estudo de caso não podem ser consideradas de grande porte. Muito provavelmente para aplicações de grande porte o tempo para aplicação das refatorações seria maior. Salienta-se que o tempo para aplicação das refatorações aqui mencionados corresponde ao tempo gasto pelo usuário para

determinar quais refatorações devem ser aplicadas. Quanto ao tempo de execução da ferramenta, não foi realizada uma análise mais específica, entretanto, como as refatorações são aplicadas sobre modelos de classes, de modo incremental, o tempo de execução do MoBRe não apresentou-se como um fator critico para as aplicações utilizadas no estudo de caso.

Ainda com relação ao *plug-in* MoBRe sabe-se que durante a aplicação de algumas refatorações, por exemplo, a *R-Logging* (Seção 5.4) o Engenheiro de Software pode escolher uma dentre as estratégias para criação dos aspectos: i) criar um aspecto para cada classe afetada; ou ii) criar um aspecto para cada pacote de classes afetadas. Essas estratégias podem não ser adequadas para sistemas de médio e grande porte que possuem muitos elementos afetados, pois produzirão um grande número de novos componentes (aspectos) no software legado. Uma solução mais adequada seria deixar que o Engenheiro de Software decidisse quais classes seriam afetadas por um determinado aspecto, a fim de os aspectos fossem criados por responsabilidade. Por exemplo, o Engenheiro de Software pode criar um aspecto que afete todas as classes com responsabilidade de tratar pedidos de venda mesmo que elas estejam em pacotes diferentes.

A falta de uma avaliação por meio de um experimento controlado do apoio computacional é outra limitação deste trabalho. A dificuldade em conduzir o experimento pode ser atribuída ao escasso tempo disponível dos integrantes dos grupos e quanto à obtenção de participantes para realização do experimento.

8.3 Contribuições e Trabalhos Futuros

Pode-se citar como contribuições deste trabalho:

- As definições de refatorações baseadas em modelos para interesses transversais bem conhecidos pela comunidade científica é um passo em busca de um processo reengenharia de software OO para OA que leve em consideração as diferenças semânticas entre essas duas tecnologias.
- A construção do *plug-in* MoBRe permitiu a avaliação da viabilidade das refatorações propostas e de apoios computacionais que automatizem a

aplicação das mesmas. Os próximos trabalhos e experimentos relacionados à modularização de interssses transversais podem usar o MoBRe, devido sua característica de ser extensível.

- MoBRe apresenta a vantagem de realizar refatorações em modelos de classes OO anotados visando a modularização de interesses transversais. Outro ponto relevante é a integração dessa ferramenta com outras para identificação de interesses transversais e para a realização de engenheria reversa. Em geral, as ferramentas para refatoração de interesses transversais aplicam refatorações apenas em nível de código e atendem apenas à função de refatoração dos interesses. Dessa forma, geralmente, a identificação e visualização dos interesses transversais eram realizados sem o apoio de ferramentas.
- O MoBRe por ser um *plug-in* do ambiente Eclipse permite a utilização conjunta de outros recursos existentes nesse ambiente, como é o caso do framework GMF (*Graphical Modeling Framework*) capaz de exibir modelos de classes da UML. Além disso, o usuário pode realizar as tarefas relacionadas à modularização dos interesses transversais em um único ambiente, podendo assim, diminuir a curva de aprendizagem por parte do Engenheiro de Software com relação à utilização dos *plug-ins*.

Como trabalhos futuros pretende-se:

- i) averiguar, por meio de um experimento controlado, se o modelo de projeto OA gerado com a utilização das diretrizes de refatoração apresenta melhores benefícios do que um projeto OA que foi obtido somente com base em refatorações de código;
- ii) elaborar novas refatorações específicas para outros tipos de interesses, como os de segurança, de tratamento de exceção, entre outros; e
- iii) criar um módulo para detecção dos impactos que uma refatoração pode causar em um determinado sistema antes de serem aplicadas.

REFERÊNCIAS

ALDAWUD, O.; ELRAD, T.; BADER, A. UML profile for aspect-oriented software development. In: INTERNATIONAL WORKSHOP ON ASPECT-ORIENTED MODELING, Boston, USA. 2003. **Proceedings...** Boston: Aspect-Oriented Software Development – AOSD, 2003. p. 1-16.

AKSIT, M.; BERGMANS, L.; VURAL, S. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, London, UK, 1992. **Proceedings...** London: Springer-Verlag, 1992. p. 372-395.

AOSDbr – COMUNIDADE BRASILEIRA DE DESENVOLVIMENTO DE SOFTWARE ORIENTADO A ASPECTOS. Terminologia em português para orientação a aspectos. Brasília, Brasil, 2008. Disponível em: <http://twiki.dcc.ufba.br/bin/view/AOSDbr/TermosEmPortugues>. Acessado em: Março de 2011.

BINKLEY, D. *et al.* Automated Refactoring of Object Oriented Code into Aspects. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE - ICSM. Budapest, Hungary, 2005. **Proceedings…** Budapest: ICSM 2005. p. 27-36.

BINKLEY, D. *et al.* Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects. **IEEE Transactions on Software Engineering**, v. 32, n. 9, p. 698-717, setembro 2006.

BOGER, M.; STURM, T.; FRAGEMANN, P. Refactoring Browser for UML. In: REVISED PAPERS FROM THE INTERNATIONAL CONFERENCE NETOBJECTDAYS ON OBJECTS, COMPONENTS, ARCHITECTURES, SERVICES, AND APPLICATIONS FOR A NETWORKED WORLD – NODe'02, London, UK, 2002. **Proceedings…** London: Springer-Verlag, 2002. p. 366-377.

BÖLLERT, K. On Weaving Aspects. MOREIRA, A. M. D.; DEMEYER, S. (Ed.) In: WORKSHOP ON OBJECT-ORIENTED TECHNOLOGY, London, UK, 1999. **Proceedings...** London: Workshop on Object-Oriented Technology, p. 301-302.

BOUNOUR, N.; GHOUL, S; ATIL F. A Comparative Classification of Aspect Mining Approaches. **Journal of Computer Science**, v. 2, n. 4, p. 322-325, 2006.

CECCATO, M. *et al.* Applying and combining three different aspect Mining Techniques. **Software Quality Control**, v. 14, n. 3, p. 209-231, setembro 2006.

CECCATO, M.; TONELLA, P. Measuring the Effects of Software Aspectization. In: WORKSHOP ON ASPECT REVERSE ENGINEERING – WARE 2004, Delft, The Netherlands, 2004. **Proceedings...** Delft: WARE 2004. p. 1-5.

CHAVEZ, C.; LUCENA, C. A metamodel for aspect-oriented modeling. In: INTERNATIONAL WORKSHOP ON ASPECT-ORIENTED MODELING WITH UML, Enschede, The Netherlands, 2002. **Proceedings…** Enschede: Aspect-Oriented Software Development – AOSD, 2002. p. 1-6.

CLARKE, S.; BANIASSAD, E. Aspect Oriented Analysis and Design – The Theme Approach. Addison-Wesley, First Edition, 2005. p. 400.

COSTA, H. A. X. *et al.* Recovering Class Models Stereotyped With Crosscutting Concerns. In: SESSION TOOL OF WORKING CONFERENCE ON REVERSE ENGINEERING (WCRE), Lille, France, 2009. **Proceedings...** Lille: Working Conference on Reverse Engineering (WCRE), 2009. p. 311-312.

DUCASSE, S.; GÎRBA, T.; KUHN, A. Distribution Map. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE - ICSM, Dublin, Ireland, 2006. **Proceedings…** Dublin: ICSM 2006. p 203-212.

DIJKSTRA, E. W. A Discipline of Programming. Prentice-Hall. First Edition, 1976. p. 217.

ECLIPSE. Disponível em: http://www.eclipse.org. Acessado em: Março de 2011.

ELRAD, T. *et al.* Discussing Aspects of AOP. **Communications of the ACM**, v. 44, n. 10, p. 33-38, outubro 2001a.

ELRAD, T.; FILMAN, R. E.; BADER, A. Aspect-Oriented Programming: Introduction. **Communications of the ACM**, v. 44, n. 10, p. 29-32, outubro 2001b.

ELSSAMADISY, A.; SCHALLIOL, G. Recognizing and responding to bad smells in extreme programming. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING – ICSE, Orlando, USA, 2002. **Proceedings…** Orlando: ICSE 2002, p 617-622.

EVERMANN, J. A MetaLevel Specification and Profile for AspectJ in UML. In: ASPECT ORIENTED SOFTWARE DEVELOPMENT – AOSD, Wellington, New Zealand, 2007. **Proceedings…** Wellington 2007. p. 21-27.

FIGUEIREDO E. C, *et al*. Applying and Evaluating Concern-Sensitive Design Heuristics. **Proceedings…** 23rd Brazilian Symposium on Software Engineering (SBES). Fortaleza, 2009a.

FIGUEIREDO, E. C. *et al*. Crosscutting Patterns and Design Stability: An Exploratory Analysis. **Proceedings…** 17th International Conference on Program Comprehension (ICPC), 2009, pp. 138-147.

FRANCE, R. B. *et al.* Aspect-oriented approach to early design modelling. **IEEE Software**, v. 151. n. 4, p. 173–186. 2004.

FOWLER, M. *et al.* Refactoring: improving the design of existing code. Addison-Wesley Professional. First Edition, 1999. p. 464.

GAMMA, E. *et al.* Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, First Edition, 1995. p. 416.

GARCIA, V. C. *et al.* Reengenharia de sistemas orientados a objetos através de transformações e mineração de aspectos. In: INTERNATIONAL INFORMATION AND TELECOMMUNICATION TECHNOLOGIES SYMPOSIUM - I2TS, São Carlos, Brasil, 2004. **Anais...** São Carlos: I2TS 2004.

GENSSLER T. *et al.* On the Computer Aided Introduction of Design Patterns into Object-Oriented Systems. In: TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES - TOOLS, Beijing, China, 1998. **Proceedings…** Beijing: TOOLS 1998. p. 258-267.

GRADECK, J.; LESIECKI, N. Mastering AspectJ: Aspect-Oriented Programming in Java. Wiley, First Edition, 2003. p. 456.

GREENWOOD, P. *et al*. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. **Proceedings…** Object-Oriented Programming In ECOOP 2007 – Object-Oriented Programming, Vol. 4609 (2007), pp. 176-200.

HANENBERG, S.; OBERSCHULTE, C.; UNLAND, R. Refactoring of Aspect-Oriented Software. In: NET OBJECTDAYS CONFERENCE - NODE, Thuringia, Germany, 2003. **Proceedings...** Thuringia: NODE. p. 1-17.

HANNEMANN, J.; MURPHY, G. C.; KICZALES, G. Role-based refactoring of crosscutting concerns. In: ASPECT-ORIENTED SOFTWARE DEVELOPMENT - AOSD, New York, USA, 2005. **Proceedings...** New York: AOSD 2005. p.135–146.

HANNEMANN, J. Aspect-Oriented Refactoring: classification and challenges. In: INTERNATIONAL WORKSHOP ON LINKING ASPECT TECHNOLOGY AND EVOLUTION, Bonn, Germany, 2006. **Proceedings...** Bonn: Aspect-Oriented Software Development – AOSD 2006. p. 1-5.

HANNEMANN, J.; G. KICZALES, Overcoming the prevalent decomposition in legacy code. In: WORKSHOP ON ADVANCED SEPARATION OF CONCERNS, Toronto, Canada, 2001. **Proceedings…** International Conference on Software Engineering – ICSE 2001. p. 1-5.

HANNEMANN, J.; G. KICZALES, Design pattern implementation in Java and AspectJ, **ACM SIGPLAN Notices**. v. 37, n.11, p. 161-173, novembro 2002.

IWAMOTO, M.; ZHAO, J. Refactoring Aspect-Oriented Programs. In: INTERNATIONAL WORKSHOP ON ASPECT-ORIENTED MODELING WITH UML, Boston, USA, 2003. **Proceedings…** Boston: Aspect-Oriented Software Development – AOSD 2003. p. 1-7.

JSPIDER. Disponível em: http://j-spider.sourceforge.net. Acessado em: Março de 2011.

JACCOUNTING. Disponível em: https://jaccounting.dev.java.net. Acessado em: Março de 2011.

JANZEN, D.; VOLDER, K. D. Navigating and querying code without getting lost. In: ASPECT-ORIENTED SOFTWARE DEVELOPMENT - AOSD, Boston, USA, 2003. **Proceedings…** Boston: AOSD 2003, p. 178-187.

JUNIOR, V. G.; WINCK, D. V. AspectJ: Programação Orientada a Aspectos com Java. Novatec, Primeira Edição, 2006. p. 228.

KAWAKAMI, D. Um Apoio Computacional para auxiliar a Reengenharia de Sistemas Legados Java para AspectJ. Dissertação de Mestrado em Ciência da Computação. UFSCar - São Carlos, Brasil. 2007.

KAWAKAMI, D.; PENTEADO R. A. D. Apoio Computacional para Refatoração de Aspectos. In: WORKSHOP BRASILEIRO DE SOFTWARE ORIENTADO A ASPECTOS (WASP), Uberlândia, Brasil, 2005. **Anais…** Uberlândia: WASP 2005. p. 1-8.

KELLENS, A.; MENS, K.; TONELLA, P. A survey of automated code-level aspect mining techniques. **LNCS Transactions on Aspect-Oriented Software Development**, p. 143-162. 2007.

KICZALES, G. *et al.* An Overview of AspectJ. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING - ECOOP, Budapest, Hungary, 2001. **Proceedings…** Budapest: ECOOP 2001. p. 327-353.

KICZALES, G. *et al.* Aspect-Oriented Programming. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING - ECOOP, Jyväskylä, Finland, 1997. **Proceedings…** Jyväskylä: ECOOP 1997. p. 220-242.

LADDAD, R. AspectJ in Action: Pratical Aspect-Oriented Programming. Greenwich Mannig Publication Co. Second Edition, 2003. p. 512.

LADDAD, R. Aspect Oriented Refactoring. Addison-Wesley Professional. First Edition, 2006. p. 352.

LANZA, M.; R. MARINESCU; S. DUCASSE, Object-Oriented Metrics in Practice. Springer. First Edition, 2006. p. 220.

LIEBERHERR, K.; ORLEANS, D.; OVLINGER, J. Aspect-Oriented Programming with Adaptive Methods. **Communications of the ACM**, v. 44, n. 10, p. 39-41, outubro 2001.

MARIN, M.; MOONEN, L.; VAN DEURSEN, A. An Approach to Aspect Refactoring based on Crosscutting Concern Types. **SIGSOFT Software Engineering Notes**, v.30, n.4, p.1–5, 2005.

MARIN, M.; VAN DEURSEN, A.; MOONEN, L. Identifying aspects using fan-in analysis. In: WORKING CONFERENCE ON REVERSE ENGINEERING – WCRE, Delft, The Netherlands. **Proceedings…** Delft: WCRE. p. 132-141, 2004.

MARK BASCH, A. S. Incorporating aspects into the UML. In Workshop on Aspect-oriented Modeling, Boston, USA, 2003. **Proceedings…** Boston: Aspect-Oriented Software Development – AOSD, 2003. p. 16-32.

MENS, T.; TOURWE, T. A survey of software refactoring. **IEEE Transactions on Software Engineering**, v. 30, n. 2, p. 126–139. 2004.

MONTEIRO, M. P.; FERNANDES, J. M. L. Towards a catalogue of refactorings and code smells for aspectj. **Transactions on Aspect Oriented Software Development (TAOSD) - Lecture Notes in Computer Science**, n.3880, p.214–258, 2006.

MURPHY-HILL, E.; PARNIN, C.; BLACK, A. P. HowWe Refactor, and HowWe Know It. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING – ICSE, New York, USA, 2009. **Proceedings...** New York: ICSE 2009.

NASSAU, M.; VALENTE, M. T. Transformações de Código para Extração de Aspectos. In: LATIN AMERICAN WORKSHOP ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT - LA-WASP, João Pessoa, Brasil, 2007. **Anais...** LA-WASP 2007.

OMG - OBJECT MANAGEMENT GROUP. Disponível em: http://www.omg.org. Acessado em: Março de 2011.

OPDYKE, W. F. Refactoring Object-Oriented Frameworks. Tese de Doutorado em Ciência da Computação. Illinois, USA. 1992.

OSSHER, H.; TARR, P. Multi-Dimentional Separation of Concerns in Hyperspace. In: ASPECT-ORIENTED PROGRAMMING WORKSHOP. Lisboa, Portugal, 1999. **Proceedings…** Lisboa: European Conference on Object-Oriented Programmming - ECOOP.

PARREIRA JÚNIOR, P. A. *et al.* ComSCId - Um Apoio Computacional Customizável para a Identificação de Interesses Transversais em Sistemas Legados Orientados a Objetos. In: WORKSHOP DE MANUTENÇÃO DE SOFTWARE MODERNA - WMSWM, Belém, Brasil, 2010. **Anais...** Belém: Simpósio Brasileiro de Qualidade de Software - SBQS, 2010a.

PARREIRA JÚNIOR, P. A. *et al.* Uma Abordagem Iterativa para Identificação de Interesses Transversais com o Apoio de Modelos de Classes Anotados. In: LATIN AMERICAN WORKSHOP ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT - LA-WASP, Salvador, Brasil, 2010. **Anais...** LA-WASP. I Congresso Brasileiro de Software: Teoria e Prática (CBSoft), 2010b.

PAWLAK, R.; *et al.* A UML notation for aspect-oriented software design. In INTERNATIONAL WORKSHOP ON ASPECT-ORIENTED MODELING WITH UML, Enschede, The Netherlands, 2002. **Proceedings…** Enschede: Aspect-Oriented Software Development – AOSD, 2002.

PIVETA, E. K. *et al.* Detecting Bad Smells in AspectJ. **Journal of Universal Computer Science**, v.12, n.7, p.811–827, 2006.

PIVETA, E. K. *et al.* Avoiding Bad Smells in Aspect-Oriented Software. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING - SEKE, Boston, USA, 2007. Proceedings... Boston: SEKE 2007. p.81–87.

PRESSMAN, R. S. Software Engineering: A Practitioner's Approach. McGraw-Hill. 7th Edition, 2010. p. 928.

RAMOS, R. A. Abordagem Aspecting – Migração de Sistemas OO para Sistemas AO. Dissertação de Mestrado. UFSCar - São Carlos, Brasil. 2004.

ROBILLARD, M. P.; MURPHY, G. C. Representing Concerns in Source Code. **ACM Transactions on Software Engineering and Methodology,** v. 16, n. 1, p. 1-38, 2007.

SANT'ANNA, C. *et al.* On the Modularity of Software Architectures: a concern-driven measurement framework. In: EUROPEAN CONFERENCE ON SOFTWARE ARCHITECTURE - ECSA, Aranjuez, Spain, 2007. **Proceedings...** Aranjuez: ECSA 2007. p.207–224.

SCHAUERHUBER, A. *et al.* A Survey on Aspect-Oriented Modeling Approaches. Business Informatics Group, Institute of Software Technology and Interactive Systems, Vienna University of Technology, 2007. DIsponível em: ftp://ftp.ifs.uni-linz.ac.at/pub/publications/2006/1406.pdf. Acessado em: Março de 2011.

SILVA, B. *et al.* Refactoring of Crosscutting Concerns with Metaphor-Based Heuristics. **Electronic Notes in Theoretical Computer Science (ENTCS)**, vol. 233, p. 105-125, 2009.

SILVA, B. C. Um Método de Refatoração para Modularização de Interesses Transversais, Dissertação de Mestrado. UFRGS - Porto Alegre, RS, 2009.

SOMMERVILLE, I. Software Engineering. Addison-Wesley. 8th Edition. 2006. p. 864.

STEIN, D.; HANENBERG, S.; UNLAND, R. Designing aspect-oriented crosscutting in UML. In: INTERNATIONAL WORKSHOP ON ASPECT-ORIENTED MODELING WITH UML, Enschede, The Netherlands, 2002. **Proceedings…** Enschede: Aspect-Oriented Software Development – AOSD, 2002.

SUZUKI, J.; YAMAMOTO, Y. Extending uml with aspects: Aspect support in the design phase. In: WORKSHOP ON OBJECT-ORIENTED TECHNOLOGY, London, UK, 1999. **Proceedings…** London: Workshop on Object-Oriented Technology . p. 299–300.

TOKUDA, T.; BATORY, D. S. Automated Software Evolution via Design Pattern Transformations. In: INTERNATIONAL SYMPOSIUM ON APPLIED CORPORATE COMPUTING, Texas, USA, 1995. **Proceedings…** Texas: International Symposium on Applied Corporate Computing.

UETANABARA JÚNIOR, J.; CAMARGO, V. V. de. A Preliminary Comparison Framework for Aspect-Oriented Modeling Approaches. In: CONFERENCIA LATINOAMERICANA DE INFORMÁTICA – CLEI, Santa Fe, Argentina, 2008. **Proceedings…** Santa Fe: CLEI 2008.

VAN GORP, P. *et al.* Towards Automating Source Consistent UML Refactorings. STEVENS, P.; WHITTLE, J., BOOCH, G. (Ed.) In: THE UNIFIED MODELING LANGUAGE CONFERENCE – UML, California, USA, 2003. Proceedings… UML 2003. 144-158.

ZHANG, J.; LIN, Y.; GRAY, J. Generic and Domain-Specific Model Refactoring using a Model Transformation Engine, Springer. Chapter 9, 2005. p. 199–218.

ZHANG, C.; JACOBSEN, H. A. A Prism for Research in Software Modularization through Aspect Mining. Technical report, Middleware Systems Research Group, University of Toronto. 2003. Disponível em: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.5698&rep=rep1&type=pdf. Acessado em: Março de 2011.

WHITTLE, J.; JAYARAMAN, P. MATA: A Tool for Aspect-Oriented Modeling based on Graph Transformation. In: WORKSHOP ON ASPECT-ORIENTED MODELING (AOM), Tennesse, USA, 2007. Proceeding… Tennesse: International Conference on Model Driven Engineering Languages and Systems - ODELS 2007.

WILLIAM, G. G.; KATO, Y.; YUAN, J.J. Aspect browser: Tool support for Managing Dispersed Aspects. Technical Report CS99-0640, Department of computer Science and Engineering, University of California, San Diego. Disponível em: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.188&rep=rep1&type=pdf. Acessado em: Março de 2011.

Apêndice A

CÓDIGOS FONTE

Código fonte correspondente ao pseudocódigo para modularização do interesse “ConcernX” apresentado na Seção 5.5.2.

```
public boolean refExample() {
 if (project.getClasses() != null) {
 Class cl = null; CrossCuttingConcern cc = null; Aspect aspect = null;
 Pointcut pcAffectedMethods = null;

 // 1. Captura a classe “cl” cujo interesse “ConcernX” é primário.
 List<Class> lsClasses =
  project.getClassesWithStereotypeId("Pri_ConcernX");
 if (lsClasses.size() == 1) {
  cl = lsClasses.get(0);

  // 2. Criar um interesse denominado “ConcernX”.
  cc = new CrossCuttingConcern("ConcernX");

  // 3. Criar um aspecto denominado “clAspect”.
  String aspectName = cl.getName() + "Aspect";
  aspect = new Aspect(aspectName, aspectName);

  // 4. Cria um conjunto de junção chamado “affectedMethods”.
  pcAffectedMethods = new Pointcut("affectedMethods",
  "affectedMethods", new Type("CallPointCut", "CallPointCut"));
 } else return false;

 lsClasses = new ArrayList<Class>(project.getClasses());
 Iterator<Class> itClasses = lsClasses.iterator();
 while(itClasses.hasNext()) {
 Class clAux = itClasses.next();

 // 5. Para cada classe “clAux” cujo interesse secundário seja
   “ConcernX”.
 if (clAux.isConcernExist("Sec_ConcernX")) {
  List<Attribute> lsAttribute = new
  ArrayList<Attribute>(clAux.getAttributes());
  Iterator<Attribute> itAttribute = lsAttribute.iterator();
  while(itAttribute.hasNext()) {
  Attribute clAttr = itAttribute.next();

  // 5.1. Remove a referência à classe “cl” da classe “clAux”.
  if (clAttr.getType().getId().equalsIgnoreCase(cl.getId())) {
```

```
   clAux.removeAttributeById(clAttr);
  }
  }

  List<Operation> lsOperation = new
  ArrayList<Operation>(clAux.getOperations());
  Iterator<Operation> itOperation = lsOperation.iterator();
  while(itOperation.hasNext()) {
  Operation mAux = itOperation.next();

  // 5.2. Para cada método “mAux” que possui chamada ao método
    “method1”
  if (mAux.isConcernExist("Sec_ConcernX")) {

   /* 5.2.1 Adiciona “mAux” ao conjunto de operações do conjunto de
     junção “affectedMethods”. */
   pcAffectedMethods.addOperation(mAux);

   /* 5.2.2 Remove o estereótipo relacionado ao interesse “ConcernX”
     do método “mAux” */
   mAux.removeConcernById("Sec_ConcernX");

   // 5.3 Atualiza o conjunto de estereótipos da classe “clAux”
   clAux.updateListOfConcerns(false);
  }
  }
 }

 /* 6. Se o conjunto de operações do conjunto de junção
   “affectedMethods” é maior que zero, faça: */
 if (pcAffectedMethods.getOperations().size() > 0) {
  /* 6.1 Cria um adendo “adAffectedMethods” do tipo “after” e
    relacione-o ao conjunto de junção “affectedMethods”. */
  Advice adAffectedMethods = new Advice("adAffectedMethods",
  "adAffectedMethods", null, Advice.afterType);
  adAffectedMethods.setPointcut(pcAffectedMethods);

  /* 6.2 Adicione o conjunto de junção “affectedMethods” e o adendo
    “adAffectedMethods” ao aspecto “clAspect” */
  aspect.addAdvice(adAffectedMethods);
  aspect.addPointcut(pcAffectedMethods);
 }

 // 7. Adicione o aspecto criado ao projeto.
 cc.addAspect(aspect);

 // 8. Adicione o interesse “ConcernX” ao projeto.
 project.addCrossCuttingConcern(cc);
 }
}
```