UNIVERSIDADE FEDERAL DE SÃO CARLOS
DEPARTAMENTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

"Apoio Computacional para auxiliar a Reengenharia de Sistemas Legados Java para AspectJ"

Daniel Kawakami

São Carlos – SP
Agosto/2007

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

K22ac

Kawakami, Daniel.
Apoio computacional para auxiliar a reengenharia de sistemas legados Java para AspectJ / Daniel Kawakami. -- São Carlos : UFSCar, 2007.
128 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2007.

1. Engenharia de software. 2. Reengenharia orientada a aspectos. 3. Interesses transversais. 4. Apoio computacional. 5. Sistema legado. 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

"Um Apoio Computacional para auxiliar a Reengenharia de Sistemas Legados Java para AspectJ"

DANIEL KAWAKAMI

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:

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

Prof. Dr. Antonio Francisco do Prado
(DC/UFSCar)

Prof. Dr. Valter Vieira de Camargo
(UNIVEM)

São Carlos
Agosto/2007

Pela proteção divina de Deus, zelando minha paz e saúde,
pelo amor incondicional de minha mãe,
pelo apoio de meus familiares e amigos.

Agradecimentos

A minha mãe, meu pai, minha irmã Luciana, meu cunhado Maurício e minha sobrinha Isabelle pelo amor, afeto, dedicação e atenção, sempre me acompanhando nos momentos felizes e nos difíceis.

A Profª. Rosângela Aparecida Dellosso Penteado pela excelente orientação, dedicação, grande amizade e respeito durante minha graduação e pós-graduação na UFSCar.

Aos professores do Departamento de Computação da UFSCar por transmitirem conhecimentos de valor imensuráveis para minha vida acadêmica e profissional.

Ao Ricardo Ramos pelas suas idéias, sugestões e trabalhos que colaboraram bastante no desenvolvimento do meu projeto de mestrado.

Ao Vinicius Durelli por suas sugestões e por contribuir imensamente com a execução de experimentos.

Aos alunos de pós-graduação da disciplina Projeto e Gerência de Sistemas de Software pela dedicação na realização de estudos de casos.

Aos meus amigos que me acompanham desde o colegial Renato, Lee, Michael, Patrícia, Marcelo, Momente, Karina.

A todos meus animados amigos de turma de graduação (EnC 2K).

Aos meus amigos de São Carlos Éder, Paula, Fabiana, Ariane, Cristiane, Maria, Daltinho, Lulys, Andréia, Sapo, Ramon, Escovar, Sujeirinha, Flauflau, Jorge, Vital, Michael, Tiago, Dugnani, Lucrédio, Maria Elisa, Nilva, Ruth, Viola.

Aos meus amigos de Araraquara Evandro, Renan, Birigui, Matheus, Gustavo, Denislei, Nelson, Miro, Gabriel, Maga, Nilton.

A Cristina e Mirian da secretaria de Pós-Graduação do PPG-CC.

Aos técnicos e demais funcionários do Departamento de Computação.

Ao CNPq-UFSCar/PIBIC pelo apoio financeiro em minha iniciação científica e à Profª. Maria do Carmo Nicoletti pela orientação.

Aos meus tios, primos, avós.

Às demais pessoas que me ajudaram direta e indiretamente nesta dissertação.

E obrigado a Deus pela minha vida, família, amigos e felicidade.

Resumo

Diretrizes para conduzir a reengenharia de sistemas legados implementados em linguagem Java para linguagem orientada a aspectos, AspectJ, preservando a funcionalidade original, foram propostas na abordagem Aspecting. A partir de estudos de caso realizados com essa abordagem, inferiu-se que poderia haver redução de esforços se um apoio computacional fosse criado. A Lista de Indícios (candidatos a aspectos) originalmente criada na Aspecting usava análise léxica para a identificação de interesses transversais no código legado Java. Assim, um Modelo de Indícios foi criado neste trabalho, para identificação de interesses transversais em códigos legados com base em análise sintática, por meio de AST (Abstract Syntax Tree) e para reestruturação do código Java para AspectJ. O sistema resultante apresenta melhorias em sua estrutura interna, devido à separação de interesses transversais e eliminação de problemas de espalhamento e de entrelaçamento em código, refletindo em um sistema mais modular, legível e manutenível. Um apoio computacional que automatiza parte das diretrizes envolvidas na reengenharia do sistema foi criado para viabilizar esse processo de reengenharia. Esse apoio computacional é denominado ReJAsp (apoio computacional para Reengenharia de sistemas Java para AspectJ) e foi construído como um *plug-in* do ambiente de desenvolvimento integrado Eclipse. Para avaliação do ReJAsp foram conduzidos estudos de caso a partir de sistemas implementados em Java da Internet e outros desenvolvidos em disciplinas de cursos de graduação da UFSCar.

Abstract

Guidelines that conduce the reengineering from legacy systems implemented in Java language to Aspect-Oriented language, AspectJ, preserving the original functionality, was proposed in the Aspecting method. Some case studies based on Aspecting was performed and it inferred that the reduction of efforts could be observed if computational support had been created previously and used. The List of Indications (of aspects) originally specified in Aspecting used lexical analysis for identification of crosscutting concerns found in Java code. Therefore, an Indication Model was presented in this paper in order to perform identification of crosscutting concerns in legacy code by syntactic analysis, using AST (Abstract Syntax Tree) and reorganization of Java code to AspectJ. The resulting system has enhancements in its internal structure, due to the separation of crosscutting concerns and elimination of problems related to scattering and tangling of code, observing better modularization, legibility and maintenance of system. A computational support that automates some of guidelines of system reengineering makes this migration process possible. Thus computational support called ReJAsp (apoio computacional para Reengenharia de sistemas Java para AspectJ) was built as plug-in of Integrated Development Environment Eclipse. The evaluation of ReJAsp was performed as case studies using systems written in Java, one of them found at Internet and others developed by students of computer science course of UFSCar.

Sumário

Lista de Figuras

Lista de Quadros e Tabelas

Capítulo 1

Introdução

1.1. Considerações Iniciais

A manutenção de sistemas legados pode ser realizada utilizando as técnicas de empacotamento (*wrapping*), migração (*migration*), re-desenvolvimento (*redevelopment*) ou a combinação dessas (BISBAL *et al*, 1999; SATIROĞLU, 2004).

As metodologias convencionais de manutenção de software têm seu enfoque em interesses primários (*core concerns*) do sistema, isto é, a própria funcionalidade do sistema. A manutenção desses interesses, em geral, tem impacto sobre uma única parte ou módulo, permitindo obter bons resultados quando essas metodologias são aplicadas. Entretanto, existem outros interesses de difícil localização por serem encontrados espalhados em múltiplos módulos do sistema. Esses interesses são denominados de interesses transversais e afetam negativamente diversos fatores de qualidade do software, tal como a manutenibilidade, a adaptabilidade e a reusabilidade (SATIROĞLU, 2004).

A dificuldade em manutenção de interesses transversais, devido ao esforço adicional na localização do interesse e aplicação de modificações por diversos módulos do sistema, propicia contínuas manutenções corretivas inadequadas, levando a degradação de sua estrutura interna. Mesmo que técnicas de refatoração (FOWLER, 1999) permitam melhorar a estrutura interna, tornando o código-fonte mais legível, os interesses transversais permanecem espalhados e entrelaçados no sistema.

A Programação Orientada a Aspectos foi proposta por Gregor Kiczales (1997) com o intuito de fornecer uma forma de desenvolvimento que trata separadamente os diversos interesses de um sistema, para sua posterior composição (*weaving*) em um único sistema. Seu objetivo foi introduzir uma nova dimensão para a Programação Orientada a Objetos, conhecida como aspectos, capaz de tratar os interesses transversais em uma nova abstração.

Atualmente, a linguagem de Programação Orientada a Aspectos (POA) mais popular e consolidada é a AspectJ (KICZALES *et al*, 2001). Os conceitos da POA e da linguagem AspectJ estão sendo empregados em projetos reais para a otimização de plataforma *middleware*, monitoramento e melhoria de desempenho, adição de interesses de segurança em aplicações existentes, e implementação de Integração de Aplicação Coorporativa (*Enterprise Application Integration*) (LADDAD, 2003). Todos esses projetos têm como objetivos a redução da quantidade do tempo necessário para a elaboração de produtos.

A comunidade de Desenvolvimento de Software Orientada a Aspectos (DSOA) está continuamente fornecendo várias soluções de propósito geral para a manipulação de aspectos em sistemas de software. Infelizmente, as abordagens DSOA priorizam o enfoque na identificação, especificação e implementação de aspectos para sistemas desenvolvidos desde as fases iniciais do ciclo de vida do software (SATIROĞLU, 2004). Desse modo, a pesquisa relacionada à aplicação de POA em softwares na fase de manutenção (sistemas legados) está em fase inicial e é um assunto relevante para investigação.

1.2. Objetivos

Este trabalho apresenta um apoio computacional para a reengenharia de sistemas contendo interesses transversais, com o intuito de facilitar a migração de sistemas

legados implementados em Java para sistemas em AspectJ. Assim, os benefícios da programação orientada a aspectos podem ser contemplados no sistema resultante por meio da diminuição do espalhamento e do entrelaçamento de interesses transversais no código-fonte que podem acarretar problemas de manutenibilidade do sistema. Esse apoio computacional, inicialmente, teve como base a abordagem Aspecting (RAMOS, 2004), sendo modificada conforme as necessidades de identificação e implementação de interesses transversais existentes no código-fonte legado.

A concepção desse apoio computacional foi condicionada ao requisito de usabilidade, assim sua interface deve ser simples e intuitiva para que possa ser usada por usuários iniciantes ou avançados, além de contribuir para o entendimento do código-fonte afetado por interesses transversais.

Outra característica importante é a identificação automática, precisa e configurável de indícios. Para isso, a tarefa de detectar os possíveis candidatos a interesses transversais, de agora em diante também chamados de aspectos, deve ser realizada de forma completa por meio de análises sintáticas de código-fonte, obtendo alta precisão dos resultados. Isto é, a quantidade de falsos indícios encontrados deve ser baixa ou nula, além de permitir a configuração ou ajuste dos parâmetros de identificação de indícios pelo desenvolvedor.

A automatização parcial da reestruturação de código deve ser contemplada. Nesse sentido, assistentes devem conduzir o desenvolvedor para modificações no código-fonte de modo que aspectos sejam centralizados em módulos de aspectos adequados. Ajustes realizados pelo desenvolvedor no código gerado podem ser necessários.

A economia de tempo deve ser observada, ou seja, a reengenharia de sistemas legados em Java com o auxílio do apoio computacional deve demandar menos tempo que o processo executado de modo manual. Além disso, a melhoria da qualidade deve ser verificada, por meio da maior eficiência na separação e centralização de aspectos com o uso do apoio e diminuição da incidência de erros durante a reengenharia do sistema.

1.3. Motivação

A limitação da Programação Orientada a Objetos na separação de interesses transversais resulta em problemas de espalhamento de código, que tornam a localização e alteração de interesses transversais difícil, e de entrelaçamento do interesse transversal com outros interesses, tornando o código menos legível. Desse modo, a existência e o advento de interesses transversais à medida que o software evolui agrava sua manutenibilidade, aumentando os esforços para sua manutenção.

A Abordagem Aspecting (RAMOS 2004) descreve um conjunto de diretrizes para a realização de reengenharia de sistemas legados implementados em Java para AspectJ. Os interesses transversais são identificados a partir do código legado orientado a objetos (OO). Em seguida, esses interesses são modelados, segundo diretrizes da Aspecting para serem posteriormente implementados como aspectos em AspectJ. Diagramas de casos de uso, de classes e o sistema orientado a aspectos, cujo comportamento é equivalente ao sistema legado, são alguns dos artefatos produzidos. Algumas das vantagens de realizar reengenharia por essa abordagem são: recuperação de documentação do sistema (diagrama de casos de uso e de classes), separação de interesses transversais, código mais manutenível, modular e legível. Entretanto, a principal desvantagem da abordagem é ser realizada de forma manual, a grande quantidade de diretrizes a serem executadas, consumindo muito esforço e propiciando a inserção de novos problemas, pelos próprios desenvolvedores, à medida que a abordagem é aplicada. Essas restrições são agravadas proporcionalmente pelo tamanho do sistema legado e pelo número de interesses transversais a serem separados.

A partir desse cenário, a automatização de parte das atividades necessárias na abordagem Aspecting torna-se necessária, visando à diminuição de esforço, de tempo para a identificação do aspecto e de possibilidades de inserção de novos problemas, acidentalmente, pelo desenvolvedor. Assim, a principal motivação deste trabalho é o desenvolvimento de um apoio computacional que auxilie nas tarefas da realização de reengenharia de sistemas legados em Java para a linguagem AspectJ.

Para isso, é importante realizar o estudo e análise da Lista de Indícios usada na abordagem Aspecting com o propósito de encontrar oportunidades de sua extensão e de

melhorias na abordagem Aspecting. O reconhecimento da Lista de Indícios por ferramentas apropriadas e a definição de novas representações para indícios também são pertinentes.

A construção de um apoio computacional provê a possibilidade de verificar a validade da Aspecting e a viabilidade da migração de sistemas legados em Java para AspectJ por meio da execução de estudos de casos. A partir de experiências práticas, contribuições e sugestões de desenvolvedores envolvidos em experimentos podem conduzir à maturidade da abordagem e do apoio computacional. Com a realização de experimentos por usuário inexperientes para a realização da reengenharia de sistemas Java para AspectJ, é possível verificar a facilidade de uso e de aprendizado do apoio computacional proposto, e de acordo com os resultados obtidos pode-se avaliar se o processo realizado é facilitado quando comparado ao manual.

Acredita-se que a existência da função de desfazer em apoios computacionais também representa um importante diferencial para sua aceitação e para melhorar a exploração de seus recursos pelos desenvolvedores. A possibilidade de retroceder, a qualquer momento, em alterações realizadas em passos anteriores da reengenharia do sistema, encoraja os desenvolvedores em experimentar e explorar cada vez mais os recursos disponibilizados pelo apoio computacional.

1.4. Organização da Monografia

No Capítulo 2 a Programação Orientada a Aspectos é introduzida com a descrição de seus termos mais comumente usados e a linguagem orientada a aspectos mais popular, AspectJ, é apresentada. Em seguida, manutenção de software e suas categorias também são discutidas. A abordagem Aspecting de Ramos (2004) é, então, contextualizada como uma forma de manutenção preventiva e algumas ferramentas usadas para o desenvolvimento e na manutenção de sistemas segundo a POA são mostradas.

Aprimoramentos relativos à abordagem Aspecting detectados neste trabalho são relatados no Capítulo 3, destacando revisões da Lista de Indícios para persistência de banco de dados, persistência em memória temporária e *logging*. Características de

mecanismos de buscas léxicas e sintáticas são confrontadas e comparadas, seguidas de ilustrações dos aprimoramentos sugeridos. Após analisar possíveis melhorias para a Abordagem Aspecting, o Modelo de Indícios é proposto, apresentado e contextualizado para os indícios de persistência em banco de dados, em memória temporária e de *logging*. Estendendo a abordagem Aspecting, diretrizes de identificação de indícios e de reestruturação de código, utilizando agora o Modelo de Indícios, são propostas.

A construção de um protótipo de apoio computacional, denominado ReJAsp (apoio computacional para Reengenharia de sistemas Java para AspectJ) é apresentada no Capítulo 4. Detalhes de sua arquitetura, bem como sua árvore de indícios, assistentes de reestruturação e de gerenciamento de indícios são discutidos. Todo o mecanismo de identificação de indícios realizado por ReJAsp é abordado, mostrando o uso de registro de indícios, de percurso em árvore (AST), de tabela

de variáveis e de regras de agrupamento de indícios durante essa etapa. Além disso, o mecanismo de persistência de indícios usado por ReJAsp e trabalhos similares a ele são apresentados.

Após as fases de projeto rápido e de construção no processo de prototipagem de software, é imprescindível realizar sua avaliação, permitindo um refinamento dos requisitos estabelecidos ao apoio computacional. Nesse sentido, estudos de caso foram realizados com equipes de desenvolvimento. O Capítulo 5 apresenta objetivos, hipóteses, medições e observações obtidas a partis desses estudos de caso. Para isso, planejamentos referentes à distribuição e alocação de equipes de desenvolvimento foram realizados, bem como atribuição de sistemas legados a essas equipes. Os resultados gerados pela coleta de tempo gasto nos experimentos, por inspeção de código-fonte e pela avaliação de questionários respondidos por desenvolvedores são apresentados e discutidos.

Finalmente, no Capítulo 6, discussões dos resultados obtidos, bem como as contribuições e limitações deste trabalho, e possíveis trabalhos futuros são apresentados.

Capítulo 2

Assuntos Relacionados

2.1. Considerações Iniciais

A dificuldade e o alto custo de manutenção de softwares é um problema antigo que persiste ainda com o advento do paradigma de desenvolvimento de software orientado a objetos. As abstrações proporcionadas por esse paradigma são capazes de representar os requisitos funcionais de software. Entretanto, apresenta limitações para a representação de requisitos não funcionais em sistemas complexos.

Os requisitos não funcionais ou interesses transversais de software são capazes de afetar diversos módulos do sistema. O tratamento de exceção, persistência em banco de dados e controle de concorrência, entre outros, são exemplos típicos de interesses transversais. Empregando o desenvolvimento de software orientado a objetos, os interesses transversais se apresentam espalhados e entrelaçados pelo código funcional dos vários módulos, o que acarreta baixa legibilidade e dificuldade na manutenção desse software.

Como principal diferencial em relação à programação orientada a objetos (POO), a programação orientada a aspectos (POA) permite a separação de interesses transversais, possibilitando a implementação de cada um individualmente e de forma

combinada em um sistema final. Desse modo, o código do sistema pode apresentar menor entrelaçamento e espalhamento desses interesses.

As vantagens proporcionadas quando se utiliza POA, tais como melhor legibilidade, código mais modular, e, conseqüentemente, melhor manutenibilidade, podem ser justificativas para a migração de sistemas orientados a objetos existentes para sistemas orientados a aspectos. Para isso, são necessárias técnicas e ferramentas que apóiem os desenvolvedores nesse processo.

A reengenharia de um sistema legado pode ser realizada de diversas maneiras e com variações, de acordo com o engenheiro de software responsável pela atividade. Nesse sentido, métodos de mineração de aspectos e de refatoração de código legado Orientado a Objetos (OO) auxiliam na migração de um sistema OO em um novo sistema construído em uma linguagem Orientada a Aspectos (OA).

Contudo, a reengenharia do sistema legado pode ser inviabilizada na prática, envolvendo grande esforço para o entendimento do código-fonte e sua efetiva realização. Com o intuito de facilitar sua viabilização, a construção de apoios computacionais é indicada, sendo muitas vezes implementados como *plug-ins* de ambientes de desenvolvimentos de software.

Para prover subsídios para a discussão dos assuntos em capítulos posteriores, neste capítulo os assuntos relacionados são discutidos e organizados da seguinte maneira: a Seção 2.2 apresenta os precursores da Programação Orientada a Aspectos; a Seção 2.3 descreve os principais conceitos da POA; a Seção 2.4 aborda a Linguagem AspectJ e suas construções; a Seção 2.5 apresenta conceitos gerais de manutenção de software e reengenharia; a Seção 2.6 discute a abordagem Aspecting; a Seção 2.7 trata de apoios computacionais à POA e na Seção 2.7 são feitas as considerações finais.

2.2. Precursores da Programação Orientada a Aspectos

Desde o advento dos primeiros computadores, os sistemas e as linguagens de programação têm sofrido evoluções, partindo da linguagem de máquina e Assembly para os primeiros computadores, passando pelos conceitos de tradução de fórmulas, programação procedimental, programação estruturada, programação funcional, programação lógica e programação com dados abstratos. Cada um desses passos

evolutivos melhorou a habilidade em alcançar a separação clara de interesses no nível de código-fonte (ELRAD *et al*, 2001a).

A separação de interesses (PARNAS, 1972) é a parte da Engenharia de Software que se refere à habilidade de identificar, encapsular e manipular apenas as partes do software que são relevantes para um conceito, objetivo ou propósito particular. Um interesse pode ser basicamente qualquer elemento cognitivo considerado durante a construção de um programa, tal como um protocolo, uma característica ou um requisito, entre outros (OSSHER e TARR, 2000).

Os interesses constituem a motivação mais importante para a organização e a decomposição do software em partes gerenciáveis e compreensíveis. A relevância dos interesses pode variar de acordo com o julgamento do desenvolvedor, com sua responsabilidade ou pode depender do estágio do ciclo de vida do software.

A Programação Orientada a Objetos (POO) tem sido o paradigma de programação dominante nas últimas décadas. O enfoque central da POO é de que cada interesse de um sistema de software deve ser implementado como um módulo separado, sendo visualizado como uma construção de uma coleção de classes discretas, cada uma implementando um diferente interesse, ou seja, com suas tarefas e suas responsabilidades bem definidas. (VOELTER, 2000).

Os objetos abstraem comportamentos e dados em uma simples entidade conceitual (e física). Muitos foram os métodos e ferramentas desenvolvidas com esse paradigma, facilitando também a escrita de aplicações complexas, tais como interfaces gráficas de usuário, sistemas operacionais e aplicações distribuídas, com código-fonte compreensivo (ELRAD *et al*, 2001a). Entretanto, a POO possui limitações em transformar todos os interesses do sistema em módulos (objetos), pois existem interesses que afetam várias partes do sistema. Com isso, cada um desses interesses não pode ser facilmente transformado em um único módulo.

Além disso, na tecnologia OO há dificuldade de localizar os interesses que envolvem restrições globais e comportamentos gerais, isolar interesses apropriadamente e aplicar conhecimento específico do domínio. Mecanismos de Programação Pós-Objeto que procuram aumentar a expressividade do paradigma OO são áreas promissoras e incluem linguagens específicas de domínio, programação genérica, linguagem formal para expressar restrição, reflexão, meta-programação e desenvolvimento orientado a características (ELRAD *et al*, 2001a).

Uma importante tecnologia de Programação Pós-Objetos é a Programação Orientada a Aspectos (POA) (KICZALES *et al* 1997a) que é baseada na idéia de que os sistemas de computação são mais bem programados pela especificação de diversos interesses e de algumas descrições de seus relacionamentos separadamente. Mecanismos de POA realizam a combinação e composição dos interesses em um programa coerente (ELRAD *et al*, 2001a). Desse modo, a POA possibilita implementar os interesses de maneira modular com os seguintes benefícios: código simplificado, desenvolvimento facilitado, manutenibilidade e grande potencial para o reúso (KICZALES *et al*, 2001).

2.3. Programação Orientada a Aspectos

Esta seção apresenta os termos comumente usados e referentes à POA. A tradução desses termos foi estabelecida e padronizada durante o I Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (WASP, 2004). Parte desses termos são específicos da linguagem de programação AspectJ e, por isso, são tratados apenas na Seção 2.4 .

a) Interesse Transversal (*Crosscutting Concern*)

Interesses transversais são requisitos no nível de sistema (LADDAD, 2003). Em um único módulo, são de difícil localização e tendem a se apresentar espalhados por vários módulos (ELRAD *et al*, 2001b). Quando o interesse transversal não é apropriadamente tratado, o código responsável por manipulá-lo se apresenta misturado com o código referente ao interesse primário da aplicação nos diversos módulos da aplicação, resultando em um código entrelaçado. Com isso, ocorre o aumento da complexidade e redução da qualidade do software, tal como a adaptabilidade, manutenibilidade e reusabilidade (SATIROĞLU, 2004).

As abordagens convencionais de manutenção de software orientadas a objetos, baseadas em técnicas de empacotamento, migração e re-desenvolvimento, obtiveram relativo sucesso ao tratar interesses não transversais quando esses afetam um único módulo e são de fácil localização. Entretanto, a manutenção e a evolução de interesses transversais empregando essas abordagens não são efetivas, pois as modificações devem ser realizadas em vários pontos do código-fonte do sistema (SATIROĞLU, 2004).

Exemplos desses casos são, entre outros, *logging*, gerenciamento remoto e otimização do caminho (LADDAD, 2003). Satiroğlu (2004) também aponta os interesses de distribuição, sincronização, persistência, segurança e comportamento de tempo-real como transversais.

b) Separação de Interesses (*Separation of concerns*)

Um dos princípios fundamentais de Engenharia de Software é a separação de interesses, proporcionando um sistema mais simples de entender e de maior manutenibilidade. Vários métodos e *frameworks* (também conhecidos como arcabouços) têm o intuito de apoiar esse princípio de alguma forma (LADDAD, 2003). Esse princípio afirma que um problema de projeto envolve diferentes tipos de interesses, que deve ser identificado e separado em módulos diferentes, tentando a separação do algoritmo básico de interesses de propósitos diversos. Com isso, ocorre minimização e melhoria da clareza das dependências entre interesses no nível conceitual e no nível de implementação (SATIROĞLU, 2004).

Além disso, muitos teóricos concordam que a melhor maneira de criar sistemas gerenciáveis é pela separação de interesses. Em uma publicação de 1972, David Parnas (PARNAS, 1972) propôs que o melhor caminho para se obter a separação de interesses é por meio da modularização – um processo de criação de módulos que permite ocultar determinados detalhes de outros módulos. Nos anos seguintes, pesquisadores estudaram vários meios de gerenciar interesses, surgindo a POO que permitiu separar interesses primários (referentes à regra de negócio). Porém, não foi efetiva para interesses transversais. Programação genérica, meta-programação, programação reflexiva, filtro de composição, programação adaptativa, programação orientada a assunto, POA, programação intencional emergiram como possíveis abordagens para a manipulação de interesses transversais, sendo que POA é a abordagem mais popular (LADDAD, 2003).

c) Espalhamento (*Scattering*)

Laddad (2003) classifica os sintomas de não modularização em duas categorias: espalhamento e entrelaçamento de código. Afirma que o espalhamento do código é causado quando um simples problema é implementado em múltiplos módulos. Por definição, os interesses transversais estão espalhados por muitos módulos,

conseqüentemente, as implementações relacionadas também estão espalhadas dentro de tais módulos.

A classificação do espalhamento de código pode ser feita em duas categorias diferentes: blocos de código duplicados ou complementares. A primeira é caracterizada pela repetição de códigos muito similares; por exemplo, o *pooling* de recursos tipicamente envolverá a adição de códigos muito semelhantes em múltiplos módulos com o intuito de obter um recurso do *pool*. A Figura 2.1 ilustra os blocos de código duplicados e espalhados.

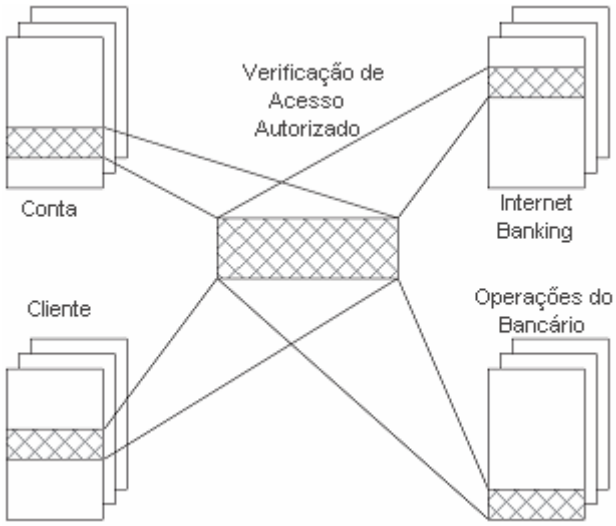

Figura 2.1: Espalhamento do código responsável pelo interesse acesso de serviços para apenas cliente autorizados (LADDAD, 2003)

O segundo tipo de espalhamento de código acontece quando diversos módulos implementam partes complementares de um interesse. Por exemplo, em um sistema de controle de acesso implementado convencionalmente, parte da autenticação é executada em um módulo, em seguida, as informações geradas por esse módulo se tornam disponíveis em outros módulos responsáveis pela finalização da autenticação. Todas essas partes devem ser fragmentadas de modo que se encaixem perfeitamente, como um quebra-cabeça, a fim de implementar a funcionalidade, como mostrado na Figura 2.2. Diversos módulos incluem código para a lógica de autenticação e verificação de acesso e devem funcionar conjuntamente para implementar a autorização de forma correta. Por exemplo, antes de checar as credenciais de um usuário (controle de acesso), deve-se verificar a autenticidade do usuário (autenticação).

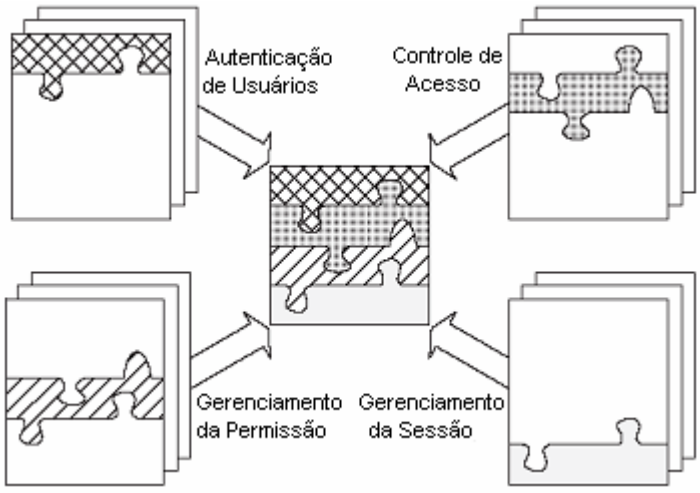

Figura 2.2: Código espalhado causado pela necessidade de inserir blocos de código em vários módulos para implementar a funcionalidade (LADDAD, 2003)

d) Entrelaçamento (*Tangling*)

Segundo Laddad (2003), o entrelaçamento de código é causado quando um módulo implementado manipula vários interesses simultaneamente. Um desenvolvedor freqüentemente considera interesses tais como regra de negócio, desempenho, sincronização, *logging* e segurança, entre outros durante a implementação do módulo. Isso causa a simultânea presença de elementos de vários interesses, resultando um código entrelaçado. A Figura 2.3 ilustra o entrelaçamento de código em um módulo.

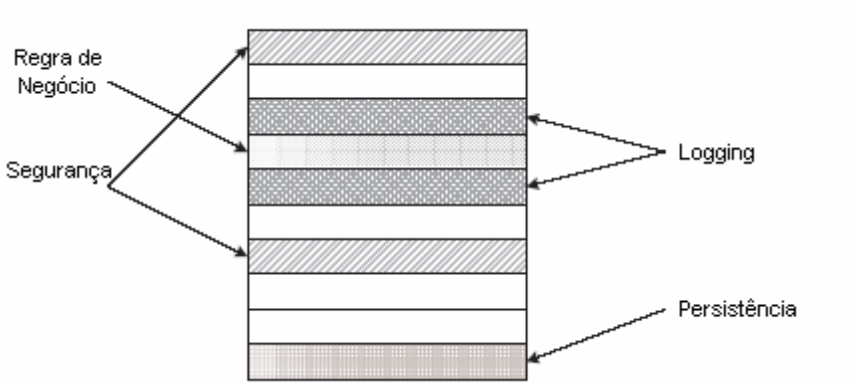

Figura 2.3: Código entrelaçado causado pelas diversas implementações simultâneas de vários interesses. (LADDAD, 2003)

e) Aspecto (*Aspect*)

Aspectos são unidades modulares de implementações que entrecortam várias partes do sistema. Declarações de aspectos, cujo formato é similar a declarações de classes, podem incluir declarações de conjunto de junção (*pointcut*), adendos (*advices*), apresentadas posteriormente, além dos outros tipos de declarações permitidas em classes (KICZALES *et al*, 2001). O aspecto tende a não ser unidade da decomposição do sistema funcional, mas certamente propriedade que afeta o desempenho ou semântica dos componentes em maneiras sistêmicas (KICZALES *et al*, 1997a).

f) Inconsciência (*Obliviousness*)

O conceito de inconsciência está relacionado com a possibilidade de um desenvolvedor escrever trechos de código-fonte referente a um interesse sem se preocupar com os demais interesses do sistema, mesmo que, tais interesses afetam o trecho implementado, sem o conhecimento explícito do desenvolvedor (FILMAN, 2001).

g) Pontos de Junção *(Join Points*)

Os pontos de junção são elementos pertencentes à semântica da linguagem do código-base que são coordenados mutuamente com os aspectos (KICZALES *et al*, 1997a). São pontos bem definidos dentro do código-base em que o interesse entrecortará a aplicação. Os pontos de junção podem ser: chamadas de métodos, chamadas de construtores, tratamento de exceção ou outros pontos na execução do programa (GRADECKI e LESIECKI, 2003).

h) Combinação (*Weaving*)

Em qualquer linguagem orientada a aspectos, a combinação é o principal mecanismo que garante a junção na execução do código contendo o aspecto com o código-base, de um modo coordenado e correto (KICZALES *et al*, 2001). Esse processo de combinação do sistema OA segue um conjunto de regras de composição, que especificam como integrar os aspectos implementados de modo a resultar em um sistema final. Essas regras podem ser bastante genéricas ou específicas com relação ao modo como interagem com o código-base. A linguagem usada para especificar as regras

de composição pode ser extensão natural da linguagem-base (linguagem empregada pelo código-base) ou totalmente diferente (LADDAD, 2003).

i) Combinador (*Aspect Weaver*)

O combinador é responsável por processar as linguagens do componente do código-base e do aspecto corretamente compondo-as para produzir toda operação desejada do sistema. Os pontos de junção são essenciais para a função do combinador, pois esse último funciona a partir da geração da representação do ponto de junção de um código-base. Essa representação é compilada e executada com o programa contendo o aspecto (KICZALES *et al*, 1997a).

O combinador pode ser implementado de várias maneiras. A maneira mais simples pode ser pela tradução código-a-código. Nesse caso, os módulos de código-fonte individuais de classes e de aspectos são pré-processados pelo compilador do aspecto a fim de produzir código fonte combinado. Com isso, o compilador do aspecto alimenta o compilador da linguagem-base com o código convertido, produzindo o código final executável. A outra abordagem consiste em, inicialmente, compilar o código-fonte usando o compilador da linguagem-base. O compilador de aspectos combina os aspectos com os arquivos gerados da compilação (LADDAD, 2003).

O termo *core concern* não teve sua tradução padronizada durante o I Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (WASP, 2004); entretanto, é utilizado neste trabalho como interesse primário.

Os interesses primários captam a funcionalidade central do módulo e estão relacionados com a lógica ou regra de negócio. Em linguagens pertencentes ao paradigma OO, tais interesses são implementados como classes. Em POA, os interesses primários são implementados usando um método base para posterior combinação com os aspectos (interesses transversais), formando o sistema final.

2.4. A Linguagem AspectJ

AspectJ (KICZALES *et al*, 2001) é uma linguagem de programação Orientada a Aspectos e extensão da linguagem de programação Java (JAVA TECHNOLOGY, 2007), ou seja, qualquer programa válido na linguagem Java também é válido em AspectJ. Seu compilador gera arquivos de extensão *"class"* que estão em conformidade

com a especificação do binário produzido pela linguagem Java, permitindo que qualquer máquina virtual Java execute os arquivos produzidos. O uso da linguagem Java, como linguagem base, confere à AspectJ todos os seus benefícios, além de facilitar o entendimento do seu código pelos programadores em linguagem Java.

AspectJ tem suas origens em antigas metas de possibilitar, de maneira clara, a manipulação de estruturas de projeto complexas em implementações de software, envolvendo trabalho em POO, reflexão e protocolos de meta-objetos (PARC, 2006).

A seguir são apresentadas as construções e outros conceitos específicos à linguagem AspectJ que tiveram suas traduções padronizadas durante o I Wasp (WASP, 2004).

a) Interesse Transversal Dinâmico (*Dynamic Crosscutting*)

A maioria dos entrecortes que ocorrem em AspectJ são dinâmicos e consistem na combinação de um novo comportamento na execução do programa. Ele acrescenta ou, até mesmo, remove o fluxo de execução do programa base de forma a entrecortar diversos módulos. Por exemplo, caso uma determinada ação seja realizada antes da execução de alguns métodos ou de tratadores de exceção em um conjunto de classes, é possível especificar em um módulo separado os pontos de combinação e a ação a ser tomada quando tais pontos são alcançados. Em AspectJ, os conjuntos de junção e os adendos compõem mecanismos denominados interesses transversais dinâmicos (LADDAD, 2003).

b) Interesse Transversal Estático (*Static Crosscutting*)

É um mecanismo que possibilita modificações em uma estrutura estática, isto é: classes, interfaces e aspectos de um sistema. Não sendo capaz de modificar a execução do comportamento do sistema, a função mais comum do mecanismo de interesse transversal estático é o suporte da implementação do mecanismo de interesse transversal dinâmico. Por exemplo, é possível a adição de novos atributos e métodos às classes e interfaces para especificarem estados e comportamentos específicos da classe que podem ser usados nas ações dos interesses transversais dinâmicos. Outro uso do mecanismo de interesse transversal estático é a declaração de avisos e de erros de tempo de compilação por vários módulos (LADDAD, 2003).

A especificação de hierarquias de classes também pode ser realizada pelo mecanismo de interesse transversal estático. A visibilidade da estrutura do sistema é importante, especialmente quando é necessário estender as classes com outras novas classes ou introduzir novos métodos e atributos às classes. Nesse caso, é importante que componentes do sistema que estejam separados possam ser vistos e acessados separadamente (SATIROĞLU, 2004).

c) Conjunto (de pontos) de junção (*Pointcut*)

Sabendo que um ponto de junção é um ponto de execução bem definido em uma aplicação, é necessária uma construção que permita a linguagem orientada a aspecto (OA) definir o momento em que o ponto de junção deve ser combinado (GRADECKI e LESIECKI, 2003). O conjunto de junção seleciona pontos de junção e coleta o contexto nesses pontos. Para distinguir os conceitos de ponto de junção e de conjunto de junção, deve-se ter em mente que os conjuntos de junção especificam as regras de combinação e os pontos de junção são as situações que satisfaçam a essa regra (LADDAD, 2003).

d) Comportamento transversal ou Adendo (*Advice*)

Adendo é um mecanismo similar a um método, usado para declarar a execução de determinado código em cada um dos pontos de junção de um conjunto de junção. AspectJ contém os adendos: *before*, *after* e *around*. Além disso, existem dois casos especiais de adendo: posterior ao retorno de método (*after returning)* e posterior ao lançamento de exceção (*after throwing)*, correspondendo às duas maneiras que uma subcomputação pode retornar por meio de um ponto de junção. Ambos os adendos anteriores e os três tipos de adendos posteriores adicionam comportamento à computação normal do ponto de junção. Entretanto, o adendo de contorno tem a capacidade de controlar a computação normal no ponto de junção. A declaração do adendo o define pela associação de um corpo de código com um conjunto de junção, além do momento relativo a cada ponto de junção e quando o código deve ser executado (KICZALES *et al*, 2001).

e) Declaração Intertipos (*Intertype declarations*)

A declaração intertipos, também chamada de introdução (*introduction*), introduz mudanças nas classes, interfaces e aspectos do sistema. As mudanças estáticas nos

módulos não afetam seu comportamento (LADDAD, 2003). Essa construção representa uma importante divergência de sistema de tipo em Java. Na linguagem padrão Java, interfaces provêm apoio à herança de interface, mas não implementam a herança. Assim, a classe concreta, que implementa uma interface, deve implementar ou herdar a implementação para cada método na interface. Essa obrigação pode requerer muito esforço, mas freqüentemente há maneiras de evitá-lo. O uso de classes abstratas poderia contornar essa situação, pois elas poderiam implementar apenas alguns métodos da interface. Entretanto, classes abstratas não são adequadas quando precisam estender módulos adicionais.

A linguagem AspectJ, por meio de declarações intertipos, pode contornar o problema das interfaces em Java, permitindo às interfaces ter comportamento concreto, bem como abstrato. Dessa maneira, programas em AspectJ possibilitam implementar heranças múltiplas (GRADECKI e LESIECKI, 2003).

2.5. Manutenção de software

A POA pode ser usada para o desenvolvimento de um novo software ou para a reengenharia de um software legado OO para outro equivalente em uma linguagem OA, podendo tornar a manutenção desse software mais fácil, devido ao mecanismo de separação de interesses transversais, o que pode resultar em um código mais modular e legível.

A manutenção de software é uma das fases mais problemáticas em seu ciclo de vida, caracterizando-se por alto custo e baixa velocidade de implementação das modificações solicitadas. Porém, é uma atividade inevitável, principalmente tratando-se de softwares para os quais os usuários constantemente solicitam modificações e agregações de novas funções (BENNET e RAJLICH, 2000). O rastreamento de defeito, a freqüente falta de documentação e entendimento de detalhes do sistema são constantes e consomem recursos durante a realização das atividades de manutenção (BISBAL *et al*, 1999).

De acordo com Bisbal *et al* (1999), diversas soluções foram propostas para os problemas envolvendo a manutenção de software. Tais soluções se enquadram de modo generalizado em três categorias:

- Re-desenvolvimento (*redevelopment*): o código-fonte da aplicação existente é totalmente reescrito e o sistema legado é completamente descartado;
- Empacotamento (*wrapping*): o componente previamente existente é empacotado em um novo componente de software tornando-o mais acessível aos mantenedores;
- Migração (*migration*): o sistema legado é transferido para um ambiente mais flexível, enquanto os dados e a funcionalidade do sistema original são mantidos.

O re-desenvolvimento é uma forma de manutenção que conduz para as modificações mais importantes, mesmo que exija a reescrita de todo código existente e perfeito entendimento do sistema existente, envolvendo muitas atividades de reengenharia (SATIROĞLU, 2004).

A reengenharia, por sua vez, é a análise e a modificação de um sistema para reconstituí-lo em uma nova forma e realizar sua subseqüente implementação (CHIKOFSKY e CROSS, 1990). A Reengenharia convencional consiste de dois estágios: (1) engenharia reversa, e (2) engenharia avante. Engenharia reversa é o processo de encontrar o projeto e a especificação do software a partir de seu código fonte. Inicialmente, na reengenharia o programa atual é submetido à engenharia reversa para se obter informações em um nível mais alto de abstração e, em seguida, o programa é implementado em uma nova linguagem de programação. O resultado é um novo programa com a mesma funcionalidade que o sistema existente ou com a sua funcionalidade melhorada, satisfazendo alguns novos requisitos além dos já existentes (SATIROĞLU, 2004). Quando se aplica reengenharia iterativa, os estágios de engenharia reversa e avante não são separados. Dessa forma, obtém-se informações do código legado e as melhorias são realizadas no código implementado na nova linguagem de programação.

Com o advento de novas tecnologias, a reengenharia de sistemas tornou-se uma opção atrativa para as organizações, embora haja muita discussão em termos de seu custo, tempo e por absorver recursos que poderiam ser usados em necessidades imediatas (PRESMAN, 2006). Assim, é necessário que a organização tenha uma estratégia pragmática para realizá-la.

2.6. Abordagem Aspecting

A abordagem Aspecting foi desenvolvida por Ramos (2004) e traz, como principal motivação, a necessidade da reengenharia de sistemas orientados a objetos existentes e implementados em Java para novos sistemas orientados a aspectos. Essa abordagem consiste de um conjunto de diretrizes que guiam o engenheiro de software em um processo iterativo.

O conjunto de códigos-fonte em linguagem Java (OO), considerados legados, serve como entrada para gerar como saída outro conjunto de códigos-fonte em linguagem AspectJ (OA). O comportamento do conjunto de códigos-fonte resultantes (OA) deve ser preservado em relação ao comportamento observado ao original (OO), isto é, sem a adição de novas funções ao sistema. A abordagem enfoca na reorganização da estrutura interna, de modo a melhorar a manutenibilidade e a legibilidade do sistema.

Adicionalmente, durante a execução da abordagem Aspecting é gerada a modelagem de interesses existentes no sistema. Tal modelagem é obtida a partir do código-fonte orientado a objetos por meio da identificação prévia de indícios (candidatos) de aspectos no sistema. Em outras palavras, partindo de uma abstração de baixo nível (código-fonte) obtém-se outra de mais alto nível. Posteriormente, um novo sistema é gerado com código mais organizado e manutenível.

A abordagem Aspecting, como apresentada na Figura 2.4, é dividida em três etapas: I) Entendimento da funcionalidade: ocorre a obtenção e a documentação de casos de uso do sistema e de detalhes de projetos; II) Tratamento de interesses: realiza a modelagem, identificação e implementação dos interesses transversais; e III) Comparação de sistema OA com o sistema OO: é verificado se o sistema OA possui o mesmo comportamento do sistema legado (OO).

Aspecting foi construída com base na realização de estudo de casos de sistemas implementados na linguagem Java, nos quais puderam ser observados indícios da existência de interesses que foram agrupados em uma lista, denominada de Lista de Indícios. Essa lista auxilia o engenheiro de software na identificação de aspectos no código fonte Java existente, de acordo com três tipos de interesses. Uma parte da Lista de Indícios, apresentada no Quadro 2.1, contém os interesses de rastreamento, de persistência em banco de dados e de persistência em memória temporária (*Buffering*).

Entretanto, os interesses de tratamento de erros, tratamento de exceção e programação paralela (*thread*) também compõem a Lista de Indícios completa encontrada em Ramos (2004). Os indícios estão expressos na terceira coluna por meio de gramáticas livres de contexto.

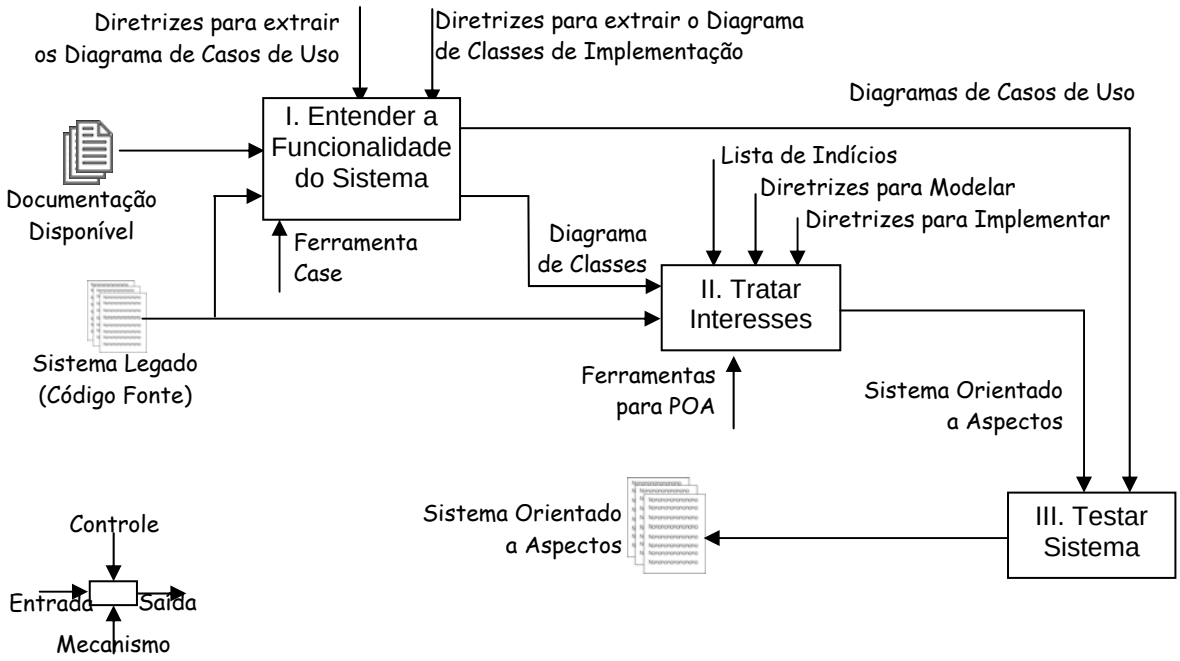

Figura 2.4: Etapas da abordagem Aspecting (RAMOS, 2004)

Após a aplicação das diretrizes da abordagem Aspecting em um sistema OO, o sistema OA obtido pode ainda não possuir o mesmo comportamento do sistema original, sendo necessário reaplicar as diretrizes. A repetição de atividades também ocorre dentro da fase de tratamento de interesse considerando a característica iterativa da abordagem. Isso faz com que o esforço despendido para a execução, principalmente, de projetos de grande porte seja significativo. O processo manual além de necessitar de grande esforço por parte do engenheiro de software, pode não ter a qualidade desejada. Assim, um apoio computacional para auxiliar o engenheiro de software a realizar o processo de reengenharia de um sistema OO para um OA é fundamental para avaliar e viabilizar a abordagem Aspecting.

Quadro 2.1 – Trecho da Lista de Indícios extraído de Ramos (2004)

Nível de Interesse	**Interesse**	**Indícios**
Desenvolvimento	1) Rastreamento (*tracing)*	Métodos que imprimem mensagens de texto, tais como: - System.out.print("<Mensagem>"); - System.out.println("<Mensagem>"); <Mensagem> contém informações ao desenvolvedor sobre os nomes ou valores dos métodos, ou indicando a posição física do método no código fonte.
Produção	2) Persistência de banco de dados	<i.Persistencia.BD> = 'Connection' <statements> | 'Connection' <statements> <SQL>| <SQL> <statements> 'Connection'| 'PreparedStatement' <statements>| 'PreparedStatement' <statements> <SQL>| <SQL> <statements> 'PreparedStatement'| 'ResultSet' <statements>| 'ResultSet' <statements> <SQL>| <SQL> <statements> 'ResultSet'
Tempo de Execução	3) Persistência em memória temporária (*Buffering*)	<i.persistência.mem.temp> = 'BufferredReader' <statements> | 'new' 'BufferredReader' '(' <statements> ')' | 'StringBuffer' <statements> | 'new' 'StringBuffer' '(' <statements> ')' | 'BufferedInputStream' <statements> | 'new' 'BufferedInputStream' '(' <statements> ')'

2.7. Ferramentas de apoio ao desenvolvimento OA

Hugunin (2001) afirma que linguagens OA são utilizáveis em sistemas reais de médio porte. A habilidade em usar as ferramentas para projetos de software reais é importante tanto para os desenvolvedores quanto para a comunidade de pesquisa que pode observar quais métodos, técnicas e ferramentas são adequadas na prática.

Há ainda obstáculos técnicos que precisam ser superados para o uso dessas ferramentas em sistemas de grande porte. Isso é importante para a avaliação do sucesso da POA, pois os benefícios de produtividade em qualquer nova ferramenta são sempre prejudicados pela falta de refinamento das mesmas.

A partir de boas ferramentas que suportem POA para sistemas de grande porte, é possível verificar os impactos de linguagens OA na produtividade de codificação e de manutenção do sistema. Além disso, o uso de linguagens e de ferramentas OA em sistemas de grande porte é importante para avaliar a premissa de que POA torna o sistema mais modular.

Algumas das ferramentas que provêm apoio ao desenvolvimento de software orientado a aspectos e auxiliam a construção de novas ferramentas são comentadas nas próximas subseções.

2.7.1. Plataforma Eclipse

Eclipse (ECLIPSE PROJECT, 2007) é uma plataforma que foi projetada inicialmente para a construção de ferramentas integradas de desenvolvimento Web e de aplicações. A plataforma por si só não provê grande número de funções ao usuário final. Seu mérito está no encorajamento ao desenvolvimento rápido de características integradas baseado no modelo de *plug-in*.

A plataforma, representada na Figura 2.5 provê um modelo de interface de usuário comum para a manipulação de ferramentas, além de executar em múltiplos sistemas operacionais enquanto fornece integração robusta com cada sistema operacional. A arquitetura da plataforma Eclipse está estruturada em subsistemas implementados como um conjunto de *plug-ins*.

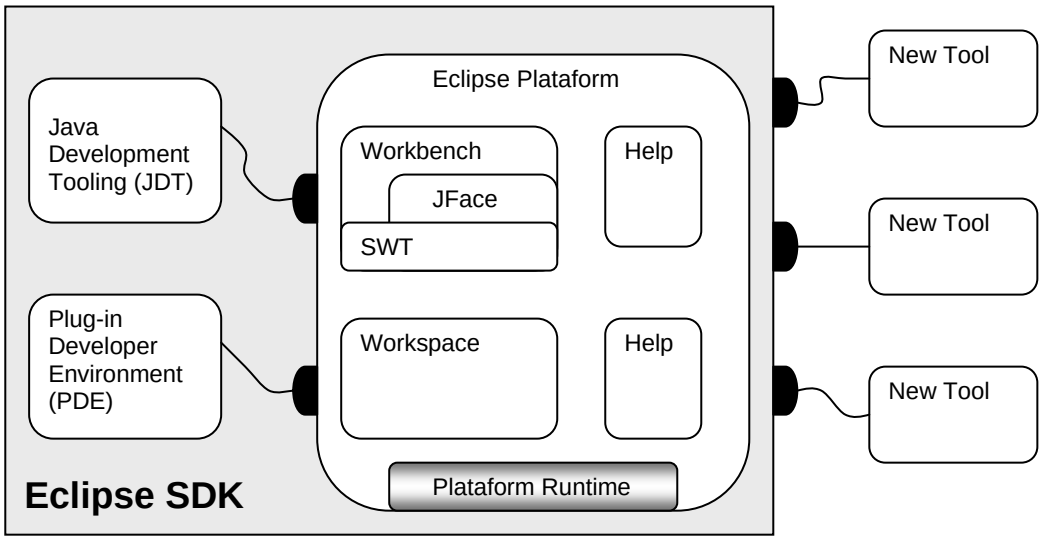

Figura 2.5: Arquitetura da Plataforma Eclipse e de suas ferramentas JDT e PDE (ECLIPSE PROJECT, 2007)

Plug-ins são pacotes estruturados de código-fonte e/ou dados que contribuem para a funcionalidade do sistema, representando códigos de bibliotecas (classes Java com API pública), extensões da plataforma e, até mesmo, documentação. Podem definir pontos de extensão, isto é, locais bem definidos em que outros *plug-ins* podem adicionar funcionalidade. Os p*lug-ins* podem ser programados a partir de API portáveis do ambiente Eclipse e serem executados, sem nenhuma mudança, em cada um dos sistemas operacionais que apóia, isto é, contém grande portabilidade entre sistemas operacionais.

A Figura 2.5 apresenta o *kit* de desenvolvimento de software Eclipse (*Eclipse SDK*), também denominado ambiente de desenvolvimento integrado, do inglês *Integrated Development Environment* – IDE. Esse ambiente inclui a plataforma básica, além de ferramentas úteis para o desenvolvimento de *plug-ins*: as ferramentas JDT que implementam um ambiente de desenvolvimento Java completo e o ambiente desenvolvedor de *plug-in* (PDE) que acrescenta ferramentas especializadas, para organizar o desenvolvimento de *plug-ins* e as extensões (ECLIPSE PROJECT, 2007). Os três blocos *New Tool* representam outros *plug-ins* que podem ser desenvolvidos posteriormente ou incluídos para atender solicitações específicas de um projeto ou ambiente de trabalho.

2.7.2. *Plug-in* AJDT

AspectJ Development Tool (ASPECTJ DEVELOPMENT TOOLS PROJECT, 2007), usualmente denominada de AJDT, é uma ferramenta que apóia a edição, a construção e a depuração de programas em AspectJ. Apresenta-se como um *plug-in* para o ambiente Eclipse e provê diversas características na programação de softwares OA, além de disponibilizar recursos que auxiliam o aprendizado e o entendimento da POA utilizando AspectJ.

A ferramenta mais poderosa que o *plug-in* AJDT disponibiliza para o entendimento do impacto do aspecto sobre todo o sistema é o visualizador de aspectos (CLEMENT *et al*, 2003), ilustrado na Figura 2.6. Cada aspecto do sistema aparece no menu do visualizador com uma cor associada. As marcas coloridas encontradas ao longo das barras verticais indicam as localizações onde os elementos do programa no arquivo são afetados por adendos. As cores das marcas referem-se às cores atribuídas pelos aspectos no menu.

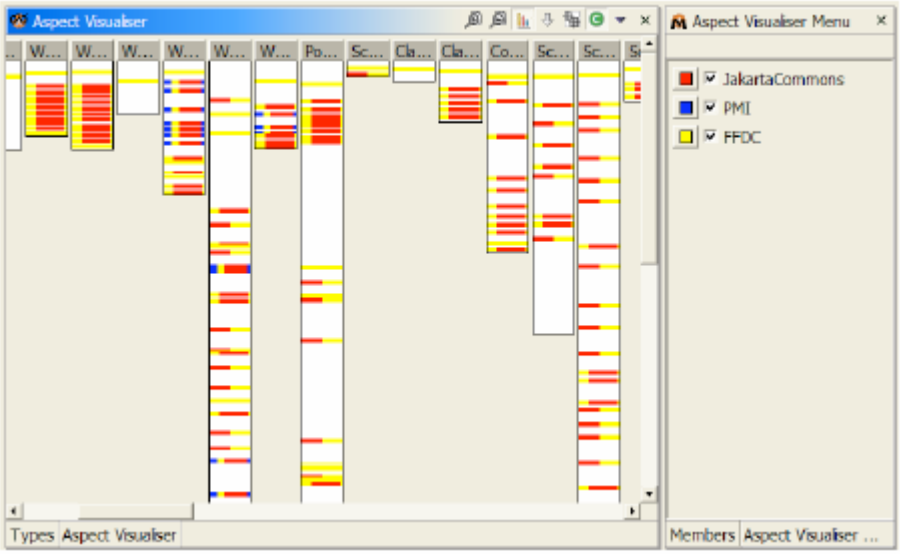

Figura 2.6: Visualizador de Aspecto, imagem extraída de Clement *et al* (2003)

A depuração encontrada no AJDT é utilizada não apenas para rastrear os problemas, mas também para observar a execução do programa passo a passo, a fim de que seu comportamento seja entendido.

2.7.3. Ferramenta de Mineração de Aspectos

A mineração de aspectos auxilia a identificação e extração de interesses transversais espalhados e entrelaçados ao longo do código-fonte. Existem duas categorias de mineração, de acordo com Hannemann e Kiczales (2001):

1. Mineração Baseada em Texto: utiliza técnicas de análise baseada em textos, tal como, o padrão de similaridade (*pattern matching*). Seu uso é recomendado quando existem convenções consistentes e rígidas de nomes de tipos, de métodos, de variáveis e de classes;
2. Mineração Baseada em Tipos: identificação de interesses transversais é realizada pela análise de tipos nos códigos-fonte, não sendo afetada pela convenção de nomes de objetos, de classes e de variáveis. O uso de tipos pode prover pistas a respeito de qualidade da modularização do projeto. Considerando que um sistema bem modular possua alta coesão e baixo acoplamento em seus subsistemas e módulos, o uso intenso de um tipo em um módulo indica alta coesão (forte dependência interna do módulo, visto que apenas poucos tipos

externos são usados) e baixo acoplamento (baixa dependência entre esse módulo com os demais).

A ferramenta de mineração de aspectos AMT – Aspect Mining Tool, desenvolvida por Hannemann e Kiczales (2001), é um arcabouço de análises multi-modal usado para a identificação de interesses transversais, provendo técnicas de análise baseada em buscas por texto e por tipo, além de ser extensível (possível de ser estendida / incrementada). Permite ao usuário definir consultas baseadas no uso de tipos e em expressões regulares, exibindo as linhas encontradas por meio de cores distintas. Caso uma linha satisfaça a vários critérios, sua visualização será separada em duas ou mais partes com cores distintas.

AMT utiliza uma versão modificada do compilador AspectJ (KICZALES *et al*, 2001) para extrair a informação de tipos para cada linha do código-fonte. A análise do código é realizada pela coleta de todas as informações relevantes a cada linha de código-fonte e, em seguida, o sistema mostra as informações extraídas graficamente. O usuário pode destacar as linhas que usam um determinado tipo, satisfazem a um padrão ou a combinação de ambos.

2.7.4. JQuery

JQuery (JAZEN e VOLDER, 2003) é uma ferramenta de navegação com uma linguagem expressiva de consulta lógica, combinando as vantagens de navegador hierárquico com a flexibilidade de uma ferramenta de consulta.

Um navegador hierárquico, do inglês *hierarchy browser*, é uma ferramenta que apóia a navegação baseada em tipos particulares de relacionamentos. Por exemplo, um navegador de hierarquia de classes permite a navegação ao longo de relacionamentos de herança, enquanto que o navegador de grafo de chamada possibilita a navegação dependências de chamadas estáticas. A interface típica de navegadores é uma árvore com nós que se expandem e retraem. À medida que os nós são expandidos, os sub-nós revelam outros elementos conectados com esse sub-nó por meio de uma relação específica.

A vantagem de navegadores hierárquicos é que fornecem um mapeamento explícito dos caminhos da navegação e o histórico da exploração do usuário é captado em uma coleção de nós que foram expandidos. Entretanto, esses navegadores são explorações especializadas e limitadas a tipos particulares de relacionamentos. Conseqüentemente, quando se deseja navegar pelo código por tipos de relacionamentos diferentes daqueles providos pelo navegador, o desenvolvedor é forçado a recorrer a diferentes navegadores. Como não há um modo ordenado para alternar entre diversos navegadores, o caminho de exploração fica dividido em fragmentos espalhados por diversas visões desconectadas. Com isso, os desenvolvedores perdem a rastreabilidade da situação atual em relação à tarefa de exploração.

Uma outra forma de exploração do código-fonte é por meio de linguagens de consulta que apresentam maior flexibilidade quanto à exploração de relacionamentos. É possível construir consultas usando combinações complexas de relacionamentos, permitindo também a extração de informações úteis e podem ser usadas com o propósito de construção de visões de navegação de código-fonte. Em geral, não é possível formular uma única consulta que encontre todas as informações desejadas de uma tarefa de navegação. Além disso, a formulação de várias consultas acarreta perda de rastreabilidade do caminho de exploração que conectam as consultas.

Jazen e Volder (2003) apresentam os seguintes benefícios de um navegador hierárquico combinado às características da ferramenta de consulta:

- Prover uma representação explicita dos caminhos de exploração pelo desenvolvedor do mesmo modo que um navegador hierárquico;
- Apoiar buscas diretas para subconjunto específico de elementos de um código de acordo com os critérios especificados pela consulta do mesmo modo que uma ferramenta de consulta;
- Apoiar exploração em termos de uma grande variação de relacionamentos entre unidades de código.

A ferramenta JQuery reduz a carga cognitiva associada com a exploração quando comparada a outros navegadores e ferramentas baseadas em consulta, da seguinte forma (JAZEN e VOLDER, 2003):

- Fornecendo representação de exploração de caminhos explícita e desfragmentada, ajudando a manter o senso de orientação dentro do contexto de uma tarefa de exploração;
- A flexibilidade da ferramenta para explorar grande variedade de diferentes relacionamentos e consultas com visão simples e integrada reduz a necessidade de se alternar entre visões e ferramentas.

2.7.5. AspectBrowser

Implementado como um *plug-in* para o ambiente Eclipse, AspectBrowser (GRISWOLD *et al* 2000) é uma ferramenta que permite a visualização de aspectos existentes no código-fonte, mediante a definição de uma expressão regular utilizando o padrão *grep* definido pelo usuário. Assim, buscas de trechos de código-fonte que satisfaçam essa expressão regular são realizadas automaticamente. Os resultados podem ser visualizados graficamente, assemelhando-se muito com a representação gráfica do da visão Aspect Visualization, contida no *plug-in* AJDT (ASPECTJ DEVELOPMENT TOOLS PROJECT, 2007).

Para cada expressão regular definida pelo usuário, um novo item (aspecto) na visão de árvore de aspectos é acrescentado e todas as suas ocorrências no código são visualizadas por linhas em barras verticais. Cada barra vertical representa um arquivo e as pequenas linhas que entrecortam as barras verticais correspondem às ocorrências no código que satisfazem a uma expressão regular.

2.7.6. HAM

Breu *et al* (2006) defendem que os aspectos emergem com o tempo e, para isso, propõem que a identificação de aspectos pode ser baseada no histórico do sistema de versões CVS (do inglês, *Concurrent Version System*), ao invés de apenas uma única versão de software como é observado nos métodos convencionais. O processo de identificação de indícios verifica a freqüência com que a chamada dos métodos é realizada e as mais freqüentes são consideradas candidatos a aspectos. Esse método foi desenvolvido em um protótipo denominado HAM.

2.7.7. AOP Migrator

Com o enfoque diferente das demais ferramentas apresentadas, AOP Migrator (BINKLEY *et al*, 2006) tem o objetivo de refatorar sistemas em Java de modo iterativo por meio de transformações com preservação de semântica, isto é, o código final resultante mantém seu comportamento original enquanto que sua estrutura interna é melhorada.

Para isso contempla o suporte para as refatorações de extração de: início/fim de método/tratamento de exceção, início/fim de chamada; condicional, pré-retorno, *wrapper* e tratamento de exceção. As atividades envolvidas na ferramenta AOP Migrator são divididas em quatro fases:

1. Descoberta: o código-fonte é analisado e as oportunidades de refatoração são determinadas. Para um mesmo trecho de código marcado, vários métodos de refatoração podem ser aplicados;
2. Transformações: transformações OO podem ser aplicadas para trechos de código marcados que não permitem a aplicação de nenhum método de refatoração, após a transformação do código, oportunidades de refatoração são verificadas novamente;
3. Seleção: a seleção do método de refatoração é aplicada quando um mesmo trecho de código marcado contém mais de um método de refatoração possível de ser aplicado. Uma escala de prioridades entre as refatorações disponíveis é sugerida por Binkley *et al* (2006), podendo ser usada para decidir qual refatoração aplicar;
4. Refatoração: transformação do código, preservando seu comportamento externo e centralização de aspectos.

O processo de migração acima usado pela ferramenta é repetido enquanto existir código “marcado”.

2.8. Considerações Finais

Neste capítulo foram apresentados os conceitos e fundamentos necessários para a elaboração do projeto de pesquisa desta dissertação e discutido em Seções posteriores.

Além disso, as discussões contidas no Capítulo 3 requerem o entendimento dos conceitos fundamentais da Programação Orientada a Aspectos, da linguagem *AspectJ*, de suas construções, de noções de manutenção de software, de reengenharia e da abordagem Aspecting. Por outro lado, a Seção 2.7 apresentou ferramentas úteis à POA cujos conceitos serão empregados em discussões do Capítulo 4 com comparações de algumas ferramentas com o protótipo de apoio computacional para a reengenharia de interesses transversais especificado e implementado neste trabalho.

As ferramentas discutidas neste Capítulo e que contribuem para a realização de atividades de reengenharia de sistema legado OO para outro equivalente em OA são comparadas no Quadro 2.2. A primeira coluna, Apoio à Identificação, indica quais ferramentas podem ser usadas para a identificação de possíveis candidatos a aspectos. A coluna de Modificação de Código informa quais ferramentas auxiliam para realização de alterações de código-fonte. Algumas ferramentas dependem de pré-requisitos informados na coluna de Dependência. O símbolo de hífen (-) indica que a informação é desconhecida.

Quadro 2.2 – Comparativo de ferramentas relacionadas à reengenharia de sistemas OO para OA

Ferramenta	**Apoio à Identificação**	**Modificação de Código**	**Integração com IDE**	**Dependência**	**Interação**
AMT	Sim	Não	Não	JDK 1.3	Comandos de linha e GUI
JQuery	Sim	Não	Sim	-	GUI e consulta por comandos
Aspect Browser	Sim	Não	Sim	-	GUI
HAM	Sim	Não	Não	-	-
AOP Migrator	Não	Sim	Sim	JVM 1.4.2 Eclipse 3.0.2 AJDT 1.2.0	GUI

Capítulo 3

Modelo de Indícios

3.1. Considerações Iniciais

A abordagem Aspecting (RAMOS *et al*, 2004) permite a reengenharia de um sistema legado em Java para um novo sistema em AspectJ, podendo tornar o sistema mais modular, legível e manutenível. Para isso, Aspecting requer a execução de grande quantidade de diretrizes de modelagem e de implementação do novo sistema orientado a aspectos, inviabilizando sua aplicação para sistemas legados de médio a grande porte.

Por esse motivo, estudos e experimentos envolvendo a abordagem Aspecting foram realizados com o intuito de observar meios de viabilizar essa abordagem e estender a Lista de Indícios, considerando novos interesses transversais. Além disso, revisões na abordagem Aspecting foram realizadas, tendo como objetivo a representação de indícios de forma mais facilitada para que novas ferramentas de identificação de indícios possam ser construídas baseando-se nessa abordagem. Essa representação é denominada de Modelo de Indícios.

Este capítulo aponta algumas limitações da abordagem Aspecting, sugerindo possíveis melhorias, por meio da extensão da Lista de Indícios e alteração de seu

mecanismo de identificação de indícios na Seção 3.2. Uma nova representação, o Modelo de Indícios, é proposta na Seção 3.3 e os interesses transversais de persistência em banco de dados, em memória temporária e *logging* são expressos segundo esse modelo. Em seguida, na Seção 3.4, diretrizes de identificação de indícios e de reestruturação de código-fonte, empregando o Modelo de Indícios, são apresentadas e detalhadas. Por fim, na Seção 3.5 são apresentadas as considerações finais deste capítulo.

3.2. Aprimoramento da Abordagem Aspecting

Para melhorar o processo reengenharia de sistemas legados em Java para novos sistemas em AspectJ da abordagem Aspecting, estudos e experimentos foram conduzidos e a necessidade de evoluções da abordagem foi constatada.

A Lista de Indícios originalmente proposta na abordagem Aspecting (RAMOS, 2004) apresenta uma característica em comum nas gramáticas livre de contexto para os interesses de persistência de banco de dados e de persistência em memória temporária: a utilização de tipos (classes e interfaces) de pacotes específicos da API do JDK, Java 2™ Platform Standard Edition Development Kit (JAVA TECHNOLOGY, 2007). Para que esses tipos possam se utilizados corretamente de forma computacional, instruções de importação desses tipos devem ser incluídas. Dessa forma, a ocorrência dessas instruções também se torna indício e deve ser incluída na Lista de Indícios.

A atualização da Lista de Indícios para os interesses de persistência em banco de dados e em memória temporária ocorreu, pois há dependência da Lista de Indícios com alguns tipos existentes na API do JDK. Além disso, novas versões do JDK podem disponibilizar novos tipos que substituem, complementam ou operam de modo independente em relação aos tipos antigos. Nas próximas subseções, as propostas para a atualização da Lista de Indícios são apresentadas para os interesses de persistência em banco de dados, de persistência em memória temporária e de *logging*.

3.2.1. Persistência em banco de dados

A persistência em banco de dados refere-se a comandos da linguagem SQL e a objetos específicos da API do JDK que fazem a conexão com o banco de dados e executam comandos em linguagem SQL (RAMOS, 2004). A gramática livre de

contexto para indícios de persistência em banco de dados apresentada no Quadro 2.1 do Capítulo 2, contém os tipos `Connection`, `PreparedStatement` e `ResultSet` pertencentes ao pacote `java.sql`. Pela documentação desse pacote pôde-se observar a existência de outros tipos que também podem ser utilizados na implementação do interesse de persistência em banco de dados, tais como: `Array`, `Blob`, `CallableStatement`, `Clob`, `DatabaseMetaData`, `Driver`, `ParameterMetaData`, `Driver`, `ParameterMetaData`, `Ref`, `ResultSetMetaData`, `SavePoint`, `SQLData`, `SQLInput`, `SQLOutput`, `Statement`, `Struct`, `Date`, `DriveManager`, `DriverPropertyInfo`, `SQLPermission`, `Time`, `Timestamp` e `Types`. Dessa forma, a atualização da Lista de Indícios consiste da adição de instruções de declaração desses tipos e de suas instruções de importação, pertencentes ao pacote `java.sql`.

A existência de indício de persistência em banco de dados também pode ser verificada por meio de comandos em linguagem SQL, denotados <SQL> no código fonte (RAMOS *et al* 2004). Assim, as operações mais comuns nesse caso são: inserção, atualização, remoção e consulta a dados. Dessa forma, a gramática livre de contexto original da Lista de Indícios foi complementada como apresentada na Figura 3.1.

```
<SQL> = 'insert into ' <substring> |
            'update ' <substring>|
            'delete from ' <substring>|
            'select ' <substring>
```

Figura 3.1: Gramática livre de contexto que complementa a Lista de Indícios original

3.2.2. Persistência em memória temporária

Este interesse, também conhecido por *buffering*, é responsável pelo armazenamento e manipulação de dados em memória temporária enquanto tais dados são processados. Esse interesse é freqüentemente confundido com *Caching*, que consiste no armazenamento de dados ou instruções que são acessadas com grande freqüência, permitindo uma considerável melhora de desempenho. Na Lista de Indícios original, os indícios de persistência em memória temporária estão relacionados à declaração e à alocação de objetos do tipo `BufferedReader`, `StringBuffer` e `BufferedInputStream`. O primeiro e terceiro tipo podem ser utilizados no código-fonte mediante a importação do pacote `java.io` da API do JDK e o tipo `StringBuffer` é um tipo pertencente ao pacote java.lang.

Segundo Deursen *et al* (2003), o tipo `FileOutputStream`, pertencente ao pacote `java.io`, é utilizado para a implementação de persistência em memória temporária. Nesse sentido, a atualização dos indícios de persistência em memória temporária contempla o acréscimo das declarações e das alocações de `FileOutputStream` e de instruções de importação desse tipo.

3.2.3. Logging

Logging está associado à coleta de informações referentes aos eventos ocorridos durante a execução de um sistema, em geral, associados a uma data e/ou hora, podendo ser armazenados em arquivos. Essas informações podem ser utilizadas para testes ou para diagnosticarem falhas de sistema. Devido às características de *logging,* as instruções desse interesse apresentam-se espalhadas pelo vários módulos do sistema e entrelaçadas com outros interesses, caracterizando-o como um interesse transversal. Este interesse não pertencia a Lista de Indícios original.

A partir da documentação da API do JDK (JAVA, 2007), verificou-se a existência do pacote `java.util.logging`, que disponibiliza uma série de classes e interfaces que possibilitam a implementação do interesse de *logging* de um sistema. A utilização de *logging* por uma aplicação pode ser feita por chamadas a partir de objetos da classe `Logger`, especificado pelo pacote `java.util.logging`. Por meio desses objetos, outros objetos `LogRecord` são alocados e passados por parâmetros para objetos da classe `Handler` para escrita de informações de *logging* em um dispositivo de entrada e saída (E/S). Os objetos `Logger` e `Handler` podem utilizar objetos da classe `Level` e opcionalmente filtros, por meio de objetos `Filter` conforme mostra a Figura 3.2. Desse modo, é possível determinar quais objetos `LogRecord` terão suas informações associadas a mensagens de *logging*. Objetos `Formatter` podem ser usados para localizar e formatar mensagens antes de sua escrita em um dispositivo de E/S.

Outra proposta de atualização da Lista de Indícios consiste na adição do interesse de *logging*, conforme mostra a gramática livre de contexto, Figura 3.3. Necessariamente, o código com o indício de *logging* deverá conter a importação de todos ou de alguns tipos existentes no pacote, utilizando respectivamente a instrução `import java.util.logging.*` ou instruções `import java.util.logging.`<Nome do Tipo>, onde <Nome do Tipo> pode ser qualquer tipo existente do pacote `java.util.logging`. Por esse motivo, o operador lógico "E", representado pelo

caractere "&" é inserido, expressando a obrigatoriedade na presença da importação desses tipos, como pode ser visto na Figura 3.3.

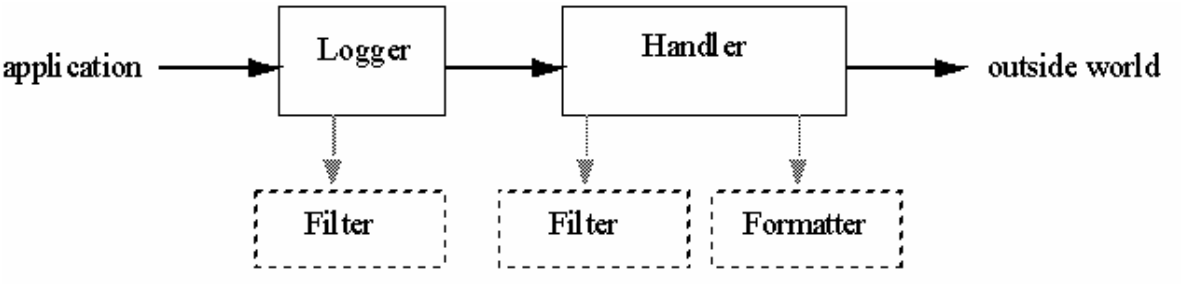

Figura 3.2: Utilização de *Logging* por uma aplicação

```
<i.Logging> =
  'import' 'java.util.logging'.('*' | <tipo.Logging>) &
  (<statements> <tipo.Logging> <statements>)*

<tipo.Logging> =
  'Filter' | 'ConsoleHandler' | 'ErrorManager' |
  'FileHandler' | 'Formatter' | 'Handler' | 'Level' |
  'Logger' | 'LoggingPermission' | 'LogManager' |
  'LogRecord' | 'MemoryHandler' | 'SimpleFormatter' |
  'SocketHandler' | 'StreamHandler' | 'XMLFormatter'
```

Figura 3.3: Gramática livre de contexto proposta para indícios de *Logging*

A linguagem Java permite a utilização de tipos com o mesmo nome desde que pertençam a pacotes diferentes. Isso permite que falsos indícios sejam selecionados. Por exemplo, considerando a implementação de um sistema de *desktop* que faça sincronização de dados com um dispositivo embarcado, o tipo `Connection` pode estar associado às conexões estabelecidas com esse dispositivo. Com a execução da identificação de indícios, os trechos de códigos-fonte com declarações do tipo `Connection` serão reconhecidos como falsos indícios de persistência de banco dados. Por isso, ao definir a gramática livre de contexto para indícios de *logging*, houve a preocupação em expressar a presença das instruções de importação de tipos.

Outra maneira de contornar a identificação desses falsos indícios consiste na associação de tipos aos seus pacotes, permitindo diferenciar tipos distintos com mesmo nome. Dessa forma, a precisão dos indícios identificados é melhorada. A vinculação das informações de tipos e de seus pacotes poderia ser inserida na Lista de Indícios, Figura 3.3, porém infere-se que suas gramáticas podem se tornar mais confusas, dependendo da

quantidade de pacotes. Na Seção 3.3, uma nova forma de representação dos indícios é apresentada.

3.2.4. Mecanismo de buscas léxicas e sintáticas

As gramáticas livres de contexto apresentadas para os indícios discutidos anteriormente são capazes de identificar instruções de declarações de objetos de determinados tipos. Entretanto, ao se realizar experimentos com a abordagem Aspecting, as instruções que utilizam os objetos declarados com esses tipos não são consideradas indícios. Além disso, a representação utilizada para a Lista de Indícios não é capaz de expressar uma estratégia de busca que considere as instruções com objetos, cuja declaração é reconhecida como um indício. Essa restrição acontece porque a Lista de Indícios sugere a execução de buscas léxicas por indícios. Nessas buscas, simplesmente é feita uma varredura pelo código a procura de um conjunto / padrão de caracteres, que inclui os caracteres pertencentes às palavras reservadas da linguagem, comentários, nomes e valores de variáveis, entre outros. O resultado gerado pode exibir informações irrelevantes e pouco precisas. Em uma situação hipotética realizando a busca léxica pelo tipo `Connection`, é possível que existam comentários, nomes de variáveis, mensagens e conteúdos de cadeias de caracteres com a palavra `Connection`, sendo associados a indícios, conforme é ilustrada na seção seguinte. Na verdade, esses resultados não correspondem realmente a indícios, por não constituírem uma instrução de declaração de objetos do tipo `Connection`. Essa é uma das características que difere nossa abordagem automatizada de outras existentes (GRISWOLD *et al*, 2000).

Por outro lado, utilizando a análise sintática do código-fonte, a manipulação de uma Árvore Abstrata Sintática (do inglês, *abstract syntax tree* (AST)) se torna possível, permitindo elicitar detalhes adicionais, além dos textuais, permitindo uma filtragem melhor dos resultados obtidos no processo de identificação de indícios. Ao invés de pesquisar por padrões textuais no código fonte, agora com a análise da AST as construções da linguagem são conhecidas e a pesquisa de declarações de objetos de tipo `Connection` não conduzem a resultados indesejados, pois apenas declarações de objetos são selecionadas.

3.2.5. Exemplificação dos aprimoramentos realizados na abordagem Aspecting

Empregando a gramática livre de contexto existente inicialmente na abordagem Aspecting, exibida no Quadro 2.1, Capítulo 2, e observando o código apresentado na Figura 3.4, pode-se notar que apenas a instrução (1) e a declaração do atributo `connection` seriam considerados indícios. Pelo que foi apresentado nas Seções anteriores, a instrução (2) deveria ser incluída na Lista de Indícios por apresentar a declaração de uma variável do tipo `Statement`, (Seção 3.2.1). As instruções (3) e (4), por utilizar uma variável do tipo `Statement`, também deveriam ser adicionadas ao conjunto de indícios encontrados (Seção 3.2.4). Por fim, a instrução (5) deve ser considerada um indício por realizar uma chamada ao método `commit`, a partir de um atributo do tipo `Connection`, de acordo com a Seção anterior. Além disso, a identificação realizada a partir da Lista de Indícios acrescentaria como indícios a palavra `Connection`, presente no comentário do código Java apresentado na Figura 3.4, localizado acima da declaração do atributo. Nesse caso, constata-se um falso indício, pois trechos de comentários não são aspectos. Esse resultado é gerado devido análise léxica implícita na Lista de Indícios.

```
public class Account {

  // Tipo Connection usado para estabelecer conexão com banco de dados
  private Connection connection;

  public void save() {
    try {
        String sql = "UPDATE ACCOUNTS SET CLIENT_ID = " + clientId +
                (1) ", ACTUAL_TYPE = '" + type + "', ACTUAL_VALUE = " + value +
                ", DESCR = '" + descry + "' WHERE ID = " + id;

                (2) Statement stm = this.con.createStatement();
                (3) stm.executeUpdate(sql);
                (4) stm.close();
                (5) this.connection.commit();
            }
            catch (SQLException e) {
                   System.out.println("Exception : " + e.toString());
    }
  }
}
```

Figura 3.4: Trechos de código com indícios de interesse de persistência em banco de dados com as novas diretivas propostas neste trabalho

3.3. Modelo de Indícios

A melhoria do processo de identificação de indícios pode ser viabilizada pela utilização de um mecanismo baseado em análise sintática do código-fonte. Existem ferramentas capazes de construir e exibir uma AST graficamente, além de proporcionar pesquisas por essa representação (AST VIEW, 2007). Essas ferramentas são necessárias devido à inviabilidade da construção e exibição de uma AST de forma manual.

A Lista de Indícios, que sugere buscas léxicas pelo código-fonte, deve ser substituída por uma outra representação que, além de armazenar os tipos característicos de indícios, também associe os tipos aos seus pacotes, permitindo a identificação única de cada tipo utilizado no sistema. Essa nova representação, denominada de Modelo de Indícios, deve facilitar as buscas por análise sintática pelo código para que novas ferramentas possam implementar o mecanismo de identificação apresentado na Seção 3.4.1. Nesta Seção, o Modelo de Indícios é apresentado e pode ser aplicado para representar as informações dos indícios que dependem de tipos existentes na API do JDK. Neste trabalho foram tratados os indícios de persistência em banco de dados, persistência em memória temporária e *logging*.

O Modelo de Indícios, mostrado na Figura 3.5, foi elaborado como um diagrama de classes. `Indication` é a classe principal e representa uma categoria de indícios, isto é, um conjunto de indícios que se referem a um mesmo interesse transversal. Cada categoria de indícios deve conter ao menos um pacote, representado por `IndicationPackage`. Assim como ocorre com a persistência em memória temporária, uma categoria de indícios pode conter diversos pacotes. Como discutido previamente, os tipos devem estar associados aos seus pacotes, de modo que tipos distintos de mesmo nome não sejam confundidos. Além disso, os tipos podem ser classes ou interfaces, correspondendo respectivamente a `IndicationClass` e `IndicationInterface`. As classes de exceção, `IndicationException`, são classes especializadas do tipo `Exception` e usadas no tratamento de exceção das operações realizadas por objetos com indícios. Um exemplo típico de classe de exceção é a classe `SQLException`. Mesmo pertencendo ao interesse transversal de tratamento de exceção, essa classe é usada quando problemas nas operações com o banco de dados ocorrem e requerem tratamento por parte da aplicação. Por esse motivo, `SQLException` também é parte integrante do interesse transversal de persistência em banco de dados. As classes recentemente

discutidas possuem um nome que as identificam e uma descrição opcional, permitindo documentar ou fazer anotações a respeito da informação registrada.

A classe `MatchText`, Figura 3.5, armazena algumas regras de buscas que suportam identificações de padrões de cadeias de caracteres <SQL>, mostradas na Figura 3.1. O atributo `target` indica qual construção alvo da AST que deve ser comparada com uma ou mais palavras especificadas no atributo `words`. Por meio do atributo `rule`, a comparação pode considerar as palavras encontradas apenas no início, final ou meio do texto da construção alvo e a diferenciação de letras maiúsculas e minúsculas pode ocorrer de acordo com o valor contido em `caseSensity`. As informações da Figura 3.1 podem ser expressas da seguinte maneira: `target` é qualquer conteúdo armazenado em uma variável de cadeia de caracteres, as palavras a serem buscadas são "insert into", "update", "delete from" e "select", `caseSensity` está com o valor “true” e `rule` define que as palavras encontradas no início dos conteúdos de variáveis de cadeia de caracteres são indícios.

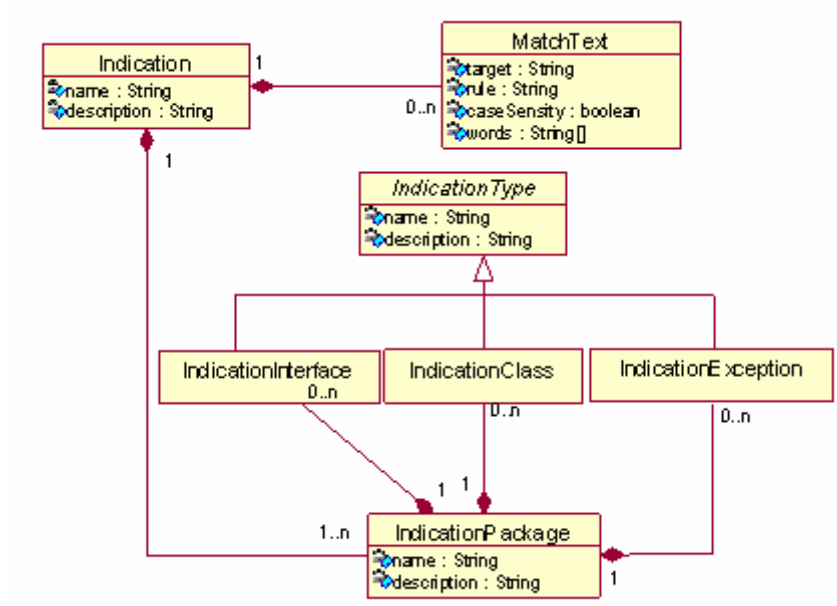

Figura 3.5: Modelo de Indícios

3.3.1. Modelo de Indícios para persistência em banco de dados

O uso do Modelo de Indícios para persistência em banco de dados é exibido por meio das Tabelas 3.1, 3.2 e 3.3. Na Tabela 3.1, aparecem os dados da categoria de indícios de persistência em banco de dados relacionados com o pacote `java.sql`. As duas primeiras colunas se referem à classe `Indication` e as duas últimas à classe

`IndicationPackage`. Nesse caso em especial, apenas o pacote `java.sql` está associado à categoria de indícios. Entretanto, dois ou mais pacotes podem estar relacionados com uma mesma categoria de indícios. Esse pacote possui 7 classes, 18 interfaces e 4 classes de exceção representadas no Modelo de Indícios respectivamente pelas Tabelas 3.3, 3.4 e 3.5. Na Tabela 3.2, dados de `MatchText` são mostrados para o indício de persistência em banco de dados. As descrições do pacote, de classes, de interfaces, de classes de exceção foram obtidas da documentação da API do JDK (JAVA TECHNOLOGY, 2007).

Tabela 3.1 – Informações de `Indication` e `IndicationPackage` para persistência em banco de dados

Indício	Descrição do Indício	Pacote	Descrição do Pacote
Persistência em Banco de Dados	Indicação de persistência em BD usando pacotes java.sql e regras de busca.	java.sql	Disponibiliza a API para acesso e processamento de dados armazenados em uma fonte de dados (geralmente uma base de dados relacional) usando JavaTM.

Tabela 3.2 – Informações de `MatchText` para persistência em banco de dados

Indício	Descrição do Indício	Alvo	Case sensitive	Regra	Palavras
Persistência em Banco de Dados	Indícios de persistência em BD usando pacotes java.sql e regras de busca.	Nome de variável	Não	Contém	sql
		Cadeia de caracteres	Não	Iniciam com	insert into
					select
					update
					delete from

Na Tabela 3.3, os atributos `name` e `description` da classe `IndicationClass`, terceira e quarta colunas, respectivamente, estão relacionados ao pacote `java.sql` apresentados na primeira e segunda coluna dessa tabela. Os dados referentes às interfaces `IndicationInterface` e às classes de exceção `IndicationException` são apresentados nas Tabelas 3.4 e 3.5. Como `IndicationClass`, `IndicationInterface` e `IndicationException` herdam de `IndicationType`, os formatos das Tabelas 3.3, 3.4 e 3.5 são similares.

Tabela 3.3 – Informações de `IndicationClass` associadas com `IndicationPackage` para persistência em banco de dados

Pacote	Descrição do Pacote	Nome	Descrição
java.sql	Disponibiliza a API para acesso e processamento de dados armazenados em uma fonte de dados (geralmente uma base de dados relacional) usando JavaTM.	Date	Um encapsulamento sutil do valor em milissegundos que permite JDBC identificá-lo como um valor SQL DATE.
		DriverManager	Serviço básico para o gerenciamento de um conjunto de *drivers* JDBC.
		DriverPropertyInfo	Propriedades de *driver* usado para estabelecer conexão.
		SQLPermission	Permissão para que objeto `SecurityManager` verifique quando o código executando em uma *applet* chama o método `DriverManager.setLogWriter` ou o método `DriverManager.setLogStream`.
		Time	Um encapsulamento sutil da classe `java.util.Date` que permite a API do JBDC identificá-lo como um valor SQL TIME.
		Timestamp	Um encapsulamento sutil de `java.util.Date` que permite a API do JDBC API identificá-lo como um valor SQL TIMESTAMP.
		Types	A classe que define as constantes usadas para identificar tipos SQL genéricos, denominados de tipos JDBC.

Tabela 3.4 – Informações de `IndicationInterface` associadas com `IndicationPackage` para persistência em banco de dados

Pacote	Descrição do Pacote	Interface	Descrição da Interface
java.sql	Disponibiliza a API para acesso e processamento de dados armazenados em uma fonte de dados (geralmente uma base de dados relacional) usando JavaTM.	Array	O mapeamento na linguagem de programação Java para o tipo ARRAY em SQL.
		Blob	A representação (mapeamento) na linguagem de programação Java para um valor BLOB em SQL.
		CallableStatement	A interface usada para executar *stored procedures* em SQL.
		Clob	O mapeamento na linguagem de programação Java para o tipo CLOB em SQL.
		Connection	Uma conexão com um banco de dados específico.
		DatabaseMetaData	Informação compreensiva sobre o banco de dados como u m todo.
		Driver	A interface que cada classe de *driver* deve implementar.

java.sql	Disponibiliza a API para acesso e processamento de dados armazenados em uma fonte de dados (geralmente uma base de dados relacional) usando JavaTM.	ParameterMetaData	Um objeto que pode ser usado para obter informação de tipos e propriedades de parâmetros em um objeto PreparedStatement.
		PreparedStatement	Um objeto que representa uma instrução em SQL pré-compilada.
		Ref	O mapeamento na linguagem de programação Java de um valor REF em SQL, que é referenciado para um objeto estruturado referente ao banco de dados.
		ResultSet	Uma tabela de dados representando um conjunto de dados de um banco de dados, que é gerado pela execução de uma instrução de consulta no banco de dados.
		ResultSetMetaData	Um objeto que pode ser usado para obter informação sobre tipos e propriedades das colunas em um objeto de ResultSet.
		Savepoint	A representação de um ponto dentro de uma transação que pode ser referenciada do método Connection.rollback.
		SQLData	A interface usada para o mapeamento de um tipo em SQL definido pelo usuário para uma classe na Linguagem de Programação Java.
		SQLInput	Uma entrada que contém um fluxo de valores representando uma instância de um tipo estruturado em SQL ou de um tipo SQL distinto.
		SQLOutput	O fluxo de saída para a escrita de atributos de um tipo definido pelo usuário para o banco de dados.
		Statement	Objeto usado para a execução de uma instrução SQL estática, retornando os resultados por ele produzido.
		Struct	O mapeamento padrão na linguagem de programação Java para um tipo estruturado em SQL.

Tabela 3.5 – Informações de `IndicationException` associadas com `IndicationPackage` para persistência em banco de dados

Pacote	Descrição do Pacote	Exceção	Descrição da Exceção
java.sql	Disponibiliza a API para acesso e processamento de dados armazenados em uma fonte de dados (geralmente uma base de dados relacional) usando JavaTM.	BatchUpdateException	Uma exceção lançada quando um erro ocorre durante uma operação de atualização em lote.
		DataTruncation	Uma exceção que reporta um aviso DataTruncation ou lança uma exceção DataTruncation quando o JDBC inesperadamente trunca um valor.
		SQLException	Uma exceção que disponibiliza informação de um erro de acesso de um banco de dados ou outros erros.
		SQLWarning	Uma exceção que prove informação sobre avisos de acesso de banco de dados.

3.3.2. Modelo de Indícios para persistência em memória temporária

O Modelo de Indícios para o interesse de persistência em memória temporária contém a organização apresentada nas Tabelas 3.6 e 3.7. Como pode ser observado, ele está relacionado com a utilização de algumas classes dos pacotes `java.lang` e `java.io`. Uma particularidade, nesse caso, é ausência de interfaces e de classes de exceção na composição dessa categoria de indícios. Da mesma forma que ocorreu com o Modelo de Indícios para persistência em banco de dados, as descrições dos pacotes e de classes foram obtidas da documentação da API do JDK (JAVA TECHNOLOGY, 2007).

Tabela 3.6 – Informações de `Indication` e `IndicationPackage` para persistência em memória temporária

Indício	Descrição do Indício	Pacote	Descrição do Pacote
Persistência em memória temporária	Indício de persistência em memória temporária.	java.lang	Disponibiliza classes fundamentais para programação na linguagem Java.
		java.io	Disponibiliza entradas e saídas dos sistemas por meio de fluxo de dados, serialização e sistema de arquivos.

Tabela 3.7 – Informações de `IndicationClass` associadas com `IndicationPackage` para persistência em memória temporária

Pacote	Classe	Descrição da Classe
java.lang	StringBuffer	Uma seqüência mutável de caracteres protegidas por Thread.
java.io	BufferedInputStream	Adiciona funcionalidade para outro fluxo de entrada, a habilidade de armazenar a entrada e suportar métodos de `mark` e `reset`.
	BufferedReader	Lê textos de um fluxo de entrada de caracteres, armazenando caracteres do mesmo modo que provê leituras eficientes de caracteres, arrays e linhas.
	FileOutputStream	Um fluxo de saída de arquivo para escrita de dados para arquivos propriamente ditos ou descritores de arquivos.

3.3.3. Modelo de Indícios para *logging*

O Modelo de Indícios para o interesse transversal de *logging* contém apenas o pacote `java.util.logging` que é composto por 15 interfaces e 2 classes. A descrição do indício e do pacote no Modelo de Indícios é apresentada na Tabela 3.8. As demais informações para interfaces e classes de *logging* foram relatadas nas Tabelas 3.9 e 3.10 respectivamente. As descrições dos pacotes, de classes e de interfaces também foram obtidas a partir da documentação da API do JDK (JAVA TECHNOLOGY, 2007).

Tabela 3.8 – Informações de `Indication` e `IndicationPackage` para *logging*

Indício	Descrição do Indício	Pacote	Descrição do Pacote
Logging	Indícios de logging encontradas em aplicações.	java.util.logging	Disponibiliza classes e interfaces das facilidades de logging existentes no JDK.

Tabela 3.9 – Informações de `IndicationClass` associadas com `IndicationPackage` para *logging*

Pacote	Classe	Descrição da Classe
java.util.logging	ConsoleHandler	Este manipulador serve para publicar registros para System.err.
	ErrorManager	Pode ser anexado a manipuladores para processar qualquer erro que ocorra em um manipulador durante o *logging*.
	FileHandler	Manipulador de logging de arquivo.
	Formatter	Um formatador que prove suporte para formatação de registros de *logging*.
	Handler	Um manipulador que exporta mensagens de *logging*.
	Level	Define um conjunto de níveis padrão de *logging* que podem ser usados para controlar a saída de *logging*.
	Logger	Usado para registrar mensagens em um sistema específico ou componente de aplicação.
	LoggingPermission	A permissão que checa o código sendo executado por meio de chamadas de métodos de controle de *logging* a partir de objetos SecurityManager.
	LogManager	Usada para manter um conjunto de estados compartilhados referentes a serviços de *logging*.
	LogRecord	Usado para passar solicitações de *logging* entre o *framework* de *logging* e manipuladores individuais de *logging*.
	MemoryHandler	Armazenam requisições em um *buffer* circular em memória.
	SimpleFormatter	Imprimem um breve sumário do registro de *logging* em um formato humanamente legível.
	SocketHandler	Simples *logging* para rede.
	StreamHandler	Fluxo baseado em *logging*.
	XMLFormatter	Formata um registro de *logging* em um padrão de XML.

Tabela 3.10 – Informações de `IndicationInterface` associadas com `IndicationPackage` para *logging*

Pacote	Interface	Descrição da Interface
java.util.logging	Filter	Um filtro pode ser usado para prover um pequeno controle sobre o que é logado, além dos controles providos pelos níveis de *logging*.
	LoggingMXBean	A interface de gerenciamento para facilidades de *logging*.

3.4. Processo de Reestruturação de Interesses Transversais em Sistemas Legados

Nesta seção discute-se como a reestruturação de interesses transversais em sistemas legados é realizada para persistência em banco de dados, *logging* e persistência em memória temporária. Todo o processo dessa atividade é resumido pelos passos do algoritmo apresentado na Figura 3.6. Como se pode observar a iteratividade do processo depende da quantidade de categorias de indícios sendo analisada e do número de indícios encontrado no código-fonte.

```
1  Escolha do projeto p do sistema legado;
2  Conversão de p escrito em Java para AspectJ;
3  Para cada Categoria de Indícios ci faça
4    identificarIndicio(p, ci);
5      Para cada Indício i encontrado faça
6        Se i é aspecto então
7          Reestruturar(p, i);
8        Senão
9          Ignorar indício;
10       Fim-Se
11     Fim-Para
12 Fim-Para
```

Figura 3.6: Algoritmo em alto nível da reestruturação de interesses transversais

O processo é iniciado pela seleção do projeto contendo os códigos-fonte do sistema legado implementado na linguagem Java. Em seguida, deve-se realizar a conversão do projeto selecionado para a linguagem AspectJ, se necessário, com apoio do ambiente de desenvolvimento que estiver sendo usado.

O desenvolvedor tem que decidir quais categorias de indícios deve reestruturar e qual a ordem em que cada uma é reestruturada. Assim, seleciona-se uma das categorias de indícios e, em seguida, aplicam-se diretrizes de identificação de indícios, discutidas com mais detalhes na Seção 3.4.1. Nessa etapa, um conjunto de indícios está disponível e o desenvolvedor deve verificar para cada um se realmente trata-se de um aspecto ou de um falso indício. Sendo um aspecto, as diretrizes de reestruturação do aspecto são executadas (linha 7). Caso contrário, o falso indício encontrado deve ser ignorado.

3.4.1. Diretrizes para identificação de indícios

Para realizar a identificação de indícios nos módulos do sistema é necessária a execução dos passos do algoritmo da Figura 3.7, que mostram a iteração dos pacotes do sistema e de arquivos escritos em Java, contidos nesses pacotes. Para cada um dos arquivos analisados deve-se determinar quais tipos associados à categoria de indícios são buscados para, em seguida, efetivamente reconhecê-los.

```
1  Para cada pacote P ⊂ projeto faça
2    Para cada arquivo Java A ⊂ P faça
3      buscarTipos(A);
4      identificarIndicios(A);
5    Fim-Para
6  Fim-Para
```

Figura 3.7: Iteração do processo de identificação por pacotes e arquivos

A atividade `buscarTipos` é detalhada no algoritmo apresentado na Figura 3.8. A linguagem Java permite a importação de outros tipos de pacotes fora o corrente, que contém o arquivo Java que está sendo analisado. A cada iteração, cada uma das instruções de importação é referenciada por `Pi` para, posterior, comparação com cada pacote associado à categoria de indícios, `Pc`. Caso ambos os pacotes sejam iguais, deve-se verificar se o pacote do tipo importado possui o wildcard (*), isto é, todos os tipos desse pacote foram importados. Se isso ocorrer, todos os tipos associados ao pacote da categoria de indícios (`Pc`) devem ser anotados. Caso contrário, o tipo importado é comparado com cada um dos tipos do pacote da categoria de indícios. Quando se verifica a ocorrência do tipo importado em `Pc`, esse tipo é anotado e associado ao pacote e ao arquivo analisado e referenciado como `Pc.A`, sendo `Pc` o pacote da categoria de indício que está sendo analisada e `A` é o arquivo corrente.

```
1  Para cada pacote da importação de tipo Pi faça
2    Para cada pacote da categoria de indício Pc faça
3      Se Pi = Pc então
4        Se Pi possui wildcard (*) então
5          Para cada tipo T ⊂ Pc faça
6            Anotar T, associado à Pc.A;
7          Fim-Para
8        Senão
9          Para cada tipo T ⊂ Pc faça
10           Se T = Ti ⊂ Pi então
11             Anotar T, associado à Pc.A;
12           Fim-Se
13         Fim-Para
14       Fim-Se
15     Fim-Se
16   Fim-Para
17 Fim-Para
```

Figura 3.8: Determinação de tipos a serem analisados no código-fonte

O procedimento até aqui realizado é para buscar tipos que devem ser analisados no código fonte. Deve-se agora identificar os indícios e para isso os passos do algoritmo apresentado na Figura 3.9 devem ser realizados. Caso nenhuma anotação de tipos tenha sido realizada, deduz-se que não existiam indícios no arquivo analisado.

O processo de identificação dos indícios pelo código-fonte se aplica à definição de qualquer número de tipos (interfaces ou classes) existente no arquivo. Em cada um dos tipos contidos no arquivo, todos os atributos são analisados. Caso o atributo seja declarado com um dos tipos anotados previamente em `Pc.A`, a declaração do atributo é marcada e seu nome é anotado em uma lista de nomes, `Ln`. Todos os métodos de cada tipo são também verificados, iniciando-se pela lista de parâmetros. Cada um dos parâmetros tem seu tipo comparado com os tipos anotados em `Pc.A` e caso seu tipo esteja incluído nas anotações, esse parâmetro é marcado e seu nome é incluído à lista de nomes (linha 12). Em seguida, cada uma das instruções existentes nos corpos de métodos é avaliada. Caso a instrução trate de uma declaração deve ser observado se o tipo utilizado está contido nas anotações de `Pc.A` (linha 17). Em caso positivo, a instrução é marcada e o nome do objeto empregado na declaração é adicionado à lista de nomes `Ln`. As demais instruções que não expressam declarações de objetos são analisadas, pois caso exista a utilização de algum objeto na instrução, o nome desse objeto é procurado na lista de nomes `Ln`. As instruções contendo nomes de objetos encontrados nessa lista são marcadas.

```
1  Para cada tipo t faça
2    Para cada atributo at ∈ t faça
3      Se at.tipo ∈ tipos anotados então
4        Marcar(at);
5        Anotar at.nome na lista de nomes Ln;
6      Fim-Se
7    Fim-Para
8    Para cada método m ∈ t faça
9      Para cada parâmetro m.par_i ∈ m faça
10       Se m.par_i ∈ tipos anotados então
11         Marcar(m.par_i);
12         Anotar m.par_i.nome em Ln
13       Fim-Se
14     Fim-Para
15     Para cada instrução ins ∈ m faça
16       Se ins é declaração então
17         Se ins.tipo ∈ tipos anotados então
18           Marcar(ins);
19           Anotar ins.nomeObjeto em Ln;
20         Fim-Se
21       Senão
22         Para cada nome n ∈ Ln faça
23           Se ins ⊃ objeto o && o.nome = n então
24             Marcar(ins);
25           Fim-Se
26         Fim-Para
27       Fim-Se
28     Fim-Para
29   Fim-Para
30 Fim-Para
```

Figura 3.9: Algoritmo para identificação de indícios

Os passos de identificação de indícios são aplicáveis quando os tipos usados em declarações de variáveis, atributos e parâmetros são previamente importados do código-fonte. A utilização de tipos contendo o caminho completo do arquivo (*Fully Qualified Names*) possibilita a utilização de um tipo sem sua importação. Caso o engenheiro de software constate a utilização de tipos com caminho completo do arquivo em diversas partes do sistema legado, todos os tipos não importados deverão ser verificados mesmo que o arquivo Java não contenha nenhum tipo importado e relacionado à categoria de indício que está sendo analisada.

Ao final da execução dos passos do algoritmo apresentado na Figura 3.9, os indícios correspondem a cada um dos trechos de código marcados. Para isso, é utilizado o mesmo padrão de marcação do código da abordagem Aspecting, ou seja, comentários devem ser inseridos ao final da linha contendo o nome do interesse encontrado seguido de um número seqüencial, que indica a ordem em que o trecho aparece na classe ou interface.

3.4.2. Diretrizes para reestruturação do código-fonte

Após a identificação dos indícios descrita na seção anterior, o engenheiro de software usa seu conhecimento e determina se o indício realmente faz parte de um aspecto ou não. Para o último caso, nenhuma modificação é realizada. Entretanto, quando é constatada a existência de aspectos, as instruções e declarações que expressam tais aspectos devem ser separadas do código e inseridas em um módulo apropriado. No caso de sistemas em linguagem AspectJ, esses módulos são arquivos de extensão aj e o combinador (*weaver*) da linguagem reconhecem construções próprias capazes de descrever o comportamento transversal dos aspectos nesses módulos. Nesta seção são descritas as diretrizes que podem ser seguidas para obtenção de um novo sistema em AspectJ a partir de um sistema legado em Java.

A Figura 3.10 apresenta o algoritmo com a visão geral da atividade de reestruturação do código do sistema legado. Como pode-se observar a iteratividade da reestruturação depende da quantidade de interesses transversais a serem encapsulados, dos pacotes do sistema e dos arquivos com marcações.

```
1  Para cada interesse transversal faça
2    criar módulo de aspecto A;
3    Para cada pacote do sistema p faça
4      Para cada arquivo com marcações a ⊂ p faça
5        Se ∃ atributo privado com marcações então
6          A.modificador := privileged;
7        Fim-Se
8        Para cada método m ⊂ a faça
9          Se todas as instruções de código possuem marcações então
10           introduzir(m, a);
11         Senão
12           reordenar instruções;
13           Se ∃ marcações no começo de método então
14             reestruturarInicioMetodo();
15           Fim-Se
16           Se ∃ marcações no fim de método então
17             reestruturarFimMetodo();
18           Fim-Se
19         Fim-Se
20       Fim-Para
21       Para cada atributo com marcações at ⊂ a faça
22         introduzir(at, a);
23       Fim-Para
24       Se quantidade de aspectos em a = 0 então
25         retirar instruções de importação de tipos do int. transversal;
26       Fim-Se
27     Fim-Para
28   Fim-Para
29 Fim-Para
```

Figura 3.10: Visão geral da reestruturação do sistema legado em Java

É criado um módulo de aspecto que centraliza a implementação de cada interesse transversal espalhada por todo o sistema. Em cada um dos pacotes do sistema, todos os arquivos com marcações são analisados. Caso se constate a existência de ao menos um atributo privado com aspectos, é adicionado à declaração do módulo de aspecto o modificador `privileged`, permitindo que atributos privados sejam acessíveis ao módulo. Em seguida, cada método do arquivo é avaliado, iniciando pela verificação de marcações em todas as instruções do método. Caso essa verificação seja verdadeira, todo o método é introduzido no aspecto. Para isso, o módulo de aspecto deve importar a classe ou a interface que contém o método e os tipos usados no método, ou seja, tipos encontrados na lista de parâmetros, em instruções e no valor de retorno do método. Por fim, o nome da classe / interface cujo método foi retirado e introduzido no módulo de aspecto deve anteceder o nome do método.

Também deve ser verificado o modificador de acesso do método. Caso o mesmo seja privado, a classe ou interface que utiliza esse método não poderá mais acessá-lo após a introdução desse método no módulo de aspecto. Modificando o acesso ao método de privado para público, a classe / interface será capaz de utilizar esse método, contudo, o acesso se torna possível para os demais módulos do sistema. A solução, nesse caso, é fazer marcações nas chamadas desse método para, posteriormente, transferir essas chamadas ao módulo de aspecto por meio de conjuntos de junção e adendos. Como o acesso ao método chamado é permitido dentro do aspecto, esse problema é contornado.

Ao analisar o método, apenas algumas instruções podem conter marcações. Para esse caso, deve-se verificar a possibilidade de reordenar as instruções de modo que as que contém marcações se concentrem no início ou no final do método. É importante garantir que a reordenação das instruções não altere a lógica do programa. Em seguida, se existirem Instruções com Marcações (IM) no início e/ou no fim do método, a reestruturação de instruções é aplicada. Essa tarefa consiste em definir um conjunto de junção, que capte o ponto de junção, no início ou fim de método, e o adendo que conterá as instruções marcadas no método. O conjunto de junção emprega os conjuntos de junção primitivos: `execution(<assinatura do método>)` e `this`, caso atributos e métodos da classe ou interface corrente sejam usados em IM; `args` se as variáveis declaradas na lista de parâmetros do método sejam utilizadas em IM. Para definir o adendo é necessário referenciar o conjunto de junção previamente criado e empregar a

palavra reservada `before`, caso as instruções sejam de início de método ou `after` caso contrário. Por fim, as instruções marcadas são removidas do método.

Com o término da reestruturação dos métodos com indícios, os atributos marcados são introduzidos do arquivo corrente para o módulo de aspecto. Essa tarefa é realizada após a reestruturação de métodos inteiros e de instruções com aspectos. Essa imposição é justificada devido à utilização de atributos em instruções. A introdução precoce de atributos gera erros de compilação, pois os atributos deixam de ser visíveis às classes/interfaces, sendo corrigidos apenas quando todas as instruções que referenciam tais atributos têm sua reestruturação concluída, podendo isso acontecer apenas ao final do processo de reestruturação do arquivo corrente. Nesse caso, testes intermediários se tornam impossíveis de serem realizados visto que o sistema não pode ser compilado.

Finalmente, é verificado se todos os trechos de código marcados foram reestruturados. Se isso for constatado, importações dos tipos associados a indícios podem ser eliminadas do código-fonte. Todas essas atividades são repetidas para cada arquivo, pacote e interesse transversal.

3.5. Considerações Finais

Este capítulo apontou algumas limitações da abordagem Aspecting (RAMOS, 2004) e algumas alterações foram sugeridas: extensão da Lista de Indícios para os interesses transversais de persistência em banco de dados, persistência em memória temporária e *logging*; necessidade em relacionar tipos com seus respectivos pacotes de modo que a diferenciação de tipos de mesmo nome seja possível; e utilização de análise sintática ao invés de léxica.

Nesse sentido, foi proposto o Modelo de Indícios com o objetivo de tornar a representação de indícios de mais fácil manipulação por ferramentas cujo mecanismo de identificação de indícios se baseie na abordagem Aspecting, além de permitir maior precisão de resultados empregando análise sintática. Os Modelos de Indícios para persistência em banco de dados, em memória temporária e *logging* foram apresentados de forma tabular e as descrições de pacotes, de classes, de interfaces e de classes de exceção foram obtidas a partir da documentação da API do JDK. Tais descrições não

afetam os resultados finais do processo de revitalização do sistema legado, simplesmente servem de documentação de indícios.

Do mesmo modo que a abordagem Aspecting utiliza a Lista de Indícios para especificar as ações a serem tomadas para realizar a identificação indícios e alteração de código, neste Capítulo foram mostradas novas diretrizes que empregam o Modelo de Indícios para a identificação de indícios e para reestruturação do código fonte utilizando construções da linguagem AspectJ.

As diretrizes existentes na abordagem Aspecting e as diretrizes apresentadas para o Modelo de Indícios possuem característica em comum, as duas demandam grande esforço em sua execução, principalmente considerando sistemas de médio à grande porte, devido sua natureza iterativa. Por esse motivo, a aplicação manual dessas diretrizes em um cenário real pode ser impraticável, não justificando o esforço despendido.

Mesmo que a reengenharia de um sistema legado envolva tomada de decisão por um projetista ou engenheiro de software, existem algumas atividades repetitivas que não requerem discernimento humano, precisam somente serem executadas, tal como a análise sintática do código, busca de tipos e objetos, geração de conjuntos de junção, de adendos e de declarações intertipos simples. O Capítulo 4 apresenta uma proposta de automatização dessas atividades com o objetivo de facilitar a reengenharia de sistemas legados.

Capítulo 4

Um Apoio Computacional para Reengenharia de Sistemas que apresentam Interesses Transversais em seu Código Fonte Java

4.1. Considerações Iniciais

A identificação de indícios de interesses transversais em sistemas implementados na linguagem Java requer habilidade e conhecimento por parte do engenheiro de software envolvido. Assim, no Capítulo 3, foi apresentado um modo para identificar indícios de interesses transversais por meio do Modelo de Indícios. Diretrizes para identificação desses indícios e para reestruturação do código-fonte, possibilitando a separação de aspectos de *logging*, persistência em banco de dados e persistência em memória temporária foram descritas. Algumas limitações da abordagem Aspecting (RAMOS, 2004) foram apontadas, sendo apresentada uma forma para contorná-las, via utilização do Modelo de Indícios.

Em conseqüência da quantidade de diretrizes que devem ser seguidas tanto pela abordagem Aspecting quanto às referente ao Modelo de Indícios apresentado no

Capítulo 3, a reengenharia de sistemas legados realizada manualmente se torna inviável, dependendo do tamanho do sistema. Por esse motivo, há necessidade de algum tipo de apoio computacional para automatizar parte dessas atividades, a fim de possibilitar o aumento de produtividade e a diminuição de erros inseridos pelos próprios engenheiros de software responsáveis pela migração do sistema legado.

A partir das considerações anteriores, a concepção de um apoio computacional para auxiliar a reengenharia de sistema legado implementado em Java para AspectJ foi realizada por meio de prototipação. São tratados os interesses transversais referentes à persistência de dados e *logging*, espalhados em seu código fonte. Esse apoio computacional é denominado ReJAsp (apoio computacional para Reengenharia de sistemas Java para AspectJ).

A primeira versão desse apoio computacional, desenvolvido em Java, realizava a identificação de indícios por meio de análise sintática, mas não possuía integração com nenhum ambiente de desenvolvimento (IDE) (KAWAKAMI e PENTEADO, 2005a, 2005b). Essa versão foi bastante útil para detectar alguns pontos críticos que deveriam ser corrigidos. Entre esses pode-se citar: a ausência de integração com um IDE, de assistentes (*wizards*), de função de desfazer e de uma representação de indícios gerenciáveis, ou seja, que permita cadastrar, atualizar e excluir informações de indícios em tempo de execução pelo próprio usuário. A partir desse aprendizado, um novo protótipo foi construído e é discutido neste Capítulo.

Na Seção 4.2 a visão geral do apoio computacional ReJAsp e suas características são apresentadas; na Seção 4.3 detalhes da arquitetura usada no apoio computacional são discutidos. A Seção 4.4 descreve o mecanismo de identificação de indícios utilizado; na Seção 4.5 os assistentes de reestruturação de código são apresentados; na Seção 4.6 questões de persistência do Modelo de Indícios são abordadas. Alguns trabalhos, existentes na literatura, que têm similaridades com o apoio computacional ReJAsp são brevemente discutidos e comparados na Seção 4.7 e, na Seção 4.8 as considerações finais sobre este Capítulo são comentadas.

4.2. Visão geral do Apoio Computacional ReJAsp

A segunda versão do apoio computacional para realizar reengenharia em sistemas que possuem interesses transversais foi implementada na linguagem de

programação Java (JAVA TECHNOLOGY, 2007) e integrado ao ambiente Eclipse (ECLIPSE PROJECT, 2007), isto é, foi implementado como um *plug-in* desse ambiente. Algumas das razões que justificam essa integração como um *plug-in* do Eclipse são:

- Utilização de bibliotecas: o ambiente Eclipse contém grande acervo de bibliotecas que podem ser usadas por programas, se forem implementados como *plug-in* desse ambiente Dentre essas se tem: bibliotecas de construção de interface (SWT, JFace), de manipulação de código Java (JDT) e leitura de arquivo XML (DOM), entre outras. Essa facilidade permite a utilização de interfaces com aparência bastante similares às do ambiente Eclipse e outros *plug-ins* dessa IDE, possibilitando adaptação mais rápida do usuário já habituado com essas interfaces;
- Uso de extensões: a utilização de pontos de extensão, a partir de ferramentas ou de outros *plug-ins* do Eclipse, promove economia de linhas de código implementadas e permite o desenvolvimento de novas funções, contribuindo com a melhoria da funcionalidade do ambiente de programação;
- Análise sintática do código-fonte: utilizando classes do Eclipse e do JDT é possível percorrer uma estrutura de dados similar à da AST (*Abstract Syntax Tree* – árvore abstrata sintática), sem que seja necessária a criação de um compilador para a construção e manipulação da mesma. Com isso, a implementação é abreviada e a confiabilidade do apoio computacional é maior, pois as classes do Eclipse e de seus *plug-ins* já foram bem testadas e validadas;
- Colaboração com AJDT: o apoio computacional ReJAsp utiliza bibliotecas do AJDT (ASPECTJ DEVELOPMENT TOOLS, 2007) para a obtenção do código-fonte proveniente de módulos de aspectos e para a criação de um novo aspecto;
- Colaboração com outros *plug-ins*: além do AJDT, existem diversas ferramentas que podem ser usadas para realizar a programação orientada a aspectos, reengenharia de sistemas e testes em conjunto com o apoio computacional, facilitando a migração de um sistema legado OO para um sistema OA;
- Ambiente de desenvolvimento amplamente utilizado: o Eclipse é um dos ambientes de desenvolvimento mais usados pela comunidade de

desenvolvedores em Java, o que favorece a utilização do apoio computacional ReJAsp.

A instalação do ReJAsp é bastante simplificada, entretanto, depende da instalação prévia dos seguintes componentes: Sistema Operacional Windows XP; Java™ 2 Plataform Standard Edition 5.0 Development Kit (JDK 5.0); IDE Eclipse 3.2; *Plug-in* AJDT 1.4.

O apoio computacional para reengenharia de interesses transversais, ReJAsp, apresentado neste trabalho, é capaz de identificar duas das seis categorias de indícios apresentadas na Lista de Indícios de Ramos (2004). As categorias contempladas são as de persistência em memória temporária (*buffering*) e de persistência em banco de dados, que sofreram atualizações neste trabalho e uma nova categoria foi adicionada: *logging*, conforme discutido no Capítulo 3. Maiores detalhes a respeito de procedimentos de instalação e de uso do ReJAsp, podem ser encontrados no Guia do Usuário[1] (KAWAKAMI e PENTEADO, 2007.

4.3. Arquitetura do Apoio Computacional ReJAsp

A arquitetura do ReJAsp é baseada na extensão de componentes como menus, perspectivas, visões e ações. A extensão de componentes e de ferramentas é bastante comum na construção de *plug-ins* e utiliza o ambiente de desenvolvimento de plug-ins, comumente denominado de PDE (em inglês, *Plug-in Development* Environment) (ECLIPSE PROJECT, 2007). A interface do PDE provê um conjunto de ferramentas para criação, desenvolvimento, teste, depuração e distribuição de *plug-ins*.

Além das extensões de componentes, várias classes do apoio computacional possuem herança com as classes da biblioteca do Eclipse incluindo telas, assistentes e outros componentes visuais. A Figura 4.1. exibe a arquitetura do apoio computacional ReJAsp, dividida em módulos de ações (*actions*), de perspectiva do apoio, de registro de indícios, de árvore de indícios, de assistentes (*wizards*), de Modelo de Indícios e de persistência de indícios em XML. Os módulos relacionados às classes da plataforma Eclipse são também representados

[1] http://www.dc.ufscar.br/~rosangel/ACRA/

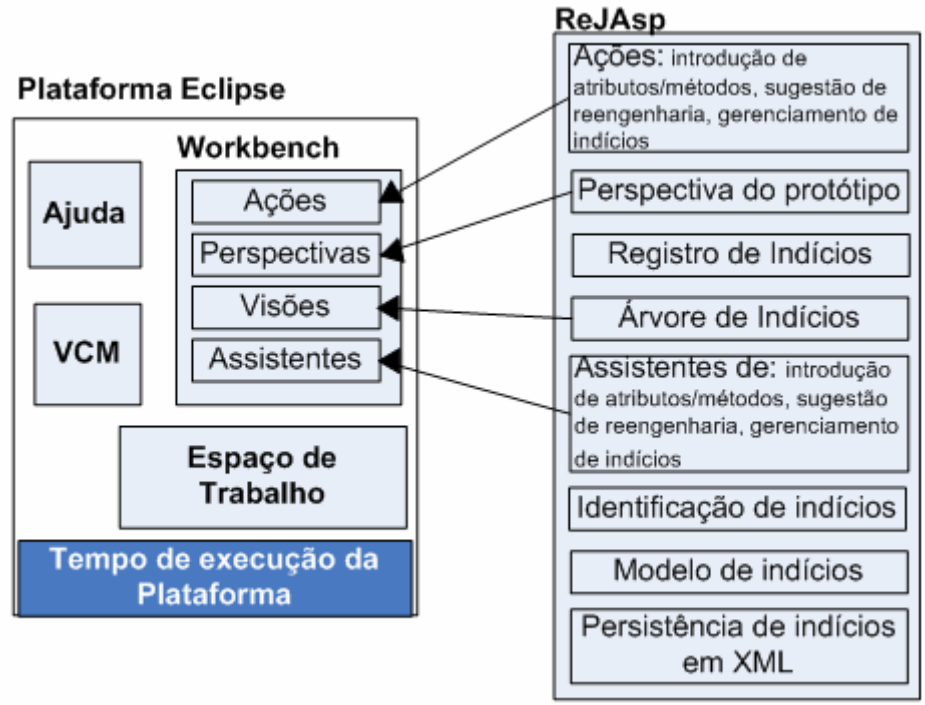

Figura 4.1: Arquitetura do apoio computacional para reengenharia de interesses transversais

O módulo de Ações do ReJAsp é constituído por classes que determinam as respostas quando algum evento ocorre, geralmente, proveniente da interação do usuário com o ambiente. As principais Ações são aquelas em que o usuário solicitou as funções de introdução de atributos / métodos, de reestruturação do código ou de gerenciamento de indícios a partir de um menu ou da barra de ferramentas (*toolbar*). O módulo de Ações foi implementado no ReJAsp por meio de pontos de extensão de Ações previamente existentes no Eclipse.

Perspectivas determinam a visibilidade de Ações e de Visões dentro de uma janela do Eclipse, além de prover mecanismo para interação orientada a tarefas, a partir de recursos (projetos, pastas e arquivos) disponíveis na plataforma Eclipse; mecanismos de tarefas múltiplas e filtro de informações (ECLIPSE PROJECT, 2007). O apoio computacional estende Perspectivas do Eclipse, associando-o com as Visões: Árvore de Indícios, *Package Explorer*, navegador de recursos, tarefas e problemas.

4.3.1. Registro de indícios no código

Um conjunto de classes foi definido com a função de geração de objetos que armazenam as informações de indícios encontradas no código: de arquivos, classes,

interfaces, atributos e métodos associados ao indício, conforme apresentada na Figura 4.2. Os registros de indícios no código-fonte são obtidos quando a identificação automatizada de indícios é acionada, podendo ser usados para visualização de possível reestruturação de código, em filtros de assistentes de reestruturação de código e na geração de conjuntos de junção e adendos.

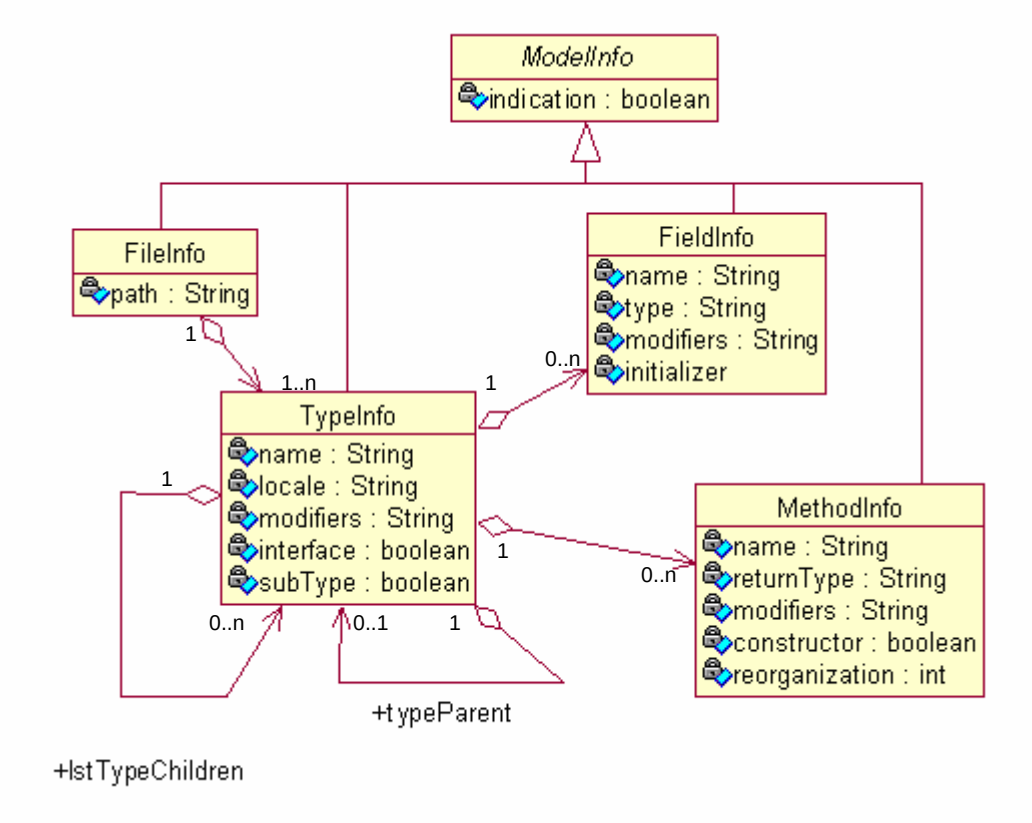

Figura 4.2: Classes de armazenamento de artefatos de projeto

No diagrama de classes da Figura 4.2, a classe `ModelInfo` foi definida como abstrata contendo o atributo `indication`, para indicar a presença ou ausência de indícios. As demais classes do diagrama são especializações de `ModelInfo` e, conseqüentemente, herdam o atributo `indication`.

A classe `FileInfo`, Figura 4.2, armazena o caminho completo de um arquivo com código-fonte Java. Por possuir herança com `ModelInfo`, é possível expressar a ocorrência ou ausência de indícios em um arquivo. Durante a identificação de indícios no `ReJAsp`, cada arquivo do sistema legado que contenha código em Java é processado e um respectivo objeto da classe `FileInfo` é gerado. Por meio do relacionamento de

agregação entre `FileInfo` e `TypeInfo`, é possível observar que um ou vários tipos (classes / interfaces) podem ser especificados em um mesmo arquivo.

O nome e o pacote do tipo são representados pelos atributos `name` e `locale` respectivamente. O atributo `Modifiers` é usado para especificar os modificadores do tipo, indicando seu acesso (público, privado ou protegido). Para diferenciar classes e interfaces foi usado o atributo `interface`.

Uma característica importante de `TypeInfo` é o auto-relacionamento. O atributo `subType` contém o valor verdadeiro quando o tipo for uma especialização, ou seja, utiliza as palavras-chave `extends` ou `implements` na declaração do tipo. Complementando esse atributo, `TypeInfo` pode conter a referência `typeParent` para um objeto que representa o tipo pai. A classe ou interface generalizada, por sua vez, contém uma coleção de um ou mais especializações, representada por `lstTypeChildren`.

No modelo da Figura 4.2, os atributos são descritos por objetos `FieldInfo` e estão relacionados aos objetos `TypeInfo` por meio de agregação, representando o relacionamento de vários atributos para uma mesma classe ou interface. A herança existente entre `FieldInfo` e `ModelInfo` justifica-se pelo fato de um atributo poder conter indícios ou não. As informações de atributos usadas neste trabalho são: nome, tipo, modificadores e inicializadores.

Do mesmo modo que as demais classes do modelo, a classe `MethodInfo` também é uma classe especializada de `ModelInfo`. Os objetos de `MethodInfo` armazenam o nome, o conjunto de modificadores e o tipo de retorno de métodos. Os construtores também são armazenados por `MethodInfo` e sua distinção com métodos é feita por meio do atributo `constructor`. Quando `constructor` recebe o valor verdadeiro o objeto de `MethodInfo` armazena um construtor e caso contrário, o objeto contém informações de um método.

Existem três modos de reestruturação de métodos: (1) introdução de métodos, (2) extração de instruções de início método e (3) extração de instruções de final de método. De acordo com a disposição dos indícios pelo método, a aplicação de um ou mais modos de reestruturação podem ser sugerido por ReJAsp. Para isso, o atributo `reorganization` armazena quais reestruturações são pertinentes ao método. Maiores detalhes de como `reorganization` é manipulado pelo apoio computacional pode ser encontrado no Apêndice 1.

Uma alternativa ao uso do modelo visto refere-se ao processamento de identificação de indícios de todos os artefatos do projeto (arquivos e pastas) auxiliado pela invocação da unidade de compilação (*Compilation Unit*) do JDT. Entretanto, as operações em assistentes e em Árvore de Indícios que usam informações desse modelo se tornariam bastante demoradas, prejudicando o uso do ReJAsp. A lentidão ocorre devido à construção de uma estrutura similar à AST pelo JDT, a qual consome tempo e recurso considerável da máquina. A limitação dessa alternativa foi verificada experimentalmente durante os testes e motivou a criação dos registros de indícios apresentados na Figura 4.2.

4.3.2. Visões

A visão de Árvore de Indícios trata-se de uma implementação própria, feita no apoio computacional, que permite a visualização de pacotes, de arquivos e de indícios. Em contrapartida, as demais visões são implementações já existentes no próprio ambiente Eclipse. *Package Explorer* é usada para a visualização de pacotes e classes Java, sendo incluída na Perspectiva do ReJAsp devido à necessidade de realizar operações, refatorações ou alterações em módulos implementados na linguagem Java. A utilização da visão *Package Explorer* pode se tornar necessária à medida que o sistema legado OO é reestruturado para uma linguagem OA, devido às modificações no código OO. Assim, ressalta-se a importância em corrigir os problemas no código legado OO previamente, uma vez que a POA tem o intuito apenas de corrigir os problemas de espalhamento e de entrelaçamento não possíveis de serem tratadas em linguagens OO.

A visão de tarefas é responsável pela exibição de tarefas associadas à codificação, podendo ser criadas e identificadas a partir do código-fonte por comentário padronizado com a palavra “TODO”. No caso do apoio computacional, em especial, cada indício é considerado como uma tarefa, mesmo que o engenheiro de software o classifique como um falso indício. De qualquer forma, esse falso indício ainda estará associado à tarefa de classificação de indício como sendo um aspecto de fato ou de falso indício pelo engenheiro de software. Caso seja verificado que se trata de um aspecto, a tarefa ainda pode envolver a atividade de separação do trecho de código com aspectos do código base, para um módulo de aspectos, disponibilizado pela linguagem AspectJ. Por esse motivo, são realizadas manipulações na visão de tarefas, cadastrando os

indícios como tarefas automaticamente, no momento em que a função de identificação de indícios é acionada.

A Árvore de Indícios estende a classe de Visão do Eclipse, adicionando comportamentos próprios, facilitando a visualização e navegação pelo código-fonte com indícios. Antes de realizar a geração da Árvore de Indícios, deve ser feita a escolha do interesse transversal a ser separado e o projeto que contém o sistema legado. Com isso, o nome do projeto é inserido no topo da Árvore de Indícios. A interface gráfica de árvore, disponibilizada pela API do Eclipse, permite organizar e relacionar itens. Cada item da árvore pode ser chamado de nó e, em termos de interface, esses nós podem ser expandidos (os nós agregados e esse nó são exibidos) ou contraídos (os nós agregados a esse nó são ocultados). Na Figura 4.3, a Árvore de Indícios é apresentada e seus nós componentes identificados por rótulos em vermelho são descritos a seguir:

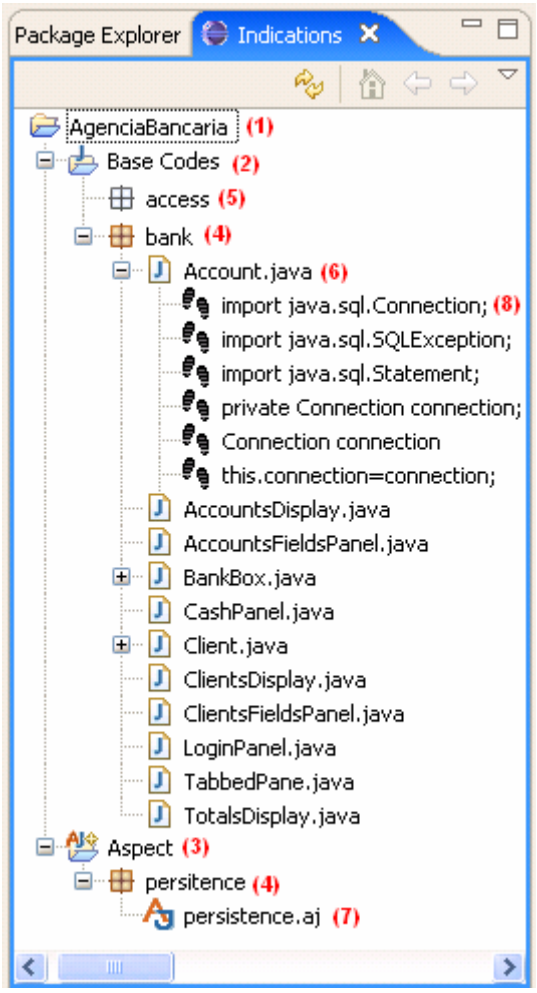

Figura 4.3: Árvore de indícios

1. **Raiz**: é o nó principal da Árvore de Indícios. A partir dele é possível expandir seus nós descendentes e ter acesso a todos os demais nós da Árvore de Indícios. Esse nó é único em uma Árvore de Indícios e não é agregado por nenhum outro nó. Seu rótulo sempre é identificado pelo nome do projeto selecionado;
2. **Códigos-base** (*Base Codes*): nó contido no nó raiz, tem a função de agregar os pacotes, os arquivos escritos em "java" e os indícios de aspecto;
3. **Aspecto** (*Aspect*): a partir desse nó, é possível ter acesso aos módulos de aspecto (de extensão "aj") e os pacotes que contém esses arquivos;
4. **Pacotes**: correspondem a nós agregados dos nós de códigos-base e dos de aspectos. Representam, justamente, os pacotes utilizados na linguagem Java ou em AspectJ, isto é, diretórios que contém os arquivos em Java ou em AspectJ. Devido à possibilidade de um mesmo diretório conter arquivos de classes Java e de módulos de aspectos, um nó de pacote pode coexistir tanto no nó de códigos-base quanto no de aspectos;
5. **Pacotes vazios**: são diretórios sem arquivos de extensão "java" ou "aj", mas que, posteriormente, podem ser preenchidos por esses arquivos. Quando isso ocorrer, sua categoria passa de pacote vazio para simplesmente pacote;
6. **Arquivos Java**: o nó com essa classificação, referencia um arquivo de extensão ".java" com rótulo que é o próprio nome do arquivo. Esses nós podem ou não conter outros nós de indícios;
7. **Arquivos de Aspecto**: apresentam os arquivos de aspecto do sistema, podem estar contidos no nó de aspectos ou em um de seus pacotes. Esses nós não apresentam nós agregados;
8. **Indícios**: esses nós representam instruções ou trechos de instruções que contém indícios (candidatos) de aspectos. Estão diretamente relacionados aos nós de arquivos ".java", mas não agregam outros nós.

Cada nó da Árvore de Indícios pode ser selecionado, os que contêm agregados simplesmente se expandem, exibindo seus nós filhos, ou se contraem, ocultando-os. O editor de texto é aberto a partir de um clique duplo no nó de arquivos Java ou de Aspectos, exibindo o código-fonte correspondente. O clique duplo no nó de indício mostra o arquivo que contém o seu trecho de código associado e posiciona o editor no local com o trecho com indício, destacando-o.

4.4. Mecanismo de Identificação de Indícios

A identificação de indícios é um mecanismo ativado a partir da Árvore de Indícios quando ocorre o:

- Carregamento da Árvore de Indícios;
- Seleção de um projeto;
- Solicitação de atualização na Árvore de Indícios.

Para realizar a identificação da Árvore de Indícios é necessário que o projeto com o sistema legado esteja armazenado no local de *workspace* padrão, aberto para alteração (a visão *Package Explorer* pode ser usada para abrir ou fechar o modo de alteração) e selecionado na Árvore de Indícios. Também é necessário certificar que o interesse transversal escolhido é o referente aos indícios que são buscados. É possível implementar a aplicação da identificação de indícios de múltiplos interesses transversais e para vários projetos. Contudo, o mecanismo de identificação só é executado em um único projeto para um interesse transversal, para evitar longo tempo de espera no processamento.

Ao iniciar a identificação, os passos dos algoritmos para determinação de tipos a serem analisados no código-fonte e para identificação de indícios, exibidos, respectivamente nas Figuras 3.8 e 3.9, são executados automaticamente pelo apoio computacional. A principal diferença entre o processo descrito no Capítulo 3 e a identificação realizada pelo ReJAsp é que as marcações não são registradas por comentários padronizados. Ao invés disso, essas informações são mantidas e gerenciadas internamente pelo apoio computacional, tornando esse mecanismo transparente ao usuário e diminuindo a propensão de erros humanos, como por exemplo, a remoção acidental dos comentários no código-fonte com as informações de indícios por parte do desenvolvedor.

Conforme comentado no Capítulo 3, os tipos (classes ou interfaces) podem conter o caminho completo de arquivo (*Fully Qualified Names*), possibilitando a declaração de um objeto com um tipo, sem sua importação prévia no código-fonte. Assim, objetos e declarações desses tipos podem conter indícios que não são identificados pelos algoritmos exibidos nas Figuras 3.8 e 3.9. Para solucionar esse caso,

o ReJAsp permite a seleção de uma estratégia de identificação, dentre duas existentes: uma baseada em tipos importados e a outra usando análise de tipos com caminho completo de arquivo. A primeira estratégia apenas considera tipos previamente importados, filtrando arquivos sem importação de tipos pertencentes à categoria de indícios. Por outro lado, a segunda estratégia, além de verificar indícios nos tipos importados analisa, também, todos os tipos com caminho completo de arquivo. Assim, o tempo de espera na identificação de indícios é menor para a primeira estratégia, porém há melhor precisão dos indícios identificados quando a segunda estratégia é adotada e o uso de tipos com caminho completo de arquivo é comum no sistema legado.

4.4.1. Percurso em árvore

A base para a execução da identificação de indícios está na utilização e percurso de estruturas de dados providas pelo JDT. Essas estruturas de dados estão condizentes com a AST da linguagem Java e, conseqüentemente, podem ser combinadas para expressar qualquer código implementado nessa linguagem.

Uma forma simplificada de descrever a AST de uma linguagem é pela utilização de gramáticas livre de contexto. Para cada gramática, os símbolos terminais são obtidos a partir de um alfabeto específico. Neste projeto a documentação completa da linguagem Java, de suas gramáticas léxicas e sintáticas podem ser encontradas em Java Technology (2007).

Regras de produções foram especificadas para cada classe relacionada à manipulação da AST a partir da documentação em JAVADOC das classes do JDT, usadas para essa manipulação. Mesmo que a notação da gramática seja a mesma e as regras de produção muito similares, as gramáticas livres de contexto documentadas no JAVADOC são diferentes das gramáticas especificadas pela SUN, criadora da linguagem Java. Entretanto, ambas geram as mesmas saídas de conjunto de terminais. As gramáticas livres de contexto usadas pelo JDT e por este projeto estão disponibilizadas no Apêndice 1.

Uma das diferenças entre as gramáticas é a nomenclatura de não-terminais, que no caso do JDT, também são usadas como nome de classes responsáveis no percurso e análise da AST. Do mesmo modo que não-terminais derivam em um conjunto de não-terminais e terminais, suas classes correspondentes podem estar associadas a um

conjunto de objetos, cujas classes equivalem a não-terminais da gramática e de palavras-chave que correspondem aos terminais.

Por esse motivo na construção do apoio computacional, adotou-se uma estratégia de percurso pela AST e análise do código por meio de funções recursivas ou que possam levar à recursividade indiretamente. Esse último caso, acontece quando a função que trata de um não-terminal A, por exemplo, gera uma série de outros não-terminais após sucessivas derivações. Um ou mais desses não-terminais gerados podem derivar o não terminal A. Assim, a função que trata do não-terminal A será novamente chamada.

A utilização da recursividade possui a desvantagem de consumo de tempo e memória, já que a cada chamada recursiva, o estado atual de processamento é registrado. Entretanto, o problema de percurso em árvore possui uma solução inerentemente recursiva, visto que é necessário registrar o estado anterior, para apresentar uma maior clareza e facilidade de manutenção dos códigos com a implementação do percurso do ReJAsp. Mais detalhes sobre do mecanismo de percurso de AST usado no apoio computacional podem ser encontrados no Apêndice 1.

4.4.2. Regras de agrupamento de indícios

A partir do percurso na AST, realizado no ReJAsp, os indícios são encontrados nos terminais de tipo, de identificador e de literal de cadeia de caracteres. A criação de um registro de indício para cada não-terminal considerado pode tornar a visualização, entendimento e transformação do código confuso e pouco eficiente. Isso acontece porque em uma instrução ou expressão pode derivar grande quantidade de identificadores, tipos e / ou literais de cadeia de caracteres. Assim, para uma mesma instrução ou expressão, diversos registros de indícios poderiam ser criados, tornando os ramos da Árvore de Indícios numerosos, sobrecarregando a lista de tarefas e dificultando a criação de aspectos pelos assistentes de reestruturação de código.

Durante a identificação de indícios, adotou-se o agrupamento de indícios por instrução. Entretanto, instruções podem conter outras instruções e, desse modo, é necessário definir se agrupamento considera frações de instruções menores ou maiores. Por exemplo, a existência de um indício, em uma instrução de atribuição é localizada dentro de uma instrução de laço de repetição `for;` nesse caso é necessário definir se apenas a instrução de atribuição é considerada indício ou toda a instrução `for`. A

princípio, considerar a instrução mais próxima ao indício pode parecer a solução mais indicada, contudo, a reestruturação de uma instrução contida dentro de outras é complexa, mesmo que se utilize mecanismos da linguagem AspectJ. Em geral, os adendos só conseguem executar antes e / ou após um determinado ponto de junção. A determinação de uma expressão que entrecorte justamente o local da instrução que esteja interna a outra instrução é difícil e aumenta a propensão de inserção de erros ou nem sempre possível de ser feita. Por esse motivo, adotou-se o agrupamento de indícios considerando a instrução mais global.

4.4.3. Tabela de variáveis

A linguagem Java gerencia o escopo de variáveis, isto é, determina a visibilidade de variáveis nas diferentes partes do programa. O escopo de um atributo, por exemplo, é válido por toda a classe a partir de sua declaração, além de incluir todas as classes internas (*inner classes*) e métodos. Por outro lado, o escopo de um parâmetro abrange todo o corpo de seu método e as variáveis locais são visíveis apenas no bloco de sua declaração.

Para que o módulo de identificação de indícios do ReJAsp possa identificar cada variável de um código em Java com indícios, foi criada a tabela de variáveis que consiste de uma pilha, denominada de pilha de níveis, contendo listas que armazenam informações de variáveis de um único nível (ou escopo).

4.5. Assistentes

Os Assistentes (*wizard*) implementados no apoio computacional ReJAsp também são especializações de um *template* de assistentes do Eclipse e foram escolhidos pelas seguintes razões:

- **Divisão de tarefas em passos**: tarefas complexas são difíceis de serem executadas e devem ser divididas em tarefas menores ou passos, a fim de facilitar sua execução. Mesmo assim, há chances consideráveis do executor se perder no meio do processo ou inserir erros, por exemplo, a utilização de seqüências incorretas na execução de passos. Além disso, a seqüência de passos pode não ser linear, ou seja, podem haver diferenças entre execuções de uma mesma tarefa em um passo e outro. O assistente, por sua vez, também divide a

tarefa em passos e gerencia automaticamente o cumprimento da tarefa na seqüência correta de passos, determinando o próximo passo de acordo com as informações coletadas anteriormente. Com isso, há a garantia de que todos os passos necessários são executados na seqüência correta;

- **Usuário não precisa conhecer todo o processo**: o assistente guia o usuário por todo o processo;
- **Automatização de parte da tarefa**: o assistente solicita a entrada de dados pelo usuário e realiza o processamento dessas informações possibilitando a automatização de parte da tarefa e a economia de tempo;
- **Retroceder e desfazer**: a qualquer momento o usuário é capaz de retornar aos passos anteriores e modificar os dados de entrada. As alterações são efetivadas somente ao final do processo;
- **Pré-visualização**: os assistentes do ReJAsp, em particular, têm telas de pré-visualização das alterações no código-fonte que podem ser confirmadas ou recusadas pelo usuário. Os trechos de código-fonte inseridos, alterados ou excluídos recebem destaques nas telas de pré-visualização, facilitando identificação e compreensão das modificações sugeridas pelo apoio computacional.

Os três assistentes implementados no ReJAsp são descritos a seguir.

a) Assistente de Introdução de Atributos

Responsável pela criação de declarações intertipos de atributos de classes, possibilita a seleção de vários atributos com ou sem indícios de um ou mais tipos (classes ou interfaces), podendo abreviar o trabalho do usuário. Sem essa característica, a criação de declarações intertipos de múltiplos atributos necessitaria da execução desse assistente várias vezes. A Figura 4.4 apresenta a seqüência de telas exibidas por esse assistente. Maiores detalhes, objetivos e a utilização de cada uma das telas podem ser encontrados no Guia do Usuário (KAWAKAMI e PENTEADO, 2007).

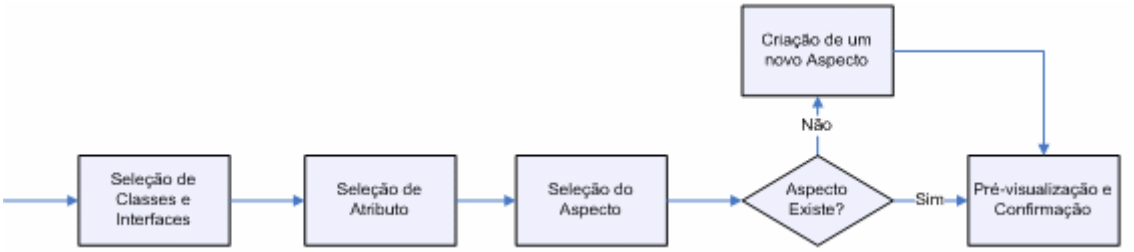

Figura 4.4: Seqüência de telas do Assistente de Introdução de Atributos para aspectos

b) Assistente de Introdução de Métodos

Tem o funcionamento muito similar ao Assistente de Introdução de Atributos apresentado anteriormente. Parte das informações usadas nesse assistente são provenientes de objetos `MethodInfo`. A seqüência de telas disponíveis nesse assistente é mostrada na Figura 4.5.

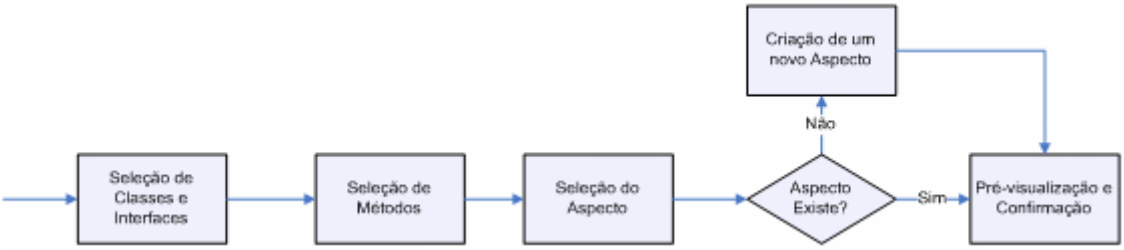

Figura 4.5: Seqüência de telas do Assistente de Introdução de Métodos para aspectos

c) Assistente para Reestruturação de Instruções

Enquanto, as introduções de atributos e de métodos implementam mecanismos de interesses transversais estáticos pelo uso de declarações intertipo, a reestruturação de instruções, Figura 4.6, guia o usuário ao desenvolvimento de interesses transversais dinâmicos, devido à especificação de conjuntos de junção e de adendos.

A reestruturação de instruções pode ser feita de dois modos: (1) pela extração de início de método e (2) pela extração de final de método. A primeira realiza a separação de instruções com indícios que estão localizadas no início de um método que passam a pertencer a um módulo de aspectos. A segunda utiliza instruções com indícios que estão localizadas no final do método. Adendos anteriores (*before*) ou posteriores (*after*) são gerados nas reestruturações de extração de início e de final de método, respectivamente.

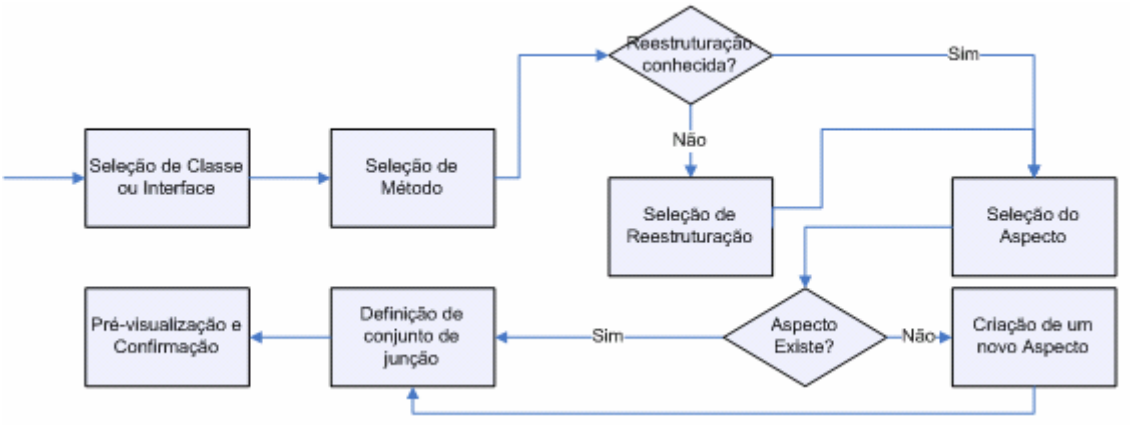

Figura 4.6: Seqüência de telas do Assistente de Reestruturação de Instruções

A tela final desse assistente é a de pré-visualização e confirmação, para coletar a quantidade de instruções extraídas do método e visualizar as alterações na classe modificada e no módulo de aspecto, que recebe um conjunto de junção e um adendo.

Na Figura 4.7, o código-fonte da classe é mostrado à esquerda, juntamente com indicações do trecho que será extraído da classe (trecho em vermelho e fonte Tachado) . À medida que a numeração do componente acima do editor da classe é alterada, mais instruções são marcadas para extração e ambos editores são atualizados. Nesse caso, a reestruturação de extração de início de método foi selecionada, pois a primeira instrução marcada é a primeira que aparece no método. Se duas instruções tivessem sido selecionadas, as duas primeiras instruções seriam marcadas e assim por diante. Da mesma forma, se a reestruturação de extração de fim de método fosse aplicada, a última instrução seria marcada quando o número de instruções a ser extraído fosse um. Considerando duas instruções selecionadas, as duas últimas instruções do método seriam marcadas e assim sucessivamente.

Além das marcações em vermelho, podem também ter marcações em amarelo feitas pelo editor de classes, para indicar que o trecho apresenta indícios, porém não foram selecionados para serem extraídos. Por outro lado, as marcações em amarelo no editor do módulo de aspectos, localizado na parte direita da tela, destacam o conjunto de junção e adendo a ser criado para acomodar as instruções extraídas da classe.

Dependendo das informações coletadas, o campo de recomendação de reestruturação pode trazer textos indicando as ações que devem ser tomadas, antes ou depois que as alterações desse assistente foram aplicadas, para que o código seja compilado corretamente. Essas recomendações podem ser: a necessidade de adicionar

modificador de módulos de aspectos, a de reordenar instruções, a de introduzir atributos, etc. No exemplo da Figura 4.7 nenhuma recomendação foi gerada.

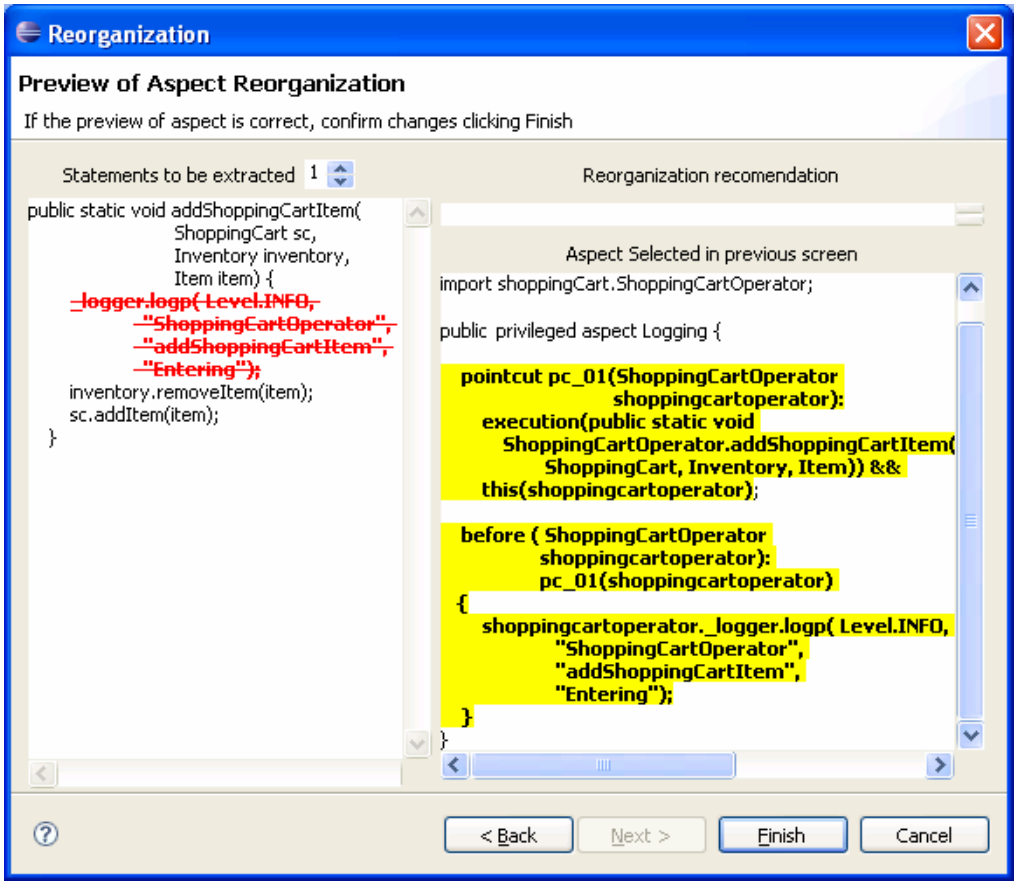

Figura 4.7: Pré-visualização de alterações

4.6. Persistência do Modelo de Indícios

Informações usadas para identificação de indícios não são fixas visando maior flexibilidade ao usuário. Assim, no ReJAsp, as funções de cadastro, atualização e remoção foram implementadas para que o usuário possa definir tipos e padrões de nomenclatura de variáveis e de cadeias de caracteres que sob seu ponto de vista, são indícios de um determinado aspecto, ao invés de mantê-los engessados no código do ReJAsp.

Além disso, alterações da API da linguagem Java resultantes da adição ou atualização de pacotes, de classes e de interfaces exigem a revisão de indícios já conhecidos e promovem oportunidades para descoberta de novos. A Figura 4.8 exibe a

seqüência de telas desse assistente, sendo que mais detalhes sobre o seu uso podem ser obtidos no Guia do Usuário (KAWAKAMI e PENTEADO, 2007).

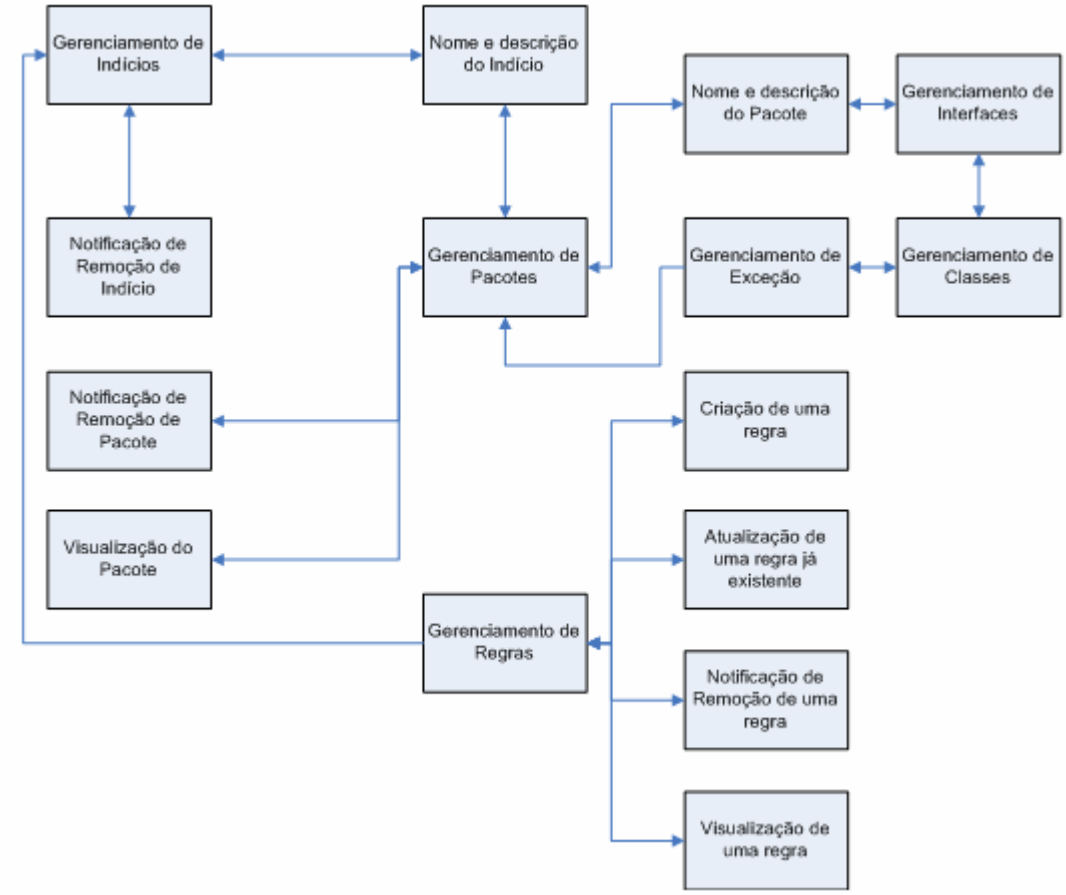

Figura 4.8: Seqüência de telas do Assistente de Gerenciamento de Indícios

4.6.1. Persistência em arquivo XML

A persistência de indícios é realizada em arquivo XML e sua leitura e escrita por meio da biblioteca DOM (*W3C Document Object Model*). Essa biblioteca manipula o documento XML como uma estrutura de árvore, tratando os seus elementos, atributos e textos como nós. O criador da DOM é um consórcio de empresas de tecnologia, conhecida por W3C, *World Wide Web Consortium* (WORLD WIDE WEB CONSORTIUM, 2007), e define essa biblioteca como uma interface neutra de plataforma e de linguagem, permitindo que programas e scripts acessem e atualizem dinamicamente o conteúdo, estrutura e estilo de um documento.

De acordo com Harold (2002), DOM possui vários problemas relacionados à dificuldade de uso, assim ele defende a utilização de JDOM (JDOM, 2007), uma API para o processamento de documentos XML baseada também em uma representação de árvore. Ainda segundo o autor, JDOM por ser mais intuitivo possibilita a geração de código com maior clareza e simplicidade. Durante a implementação do módulo de leitura e de escrita de indícios em XML, pôde-se constatar as características positivas da JDOM. Entretanto, com a atualização do ambiente de desenvolvimento do ReJAsp, que passou a utilizar a versão 3.2 do Eclipse e JDK 5, os módulos de leitura e de escrita de XML deixaram de funcionar. Foram realizadas pesquisas sem sucesso na busca de versões mais atualizadas do JDOM. A ausência de um suporte confiável e de atualizações periódicas tornou a utilização de JDOM discutível, pois o ReJAsp ficaria atrelado a uma base tecnológica antiga compatível ao JDOM. Por esse motivo, decidiu-se descartar o módulo de leitura e de escrita de XML por JDOM. O novo módulo de processamento de XML do apoio computacional teve como base a biblioteca DOM que apresenta atualização periódica.

O XML usado pelo ReJAsp está condizente com as informações do DTD, expressas na Figura 4.9. A definição de tipo de documento (*Document Type Definition* - DTD) especifica blocos de construções permitidas para XML, determinando uma estrutura de documento contendo uma lista de elementos e de atributos. Como pode ser observada na Figura 4.9, a DTD pode ser declarada em um documento XML.

A Figura 4.10 apresenta um trecho de XML contendo as informações dos indícios cadastrados inicialmente no ReJAsp.

4.7. Abordagens Similares ao ReJAsp

Além da abordagem Aspecting, existem outras que auxiliam a identificação de indícios ou candidatos a aspectos (mineração de aspectos) que são comentados a seguir.

AspectBrowser (GRISWOLD *et al*, 2000) é uma ferramenta desenvolvida também como um *plug-in* do ambiente Eclipse que permite a visualização dos indícios no código-fonte, mediante a definição prévia de expressões regulares, definidas pelo usuário no padrão *grep*. Os resultados são visualizados em barras verticais, assemelhando-se com a representação gráfica da visão *Aspect Visualization*, contida no *plug-in* AJDT (ASPECTJ DEVELOPMENT TOOLS, 2007). O ponto fraco dessa

ferramenta é a utilização de buscas léxicas pelo código-fonte, discutido na Seção 3.2.4, possibilitando a ocorrência de uma quantidade considerável de falsos indícios.

```
<!DOCTYPE Indications [

<!ELEMENT Indications (indication+)>
<!ELEMENT indication (name, description, Packages, Rules*)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT Packages (package+)>
<!ELEMENT package (name, description, Interfaces?, Classes?,
Exceptions?)>
<!ELEMENT Interfaces (interface+)>
<!ELEMENT Classes (class+)>
<!ELEMENT Exceptions (exception+)>
<!ELEMENT interface (name, description)>
<!ELEMENT class (name, description)>
<!ELEMENT exception+ (name, description)>
<!ELEMENT Rules (rule+)>
<!ELEMENT rule (target, matchingRule, caseSensity, Words)>
<!ELEMENT target (#PCDATA)>
<!ELEMENT matchingRule (#PCDATA)>
<!ELEMENT caseSensity (#PCDATA)>
<!ELEMENT Words (word+)>
<!ELEMENT word (#PCDATA)>

<!ATTLIST indication index ID #REQUIRED>
<!ATTLIST package index ID #REQUIRED>
<!ATTLIST interface index ID #REQUIRED>
<!ATTLIST class index ID #REQUIRED>
<!ATTLIST exception index ID #REQUIRED>
<!ATTLIST rule index ID #REQUIRED>

]>
```

Figura 4.9: DTD do arquivo XML de indícios

A ferramenta AMT, *Aspect Mining Tool* (HANNEMANN e KICZALES, 2001), emprega duas técnicas distintas: mineração de aspectos baseada em texto e baseada em tipo. A primeira depende da presença de convenções de nomenclaturas de tipos, métodos, variáveis e classes. Por outro lado, a técnica baseada em tipo reconhece a declaração de tipos e seus objetos, não dependendo de convenções de nomes no código. Mesmo que o processo de identificação de indícios apresentado na Seção 3.4.1, também seja baseado na análise de tipos, eles se diferenciam. Os tipos analisados na AMT não são associados a pacotes, possibilitando maior ocorrência de falsos indícios, conforme foi discutido na Seção 3.3. Um dos problemas verificados na AMT foi a não integração

da ferramenta com um ambiente de desenvolvimentos integrado e a necessidade em utilizar uma versão muito antiga do JDK.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Indications>
  <indication index="0">
    <name>Database Persistence</name>
    <description>Indication for database </description>
    <Packages>
      <package index="0">
        <name>java.sql</name>
        <description> API for accessing database</description>
        <Interfaces>
          <interface index="0">
            <name>Connection</name>
            <description>A connection (session) </description>
          </interface>
          <interface index="1">
            <name>PreparedStatement</name>
            <description>An object SQL statement.</description>
          </interface>
        </Interfaces>
        <Classes>
          <class index="0">
            <name>SQLPermission</name>
            <description>The permission data.</description>
          </class>
        </Classes>
        <Exceptions>
          <exception index="0">
            <name>SQLException</name>
```

Figura 4.10: Trecho de um exemplo de arquivo XML

No apoio computacional desenvolvido a navegação e a visualização de indícios encontrados no código-fonte são realizadas com base na Árvore de Indícios. Uma alternativa é o uso de JQuery (JANZEN e VOLDER, 2003) que permite a busca de subconjuntos específicos de elementos do código-fonte, inclusive candidatos a aspectos. Para isso, essa ferramenta combina características de navegadores hierárquicos e exploração do código com a especificação de linguagens de consultas. O ponto negativo dessa ferramenta é quanto à sua usabilidade, pois a definição de consultas demanda conhecimento de sua linguagem. Além disso, Jazen e Volder (2003) afirmam que JQuery não provê apoio direto à manipulação de aspectos. Nesse sentido, a Árvore de Indícios e os assistentes existentes no ReJAsp apresentam maior facilidade de uso e maior enfoque nas atividades de identificação e reestruturação de aspectos.

Breu *et al* (2006) defendem que os aspectos emergem com o tempo e, para isso, propõem que a identificação de indícios pode ser baseada no histórico do sistema de versões CVS (do inglês, *Concurrent Version System*), ao invés de apenas uma única versão de software como é observado nos métodos convencionais. O processo de identificação de indícios verifica a freqüência com que a chamada dos métodos é realizada, considerando as mais freqüentes como indícios. Esse método foi desenvolvido em um protótipo denominado HAM, com a desvantagem de não estar integrado a nenhum ambiente de desenvolvimento. O experimento conduzido pelos autores em sistemas pequenos e com histórico do CVS reduzido obteve baixa precisão, em torno de 60% de acerto. Como não existem garantias de que o sistema legado seja mantido pelo CVS ou que modificações não sejam feitas com freqüência, pode ser discutível a utilização desse protótipo. Nesse ponto, o ReJAsp, apresentado neste Capítulo, tem a vantagem de integração com o IDE Eclipse e eficiência independente da utilização de um sistema de controle de versão e do tamanho do sistema legado.

Com o enfoque diferente das demais ferramentas apresentadas, AOP Migrator (BINKLEY *et al*, 2006) tem o objetivo de refatorar sistemas em Java de modo iterativo. Para isso, contempla o apoio para as refatorações de extração de: início/fim de método/tratamento de exceção, início/fim de chamada, condicional, pré-retorno, *wrapper* e tratamento de exceção. A sua instalação depende da instalação de versões antigas do IDE Eclipse e do *plug-in* AJDT. Em comparação com o ReJAsp, esse tem a vantagem de poder ser utilizado em versões mais atuais do Eclipse e do AJDT, além de ter a função de identificação de indícios, que não é contemplada no AOP Migrator.

4.8. Considerações Finais

A construção de um apoio computacional para a reengenharia de interesses transversais não é uma atividade trivial, pois diversas características são imprescindíveis em sua concepção: facilidade de uso, compatibilidade com ambiente de desenvolvimento integrado (IDE), desempenho, persistência de informações, acurácia e portabilidade. Além disso, deve cumprir com sua função primordial: auxiliar na migração de sistemas legados OO implementados em Java para sistemas OA em AspectJ, mantendo sua funcionalidade. Devido à complexidade envolvida em sua implementação, o modelo de prototipação de software foi adotado, possibilitando

fragmentar todo o desenvolvimento em iterações ou ciclos. Assim, a cada iteração foi possível avaliar e testar o apoio computacional, verificando a pertinência de cada função na prática, revisando os requisitos estabelecidos inicialmente.

Mesmo se tratando de uma versão do protótipo do ReJAsp, o esforço em definir a arquitetura, o funcionamento, a interface, as heurísticas de identificação de indícios e a persistência de dados foi bastante significativo.

Ao invés de prover um sistema de ajuda acoplado ao ReJAsp, decidiu-se documentar os procedimentos de instalação e uso em um documento denominado de Guia do Usuário. Como ainda não se trata de um software final, o tempo gasto para elaborar um sistema de ajuda completo e integrado não se justifica, devido às revisões a serem feitas até a versão final seja alcançada.

A depuração de código-fonte e diversos testes de desenvolvimento foram realizados para que o ReJAsp esteja operacional e correto. Mesmo que esses testes contribuam com a maturidade do apoio computacional, não é o modo indicado para sua avaliação. Assim, é necessário que desenvolvedores ou mantenedores de software, não envolvidos em sua construção participem de sua avaliação. Nesse sentido, dados qualitativos e quantitativos foram coletados a partir de estudos de caso, sendo apresentados e discutidos no Capítulo 5.

Capítulo 5

Estudos de Caso

5.1. Considerações Iniciais

Um protótipo para o apoio computacional para a reengenharia de interesses transversais, denominado ReJAsp, foi proposto no Capítulo 4, cuja construção foi realizada como um *plug-in* para o ambiente de desenvolvimento Eclipse. Por se tratar de uma aplicação cuja interação do desenvolvedor é freqüente e determinante no processo de transformação de um sistema legado em Java para outro equivalente em AspectJ, questões de usabilidade foram consideradas. O mecanismo de identificação de indícios baseada em análise sintática do código-fonte apresenta certo grau de complexidade para sua implementação e foi considerada importante na elaboração do protótipo de apoio por gerar melhores resultados quando comparada com análise léxica.

Neste Capítulo é discutida a avaliação do protótipo ReJAsp, realizada por meio de estudos de caso e são brevemente comentadas algumas questões relacionadas ao seu refinamento. Os capítulos anteriores trataram das primeiras fases do modelo de

prototipação de software: obtenção de requisitos, projeto rápido, construção, avaliação, refinamento e construção de produto final.

A Seção 5.2 descreve como foi a condução dos estudos de caso, ou seja, a definição dos objetivos, hipóteses, medições, objetos de análise, alocação de grupos, seleção de sistemas legados, planejamento do experimento e das etapas dos estudos de caso. Na Seção 5.3 os resultados obtidos em cada etapa dos estudos de caso para os sistemas considerados são apresentados. Na Seção 5.4 são discutidos os resultados obtidos a partir das informações coletadas de questionários aplicados às grupos envolvidas nos experimentos. Na Seção 5.5 as considerações finais sobre os estudos de caso finalizam a discussão deste Capítulo.

5.2. A Condução dos Estudos de Caso

A avaliação do protótipo desenvolvido é necessária para verificar a validade e o refinamento dos requisitos de software especificados. Assim, os resultados obtidos na avaliação são usados para determinar se um novo protótipo ou o produto final deve ser construído. Para que o ReJAsp possa ser avaliado foram conduzidos estudos de caso envolvendo sistemas de informação de pequeno porte implementados em Java. A discussão dos resultados obtidos com a reengenharia realizada é feita com base na avaliação dos esforços gastos quando ela é feita manualmente e com o apoio computacional.

5.2.1. Objetivos e hipóteses

Os estudos de caso têm o objetivo de verificar o tempo gasto com a reengenharia de sistemas legados em Java para AspectJ feita de modo manual e utilizando o ReJAsp. Dessa forma, deseja-se verificar se existe economia de tempo quando o apoio computacional é usado e se a mesma é significativa.

Outro ponto a ser avaliado nos estudos de caso é a usabilidade do ReJAsp, para constatar a sua facilidade de aprendizado e de uso, além das chances de sua aceitação pelos desenvolvedores. Devido à utilização de um processo parcialmente automático, isto é, ocorrência de grande interação com o desenvolvedor, questões de usabilidade foram consideradas durante a fase de construção e de avaliação.

A qualidade e eficácia da reengenharia de sistemas legados com e sem o uso do apoio computacional também é discutida com base na análise de código-fonte dos sistemas implementados.

5.2.2. Medições e observações

Para que os dados pudessem ser medidos e observados foi elaborada e distribuída aos desenvolvedores uma planilha para que os tempos gastos com o processo de reengenharia dos sistemas manual e parcialmente automatizado (usando ReJAsp) fossem registrados. Esses tempos incluem a quantidade de horas usadas para o entendimento do código do sistema legado e também o de implementação em AspectJ do sistema. Outras informações que também constam dessa planilha são: a quantidade de membros do grupo presente, descrições e comentários a respeito da atividade desenvolvida. O tempo final é calculado com base na somatória de tempos registrados previamente multiplicados pela quantidade de membros presentes para cada atividade. As descrições e comentários são úteis para a identificação de eventuais problemas encontrados pelo grupo e que futuramente podem ser utilizados para melhorar o apoio computacional.

As seguintes métricas foram coletadas a partir do sistema legado (Java) e do sistema resultante em AspectJ: LOC, quantidade de classes, de métodos e de atributos, entre outras. Outra medida gerada a partir do ReJAsp refere-s à quantidade de indícios de aspectos encontrados. Os resultados considerados ideais, depois da aplicação do processo de reengenharia, são aqueles em que os sistemas resultantes não contenham mais aspectos espalhados no sistema. Entretanto, diversas restrições impedem que essa situação seja alcançada, entre elas pode-se citar: falta de experiência de desenvolvedores em programação orientada a aspectos, prazos limitados para execução dos estudos de caso, alocação não-integral dos grupos de desenvolvimento para a realização desses estudos. Por esse motivo, os sistemas resultantes podem conter aspectos remanescentes no código e a medida de indícios pode ser usada para evidenciar uma dimensão aproximada da quantidade desses que ainda restam em cada sistema e quantidade dos que foram tratados, ou seja, separados em módulos de aspectos.

Não foi estabelecida nenhuma medida de confiabilidade dos sistemas implementados. Contudo, os grupos foram instruídos para a realização de testes de

regressão de modo que se possa garantir que o comportamento externo do sistema resultante é equivalente ao do sistema legado.

5.2.3. Planejamento do experimento

Para a montagem dos grupos participantes foi distribuído um questionário para obter informações: a) relacionadas ao conhecimento de cada membro do grupo com relação à linguagem Java, programação orientada a aspectos, AspectJ, Eclipse; b) relacionadas ao uso do ReJAsp quanto à sua usabilidade, eficácia do identificador de indícios, praticidade de assistentes, utilidade do visualizador de indícios e a documentação para uso (KAWAKAMI e PENTEADO, 2007).

5.2.4. Grupos de desenvolvimento

Para a realização dos estudos de caso grupos foram formados a partir do conhecimento de cada um dos desenvolvedores (inicialmente, 17) considerando-se as respostas obtidas quando da aplicação dos questionários antes do início do experimento, como citado na Seção anterior. A atividade de aplicação do questionário foi importante para determinar e balancear os grupos, a fim de que as comparações e discussões de resultados sejam mais representativas e apropriadas. Dos 17 desenvolvedores, 2 foram descartados das primeiras etapas do experimento por apresentarem conhecimento muito mais avançado em Java, POA, AspectJ e Eclipse, porém alocados em um estudo de caso separado. Quatro grupos foram formados: G1 com 3 desenvolvedores (grupo com maior experiência) e os demais grupos G2 (três integrantes com conhecimento de AspectJ e OA, todos com bom conhecimento de Java), G3 e G4 com quatro desenvolvedores (dois integrantes com algum conhecimento de AspectJ e OA, todos com conhecimento de Java em diferentes níveis).

5.2.5. Código-fonte de sistemas legados

Para a realização do experimento, quatro sistemas legados em Java foram selecionados: de agência bancária, de biblioteca, de loja de calçados e de gestão de padarias. O sistema de biblioteca é um bastante simples, *opensource,* obtido de JLibrary (2007). Os demais sistemas foram obtidos a partir de trabalhos realizados por alunos de graduação do curso de Bacharelado em Ciência da Computação da UFSCar.

Antes de submeter os sistemas aos grupos, a reengenharia desses sistemas para AspectJ foi realizada por esse autor, manualmente, com o intuito de verificar a presença de indícios de *logging*, persistência em banco de dados ou em memória temporária. Ressalta-se que a planilha com as informações de horas gastas e LOC implementadas foram anotadas para analisar o grau de dificuldade no entendimento e na reengenharia desses sistemas. A partir desses dados, os sistemas que demandaram maior esforço foram considerados mais complexos e atribuídos aos grupos mais bem preparados.

5.2.6. Execução dos experimentos

Os experimentos foram conduzidos em duas etapas: a primeira manualmente e a segunda com o uso do ReJAsp.

Para amenizar a diferença de conhecimento entre os desenvolvedores foi realizado breve treinamento envolvendo a base teórica de POA e da linguagem AspectJ, antes da primeira etapa dos estudos de caso. Os sistemas legados considerados mais complexos e trabalhosos para a realização da reengenharia, de gestão de padarias e de loja de calçados, foram atribuídos aos grupos com maior experiência, G1 e G2, respectivamente, como descrito na seção anterior. O grupo G3 recebeu o sistema de agência bancária e G4 recebeu o de biblioteca. Durante a realização dessa primeira etapa os desenvolvedores puderam consultar o avaliador (este autor) dos estudos de caso para eventuais dúvidas técnicas.

Findada a primeira etapa e antes de iniciar a seguinte, o apoio computacional foi apresentado aos grupos exemplificando o seu uso e o Guia do Usuário (KAWAKAMI e PENTEADO, 2007) foi disponibilizado aos grupos.

Para a segunda etapa dos estudos de caso, cada grupo recebeu um dos quatro sistemas implementados em Java utilizados na primeira etapa, porém diferente do primeiro, para realizar a reengenharia com o uso do ReJAsp. Os sistemas foram trocados entre os grupo, observando-se a experiência de cada um deles. Assim, G1 recebeu o sistema de loja de calçados; G2 recebeu o de gestão de padarias; G3 recebeu o de biblioteca e G4 o de agência bancária. Foi solicitado aos grupos que não trocassem informações sobre os sistemas e que, se necessário, esclarecessem qualquer dúvida com este autor.

Os dois desenvolvedores não considerados anteriormente, com conhecimentos avançados sobre Java, conceitos de programação orientada a aspectos, AspectJ e

Eclipse, fizeram estudos de caso separadamente. D1 fez experimentos com o sistema de agência bancária manualmente e usando ReJAsp com o sistema de loja de calçados; enquanto que D2 realizou a reengenharia dos mesmos sistemas de D1 utilizando somente ReJAsp, pois já havia realizado a reengenharia manual. O apoio computacional foi apresentado a D1 e disponibilizado o Guia do Usuário.

5.3. Resultados obtidos com a reengenharia dos sistemas

Os resultados obtidos, a partir da realização dos experimentos, são apresentados nas subseções seguintes para um mesmo sistema, de forma agrupada.

5.3.1. Sistema de biblioteca

No sistema resultante do processo de reengenharia manual realizado por G4 o código legado não foi alterado, devido à dificuldade do grupo na realização do estudo de caso. Assim, os dados desse grupo para esse estudo de caso foram usados para comparações.

Na segunda etapa, G3 gastou 26 horas para realizar a reengenharia do mesmo sistema usando ReJAsp e um único módulo de aspecto foi implementado contendo 195 linhas, 3 atributos, 2 constantes, 8 conjuntos de junção e 8 adendos. As informações coletadas do sistema original e as dos sistemas após a realização do experimento são mostradas na Tabela 5.1. Por convenção é usado o hífen (-) para indicar um dado não aplicável, como por exemplo, o tempo de reengenharia do sistema legado que é uma medição não coletada, ou para representar dados inválidos.

Os indícios identificados foram obtidos pela aplicação da função de identificação de indícios implementada no ReJAsp conforme descrita no Capítulo 4. A informação contida na linha de “Classes tratadas” são referentes àquelas classes que originalmente continham indícios no legado e que foram modificadas após a reengenharia. Assim, G3 conseguiu modificar 9 de 12 classes inicialmente com indícios, sendo que em 5 delas todos os indícios foram separados. Entretanto, a quantidade de indícios remanescente ainda é alta, 101 dos 169 existentes no legado. Com esses resultados o percentual é de 59,76 para indícios não separados. As possíveis razões desse resultado são a inexperiência e alocação em tempo parcial do integrantes do grupo.

Tabela 5.1 – Resultados para o sistema de Biblioteca

Métrica	Biblioteca		
	Legado	**Etapa 1 (G4)**	**Etapa 2 (G3)**
Métodos estáticos	2	-	2
LOC	3487	-	3537
LOC de aspecto	-	-	195
LOC de código base	3487	-	3342
Classes	31	-	31
Atributos	373	-	365
Pacotes	1	-	1
LOC em métodos	2605	-	2458
Métodos sobrescritos	2	-	2
Atributos estáticos	14	-	6
Métodos	101	-	114
Interfaces	0	-	0
Indícios	169	-	101
Classes com indícios	12	-	4
Classes tratadas	-	-	9
Tempo de Reengenharia	-	14	26

5.3.2. Sistema de gestão de padarias

O sistema resultante do processo de reengenharia manual realizado por G1, consumiu 12 horas e 25 minutos. A quantidade de linhas geradas no módulo de aspectos foi de 44, 6 conjuntos de junção, 6 adendos e 1 declaração intertipos foram criados.

A Tabela 5.2 mostra que a quantidade de indícios se manteve a mesma que a do sistema legado quando o apoio computacional não foi usado. Já o aumento de classes com indícios no processo manual para 9, deve-se ao fato de que G1 ao declarar estaticamente relacionamentos de classes com a interface `Compatible`, essas passaram a implementar o método `setComando` adicionando mais indícios de persistência ao banco de dados. Entretanto, alguns indícios foram separados de 3 classes.

O tempo gasto na segunda etapa foi de aproximadamente 13 horas e 30 minutos. O módulo de aspecto desse sistema contém 32 linhas de código, 3 conjuntos de junção, 3 adendos e 7 atributos introduzidos.

Comparando os resultados das etapas 1 e 2 verifica-se o experimento com ReJAsp demandou maior tempo. Entretanto, manualmente o grupo não foi capaz de diminuir a quantidade de indícios existentes no código, ou seja, os indícios existentes continuaram presentes, o que mostra que se um apoio computacional fosse usado poderia contribuir para que o grupo percebesse a existência de indícios de aspectos. Por

outro lado, o grupo que usou o apoio computacional teve mais sucesso em realizar a separação dos indícios, com a redução desses de 204 para 185.

Tabela 5.2 – Resultados para o sistema de gestão de padarias

Métrica	Sist. de Gestão de Padarias		
	Legado	Etapa 1 (G1)	Etapa 2 (G2)
Métodos estáticos	8	14	8
LOC	3104	3196	3119
LOC de aspecto	-	44	32
LOC de código base	3104	3152	3087
Classes	39	39	39
Atributos	199	199	192
Pacotes	1	1	1
LOC em métodos	1897	1880	1891
Métodos sobrescritos	0	0	0
Atributos estáticos	1	5	1
Métodos	362	382	362
Interfaces	0	1	0
Indícios	204	204	185
Classes com indícios	6	9	6
Classes tratadas	-	3	4
Tempo de Reengenharia	-	12:15	13:32

5.3.3. Loja de Calçados

O sistema resultante do processo de reengenharia manual realizado por G2, consumiu 30 horas e 30 minutos. Inspecionando os códigos entregues por G2, não foi verificada alteração nos arquivos de extensão java, ou seja, não foram reconhecidos indícios de aspectos. O grupo incluiu aspectos de *Logging* e de inserção de mensagens, não existentes no sistema original, no código em AspectJ. Infere-se também que a separação dos aspectos existentes não foi realizada no experimento provavelmente por dificuldades encontradas pelo grupo e os resultados dessa etapa não são usados em discussões.

Na segunda etapa do estudo de caso, G1 gastou 8 horas 45 minutos para realizar a reengenharia com auxilio do apoio computacional, sendo geradas 875 linhas, 43 conjuntos de junção e 43 adendos.

È interessante observar que quando G1 realizou manualmente a reengenharia do sistema gestão de padarias, Seção 5.3.2, apenas conseguiu separar uma parcela muito pequena de aspectos do código-fonte, 44 LOC no módulo de aspectos, enquanto que na segunda etapa foram 875 LOC e um maior percentual de indícios separados. Infere-se

que uma razão para a notável evolução do grupo é a utilização da árvore de indícios, permitindo tornar o desenvolvedor ciente dos possíveis aspectos remanescentes e que mereçam tratamento.

Tabela 5.3 – Resultados para o sistema de loja de calçados

Métrica	Loja de calçados				
	Legado	**Etapa 1 (G2)**	**Etapa 2 (G1)**	**D1 (Prot.)**	**D2 (Prot.)**
Métodos estáticos	20	-	21	20	9
LOC	4631	-	4920	4692	4335
LOC de aspecto	-	-	875	149	793
LOC de código base	4631	-	4045	4543	3542
Classes	55	-	55	55	43
Atributos	316	-	331	313	285
Pacotes	4	-	4	5	5
LOC em métodos	2973	-	2375	2897	2191
Métodos sobrescritos	0	-	0	0	0
Atributos estáticos	38	-	39	38	26
Métodos	423	-	423	423	378
Interfaces	0	-	0	0	0
Indícios	389	-	125	345	20
Classes com indícios	24	-	24	24	22
Classes tratadas	-	-	13	3	22
Tempo de Reengenharia	-	30:20	08:45	03:54	10:00

Em paralelo aos experimentos realizados pelos grupos, estudos de casos foram realizados pelos desenvolvedores D1 e D2, conforme mencionado anteriormente. Ambos aplicaram a reengenharia do sistema de loja de calçados usando ReJAsp. O tempo gasto por D1 foi aproximadamente de 4 horas e foram tratados indícios em 3 das 24 classes que continham indícios, com a presença de 345 indícios remanescente de um total de 389 inicialmente. Assim, é possível inferir que apenas 44 indícios do código-fonte foram tratados.

Já a reengenharia realizada pelo desenvolvedor D2 consumiu 10 horas, observando que 22 das 24 classes tiveram todos os indícios separados. Consultando suas anotações, D2 após fazer a separação de indícios, constatou que o código modificado para algumas classes poderia se tornar parte integrante de outras classes relacionadas. Por esse motivo, a quantidade de linhas de código do sistema resultante foi menor se comparado com o sistema legado.

5.3.4. Agência Bancária

O sistema resultante do processo de reengenharia manual realizado por G3, consumiu 42 horas. O módulo de aspecto criado é composto por 316 linhas de código, 1 atributo, 1 método, 8 conjuntos de junção e 8 adendos.

Tabela 5.4 – Resultados para o sistema de agência bancária

Métrica	Agência Bancária				
	Legado	**Etapa 1 (G3)**	**Etapa 2 (G4)**	**D1 (Man.)**	**D2 (Prot.)**
Métodos estáticos	1	1	-	1	1
LOC	1321	1392	-	1312	1439
LOC de aspecto	-	253	-	103	489
LOC de código base	1321	1139	-	1209	950
Classes	11	11	-	14	11
Atributos	101	102	-	97	94
Pacotes	1	1	-	3	2
LOC em métodos	995	800	-	1002	663
Métodos sobrescritos	1	1	-	1	1
Atributos estáticos	0	0	-	0	0
Métodos	96	100	-	101	81
Interfaces	0	0	-	0	0
Indícios	196	78	-	167	0
Classes com indícios	3	3	-	3	0
Classes tratadas	-	2	-	3	3
Tempo de Reengenharia	-	42:00	6:00	02:58	04:27

Na segunda etapa do estudo de caso, G4 gastou 6 horas, porém o sistema não compilava e, por ao compará-lo com o código original, notou-se que suas classes não sofreram qualquer alteração. Assim, por esse grupo por não ter alcançado o objetivo desse estudo de caso, esses resultados foram descartados.

D1 realizou a reengenharia do sistema de agência bancária de modo manual antes da realização da reengenharia do sistema de loja de calçados com o apoio computacional. Para esse caso, foram implementados 3 módulos de aspectos, contendo : 103 linhas de código, 2 atributos, 4 métodos, 1 conjunto de junção, 1 adendo e 3 declarações estática de tipos.

D2 realizou a reengenharia desse mesmo sistema utilizando o apoio computacional, em um tempo total de aproximadamente 4 horas e 30 minutos. O módulo de aspecto desenvolvido contempla 489 linhas de código, 3 atributos introduzidos, 15 métodos introduzidos, 12 conjuntos de junção, 12 adendos, 6 instruções de declaração estática.

O tempo gasto por D2 usando ReJAsp foi menor que o gasto por G3 para a reengenharia manual e houve também mais eficácia na separação de indícios. No caso de D1, que realizou a reengenharia mais rápida sem apoio computacional, o sistema possui alta incidência de indícios nas classes. Pode-se inferir que o apoio computacional contribuiu na reengenharia desse estudo de caso, mesmo que o consumo de tempo foi maior, quando se compara o número de indícios encontrados: 196 por D2 e 29 por D1.

5.4. Resultados obtidos por questionário

Após o término da segunda etapa do estudo de casos, questionários foram aplicados aos grupos G1, G2, G3 e G4 a fim de coletar suas opiniões a respeito do uso do ReJAsp na reengenharia de sistemas legados Java para sistemas em AspectJ. Um dos desenvolvedores de G4 não realizou o experimento, assim são consideradas 14 opiniões. Os resultados completos dos questionários e seus gráficos podem ser encontrados no Apêndice 2.

O questionário é composto de questões de múltipla escolha, variando as alternativas entre 4 e 5 dependendo da questão. O desenvolvedor pode assinalar apenas a alternativa mais adequada ou nenhuma alternativa. Há um espaço reservado, em cada questão, para que o desenvolvedor justifique a escolha da alternativa assinalada ou de nenhuma alternativa.

Quanto à facilidade de uso do ReJAsp 13 desenvolvedores consideraram fácil e somente um difícil. Muito provavelmente, esse desenvolvedor não interpretou corretamente a questão, pois ao justificar sua opinião, colocou que seu conhecimento em POA e em Java era bastante limitado, dificultando a execução do experimento, mas não apontou nenhuma falha de usabilidade no apoio computacional.

Em relação à eficiência da documentação (Guia do Usuário) houve unanimidade de que essa foi muito útil. Quanto à precisão de identificação de indícios, todos acreditam que foi correta na maior parte dos casos. Os desenvolvedores de G1, como comentário, colocaram que as instruções de importação de tipos não deveriam ser consideradas como indícios. Essa consideração não é pertinente dada à forma que o apoio computacional é baseado nas diretrizes de identificação de indícios, Capítulo 3, com todo processo de identificação iniciando-se pela análise de instruções de importação de tipos. A retirada dessa análise permitiria que tipos de mesmo nome, mas

de pacotes diferentes fossem reconhecidos como sendo o mesmo tipo, podendo gerar falsos indícios.

O grupo G3 em seus comentários relatou que indícios foram encontrados em um método abstrato, sendo que não continham indícios na opinião do grupo. Isso pode acontecer com o uso do ReJAsp quando o tipo de retorno ou o tipo de ao menos um parâmetro está relacionado a um interesse transversal. Nesse caso, provavelmente o grupo não estava ciente de que deveria estabelecer um entrecorte dinâmico para que o contexto seja captado na chamada do método e passado para o módulo de aspecto para tratar esse caso. É evidente que o entrecorte deve afetar as classes especializadas que implementam esse método.

Pelos questionários, todos os desenvolvedores relatam que a visualização de indícios foi útil durante a execução do experimento. Os integrantes de G1 e G4 enfatizam que a exibição do código, realçando o trecho com indícios, quando o mesmo é selecionado na árvore de indícios é um ponto forte do apoio computacional. Os membros de G2 acreditam que os indícios de importação de tipos poderiam ser ocultos da árvore de indícios e G3 acrescentou a sugestão de agrupamento de indícios adjacentes no código.

Para a questão relativa à facilidade de uso de assistentes de reestruturação, 10 desenvolvedores consideraram fácil, 2 difícil e 2 não opinaram. É importante ressaltar que os dois desenvolvedores que não opinaram, pertencentes a G3, tiveram o problema ao executar o assistente de reestruturação de código, mas não consultaram este autor.

O assistente de reestruturação foi considerado de difícil uso, por um dos desenvolvedores, devido à quantidade de passos nele existente, pois permite que algum passo seja ignorado ou feito incorretamente. As chances de algum passo ser ignorado são nulas no ReJAsp, pois todas as suas telas são travadas (botão de prosseguir desabilitado), até que informações válidas sejam colocadas pelo desenvolvedor. Além disso, na tela de pré-visualização as alterações a serem realizadas em código são visualizadas e os trechos alterados, tanto do código-base quanto módulo de aspecto, recebem destaque, evitando que informações incorretas sejam inseridas. Se houver qualquer problema há o recurso de usar o botão voltar e refazer algum passo. Uma forma de amenizar a quantidade de passos no assistente é a coleta de informações pelo contexto em que o assistente é acionado, discutido na seção de Trabalhos Futuros, no

próximo Capítulo. Outro desenvolvedor, que avaliou os assistentes de reestruturação de difícil uso, não justificou sua opção.

Os assistentes de reestruturação de código são usados para tratar quantidade limitada de problemas, assim, foi elaborada uma questão para avaliar a freqüência que esses assistentes foram empregados. A utilização dos assistentes na maioria dos casos foi realizada por 10 desenvolvedores,, 2 não opinaram (os mesmos que não opinaram anteriormente), 1 utilizou em metade dos casos e 1 raramente usou. O desenvolvedor que utilizou em metade dos casos, pertencia ao grupo que não conseguiu utilizar os assistentes completamente, e o desenvolvedor que utilizou raramente justificou sua opção devido a sua pouca familiaridade com ReJAsp.

O questionário continha uma pergunta para verificar se os desenvolvedores precisaram utilizar o recurso de gerenciamento de indícios. Como era esperado, a sua utilização não foi constatada na maioria dos casos por ser um recurso avançado do ReJAsp ou por não ter aplicabilidade nos sistemas implementados. Daqueles que utilizaram esse recurso, a maioria considerou de fácil uso, outra parcela avaliou como muito fácil. O único desenvolvedor que apontava esse assistente de difícil uso relatou que não usou essa função, o que é inconsistente.

O tempo de implementação variou não somente devido à complexidade dos sistemas e ao uso ou não do apoio computacional, mas também devido à dificuldade de modificação do sistema. Entretanto, cada desenvolvedor é capaz de notar possíveis economias de tempo geradas com o uso do apoio computacional. Isso foi verificado por 12 dos 14 desenvolvedores. Um desenvolvedor relatou pouca economia e o outro que observou alguma economia. É interessante ressaltar que o desenvolvedor que indicou que o apoio gerou pouca economia de tempo de reengenharia é o pertencente ao grupo que encontrou problemas na implementação do sistema de biblioteca previamente relatado. O desenvolvedor que não notou economia justifica que o tempo gasto com o aprendizado da ferramenta foi praticamente o mesmo tempo economizado na execução do experimento.

5.5. Considerações Finais

O estudo de caso apresentado foi realizado a partir de recursos limitados e disponíveis para este trabalho. Assim, resultados mais conclusivos poderiam ser

inferidos em estudos de caso ideal, utilizando uma quantidade maior de projetos e de grupos de desenvolvimento, bem como, alocação integral, homogeneidade de habilidades, presença deste autor na realização do experimento para controle estrito na coleta de dados.

Um fator que contribuiu para a geração de resultados discrepantes e não conclusivos foi concentração de atividades, um ou dois dias antes do prazo de entrega, para um dos grupos na primeira etapa do estudo de caso e para outro grupo, na segunda etapa do estudo de caso. Nesses casos, a execução do experimento teve baixa prioridade, comprometendo a qualidade da implementação.

Inspecionando os sistemas gerados, são verificadas deficiências dos desenvolvedores ainda com a programação orientada a objetos (POO). Problemas que poderiam ter sido resolvidos com POO não foram aplicados, complicando a solução usando AspectJ. A POA não tem objetivo de substituir completamente a POO e sim complementá-la, adicionando mecanismos para separação de interesses transversais, o que poderia abreviar o trabalho dos grupos.

A desistência de um dos desenvolvedores também afetou os resultados finais, já que houve desbalanceamento desse grupo em relação aos demais. Esse motivo pode justificar os resultados insatisfatórios gerados por eles.

Os resultados evidenciam que a quantidade de LOC dos sistemas implementados de modo manual ou usando ReJAsp é maior que a quantidade de LOC dos legados na maior parte dos casos. Isso se justifica pelo fato que a criação de aspectos requer a especificação de conjuntos de junção, adendos, instruções de declarações estáticas, etc que contribuem para o aumento do código.

O experimento teve como principal objetivo a separação de aspectos de sistemas legados OO para módulos de aspectos em AspectJ. Assim, melhorias adicionais como generalizações de conjuntos de junção, identificação e eliminação de trechos códigos duplicados e outras análises mais apuradas não foram incluídas, devido à restrição de tempo dos experimentos e à preocupação de se manter o enfoque principal do trabalho.

Após a criação de módulos de aspectos, identificação e separação de aspectos provenientes de códigos legados em Java, otimizações dos módulos de aspectos gerados podem ser realizadas utilizando refatorações de aspectos, definidas por Monteiro (2004). Essas otimizações fogem da proposta deste trabalho, mas são pertinentes em trabalhos futuros.

Este trabalho não teve como objetivo final a geração de um produto final de apoio computacional para a reengenharia de interesses transversais, a principal proposta foi a identificação de correções, de melhorias, de aplicabilidade, de usabilidade e da viabilidade do apoio computacional. Mesmo com as limitações encontradas nos estudos de caso realizados, a avaliação do ReJAsp foi realizada com sucesso, pontos fortes e fracos foram identificados e são discutidos no Capítulo 6, bem como os trabalhos futuros.

Capítulo 6

Considerações Finais

6.1. Introdução

A manutenção de software é uma atividade caracterizada pelo alto custo e esforço do projetista. Porém, é inevitável para qualquer tipo de software, pois os usuários constantemente solicitam modificações de ambiente operacional, a inserção de novas funções, correção de erros, entre outras. Sempre se realiza manutenção seja corretiva, adaptativa, perfectiva ou preventiva (PRESSMAN, 2006).

A Programação Orientada a Objetos (POO) permite que problemas sejam decompostos em objetos com a abstração de comportamentos e de dados em uma simples unidade (KICZALES *et al* 2007b). Embora a POO tenha obtido grande sucesso na modelagem e implementação de sistemas de software complexos, alguns problemas ainda persistem. Experiências práticas com projetos amplos e complexos mostram que programadores constantemente se deparam com código de difícil manutenção, devido à constante dificuldade na separação de interesses, principalmente os interesses transversais.

Uma importante tecnologia de Programação Pós-Objetos é a Programação Orientada a Aspectos (POA) que provê mecanismos para implementar cada interesse transversal separadamente. Por meio do combinador de aspectos (*weaving*), o código-base e os módulos que tratam os interesses transversais são combinados para a posterior composição do sistema final. Dentre as linguagens de Orientação a Aspectos (OA), AspectJ (KICZALES *et al* 1997a) é a mais difundida e utilizada.

Uma forma de manutenção preventiva de sistemas legados OO é pela separação de interesses transversais que se encontram espalhados, gerando um sistema implementado em OA. Nesse sentido, a abordagem Aspecting (RAMOS, 2004) se destaca por apresentar um modo direto e ordenado ao realizar a reengenharia do sistema implementado originalmente em Java para a linguagem AspectJ.

Entretanto, a aplicabilidade da Aspecting é comprometida devido à grande quantidade de diretrizes a serem seguidas. O esforço em sua execução se torna diretamente proporcional ao tamanho do sistema legado e ao número de interesses transversais presentes. Uma maneira de contornar esse problema é via automatização de parte das atividades envolvidas na Aspecting.

Assim, este trabalho apresentou um apoio computacional para Reengenharia de sistemas Java para AspectJ, denominado ReJAsp. Esse apoio envolveu a definição do Modelo de Indícios, uma representação de indícios configurável, atende à de persistência de dados; com a especificação de diretrizes de identificação de indícios e de reestruturação de código, Usa o Modelo de Indícios e estendeu algumas das diretrizes da abordagem Aspecting. A construção de um protótipo do ReJAsp foi realizada como *plug-in* do ambiente integrado de desenvolvimento (IDE) Eclipse e sua avaliação foi realizada por meio de estudos de caso.

A Seção 6.2 discute os resultados obtidos com o experimento realizado; a Seção 6.3 descreve as limitações do apoio computacional. As contribuições deste projeto são listadas na Seção 6.4 e, finalmente a Seção 6.5 apresenta alguns trabalhos que podem dar continuidade a este.

6.2. Discussão dos Resultados

O apoio computacional construído com as características anteriormente descritas foi avaliado por experimentos realizados em duas etapas, com quatro grupos

constituídos por 3 ou 4 de desenvolvedores com conhecimentos intermediário em Java e básico em POA como apresentado no Capítulo 5.

Um questionário foi elaborado para avaliar algumas das características do ReJAsp, conforme descrito no Capítulo 4. A compilação das respostas ao questionário indiciou que sua usabilidade é boa e que o Guia de Usuário (KAWAKAMI e PENTEADO, 2007) elaborado também contribui para o aprendizado do apoio computacional.

Outro ponto avaliado foi o mecanismo de identificação de indícios que apresentou corretitude para a maior parte dos casos. Um dos problemas relatados por um dos grupos está relacionado com a inclusão de algumas instruções de importação de tipos como indícios. De acordo com as discussões apresentadas no Capítulo 3, essas instruções são os primeiros indícios avaliados em um código-fonte Java. Se nenhum indício for encontrado ainda nas importações de tipo, o arquivo é então descartado, não havendo processamento, o que gera economia de tempo. A associação de indícios a essas instruções também é justificada devido ao relacionamento de tipos com seus respectivos pacotes, diferencial do mecanismo de identificação deste trabalho com as técnicas de mineração de aspectos baseada em tipos (HANNEMANN e KICZALES, 2001). Dessa forma, há aumento de precisão na identificação de indícios, visto que tipos de mesmo nome são distinguidos.

A visualização dos indícios também foi indicada como um ponto forte do apoio computacional. Outra característica apontada como importante pelos participantes do experimento foi a exibição do código-fonte no editor, destacando o trecho de código com indícios, quando seu nó equivalente é clicado via árvore de indícios.

Os assistentes de reestruturação de código foram usados na maioria das vezes, de acordo com dados coletados pelas respostas aos questionários, e foram classificados como fáceis de usar pela maioria dos desenvolvedores. Com isso, acredita-se que essa função de software atingiu os resultados inicialmente esperados, mesmo que melhorias nos assistentes de reestruturação de código ainda estejam previstas.

A avaliação do assistente de gerenciamento de indícios não pode ser feita efetivamente nesse experimento, devido a não utilização desse assistente pela maior parte dos grupos. Infere-se que isso tenha ocorrido por sua utilização demandar bom conhecimento de POA, do processo de identificação de indícios, do Modelo de Indícios

e do uso do ReJAsp. Alguns dos desenvolvedores que o utilizaram apontaram o seu uso como fácil

A idéia inicial de comparar o tempo gasto entre o processo de reengenharia manual e com apoio computacional foi possível somente em um dos quatro sistemas usados nos estudos de caso. Segundo as opiniões obtidas pelas respostas ao questionário, pôde-se inferir que o apoio computacional é capaz de gerar economia de tempo na atividade de reengenharia de sistemas legados. Entretanto, não foi possível estimar essa economia com os registros de tempos apresentados nas planilhas confeccionadas pelos grupos. Para uma melhor avaliação dos resultados, este autor realizou inspeção nos códigos dos sistemas entregues.

Ainda em relação ao tempo gasto com a reengenharia de sistemas, é interessante notar que de acordo com os dados obtidos, o sistema de gestão de padarias teve seu tempo menor na reengenharia manual do que utilizando ReJAsp. Esse resultado inesperado é explicado quando a quantidade de indícios coletados das duas formas é comparada. Manualmente, a quantidade de indícios no código base se manteve igual ao do legado, devido à separação de poucos aspectos e com a inserção de novos aspectos pelo grupo. Com apoio computacional, obteve-se maior sucesso na separação de indícios, devido à diminuição da quantidade de indícios do sistema entregue. Assim, a melhor qualidade do sistema pode justificar o tempo adicional de aproximadamente 1 hora gasto no segundo experimento, sendo que a utilização do ReJAsp, contribuiu para a separação mais efetiva dos aspectos.

Resultado similar foi encontrado na reengenharia do sistema de agência bancária. Novamente, a economia de tempo esperada pela utilização do ReJAsp não foi constatada, contudo, com ele foi possível o tratamento de número maior de aspectos.

A principal vantagem observada com a realização desses experimentos é a identificação e a visualização de indícios conferida pelo apoio computacional, permitindo tornar os desenvolvedores cientes dos possíveis candidatos a aspectos. Em contrapartida, o desconhecimento de indícios remanescentes no código contribui para a finalização precoce das atividades de reengenharia do sistema, mantendo grande número de aspectos espalhados e entrelaçados pelo sistema.

6.3. Limitações

Uma das limitações ao apoio computacional é identificar somente três categorias de interesses transversais, sendo que a Aspecting apresenta diretrizes para identificar outros que não foram tratados aqui. Como comentado anteriormente, isso ocorre devido à diferença de formas de reconhecimento dos interesses transversais: a abordagem Aspecting usa análise léxica e seu processo é manual; o apoio computacional aqui apresentado utiliza a análise sintática.

O processo de identificação de indícios do sistema inteiro pode consumir uma quantidade de tempo de tempo considerável, dependendo de sua quantidade de arquivos, de aspectos e de linhas de código. Para os sistemas usados nos estudos de casos, o tempo de demora não influenciou na usabilidade do ReJAsp, entretanto, é possível notar diferenças de tempos entre sistemas durante a identificação de indícios. Muito provavelmente para sistemas de grande porte a usabilidade seja significantemente prejudicada por tempos de espera maiores.

A dificuldade na avaliação 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 sistemas legados de pequeno e médio porte que contivessem os interesses transversais apoiados pelo ReJAsp. Os sistemas disponibilizados para realização do experimento, como comentado anteriormente, não foram implementados com todos os recursos existentes em linguagens POO. Dessa forma, os estudos de caso realizados foram sem a presença deste autor, o que também impede que conclusões mais significativas sejam obtidas.

6.4. Contribuições

Pode-se citar como contribuições deste projeto:

- As definições do Modelo de Indícios, das diretrizes de identificação e de reestruturação de código, se mostraram extensões pertinentes da abordagem Aspecting, bem como a revisão dos interesses transversais de persistência em banco de dados, em memória temporária e adição do tratamento de *logging*, interesse transversal bastante comum em sistemas legados.

- A construção do ReJAsp permitiu a avaliação da viabilidade da abordagem Aspecting e de apoios computacionais que automatizem parte de suas atividades. Os próximos trabalhos e experimentos podem usar o ReJAsp, devido à facilidade de aprendizado e de uso, sendo auxiliados por uma documentação bem completa, o Guia do Usuário.
- O ReJAsp apresenta a vantagem de realizar tarefas de identificação e separação de aspectos. Em geral, as ferramentas computacionais realizam somente uma dessas atividades. A ausência de integração entre essas tarefas, além de exigir esforços adicionais em transformar as saídas de um apoio em entrada válida em outro, possibilitam a inserção de erros pelo desenvolvedor.
- O ReJAsp por ser um *plug-in* do ambiente Eclipse permite a utilização conjunta de outros recursos existentes nesse ambiente e em outros apoios computacionais também implementados como *plug-ins* do Eclipse.
- O uso de AST para identificação de aspectos possibilita a obtenção de resultados mais confiáveis, pois diferentemente da análise léxica que considera apenas informações textuais, detalhes da estrutura de programa são usados na detecção de indícios.
- O uso do ReJAsp pode ser feito por desenvolvedores com conhecimentos básicos ou avançados de Java e de AspectJ, por ser de fácil uso, possuir interfaces amigáveis e devido à grande interação com o desenvolvedor possibilita a realização da reengenharia com esforço moderado.

6.5. Trabalhos Futuros

A seguir são apresentados alguns trabalhos que podem ser realizados para amenizar as limitações apresentadas na Seção 6.3.

- Adição de novas categorias de indícios por meio de estudo de sistemas que contenham os interesses transversais de segurança, de controle de acesso, de *data profiling*, de repositório de conexões (*pooling*), de imposição de políticas (*policy enforcement*). A busca e análise de bibliotecas amplamente usadas que

tratem interesses transversais também podem contribuir para adição de novas categorias de indícios;

- Atualização da categoria de indício de *logging*, considerando a biblioteca log4j;
- Busca de sistemas que utilizem a biblioteca de monitoramento e de gerenciamento da execução de máquina virtual disponíveis a partir da versão 5 do JDK (`java.lang.management` e `javax.management`) , verificando se suas instruções compõem um interesse transversal, ou seja, apresentam-se espalhadas e entrelaçadas;
- Atualmente, a unidade de comparação é baseada em tipos. É possível tornar a granularidade da identificação de indícios mais fina, permitindo identificar indícios por métodos. Assim, a busca por interesses transversais de rastreamento se tornaria possível, pois as instruções: `System.out.print(<Mensagem>)` e `System.out.println(<Mensagem>)` poderiam ser buscadas. Nesse caso, apenas as chamadas de `print` e `println` seriam tratadas como indícios ao invés de todas as instruções relacionadas ao tipo `System`;
- Ativação de assistentes de reestruturação de código pelo contexto: o ReJAsp é capaz de identificar quais as possíveis reestruturações associadas a um elemento da árvore de indícios. Assim, a exibição de um menu *popup* quando o item é selecionado, poderia permitir a seleção de uma reestruturação aplicável ao elemento. Nesse caso, a quantidade de telas de assistente poderia ser abreviada, pois parte das informações solicitadas pelo assistente já foram coletadas pelo contexto em que o elemento foi selecionado. A seleção também poderia ser feita na seleção do elemento, via editor de código;
- Gerenciamento de indícios por contexto: a seleção de pacotes, classes ou interfaces a partir da árvore de indícios possibilitaria a criação ou atualização de uma categoria de indícios rapidamente;
- Agrupamento de indícios: no Capítulo 4 foi discutido que os indícios eram agrupados por instruções. A possibilidade do desenvolvedor agrupar várias instruções, em um único indício, poderia contribuir para uma melhor visualização desses, desde que instruções estejam adjacentes umas das outras.

Além desses trabalhos, existem algumas melhorias no apoio computacional desenvolvido que podem ser realizadas consumindo menor esforço:

- Atualização automática da árvore de indícios: qualquer alteração de código feita fora do apoio computacional requer a atualização da árvore de indícios. Atualmente, essa atualização deve ser feita pelo desenvolvedor, entretanto, a atualização automática da árvore de indícios pode ser realizada;
- Função de marcação de código por comentários: o ReJAsp pode permitir que marcações por meio de comentários em código-fonte sejam feitas para todos os indícios encontrados ou apenas para os indícios escolhidos pelo próprio desenvolvedor. Também pode existir uma tela que possibilite a especificação do padrão de comentário, por exemplo, que um número seqüencial possa ser inserido automaticamente pelo apoio computacional a cada marcação. Isso possibilitaria que apoios computacionais de modificação de código, que usem comentários para determinar trechos de códigos a serem separados, sejam usados conjuntamente com o ReJAsp;
- Assistente de conjunto de junção: para realizar a separação de determinados trechos de código-fonte, conjuntos de junção mais complexos são necessários. Dessa forma, pode-se implementar um assistente de conjunto de junção ativado após a seleção de um trecho de código;
- Adição de assistentes de reestruturação de código: permite ampliar as reestruturações de código já realizadas pelo apoio computacional, dando-lhe mais robustez e melhorando sua escalabilidade;
- Função de desfazer própria: na implementação atual é possível ter acesso à função de desfazer por meio do ambiente Eclipse, condicionando o desenvolvedor a abrir os códigos-fonte que são modificados pelo ReJAsp no editor previamente. Um controle da função de desfazer pelo próprio apoio computacional pode tornar essa função mais simplificada;
- Barra de status: quando a identificação de indícios é realizada, existe um tempo de espera variável de acordo com o tamanho do projeto. A adição de uma barra de status permite melhor acompanhamento do desenvolvedor no andamento da operação;
- Importação de projetos: o desenvolvimento de um assistente de importação de projetos legados, pode permitir que o projeto em Java fosse automaticamente convertido para AspectJ (os códigos não sofrem alterações, mas o tipo de projeto

é modificado de modo que o AJDT o reconhecesse), permitindo que o ReJAsp realize a identificação de indícios de todas as categorias cadastradas. Assim, ao final da importação, o desenvolvedor pode ser informado da quantidade de indícios de cada categoria e outras métricas podem mostradas e armazenadas no projeto. Posteriormente, comparativos das métricas coletadas quando o projeto legado foi importado com as métricas do sistema atual podem ser apresentados.

Referências Bibliográficas

AOSD.06 Disponível em: <http://www.aosd.net/2006>. Acesso em: 11 dez. 2006

AspectJ Development Tools Project. Disponível em: <http://www.eclipse.org/ajdt>. Acesso em: 29 jan. 2007

Ast View Disponível em: <http://www.eclipse.org/jdt/ui/astview/index.php>. Acesso em: 15 de jun 2007.

Badros, G. J. "JavaML: A Markup Language for Java Source Code". In: Proceedings of the 9th Internacional Conference. on the World Wide Web, Amsterdam, The Netherlands, May 2000.

Bennet, K. H. e Rajlich, V. T. “Software Maintenance and Evolution: a Roadmap”. In: Anthony Finkelstein, ed. The Future of Software Engineering. Limerick, Ireland. ACM Press, Páginas 75-87. 2000.

Binkley, D.; Ceccato, M.; Harman, M.; Ricca, F.; Tonella, P. “Automated Refactoring of Object Oriented Code into Aspects”. In: Proceedings of the 21st IEEE Conference on Software Maintenance (ICMS'05), 2005.

Binkley, D.; Ceccato, M.; Harman, M.; Ricca, F.; Tonella, P. “Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects”. In: IEEE Transactions on Software Engineering, 2006.

Bisbal, J.; Lawless, D.; Wu, B.; Grimson, J. "Legacy information systems: Issues and directions." IEEE Software, 1999.

Breu, S.; Zimmermann, T.; Lindig, C. "Aspect Mining for Large Systems". In: Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, Portland, 2006.

Clement, A.; Colyer, A.; Kersten, M. "Aspect-Oriented Programming with AJDT". In ECOOP Workshop on Analysis of Aspect-Oriented Software, July, 2003.

Chikofsky, E. J.; Cross II, J. H. “Reverse Engineering and Design Recovery: A Taxonomy”. In: IEEE Software, pág. 13-17, Janeiro 1990.

Chung, L.; Nixon, B.; Yu, E.; Mylopoulos, J. "Non-functional requirements in software engineering". In: Boston: Kluwer Academic, pág. 439, 1999.

Cysneiros, L.M., Leite, J.C.S.P. "Definindo Requisitos Não Funcionais". In: Simpósio Brasileiro de Engenharia de Software (SBES'97), pág. 49-54, 1997.

Deitel, H. M.; Deitel P. J. "Java, como programar", tradução: Edson Furnankiewicz, Bookman, 3ª edition, 2001, ISBN: 85-7307-727-1, Páginas 680 - 713.

Deursen, A. V., Marin, M., Moonen, L. "Aspect Mining and Refactoring". In: Proceedings First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE). University of Waterloo, 2003.

Dijkstra, E.W. “On the role of scientific thought” (EWD 447). In: Selected Writings on Computing: A Personal Perspective. Springer-Verlag, 1992.

Eclipse Project. Disponível em: <http://www.eclipse.org>. Acesso em: 29 jan. 2007.

Elrad, T.; Filman, E. R.; Bader, A. “Aspect-Oriented Programming”. Communications of the ACM, 2001a. 44(10): p. 29-32.

Elrad, T.; Aksit, M.; Kiczales, G.; Lieberherr, K.; Ossher, H. “Discussing Aspects of AOP”. Communications of the ACM, 2001b. 44(10): p. 33-38.

Filman, R.E.; Friedman, D. P. “Aspect-Oriented Programming is Quantification and Obviousness”. In: Position paper for the Advanced Separation of Concern Workshop at the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM, Minneapolis, Minnesota, USA, 2000.

Filman, R. E. “What is Aspect-Oriented Programming, Revisited”. In: Workshop on Advanced Separation of Concerns, 15th European Conference on Object-Oriented Programming, Budapest, June 2001.

Fowler, M.; Beck, K.; Brant, J.; Opdyke, W. “Refactoring: Improving the Design of Existing Code”. Addison-Wesley, 1st edition, 1999.

Gradecki, J. D.; Lesiecki, N.; “Mastering AspectJ: Aspect-Oriented Programming in Java”. Wiley Publishing, Inc., 2003.

Griswold, W.; Kato, Y.; Yuan, J. "Aspect Browser: Tool Support for Managing Dispersed Aspects". In: Workshop on Multi-Dimensional Separation of Concerns (ICSE), Ireland, 2000.

Hannemann, J.; Kiczales, G. "Overcoming the Prevalent Decomposition in Legacy Code". Position paper for workshop on Advanced Separation of Concerns, International Conference on Software Engineering (ICSE), 2001.

Hannemann, J.; Fritz, T.; Murphy, G. C. “Refactoring to Aspects - an Interactive Approach”. In: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange, 2003.

Hannemann, J.; Muphy, G. C.; Kiczales, G. “Role-Based Refactoring of Crosscutting Concern”. In: Proceedings of the 4th international conference on Aspect-oriented software development, 2005.

Harold, E. R. "Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP, and TrAX." Addison-Wesley, 2002.

Hessellund, A. “Refactoring as a Technique for the Reengineering of Legacy Systems”. ITU, København, 2004. Disponível em: <http://www.itu.dk/people/hessellund/refactoring/refactoring.pdf>. Acesso em: 12 fev. 2006.

Hugunin, J. "The Next Steps For Aspect-Oriented Programming Languages (in Java)". In NSF Workshop on New Visions for Software Design & Productivity: Research & Applications, Vanderbilt University, Nashville, TN, Dec. 13-14 2001. National Coordination Office for Information Technology.

Janzen, D.; Volder, K. D. "Navigating and querying code without getting lost". In: Aspect-Oriented Software Engineering, pages 178–187. ACM, 2003.

Java Technology. Disponível em: <http://java.sun.com/>. Acesso em: 15 jun. 2007.

JDOM. Disponível em < www.jdom.org >. Acesso em: 25 jun. 2007.

JLibray: Open Source Document Management System from your Desktop. Disponível em: <http://jlibrary.sourceforge.net/>. Acesso em: 4 jul. 2007.

Kawakami, D.; Penteado, R. “Apoio Computacional para a Refatoração de Aspectos”, In: Anais do II Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (WASP'05), Uberlândia, MG, 2005a.

Kawakami, D.; Penteado, R. “Manutenção de Sistemas OO utilizando OA”, In: Anais do II Workshop de Manutenção de Software Moderna (WMSWM), Manaus, AM, Brasil, 2005b.

Kawakami, D.; Penteado, R. “Guia do usuário para Apoio Computacional para a Reengenharia de Interesses Transversais”. Disponível em <http://www.dc.ufscar.br/~rosangel/ReJAsp/>.

Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C. V.; Loingtier, J. M.; Irwin, J. “Aspect-Oriented Programming”, In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, 1997a.

Kiczales, G.; Lamping, J.; Mendhekar, A. “A. RG: A Case-Study for Aspect-Oriented Programming”. In: SPL97. Xerox Palo Alto Research Center, Technical Report, 1997b.

Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. “An Overview of AspectJ” In: Proceedings of the 15th European Conference on Object-Oriented Programming, 2001.

Kitchenham, B.; Pickard, L.; Pfleeger, S. L. "Case Studies for Method and Tool Evaluation". IEEE Software, vol. 12, no. 4, pp. 52-62, Jul., 1995.

Laddad, R. “AspectJ in Action: Pratical Aspect-Oriented Programming”. In: Manning Publications Company, Connecticut - USA, 2003, 512 p.

Mens, T.; Tourwe, T. "A survey of software refactoring". IEEE Transactions on Software Engineering, 2004.

Monteiro, M. P. “Catalogue of Refactorings for AspectJ”. Technical Report UM-DI-GECSD-200402, Universidade do Minho, December 2004. Disponível em: <www.di.uminho.pt/~jmf/ PUBLI/papers/2004-TR-02.pdf>. Acesso em: 29 jan. 2006.

Monteiro, M. P.; Fernandes, J. M. "Towards a catalog of aspect-oriented refactorings". In: Proc. of the 4th International Conference on Aspect-Oriented Software Development (AOSD), pages 111.122. ACM Press, PARC Palo Alto Research Center, March 2005. Disponível em: < http://www.parc.xerox.com/>. Acesso em: 29 jan. 2006.

Ossher, H.; Tarr, P. "Multi-Dimensional Separation of Concerns and The Hyperspace Approach." In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer, 2000.

Parc. Disponível em: <http://www.parc.com/>. Acesso em: 28 jan. 2006.

Parnas, D. L. "On the criteria to be used in decomposing systems into modules". Communications of the ACM, 15(12):1053–1058, December 1972.

Penteado, R. A. D. “Um Método para Engenharia Reversa Orientada a Objetos”. Tese de Doutorado, IFSC - USP, 1996.

Pressman, R. S. "Engenharia de Software" 6. ed. Brasil, McGraw Hill, 2006.

Ramos, R. A. “Abordagem Aspecting: Migração de Sistemas OO para Sistemas OA”. Dissertação de Mestrado. Departamento de Computação, Universidade Federal de São Carlos, 2004.

Ramos, R. A., Penteado, R. A. D., Masiero, P. C. "Migração de Sistemas OO para Sistemas OA com a abordagem Aspecting". In: Simpósio Brasileiro de Engenharia de Software (SBES), Brasília – DF, 2004.

Rashid, A.; Sawyer, P.; Moreira, A.; Araújo, J. "Early Aspects. A Model for Aspect-Oriented Requirements Engineering". In: Conference on Requirements Engineering, Essen, Germany, 2002.

Robillard, M. P.; Murphy, G. C. “Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies”. In: IEEE 24th International Conference on Software Engineering, Orlando, Florida, USA, May 2002.

Satiroglu, Y. “Aspect-Oriented Evolution of Legacy Information Systems”. Dissertação de Mestrado. Departamento de Engenharia de Computação e Instituto de Engenharia e Ciências da Universidade de Bilkent, 2004. p. 14 - 98.

Shepherd, D.; Gibson, E.; Pollock, L. “Design and Evaluation of an Automated Aspect Mining Tool”. In: Proceedings of the 2004 Mid-Atlantic Student Workshop on Programming Languages and Systems, 2004.

Sommerville, I. “Engenharia de Software”, 6a ed, Addison-Wesley Pub. Co., 2003.
Thomas, D. “Refactoring as Meta Programming?” In: Journal of Object Tecnology, Vol. 4, No 1, January – February 2005. Disponível em: <www.jot.fm/issues/issue_2005_01/column1.pdf> Acesso em: 12 fev. 2006.

Voelter, M. “Aspect-Oriented Programming in Java”, 2000. Disponível em <http://www.voelter.de/data/articles/aop/aop.html>. Acesso em: 12 dez. 2005.

WASP, 1, 2004, Brasília. Relatório Técnico, 2004. Disponível em <http://twiki.im.ufba.br/pub/WAsp/WebHome/WASP04-RelatorioFinal.pdf> Acesso em 4 ago. 2005.

Welie, M. V. "Pattern in Interaction Design". Disponível em <http://www.welie.com>. Acesso em: 19 fev. 2006.

World Wide Web Consortium. Disponível em <http://www.w3.org/>. Acesso em: 25 jun. 2007.

Apêndice 1

Armazenamento de Reestruturações Aplicadas a Elementos de Código Java

As reestruturações usadas pelo ReJAsp podem ser aplicadas aos métodos ou aos atributos dependendo da disposição dos indícios no código-fonte. De responsabilidade do atributo `reorganization` pertencente à classe `MethodInfo`, Figura 4.2 do Capítulo 4, essa informação é armazenada em um inteiro cuja manipulação é feita por operações binárias, possibilitando registrar todas as combinações de reestruturação do código-fonte disponíveis.

Para que isso seja possível, foi realizado o armazenamento de variáveis constantes e estáticas com os valores em binário de 000001, 000010, 000100 e 001000, indicando respectivamente oportunidades de reestruturação de código por introdução de atributos, por introdução de métodos, por extração de começo de método e por extração de final de método.

Quando o processo de detecção de indícios, em um método, é iniciado o atributo `reorganization` tem o valor 000000, sinalizando a ausência de indícios. À medida que oportunidades de reestruturação de código-fonte são encontradas, durante a fase de identificação automática de indícios, o atributo `reorganization` recebe o resultado da operação de OR (OU lógico bit-a-bit) dele mesmo com o valor binário que representa a oportunidade de reestruturação. Por exemplo, no início da detecção de indícios no método, o valor de `reorganization` é 000000. Com a execução do processo, são verificadas oportunidades de reestruturação do código de introdução de método em aspecto (de valor 000010) e de extração de final de método (001000). Com isso, o valor final do atributo `reorganization` é 001010, resultante da operação 000000 | 000010 | 001000.

Com isso, `reorganization` pode ser usado para determinar as reestruturações indicadas para um método. Na prática, ao selecionar um elemento da árvore de indícios, um menu *popup* é aberto com as oportunidades de reestruturação de código relacionadas ao elemento selecionado. Isso é possível a partir operações de E lógico bit-a-bit do atributo `reorganization` com cada constante estática associada a uma reestruturação. Caso o resultado final seja o valor zero, a oportunidade de aplicar a reestruturação não foi verificada e, se o resultado for diferente de zero, a reestruturação é sugerida ao método.

No exemplo anterior, o atributo `reorganization` com o valor 001010 sofre operações com AND (E lógico) com 000001, 000010, 000100 e 001000, gerando respectivamente os resultados: 000000, 000010, 000000 e 001000. Como 000010 e 001000 são os valores estáticos referentes às reestruturações por introdução de método e por extração de final de método, tais reestruturações podem ser aplicadas.

Gramática livre de contexto para percurso de código em Java usando JDT

Na construção do ReJAsp, adotou-se a estratégia de percurso pela AST e análise do código por meio de funções recursivas ou que possam levar à recursividade indiretamente. Para isso, o apoio computacional utiliza classes do JDT específicas para esse fim, cujo nome é idêntico aos não-terminais das gramáticas livres de contexto apresentados a seguir. Do mesmo modo que esses não-terminais derivam em um conjunto indefinido de não-terminais e terminais, as classes de mesmo nome podem conter um conjunto de objetos, cujas classes também equivalem a não-terminais dessas gramáticas, e de palavras-chave que correspondem aos terminais.

- **Gramáticas livres de contexto usados pelo JDT para validação de código em Java**

CompilationUnit:
[PackageDeclaration]
{ ImportDeclaration }
{ TypeDeclaration | EnumDeclaration | AnnotationTypeDeclaration; }

```
PackageDeclaration:
   package Name;

ImportDeclaration:
   import [ static ] Name [ . * ] ;

Name:
    SimpleName
    QualifiedName

SimpleName:
    Identifier

QualifiedName:
   Name . SimpleName
```

- **Classes e interfaces são derivadas de TypeDeclaration**

```
TypeDeclaration:
                ClassDeclaration
                InterfaceDeclaration

ClassDeclaration:
    { ExtendedModifier } class Identifier
                        [ < TypeParameter { , TypeParameter } > ]
                        [ extends Type ]
                        [ implements Type { , Type } ]
                        { { ClassBodyDeclaration | ; } }

InterfaceDeclaration:
    { ExtendedModifier } interface Identifier
                        [ < TypeParameter { , TypeParameter } > ]
                        [ extends Type { , Type } ]
                        { { InterfaceBodyDeclaration | ; } }

TypeParameter:
   TypeVariable [ extends Type { & Type } ]

TypeVariable:
   Identifier

EnumDeclaration:
    { ExtendedModifier } enum Identifier
       [ implements Type { , Type } ]
       {
       [ EnumConstantDeclaration { , EnumConstantDeclaration } ] [ , ]
       [ ; { ClassBodyDeclaration | ; } ]
       }
```

```
FieldDeclaration:
   { ExtendedModifier } Type VariableDeclarationFragment
      { , VariableDeclarationFragment } ;

VariableDeclarationFragment:
   Identifier { [] } [ = Expression ]

MethodDeclaration:
   { ExtendedModifier }
                [ < TypeParameter { , TypeParameter } > ]
     ( Type | void ) Identifier (
     [ FormalParameter
                  { , FormalParameter } ] ) {[ ] }
     [ throws TypeName { , TypeName } ] ( Block | ; )

ConstructorDeclaration:
   { ExtendedModifier }
                [ < TypeParameter { , TypeParameter } > ]
     Identifier (
                [ FormalParameter
                        { , FormalParameter } ] )
     [throws TypeName { , TypeName } ] Block

FormalParameter:
   SingleVariableDeclaration

SingleVariableDeclaration:
   { ExtendedModifier } Type [ ... ] Identifier { [] } [ = Expression ]

TypeName:
   Name
```

- **Todas as instruções têm como base o não-terminal Statement**

```
Statement:
     AssertStatement | Block | BreakStatement | ConstructorInvocation |
     ContinueStatement | DoStatement | EmptyStatement |
     EnhancedForStatement | ExpressionStatement | ForStatement | IfStatement |
     LabeledStatement | ReturnStatement | SuperConstructorInvocation |
     SwitchCase | SwitchStatement | SynchronizedStatement | ThrowStatement |
     TryStatement | TypeDeclarationStatement | VariableDeclarationStatement |
     WhileStatement

AssertStatement:
   assert Expression [ : Expression ] ;
```

```
Block:
   { { Statement } }

BreakStatement:
   break [ Identifier ] ;


ConstructorInvocation:
    [ < Type { , Type } > ]
                    this ( [ Expression { , Expression } ] ) ;

ContinueStatement:
   continue [ Identifier ] ;

DoStatement:
   do Statement while ( Expression ) ;

EmptyStatement:
   ;

EnhancedForStatement:
   for ( FormalParameter : Expression )
                        Statement

ExpressionStatement:
   StatementExpression ;

ForStatement:
   for (
                        [ ForInit ];
                        [ Expression ] ;
                        [ ForUpdate ] )
                        Statement
ForInit:
                Expression { , Expression }
ForUpdate:
                Expression { , Expression }

IfStatement:
   if ( Expression ) Statement [ else Statement]

LabeledStatement:
   Identifier : Statement

ReturnStatement:
   return [ Expression ] ;
```

```
SuperConstructorInvocation:
    [ Expression . ]
        [ < Type { , Type } > ]
        super ( [ Expression { , Expression } ] ) ;

SwitchCase:
                case Expression  :
                default :

SwitchStatement:
                switch ( Expression )
                        { { SwitchCase | Statement } } }
SwitchCase:
                case Expression  :
                default :

SynchronizedStatement:
   synchronized ( Expression ) Block

ThrowStatement:
   throw Expression ;

TryStatement:
    try Block
        { CatchClause }
        [ finally Block ]

TypeDeclarationStatement:
   TypeDeclaration
   EnumDeclaration

VariableDeclarationStatement:
   { ExtendedModifier } Type VariableDeclarationFragment
      { , VariableDeclarationFragment } ;

WhileStatement:
   while ( Expression ) Statement
```

- **As expressões podem ser derivadas de diversas construções Statement, enumerações, declaração de um atributo ou de lista de atributos**

```
Expression:
  Annotation | ArrayAccess | ArrayCreation | ArrayInitializer | Assignment |
  BooleanLiteral | CastExpression | CharacterLiteral | ClassInstanceCreation |
  ConditionalExpression | FieldAccess | InfixExpression | InstanceofExpression |
  MethodInvocation | Name | NullLiteral | NumberLiteral |
  ParenthesizedExpression | PostfixExpression | PrefixExpression | StringLiteral
```

```
  | SuperFieldAccess | SuperMethodInvocation | ThisExpression | TypeLiteral |
  VariableDeclarationExpression

ArrayAccess:
   Expression [ Expression ]

ArrayCreation:
   new PrimitiveType [ Expression ] { [ Expression ] } { [ ] }
   new TypeName [ < Type { , Type } > ]
      [ Expression ] { [ Expression ] } { [ ] }
   new PrimitiveType [ ] { [ ] } ArrayInitializer
   new TypeName [ < Type { , Type } > ]
      [ ] { [ ] } ArrayInitializer

ArrayInitializer:
         { [ Expression { , Expression} [ , ]] }

Assignment:
   Expression AssignmentOperator Expression

CastExpression:
   ( Type ) Expression

ClassInstanceCreation:
      [ Expression . ]
         new [ < Type { , Type } > ]
         Type ( [ Expression { , Expression } ] )
         [ AnonymousClassDeclaration ]

ConditionalExpression:
   Expression ? Expression : Expression

FieldAccess:
         Expression . Identifier

InfixExpression:
   Expression InfixOperator Expression { InfixOperator Expression }

InstanceofExpression:
   Expression instanceof Type

MethodInvocation:
    [ Expression . ]
       [ < Type { , Type } > ]
       Identifier ( [ Expression { , Expression } ] )

ParenthesizedExpression:
    ( Expression )
```

```
PostfixExpression:
   Expression PostfixOperator

PrefixExpression:
   PrefixOperator Expression

SuperFieldAccess:
    [ ClassName . ] super . Identifier

SuperMethodInvocation:
    [ ClassName . ] super .
       [ < Type { , Type } > ]
       Identifier ( [ Expression { , Expression } ] )

ThisExpression:
    [ ClassName . ] this

TypeLiteral:
    ( Type | void ) . class

VariableDeclarationExpression:
   { ExtendedModifier } Type VariableDeclarationFragment
       { , VariableDeclarationFragment }
```

Mecanismo de Percurso em Árvore no JDT e Identificação de Indícios

Na Figura A.1 foi ilustrado o não-terminal `IfStatement` que contém a palavra reservada `if` e os parênteses como terminais, a expressão condicional e a instrução, caso a condição seja satisfeita, são representados pelos não-terminais `Expression` e `Statement`, respectivamente. Uma parte opcional dessa produção é a parte `else` do comando `if`, contendo o terminal (palavra-chave) `else` e também o não-terminal `Statement`, referente à instrução que deve ser executada caso a condição não seja satisfeita.

```
IfStatement:
    if ( Expression ) Statement [ else Statement]
```

Figura A.1: Produção para a instrução do comando `if`

A classe que trata essa produção no JDT é a `IfStatement`, pertencente ao pacote `org.eclipse.jdt.core.dom`, permitindo o acesso ao condicional por meio de uma instância da classe `Expression` e a instrução da parte `if` do comando por meio de uma instância da classe `Statement`.

O não-terminal `Statement` pode derivar qualquer instrução em Java como apresentado na Figura A.2. Portanto, o comando `if` pode conter qualquer instrução, incluindo um bloco de instruções. Ao realizar derivações, um não-terminal `IfStatement` pode derivar uma quantidade indeterminada de não-terminais `IfStatement`, por exemplo. Por esse motivo na construção do apoio computacional, adotou-se a estratégia de percurso pela AST e a análise do código por meio de funções recursivas ou que possam levar à recursividade indiretamente.

```
Statement:
    AssertStatement | Block | BreakStatement |
    ConstructorInvocation | ContinueStatement | DoStatement |
    EmptyStatement | EnhancedForStatement | ExpressionStatement |
    ForStatement | IfStatement | LabeledStatement |
    ReturnStatement | SuperConstructorInvocation | SwitchCase |
    SwitchStatement | SynchronizedStatement | ThrowStatement |
    TryStatement | TypeDeclarationStatement |
    VariableDeclarationStatement | WhileStatement
```

Figura A.2: Produção para o não-terminal `Statement`

A utilização de recursividade possui a desvantagem de consumo de tempo e memória, pois a cada chamada recursiva, o estado atual de processamento é registrado. Entretanto, o problema de percurso em árvore possui uma solução inerentemente recursiva, visto que necessita registrar o estado anterior e a implementação do percurso apresenta maior clareza e facilidade em sua manutenção.

Quando se implementa uma solução recursiva é necessário especificar uma condição de parada. No caso do percurso em árvore, a condição de parada é quando as derivações resultam somente terminais da linguagem (palavras-chaves, operadores, identificadores de variáveis, etc). O mecanismo de identificação de indícios do ReJAsp, ao encontrar alguns dos terminais da linguagem, verifica a existência de indícios.

O processo de identificação de indícios é iniciado a partir de uma instância de `WorkspaceRoot` que contém informações da pasta de *workspace* do Eclipse. Essa pasta é escolhida pelo desenvolvedor e pode conter projetos em desenvolvimento ou em

manutenção. A partir da instância de `WorskpaceRoot`, é possível ter acesso a qualquer instância de projetos (tipo `Project`). Contudo, o ReJAsp apenas escolhe o projeto previamente selecionado pelo desenvolvedor. Conforme é apresentado na Figura A.3, a instância de projetos contém uma coleção de recursos (`Resource`), isto é, arquivos ou pastas representados por instâncias de `File` e de `Folder` respectivamente.

Durante essa varredura por instâncias de arquivos, a extensão de arquivos é verificada. Se arquivos de extensão java forem encontrados, uma instância de `FileInfo` (Figura 4.2 do Capítulo 4) é criada e o conteúdo do arquivo é usado pelo compilador do JDT para a criação da AST. O nó raiz da AST corresponde ao não-terminal `CompilationUnit`, como ilustrado na Figura A.4.

```
WorkspaceRoot:
    { Project }

Project:
    { Resource }

Resource:
    ( Folder | File )

Folder:
    { Resource }
```

Figura A.3: Produção para não-terminais de recursos do Eclipse

Ao percorrer a árvore partindo de `CompilationUnit`, é possível ter acesso a e procurar por indícios com declarações de importação (`ImportDeclaration`). Um algoritmo semelhante ao exibido na Figura 3.8 é usado para processar cada instância `ImportDeclaration`. No processo manual o tipo importado associado a um indício era anotado, já com o uso do ReJAsp armazena-se o tipo em uma lista de tipos com indícios.

Caso nenhum tipo importado contenha indícios, o arquivo é considerado desprovido de indícios e um novo arquivo de extensão java é avaliado. Contudo, se indícios em importações de tipos for verificada, instâncias `TypeDeclaration` são analisadas no arquivo. O não-terminal `TypeDeclaration` pode derivar uma classe ou interface, que por sua vez deriva declarações de atributos e métodos.

Por definição, as interfaces não possuem atributos nem instruções. A possibilidade em obter derivações de declarações de atributos e de métodos a partir de

uma interface acontece porque, ao invés de atributos, as interfaces possuem a declaração de constantes e de métodos abstratos. A constante é um caso particular de atributo, isto é, a utilização de modificadores: público (`public`), estático (`static`) e final (`final`) na declaração do atributo, bem como a inicialização de um valor. Em constantes é possível encontrar indícios, verificando o tipo associado à constante ou a inicialização caso o tipo seja uma cadeia de caracteres.

O método abstrato é um método sem corpo, isto é, sem instruções. Por meio de derivações de `MethodDeclaration`, é possível obter um método com ou sem corpo, devido ao trecho final da produção de `MethodDeclaration`: (`Block` | ;), exibido na Figura 4.3. Assim, indícios podem ser encontrados em métodos abstratos analisando os tipos usados em parâmetros e no tipo de retorno do método.

```
CompilationUnit:
    [ PackageDeclaration ]
        { ImportDeclaration }
        { TypeDeclaration | EnumDeclaration |
          AnnotationTypeDeclaration; }

FieldDeclaration:
    { ExtendedModifier } Type VariableDeclarationFragment
         { , VariableDeclarationFragment } ;

MethodDeclaration:
    { ExtendedModifier }
              [ < TypeParameter { , TypeParameter } > ]
        ( Type | void ) Identifier (
        [ FormalParameter
                 { , FormalParameter } ] ) {[ ] }
        [ throws TypeName { , TypeName } ] ( Block | ; )
```

Figura A.4: Produção para não-terminais de `CompilationUnit`, de declaração de atributos e de métodos

Considerando que as interfaces não contém instruções, apenas reestruturações por introdução de atributos ou por introdução de declarações de métodos podem ser aplicadas às mesmas. Essas reestruturações estão relacionadas com a utilização de interesses transversais estáticos que, por definição, afetam a estrutura estática de classes e de interfaces, jamais interferindo no comportamento dinâmico do sistema. Essa característica está condizente com o próprio uso da interface, cuja responsabilidade é servir de modelo para que classes a implementem, observando também sua impossibilidade de instanciação.

Ao analisar classes e interfaces, o ReJAsp cria instâncias de `TypeInfo` que são associadas às instancias de `FileInfo`. Em seguida, os atributos (constantes no caso de interfaces) são avaliados, utilizando seu tipo e inicialização na avaliação. Para isso, é verificado se o tipo pertence à lista de tipos com indícios. Caso o tipo não esteja presente na lista, é verificada sua declaração como cadeia de caracteres, isto é, de tipo `String`. Nesse último caso, o conteúdo da cadeia de caracteres usado na inicialização é avaliado.

Como pode ser observado na Figura A.4, vários atributos podem estar em uma mesma declaração, pois o não-terminal `VariableDeclarationFragment` pode ser repetido diversas vezes. Esse não-terminal deriva o identificador do atributo e uma inicialização optativa. Entretanto, independente do atributo estar disposto em uma declaração isolada ou em declaração de múltiplos atributos, uma estrutura `FieldInfo` é gerada para cada atributo encontrado e associado à instância `FieldInfo`.

Em seguida, os métodos são analisados, observe a produção `MethodDeclaration`. Para cada método encontrado, o ReJAsp armazena parte de suas informações em `MethodInfo` e associa à instância de `TypeInfo` que corresponde o tipo (classe ou interface) que contém o método. Indícios são buscados em tipos de retorno e de parâmetros de métodos; caso a presença de indícios seja constatada, `reorganization` é usada em operação de OU lógico com a constante referente à introdução de método, conforme discussões do início deste apêndice.

Os métodos de classes podem conter o corpo de método e, conseqüentemente, instruções de acordo com a produção de `MethodDeclaration`, que deriva o não-terminal `Block` contendo um conjunto de qualquer número de instruções. Para cada instrução, representada pelo não-terminal `Statement`, derivações de outros não terminais ocorrem e à medida que esses não-terminais vão sendo resolvidos, são geradas palavras chaves da linguagem, como por exemplo: `try`, `for`, `if`, `break`, etc; identificadores (nomes de atributos, variáveis locais, métodos, classes) ; não-terminais `Expression`, cuja produção é apresentada na Figura A.5; e tipos.

Com derivações de `Expression`, são obtidas palavras chaves (`instanceof`, `super`, `this`, `class`), operadores (+, -, *, /, ++, --), caracteres reservados (‘(’, ‘)’, ‘[’, ‘]’, ‘.’), identificadores, literais e tipos. A verificação de indícios é feita nos terminais de tipo, de literais e de identificadores.

```
Expression:
   Annotation | ArrayAccess | ArrayCreation | ArrayInitializer |
   Assignment | BooleanLiteral | CastExpression |
   CharacterLiteral | ClassInstanceCreation |
   ConditionalExpression | FieldAccess | InfixExpression |
   InstanceofExpression | MethodInvocation | Name | NullLiteral |
   NumberLiteral | ParenthesizedExpression | PostfixExpression |
   PrefixExpression | StringLiteral | SuperFieldAccess |
   SuperMethodInvocation | ThisExpression | TypeLiteral |
   VariableDeclarationExpression
```

Figura A.5: Produção para não-terminais com expressões (`Expression`)

A existência de cada tipo encontrado na lista de tipos com indícios é verificada. Os tipos presentes nessa lista têm a instrução que a contém considerada como um indício. Em especial, para instruções de declarações de variáveis locais, os nomes e tipos das variáveis são armazenados em uma estrutura de dados, denominada de tabela de variáveis, apresentadas no Capítulo 4 e detalhadas a seguir. Essa regra também é aplicada para atributos, constantes e parâmetros.

Os identificadores encontrados mediante as derivações de instruções e expressões são buscados na tabela de variáveis. Caso o identificador esteja presente na tabela de variáveis é o sinalizador que a instrução contendo o identificador contém indícios.

Outra forma de encontrar indícios é a partir de terminais de literais, mais especificamente os literais de cadeias de caracteres. Para isso, instâncias de `MatchText` do Modelo de Indícios, Figura 3.5, são utilizadas. Essa classe contém uma lista de palavras que são comparadas com o conteúdo do literal encontrado, considerando as regras: (1) de início de cadeia, (2) de meio de cadeia e (3) de final de cadeia. A diferenciação entre letras maiúsculas e minúsculas pode ser habilitada. Caso uma ou mais palavras da instância `MatchText` sejam encontradas, no conteúdo do literal, e as regras sejam satisfeitas, o literal é considerado um indício.

Tabela de Variáveis

A tabela de variáveis é uma estrutura de dados construída e usada no ReJAsp com o objetivo de identificar variáveis com indícios em código Java, considerando seu

escopo no programa. A execução do mecanismo de identificação de indícios pelo apoio computacional no exemplo de código da Figura A.7, é reproduzida exatamente a tabela de variáveis mostrada na Figura A.6.

a) Criação de níveis

Inicialmente, a tabela de variáveis não apresenta nenhum nível armazenado. Ao iniciar a análise da classe A, Figura A.7, o primeiro nível é criado, isto é, o escopo de atributos e constantes dessa classe é estabelecido. Ao verificar a existência de três atributos dessa classe: `At1, At2, At3`; os mesmos são registrados na lista de nível 1.

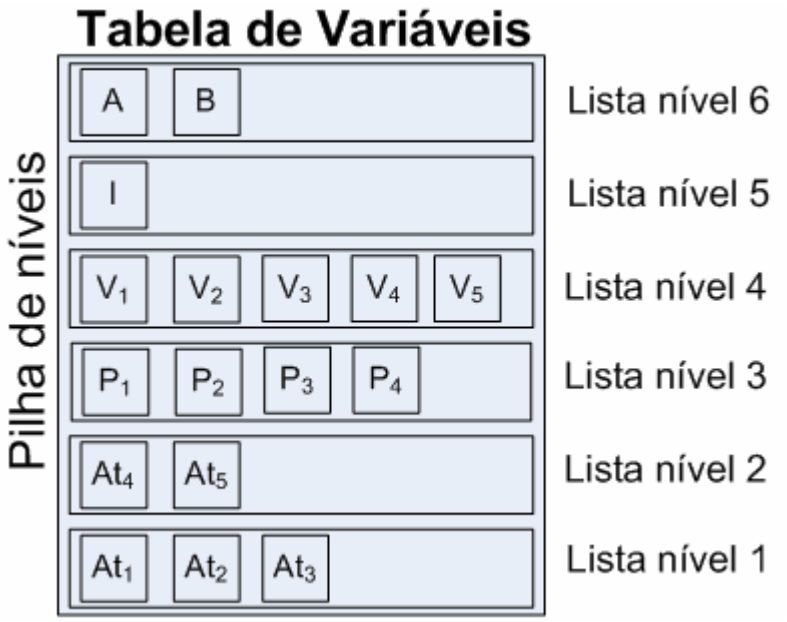

Figura A.6: Tabela de variáveis

Depois disso, o ReJAsp prossegue a identificação e encontra a ocorrência de uma classe interna B. Assim, um novo escopo de variáveis é estabelecido, ou seja, o escopo de atributos e constantes da classe B se torna conhecida e a lista de nível 2 é criada. Nessa lista adicionam-se os atributos da classe B: `At4` e `At5`, linha 5 da Figura A.6.

Na seqüência, o ReJAsp verifica a declaração de `metodo1` com os parâmetros: `P1, P2, P3` e `P4`. Desse modo, é criado o terceiro nível, correspondendo ao escopo de parâmetros do método, com a criação da lista de nível 3, adicionando à essa lista as informações dos quatro parâmetros.

Antes de iniciar a análise de cada instrução existente no método um novo nível é especificado, determinando o escopo de variáveis locais existentes no corpo do método. Com isso, a lista de nível 4 é criada e associada à tabela de variáveis. A primeira instrução do método, linha 8 da Figura A.6, trata-se de uma declaração de cinco variáveis: `V1, V2, V3, V4` e `V5`, que são acrescentadas à lista de nível 4.

```
1  public class A {
2    private int At1, At2, At3;
3
4    public class B {
5        private String At4, At5;
6
7        public void metodo1(int P1, int P2, int P3, int P4) {
8          double V1, V2, V3, V4, V5;
9          for ( int I = 0; I < V1; I++ ) {
10           if ( I > V2 ) {
11             char A, B;
12           }
13         }
14       }
15   }
16 }
```

Figura A.7: Exemplo de código que gera seis níveis na tabela de variáveis

Escopos podem ser definidos por instruções desde que as mesmas contenham blocos de instrução. Assim, cada bloco de instrução iniciado por '{' e finalizado por '}', define um escopo de variável próprio. A instrução de laço `for`, linhas 9 a 12 da Figura A.7, contém um bloco e, assim, define um novo escopo. A variável I pertencente ao controle do `for` apenas é acessível dentro desse bloco. Desse modo, quando ReJAsp processar o comando `for`, a lista de nível 5 é alocada e a variável I é associada.

Posteriormente, a instrução `if` é analisada e verifica-se a necessidade de criação de outro nível, para tratar o bloco referente a essa instrução. Dessa forma, a lista de nível 6 é criada adicionando-se as variáveis declaradas nesse bloco.

b) Remoção de níveis da tabela de variáveis

Existem determinados pontos no código-fonte em que a visibilidade de variáveis deixa de ser aplicável, isto é, o escopo / contexto em que a variável atua se encerra. Esses pontos podem ser caracterizados por final de bloco de instrução, de método e de classe.

O mecanismo de identificação de indícios, ao analisar o código-fonte, detecta o término desses escopos e atualiza a tabela de variáveis para que a busca de indícios seja feita adequadamente. No exemplo da Figura A.7, até a linha 11, a tabela de variáveis possui o seu estado idêntico àquele ilustrado na Figura A.6. Entretanto, a partir da linha 12 é verificado o fim do bloco `if` e com isso, as variáveis locais declaradas, nesse bloco, deixam de ser visíveis. Nesse caso, a lista de nível 6 e seus elementos se tornam irrelevantes e não aplicável ao restante do código que está sendo analisado. Por isso, essa lista é removida da tabela de variáveis juntamente com seus elementos, nesse caso, informações das variáveis A e B. De modo análogo, a partir da linha 13 onde ocorre o encerramento do bloco da instrução `for`, a lista de nível 5 é removida e a variável I deixa ser aplicada no restante do código. O encerramento das listas de níveis 4 e 3 ocorre ao final do corpo do método, isto é, na linha 14. As linhas 15 e 16 determinam a remoção das listas de nível 2 e 1 da tabela de variáveis, respectivamente. Esse comportamento descrito é característico de uma pilha, também conhecida como LIFO, acrônimo para *Last In, First Out*, ou seja, último a entrar, primeiro a sair.

c) Pesquisa e Utilização de Elementos na Tabela de Variáveis

A pesquisa de elementos na tabela de variáveis é feita sempre priorizando as variáveis do último nível para o primeiro. Devido à possibilidade de variáveis declaradas em níveis distintos do programa conterem o mesmo nome, é necessário priorizar aquela de escopo mais local. Neste projeto, variáveis de escopos mais globais são declaradas em níveis menores, enquanto que variáveis de escopo mais local são especificadas em níveis maiores na tabela de variáveis.

O exemplo apresentado na Figura A.7 é meramente ilustrativo para expressar o funcionamento da tabela de variáveis utilizando variáveis simples (inteiras, reais, cadeia de caracteres). Na prática, as variáveis armazenadas nessa estrutura de dados são apenas àquelas contendo indícios.

Apêndice 2

Gráfico de Conhecimentos e Opinião do ReJAsp

Os conhecimentos e opiniões de 14 desenvolvedores, integrantes dos quatro grupos, envolvidos nos estudos de caso foram coletados por meio de questionários, como mencionado no Capítulo 5. As respostas às questões desses questionários são do tipo de múltipla escolha, com 4 ou 5 alternativas. O desenvolvedor assinala a alternativa que melhor se aplica à sua resposta seguida de uma justificativa para essa escolha em espaço previamente reservado. Os desenvolvedores foram orientados a não assinalar nenhuma alternativa, desde que a mesma não se aplicasse, porém a justificativa é sempre obrigatória. Os resultados coletados nesses questionários são apresentados a seguir, por item, sendo que as alternativas não assinaladas foram agrupadas como Não Aplicável, NA (vide questões 8, 9 e 10). Além das informações numéricas é apresentado, também, um gráfico tipo "pizza" para ilustrar os resultados obtidos.

(1) Conhecimento em Java

Opção	Descrição	Freqüência
1	Nenhum	0
2	Alguma base teórica	2
3	Pouca prática	7
4	Bom conhecimento	4
5	Domínio avançado	1

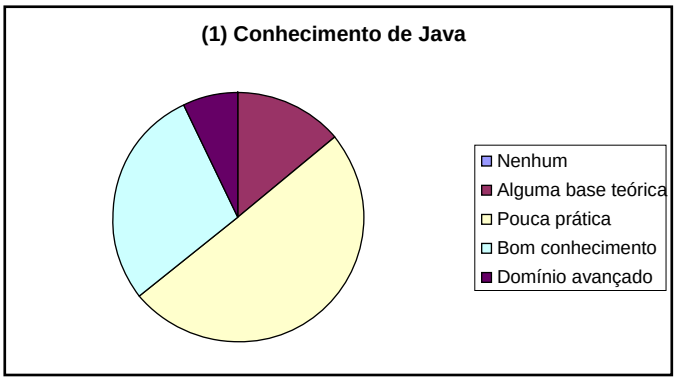

Figura A.8: Conhecimento em Java

(2) Conhecimento de POA

Opção	Descrição	Freqüência
1	Nenhuma	2
2	Pequena noção	7
3	Base teórica razoável	2
4	Familiaridade com POA	3
5	Conhecimento avançado	0

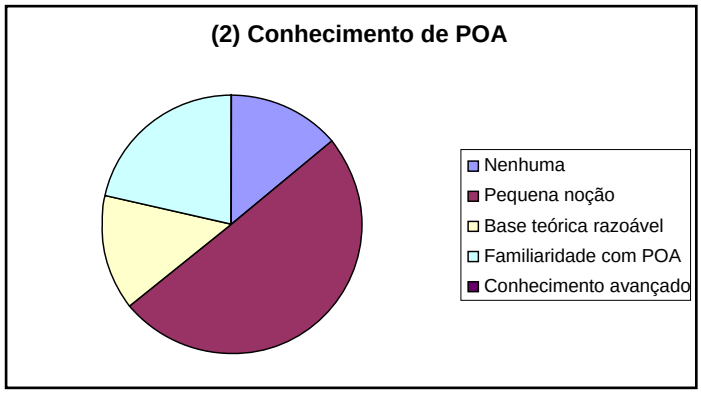

Figura A.9: Conhecimento de POA

(3) Conhecimento de AspectJ

Opção	Descrição	Freqüência
1	Nenhuma	8
2	Pequena noção	3
3	Conhecimento básico	2
4	Conhecimento intermediário	1
5	Conhecimento avançado	0

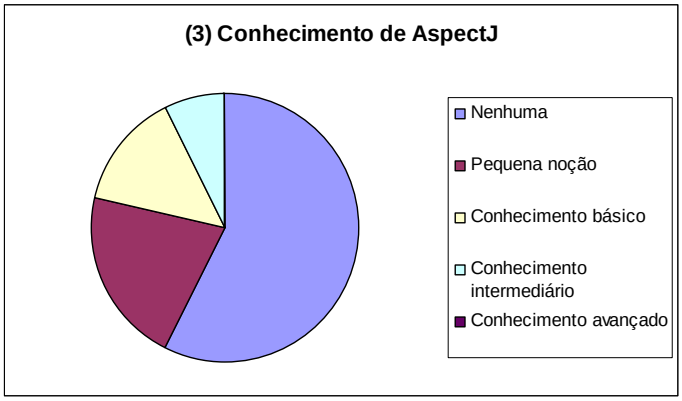

Figura A.10: Conhecimento de AspectJ

(4) Facilidade de uso do ReJAsp

Opção	Descrição	Freqüência
1	Muito difícil	0
2	Difícil	1
3	Fácil	13
4	Muito fácil	0

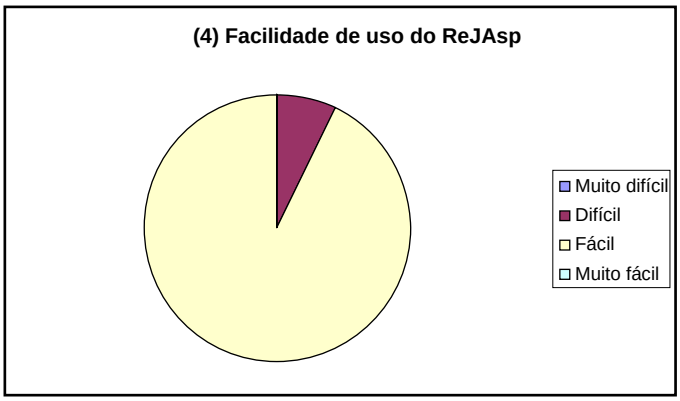

Figura A.11: Facilidade de uso do ReJAsp

(5) Eficiência do guia do usuário em atender dúvidas

Opção	Descrição	Freqüência
1	Não ajudou	0
2	Ajudou pouco	0
3	Ajudou bastante	14
4	Respondeu a todas as dúvidas	0

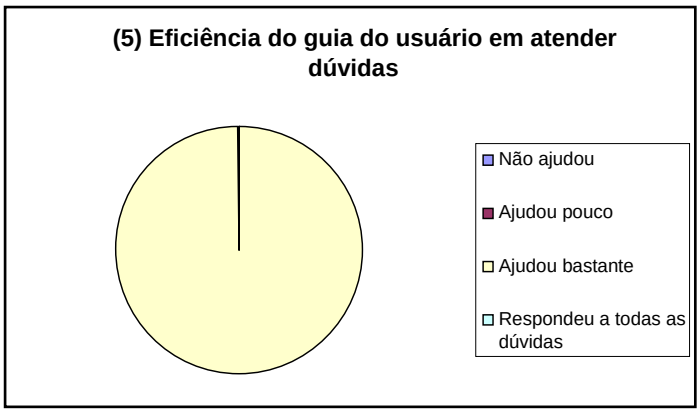

Figura A.12: Eficiência do guia do usuário em atender dúvidas

(6) Precisão da Identificação de Indícios

Opção	Descrição	Freqüência
1	Totalmente impreciso	0
2	Incorreto na maior parte dos casos	0
3	Correto na maior parte dos casos	14
4	Totalmente preciso	0

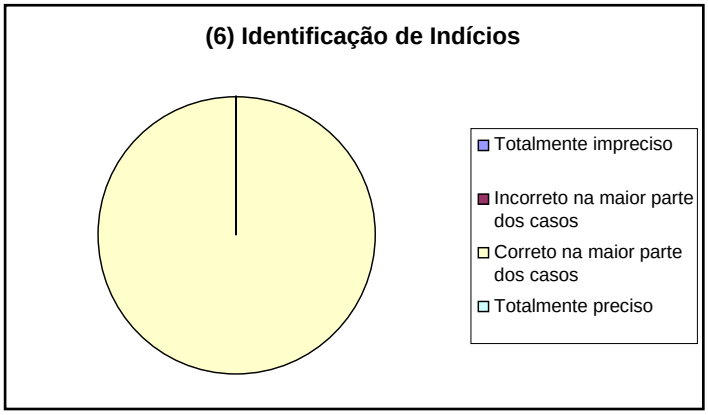

Figura A.13: Precisão da identificação de indícios

(7) Visualização de Aspectos

Opção	Descrição	Freqüência
1	Péssima	0
2	Insatisfatória	0
3	Útil	14
4	Excelente	0

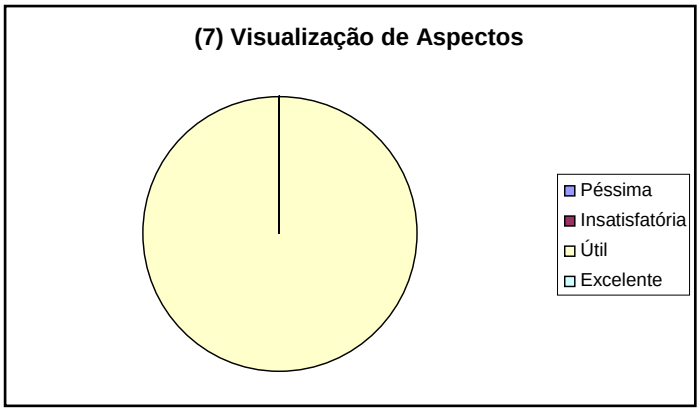

Figura A.14: Visualização de Aspectos

(8) Facilidade de uso de assistentes de reestruturação do código

Opção	Descrição	Freqüência
1	Muito difícil	0
2	Difícil	2
3	Fácil	10
4	Muito fácil	0
NA	Não Aplicável	2

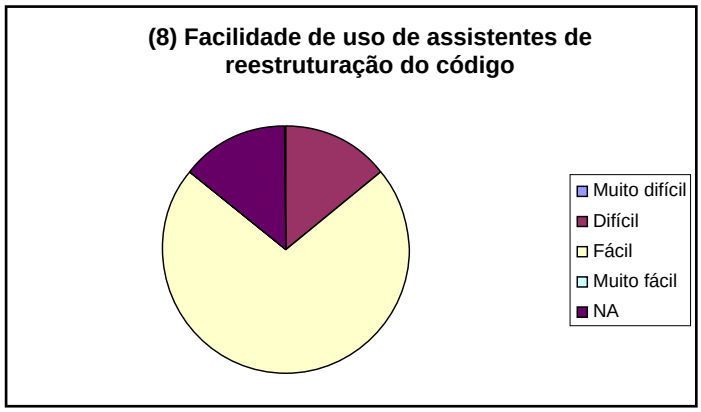

Figura A.15: Facilidade de uso de assistentes de reestruturação do código

(9) Eficácia dos assistentes de reestruturação do código

Opção	Descrição	Freqüência
1	Assistente não usado	0
2	Assistente usado raramente	1
3	Usado em metade dos casos	1
4	Usado na grande maioria dos casos	10
NA	Não Aplicável	2

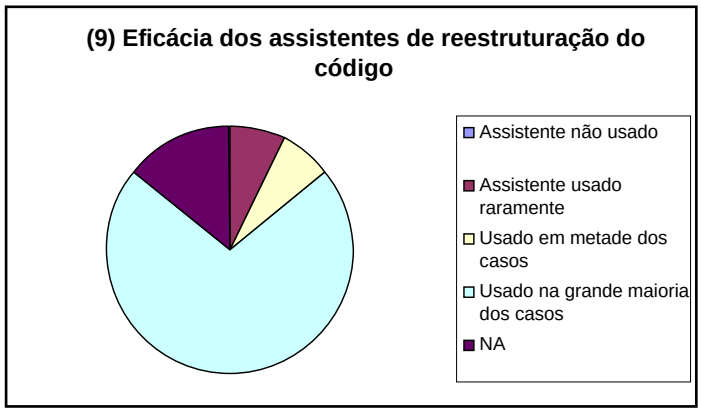

Figura A.16: Eficácia dos assistentes de reestruturação de código

(10) Facilidade de uso de assistentes de gerenciamento de indícios

Opção	Descrição	Freqüência
1	Muito difícil	0
2	Difícil	1
3	Fácil	5
4	Muito fácil	2
NA	Não Aplicável	6

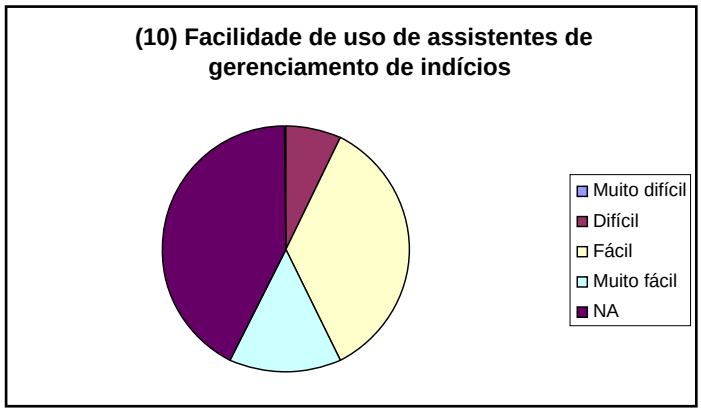

Figura A.17: Facilidade de uso do gerenciamento de indícios

(11) Economia de tempo utilizando o ReJAsp

Opção	Descrição	Freqüência
1	Gasta mais tempo	0
2	Mesmo tempo gasto	1
3	Pouca economia	1
4	Muita economia	12

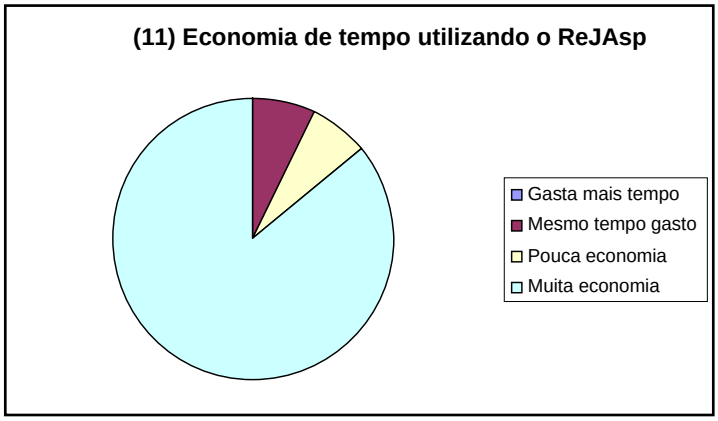

Figura A.18: Economia de tempo utilizando o ReJAsp

(12) Dificuldade entre estudos de caso, da primeira e segunda etapa, realizados pelo Grupo

Opção	Descrição	Freqüência
1	1º muito mais difícil	2
2	1º um pouco mais difícil	8
3	1º um pouco mais fácil	1
4	1º muito mais fácil	3

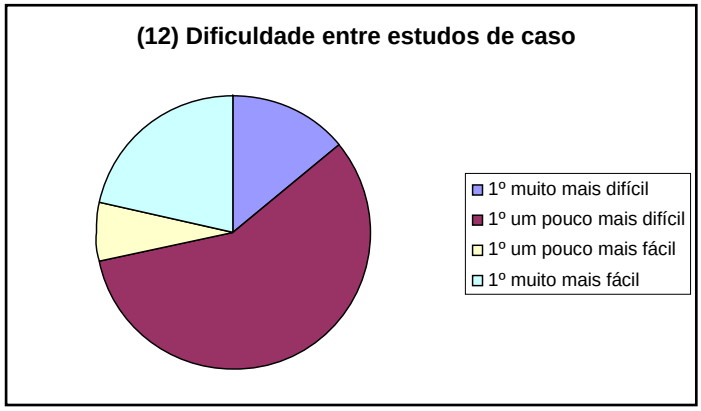

Figura A.19: Dificuldade entre estudos de caso, da primeira e segunda etapa, realizados pelo Grupo