Marcelo Nassau Malta

Transformações de Código para Extração de Aspectos

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Informática.

Belo Horizonte

Maio de 2008

FICHA CATALOGRÁFICA

Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

Malta, Marcelo Nassau
M261t Transformações de código para extração de aspectos / Marcelo Nassau Malta. – Belo Horizonte, 2008.
76 f. : il.

Orientador: Prof. Dr. Marco Túlio de Oliveira Valente.
Dissertação (mestrado) – Pontifícia Universidade Católica de Minas Gerais, Programa de Pós-graduação em Informática.
Bibliografia.

1. Programação (Computadores). 2. Engenharia de software – Teses I. Valente, Marco Túlio de Oliveira. II. Pontifícia Universidade Católica de Minas Gerais. III. Título

CDU: 681.3.03

Bibliotecário: Fernando A. Dias – CRB6/1084

PUC Minas
Programa de Pós-graduação em Informática

FOLHA DE APROVAÇÃO

"Transformações de Código para Extração de Aspectos"

Marcelo Nassau Malta

Dissertação defendida e aprovada pela seguinte banca examinadora:

Prof. Marco Túlio de Oliveira Valente - Orientador (PUC Minas)
Prof. Paulo Cesar Masiero (ICMC/USP)
Prof. Humberto Torres Marques Neto (PUC Minas)

Prof. Marco Túlio de Oliveira Valente - Orientador (PUC Minas)
Doutor em Ciência da Computação - UFMG

Prof. Paulo Cesar Masiero (ICMC/USP)
Doutor em Administração - USP

Prof. Humberto Torres Marques Neto (PUC Minas)
Doutor em Ciência da Computação - UFMG

Belo Horizonte, 09 de maio de 2008.

Anel Rodoviário – KM 23,5, Rua Walter Ianni, 255 – São Gabriel / Belo Horizonte – MG / CEP: 31980-110
Fone: (31) 3439-5204 – Fax: (31) 3439-5204 – e-mail: mestradoinfsec@pucminas.br

Resumo

Em processos de extração de aspectos em sistemas legados, após a identificação de trechos de código correspondentes a interesses transversais e antes da aplicação de uma refatoração visando a extração destes para aspectos, podem ser necessárias transformações no código base do sistema. Essas transformações são necessárias porque nem sempre tais trechos de código estão localizados em pontos do programa passíveis de captura usando o modelo para descrição de conjuntos de junção de uma linguagem orientada por aspectos. Assim, tais transformações têm como objetivo viabilizar a associação desses trechos de código a pontos de junção.

Trabalhos na área de refatoração de interesses transversais sempre mencionam a necessidade de preparação do código de uma aplicação para viabilizar a extração de aspectos, mas normalmente essa necessidade é mencionada de forma abreviada. Nesta dissertação, apresenta-se um catálogo com as principais transformações que podem ser aplicadas em sistemas orientados por objetos, visando a preparação destes para extração de aspectos. As transformações desse catálogo foram divididas em dois grupos: Movimentações de Código e Extrações de Métodos. As transformações catalogadas são descritas e algumas delas são exemplificadas em sistemas reais. Apresentam-se ainda descrições formais para cada transformação catalogada.

Adicionalmente, descreve-se o projeto e a implementação da ferramenta *TransformationMapper* que identifica necessidades de transformações com base em um mapeamento de elementos que compõem interesses transversais. São apresentados também resultados de avaliações envolvendo o uso dessa ferramenta para demonstrar a aplicação das transformações catalogadas em sistemas reais.

O desenvolvimento desta dissertação permite concluir que transformações constituem uma etapa fundamental no processo de extração de aspectos. Acredita-se que o catálogo proposto e a ferramenta *TransformationMapper* possam auxiliar desenvolvedores de sistemas em processos de refatoração de interesses transversais em sistemas legados.

Abstract

In the extraction of aspects from legacy systems, after locating fragments of code corresponding to crosscutting concerns and before extracting such code to aspects, transformations may be needed in the object-oriented code. These transformations are required because such fragments of code sometimes are situated at static locations of the base program that cannot be captured using the pointcut descriptor model of aspect-oriented languages. Therefore, such transformations aim to associate crosscutting code to join points.

Most of the works in the area of aspect-oriented refactoring frequently mention the need of preparing the base code for enabling the extraction of aspects, but usually this need is mentioned in an abbreviated way. In this master thesis, we present a catalog of the main object-oriented transformations that can be applied in object-oriented systems, in order to prepare them to aspect extraction. The catalogued transformations are divided in two groups: Statement Reordering and Method Extraction. In this dissertation, the catalogued transformations are described and some of them are exemplified in real-world systems.

Additionally, we describe the design and implementation of the *TransformationMapper* tool. This tool aims to identify the need of transformations based on mapping crosscutting concerns elements. We also present case studies in which this tool has been used to show the application of the catalogued transformations in real-world systems.

The experience gained during this master thesis has allowed us to conclude that transformations constitute an essential step in aspect extraction processes. We believe that the proposed catalog and the *TransformationMapper* tool can help system developers in refactoring crosscutting concerns from legacy systems.

Aos meus pais,
minha esposa Ivana
e meu filho Rafael.

Agradecimentos

Agradeço aos meus pais por terem me ensinado valores e princípios fundamentais na minha vida, para condução dos meus projetos pessoais, profissionais e acadêmicos. À minha esposa Ivana pelo amor, incentivo e compreensão, principalmente nos momentos em que estive ausente me dedicando ao mestrado. Ao meu filho Rafael, que é a maior inspiração para que eu alcance os meus objetivos.

Aos professores Ana Maria, Lucila Ishitani, Raquel Mini, Mark Alan e Sílvio Jamil por compartilharem seus conhecimentos nas disciplinas que tive oportunidade de cursar. À Giovanna, secretária acadêmica, pela presteza e atenção aos alunos. Aos colegas do mestrado, em especial aos do Datapuc, pois o grupo de estudos formado no início do curso foi fundamental para que eu pudesse seguir em frente nessa caminhada.

Aos professores Fábio Horácio e Humberto Torres, que na diretoria do Datapuc sempre incentivaram e permitiram que seus funcionários conciliassem atividades profissionais e acadêmicas. Gostaria de agradecer especialmente ao Professor Humberto Torres por ter sido o maior incentivador para que eu ingressasse no mestrado. À equipe do Centro de Analistas de Negócio pelo profissionalismo e competência que sempre demonstraram para suprir minha ausência nos momentos de dedicação ao mestrado.

Aos amigos que me apoiaram e ajudaram, direta ou indiretamente, ao longo desses últimos dois anos. Um agradecimento especial aos amigos Humberto Mossri e Roger Torres por todo apoio e incentivo, principalmente nos momentos mais difíceis.

Ao Professor Marco Túlio, meu orientador, apenas agradecer seria muito pouco. Ficará uma enorme gratidão pelo exemplo de pesquisador, por todos os ensinamentos, pela confiança, disponibilidade e compreensão. Enfim, por ter sempre se empenhado para viabilizar as melhores condições possíveis para que eu pudesse chegar até aqui.

Muito obrigado a todos!

Conteúdo

Lista de Figuras **vi**

Lista de Tabelas **viii**

1 Introdução **1**

1.1 Motivação 1
1.2 Objetivos 5
1.3 Estrutura da Dissertação 6

2 Revisão da Literatura **7**

2.1 Desenvolvimento Orientado por Aspectos 7
2.1.1 Programação Orientada por Aspectos 7
2.1.2 AspectJ 9
2.1.3 Tipos de Interesses Transversais 10
2.2 Mineração de Aspectos 11
2.2.1 Técnicas para Mineração de Aspectos 11
2.2.2 Ferramentas para Mineração de Aspectos 13
2.3 Refatorações Orientadas por Aspectos 15
2.4 Ferramentas para Extração de Aspectos 18
2.4.1 AOP-Migrator 20
2.4.2 Ferramenta para refatoração baseada em papéis 21
2.4.3 Ferramenta proposta por Anbalagan e Xie 22
2.5 Experiências no Uso de Aspectos 24
2.6 Considerações Finais 26

3 Transformações Orientadas por Objetos **27**
3.1 Movimentações de Código . . . 28
3.1.1 Pré-condições para Movimentações de Código . . . 33
3.1.2 Transformações *Ad Hoc* . . . 34
3.2 Extrações de Métodos . . . 34
3.2.1 Limitações para Extração de Métodos . . . 38
3.3 Definições Formais . . . 40
3.4 Considerações Finais . . . 45

4 Ferramenta para Mapeamento de Transformações **47**
4.1 Objetivos . . . 47
4.2 Principais Funcionalidades . . . 48
4.3 Implementação . . . 51
4.4 Aplicabilidade e Restrições . . . 53
4.5 Considerações Finais . . . 57

5 Avaliação **58**
5.1 JSpider . . . 58
5.2 Prevayler . . . 60
5.3 JHotDraw . . . 61
5.4 JAccounting . . . 62
5.5 Discussão . . . 63
5.5.1 Importância das Transformações . . . 65
5.5.2 Interesses Transversais Homogêneos e Heterogêneos . . . 65
5.5.3 Automatização de Transformações . . . 66
5.6 Considerações Finais . . . 67

6 Conclusões **68**
6.1 Contribuições . . . 69
6.2 Comparação com Outros Trabalhos . . . 70
6.3 Trabalhos Futuros . . . 71

Bibliografia **72**

Lista de Figuras

1.1 Processo de migração de um sistema legado para uma versão orientada por aspectos [AKT07] . . . 2
1.2 Exemplo de transformação no sistema JSpider. Comentários indicam código correspondente a um interesse transversal. . . . 3

2.1 Módulos de um sistema composto por múltiplos interesses [AKT07] . . . 8
2.2 Exemplo de aspecto para registro de *log*. [TBBV04] . . . 11

3.1 Código antes e após aplicação da Transformação 1 em método do sistema JAccounting. Comentários indicam código classificado como transversal. . 30
3.2 Código antes e após aplicação da Transformação 5 em método do sistema JAccounting. . . . 31
3.3 Código antes e após aplicação da Transformação 16 em método do sistema JHotDraw. . . . 32
3.4 Exemplo de aplicação da Transformação 17 em um método do sistema JHotDraw. . . . 33
3.5 Exemplo de aplicação da Transformação 4, seguida da aplicação de uma transformação *ad hoc*, no sistema JAccounting. . . . 35
3.6 Exemplo de aplicação da Transformação 19 em parte de uma expressão no sistema JSpider. . . . 37
3.7 Exemplo de aplicação da Transformação 19 em um bloco de comandos no sistema JSpider. . . . 38
3.8 Exemplo de aplicação da Transformação 21 no sistema Prevayler. . . . 39
3.9 Exemplo de aplicação da Transformação 23 no sistema Prevayler. . . . 40
3.10 Exemplo de aplicação da Transformação 24 no sistema JSpider. . . . 41

3.11 Transformações T1-T9 43
3.12 Transformações T10-T17 44
3.13 Transformações T18-T24 45

4.1 Mapeamento, através do *ConcernMapper*, da funcionalidade de *logging* no sistema JSpider. 48
4.2 Mapeamento, através do *TransformationMapper*, da funcionalidade de *logging* no sistema JSpider. 49
4.3 Vínculos estabelecido pelo *TransformationMapper* entre diferentes visões do Eclipse. 50
4.4 Estatísticas obtidas após mapeamento da funcionalidade de *logging* do sistema JSpider. 50
4.5 Diagrama de Pacotes do *TransformationMapper*. 53
4.6 Exemplo de método do sistema JSpider onde a funcionalidade de *logging* está presente. 54
4.7 Resultado gerados pelo *TransformationMapper* (considerando como alvo o método da Figura 4.6). 55

5.1 Código antes e após aplicação da Transformação 19 em método do sistema JSpider. 60
5.2 Código antes e após aplicação da Transformação 17 em método do sistema JHotDraw. 62
5.3 Código antes e após aplicação de transformações em método do sistema JAccounting. 64

Lista de Tabelas

2.1 Refatorações para extração de interesses transversais [MF06] 16
2.2 Refatorações para reestruturação interna de aspectos [MF06] 17
2.3 Refatorações para generalização e especialização de aspectos [MF06] 18
2.4 Leis de programação [CB05] 19

3.1 Movimentações de código 29
3.2 Extração de Métodos 36

4.1 Justificativas para os resultados gerados pelo *TransformationMapper* (considerando como alvo o método da Figura 4.6) 56
4.2 Informações sobre o código fonte do *ConcernMapper* e do *TransformationMapper* 56

5.1 Transformações no sistema JSpider 59
5.2 Transformações no sistema Prevayler 61
5.3 Transformações no sistema JHotDraw 61
5.4 Transformações no sistema JAccounting 63
5.5 Resultados das avaliações 65

Capítulo 1

Introdução

1.1 Motivação

Nos últimos dez anos, aspectos consolidaram-se como o principal paradigma de programação quando se trata de separação e modularização de interesses transversais. Normalmente, a implementação de tais interesses ocorre de forma entrelaçada e espalhada pelo código base de uma aplicação, o que prejudica o entendimento, a evolução e a manutenção destes. Exemplos de interesses com essas características incluem: mecanismos de *logging*, tratamento de exceções, persistência, controle de concorrência, distribuição e controle de transações. O desenvolvimento de software orientado por aspectos torna possível expressar tais interesses de forma modularizada, promovendo a reutilização de código e reduzindo o espalhamento e o entrelaçamento do código transversal em relação ao código de uma aplicação [KLM+97].

A consolidação dos conceitos de desenvolvimento de software orientado por aspectos motivou a realização de diversos trabalhos em duas áreas que são críticas em processos de migração de sistemas para essa nova tecnologia: mineração de aspectos [CMM+05, HK01, SGP04, AKT07] e refatoração orientada por aspectos [Lad03, HMK05, MF05, MF06]. Mineração de aspectos tem como objetivo central a identificação de interesses transversais em código orientado por objetos. Uma vez identificados os interesses transversais presentes no código fonte de um sistema, técnicas de refatoração orientada por aspectos – ou mais precisamente, de refatoração de interesses transversais – são usadas para modularizar em aspectos a implementação de tais interesses. Como em qualquer processo de refatoração, o objetivo final é melhorar a qualidade do projeto e da organização interna de um sistema, preservando seu comportamento [FBB+99]. A Figura 1.1 mostra uma representação do processo de mineração de aspectos seguido pela refatoração orientada por aspectos em um sistema legado.

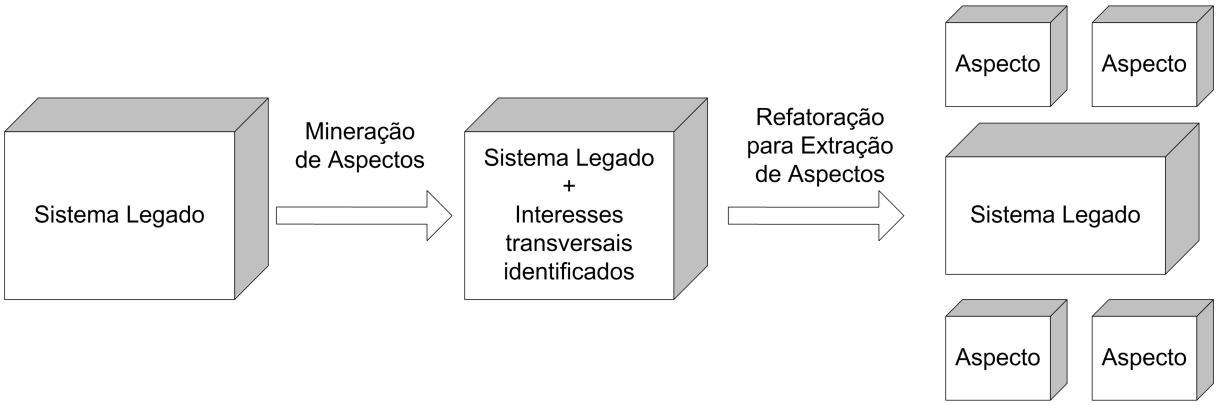

Figura 1.1: Processo de migração de um sistema legado para uma versão orientada por aspectos [AKT07]

Apesar de trazerem benefícios como modularização e facilidade de entendimento, manutenção e reutilização de código, mineração de aspectos e refatoração orientada por aspectos, quando aplicadas manualmente em sistemas legados, constituem atividades complexas, trabalhosas e sujeitas a erros. Diante disso, torna-se necessária a utilização de ferramentas de apoio por parte dos engenheiros de software para facilitar tais atividades.

No entanto, nem sempre as refatorações necessárias podem ser aplicadas de forma direta visando a extração de aspectos, mesmo quando o interesse transversal já tenha sido corretamente identificado e delimitado no código fonte de um sistema [MF05, BCH+05, BCH+06]. Esse fato ocorre porque linguagens orientadas por aspectos permitem a introdução de comportamento transversal apenas em pontos de junção, os quais correspondem a pontos pré-definidos da execução de um programa. Por exemplo, em AspectJ estes pontos incluem chamadas e execução de métodos e construtores, leitura e atribuição em campos de classes, execução de tratadores de exceção, dentre outros [KLM+97, KHH+01]. Contudo, em se tratando de sistemas legados, não se pode garantir que o código que deve ser extraído para um aspecto ocorre exatamente antes, depois ou no lugar de um ponto de junção (como requerido por AspectJ). Como conseqüência, após a identificação de um código transversal e antes da aplicação de uma refatoração visando a sua extração para um aspecto, normalmente é necessária uma *transformação* no programa base, de forma a associar este código a um ponto do programa passível de captura usando os recursos para descrição de conjuntos de junção de uma linguagem orientada por aspectos [BCH+05, BCH+06].

Utiliza-se nesta dissertação o termo *transformações* para caracterizar as tarefas que fazem parte do processo de preparação do código base de um sistema para a extração de aspectos, pois pretende-se diferenciar essas tarefas daquelas que fazem parte do processo

de refatoração de interesses transversais propriamente dito. Essas tarefas também não podem ser caracterizadas como refatorações orientadas por objetos, pois nem sempre elas promovem melhorias no projeto de código de um sistema.

Para exemplificar uma transformação orientada por objetos, mostra-se na Figura 1.2 um exemplo onde o código correspondente a um interesse transversal se encontra em um ponto do método que não pode ser interceptado por AspectJ. A transformação realizada consiste em extrair um método que retorne o valor atribuído à variável local `interval` e substituir a atribuição da constante `INTERVAL_MIN` a essa variável pela chamada do novo método. Após essa transformação, pode-se criar um conjunto de junção que capture a chamada do novo método e associá-lo um adendo do tipo `before` contendo o código identificado como transversal.

```
public Throttle createThrottle(PropertySet props) {
  int interval = props.getInteger(INTERVAL, INTERVAL_DEFAULT);
  ...
  if (interval < INTERVAL_MIN) {
    log.warn("Throttle interval < " + INTERVAL_MIN +       // aspecto
      " ms is dangereous - set to minimum allowed of " +   // aspecto
      INTERVAL_MIN + " ms");                               // aspecto
    interval = INTERVAL_MIN;
  } ...
}
```

⇓

```
public Throttle createThrottle(PropertySet props) {
  int interval = props.getInteger(INTERVAL, INTERVAL_DEFAULT);
  ...
  if (interval < INTERVAL_MIN) {
    log.warn("Throttle interval < " + INTERVAL_MIN +       // aspecto
    " ms is dangereous - set to minimum allowed of " +     // aspecto
    INTERVAL_MIN + " ms");                                 // aspecto
    interval = setToMininum();
  } ...
}

private int setToMininum() {                        // método extraído
  return INTERVAL_MIN;
}
```

Figura 1.2: Exemplo de transformação no sistema JSpider. Comentários indicam código correspondente a um interesse transversal.

O caminho entre mineração de aspectos e refatoração orientada por aspectos pode se

tornar longo quando as pré-condições requeridas pelas ferramentas que suportam processos automatizados de extração de aspectos não são atendidas. Esse esforço pode caracterizar uma lacuna entre o processo de mineração de aspectos e o processo de refatoração [YR07].

Em geral, trabalhos na área de refatoração de interesses transversais sempre mencionam a necessidade de transformações ou preparação do código de uma aplicação para viabilizar a extração de aspectos. No entanto, normalmente essa necessidade é mencionada de forma abreviada, sem um esforço mais aprofundado de descrição, classificação e quantificação das transformações requeridas. Por exemplo, em um dos seus primeiros artigos na área, Monteiro reconhece que "às vezes é necessário refatorar o código base de forma a expor pontos de junção requeridos por AspectJ" [MF03]. Contudo, mesmo em trabalhos posteriores do mesmo autor, essa necessidade não é investigada de forma mais apurada [MF05, MF06].

Em um trabalho de 2001, Murphy et al. reconhecem que um interesse pode ser mais facilmente modularizado se "for realizado um trabalho mais elaborado de preparação do código fonte" [MLWR01]. No entanto, as transformações sugeridas consistem basicamente na movimentação de comandos para o início ou para o final de métodos.

Mais recentemente, Liu, Batory e Lengauer, ao proporem uma metodologia para refatoração de *features* em sistemas legados, reconhecem que uma reordenação dos comandos de um método pode ser demandada pela metodologia proposta e "que diversas iterações podem ser requeridas até se atingir uma refatoração aceitável" [LBL06]. Todavia, essas iterações não são quantificadas no estudo de caso descrito no artigo e também não são suportadas pela ferramenta que implementa a metodologia proposta.

Binkley et al. descrevem uma versão preliminar de uma ferramenta para refatoração automática de interesses transversais, chamada AOP-Migrator [BCH+05]. No estudo de caso descrito, informa-se que "a grande maioria das refatorações realizadas (80.2%) foram aplicadas sem a necessidade de qualquer transformação orientada por objetos". Porém, em um trabalho recente que descreve uma versão mais elaborada da ferramenta AOP-Migrator [BCH+06], os mesmos autores declaram que 40% das refatorações aplicadas em novos estudos de caso consistiram em transformações no código orientado por objetos dos sistemas estudados.

Finalmente, Cole e Borba, ao propor um conjunto de leis para formalizar refatorações em AspectJ reconhecem que "a maioria das leis propostas requerem pré-condições bastante específicas" e que "a determinação de tais pré-condições foi certamente a tarefa mais difícil" da pesquisa realizada [CB05]. Contudo, os autores não apresentam um estudo sobre possíveis transformações no código base de forma que as pré-condições requeridas pelas leis propostas sejam atendidas.

1.2 Objetivos

O objetivo geral desta dissertação de mestrado é investigar transformações necessárias no código-fonte de sistemas legados após a identificação de interesses transversais, a fim de viabilizar a extração de aspectos. Os objetivos específicos desta pesquisa são relacionados a seguir:

- Catalogar, descrever, formalizar e exemplificar as principais transformações orientadas por objetos necessárias para expor pontos de junção correspondentes a interesses transversais presentes no código base de aplicações legadas ou para atender a pré-condições requeridas por processos de refatoração de interesses transversais.

- Projetar e implementar uma ferramenta que possa apoiar desenvolvedores no processo de identificação das transformações necessárias no código base de sistemas legados para extração de interesses transversais previamente identificados. Essa ferramenta deve localizar as ocorrências de interesses transversais em todas as classes de um projeto e avaliar a necessidade de transformação para cada ocorrência encontrada. Como premissas para utilização dessa ferramenta serão assumidas aplicações desenvolvidas na linguagem Java, usando AspectJ como linguagem para implementação de aspectos e Eclipse como ambiente para desenvolvimento e manutenção dessas aplicações.

- Com o apoio da ferramenta proposta, pretende-se quantificar o uso das transformações catalogadas em sistemas legados que já foram alvo de migração para aspectos e analisar a importância dessas transformações em processos de refatoração orientada por aspectos. Pretende-se ainda correlacionar as transformações identificadas com o tipo de interesse transversal refatorado, pois acredita-se que a necessidade de transformações no código orientado por objetos depende não só do local onde o interesse está presente no código base de uma aplicação, mas também de características do interesse em questão.

Os resultados da pesquisa proposta podem ser úteis em processos de migração de sistemas legados para sistemas orientados por aspectos, bem como para projetistas e usuários de ferramentas de refatoração automática ou semi-automática de interesses transversais. Com o trabalho, pretende-se também demonstrar e avaliar o tamanho da lacuna existente entre os processos de mineração e de refatoração orientada por aspectos, a qual considera-se que não tenha sido devidamente abordada na literatura sobre aspectos até o momento.

1.3 Estrutura da Dissertação

Descreve-se nesta seção como está organizado o restante desta dissertação.

No Capítulo 2, realiza-se uma revisão da literatura diretamente relacionada ao desenvolvimento desta dissertação. Essa revisão inclui Desenvolvimento Orientado por Aspectos, Mineração de Aspectos, Refatorações Orientadas por Aspectos, Ferramentas para Extração de Aspectos e Experiências no Uso de Aspectos.

No Capítulo 3, apresenta-se um catálogo de transformações orientadas por objetos, o qual foi divido em dois conjuntos: Movimentações de Código e Extrações de Métodos. As transformações propostas são descritas e o uso de algumas delas é exemplificado em sistemas reais. Apresentam-se também definições formais para cada uma das transformações.

No Capítulo 4, descreve-se o projeto e a implementação da ferramenta *TransformationMapper*, que é capaz de identificar a necessidade de transformações orientadas por objetos em sistemas Java através do mapeamento de elementos que compõem interesses transversais. Apresentam-se exemplos de utilização da ferramenta, suas funcionalidades e principais limitações.

No Capítulo 5, apresentam-se os resultados de avaliações realizadas em sistemas reais que passaram por refatorações orientadas por aspectos. Esses resultados foram obtidos com o apoio do *TransformationMapper* e inspeções manuais. Discutem-se também as principais lições aprendidas com os estudos realizados.

O Capítulo 6 conclui esta dissertação, apresentando suas principais contribuições e comparações com trabalhos relacionados. Para finalizar, apresentam-se propostas para trabalhos futuros que possam dar continuidade à pesquisa desenvolvida nesta dissertação de mestrado.

Capítulo 2

Revisão da Literatura

Neste capítulo, são apresentados os principais conceitos e tecnologias utilizadas no desenvolvimento dessa dissertação de mestrado. Na Seção 2.1, apresenta-se o paradigma de programação orientada por aspectos (POA), a linguagem AspectJ e os tipos de interesses transversais mais comuns em sistemas legados. Na Seção 2.2, apresentam-se técnicas e ferramentas para Mineração de Aspectos. Na Seção 2.3, são descritos os principais catálogos que já foram propostos para refatoração orientada por aspectos e, na Seção 2.4, são apresentadas algumas ferramentas para extração de aspectos em sistemas legados. Por fim, na Seção 2.5, relacionam-se experiências de uso da tecnologia de aspectos.

2.1 Desenvolvimento Orientado por Aspectos

2.1.1 Programação Orientada por Aspectos

O paradigma de programação orientada por objetos (POO) é predominante atualmente no desenvolvimento de sistemas. Como benefícios desse paradigma estão a reutilização de código, modularidade, abstração de entidades do mundo real através de objetos, redução dos custos de construção e manutenção de sistemas, dentre outros. Orientação por objetos tem sido utilizada na implementação de requisitos funcionais e não-funcionais. Requisitos funcionais correspondem à lógica de negócio e aos objetivos centrais a serem alcançados por um sistema. Já os requisitos não-funcionais correspondem a premissas, pré-condições, atributos de qualidade, restrições ou padrões estabelecidos para a operação dos requisitos funcionais. Exemplos de requisitos não-funcionais incluem controle de transações, mecanismos de auditoria, segurança, persistência, distribuição, dentre outros.

No entanto, a POO apresenta limitações para modularização de alguns requisitos, principalmente os não-funcionais, quando estes constituem interesses transversais, ou seja,

sua implementação "atravessa" o código do sistema ocorrendo em múltiplas classes. A implementação de tais interesses apresenta dois tipos de problemas: espalhamento e entrelaçamento de código [KLM+97, Lad03]. O espalhamento é caracterizado quando a implementação de interesses transversais se espalha por múltiplas classes do sistema. Já o entrelaçamento é caracterizado pela implementação de tais interesses junto com outros requisitos do sistema. Dentre as conseqüências desses problemas estão a dificuldade para manutenção e reutilização do código correspondente aos interesses transversais, acoplamento indesejado criado entre tais interesses e o código base do sistema, problemas de legibilidade e entendimento do código, dentre outros. A Figura 2.1 mostra módulos de um sistema composto por múltiplos interesses, onde interesses transversais se espalham e se entrelaçam com a lógica de negócio.

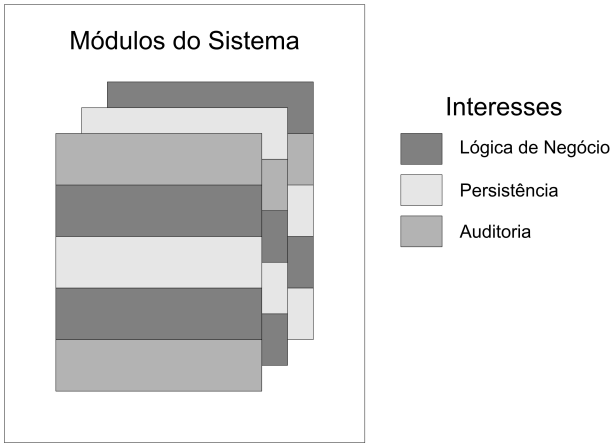

Figura 2.1: Módulos de um sistema composto por múltiplos interesses [AKT07]

Com o objetivo de apresentar uma solução para modularização de interesses transversais, o paradigma de programação orientada por aspectos (POA) foi proposto por Gregor Kiczales [KLM+97]. Nesse novo paradigma, interesses transversais podem ser modularizados de forma adequada por meio de aspectos. Assim, POA não é um paradigma que veio para substituir a POO, mas para complementá-la e enriquecê-la [Lad03].

A POA permite a implementação de interesses transversais de duas formas: (1) transversalidade estática, que permite alterações na hierarquia de classes e introdução de métodos e atributos em classes através de declarações intertipos (*inter-type declarations*), e (2) transversalidade dinâmica, que permite alterar o comportamento de um programa em pontos específicos de sua execução.

Transversalidade dinâmica ocorre a partir da definição de eventos do programa que podem ser afetados por aspectos. Tais eventos são denominados pontos de junção (*joinpoints*). Estes podem ser agrupados em conjuntos de junção (*pointcuts*) e, para cada

conjunto, são associados adendos (*advices*) que implementam regras que afetam o comportamento dos programas antes (*before*), após (*after*) ou no lugar (*around*) dos pontos de junção selecionados. Através dessas definições é possível interceptar a execução de um programa para que um interesse transversal seja introduzido.

2.1.2 AspectJ

Para modularização de interesses transversais, AspectJ [Lad03, GL03] consolidou-se como a linguagem mais comum e utilizada para sistemas desenvolvidos em Java. A linguagem possui mecanismos que suportam transversalidade estática, permitindo a introdução de membros em classes e interfaces ou a alteração de uma hierarquia de classes através da cláusula `declare parents`. Para suportar transversalidade dinâmica – permitindo a exposição de pontos onde um programa pode ser interceptado e a definição de quais ações devem ser associadas a esses pontos – AspectJ permite selecionar pontos de junção relativos aos seguintes eventos da execução de um sistema:

- **Execução e chamada de métodos:** quando a execução de um método é interceptada, o ponto de junção corresponde ao corpo do método, enquanto em uma chamada o ponto de junção corresponde ao ponto do programa onde o método é invocado. Para interceptar execução e chamada de métodos, AspectJ utiliza respectivamente as palavras reservadas `execution` e `call`, passando como parâmetro a assinatura dos métodos a serem interceptados.

- **Execução e chamada de construtores:** semelhante às definições de execução e chamada de métodos, onde substitui-se apenas o nome do método pela palavra reservada `new`.

- **Inicialização estática de classes:** o ponto de junção corresponde à execução do código de inicialização estática de uma classe. AspectJ utiliza a palavra reservada `staticinicialization`, passando como parâmetro o nome da classe.

- **Leitura e escrita em atributos de classes:** o ponto de junção corresponde a um comando de leitura ou escrita em um atributo de classe. Para interceptar esses comandos, AspectJ utiliza respectivamente as palavras reservadas `get` e `set`, passando como parâmetro os atributos a serem interceptados.

- **Execução de tratadores de exceção:** o ponto de junção corresponde à execução de um bloco `catch`. AspectJ utiliza a palavra reservada `handler` para interceptar exceções cujos tipos devem ser passados como parâmetro.

Além do modelo de pontos de junção, AspectJ também possui as seguintes características para implementação de transversalidade dinâmica:

- **Escopo de interceptação:** Para definir o escopo léxico de interceptação de pontos de junção, AspectJ utiliza as palavras reservadas `within`, para capturar pontos de junção no escopo de uma classe ou aspecto, e `withincode`, para capturar pontos de junção no escopo de um método ou construtor.

- **Captura de contexto:** AspectJ utiliza os designadores `this`, para prover acesso ao objeto corrente em um ponto de junção, `target`, para capturar o objeto que é alvo de uma chamada ou atribuição e `args`, para capturar os argumentos passados para um método.

- **Fluxo de controle:** Para capturar pontos de junção que ocorrem em um fluxo de controle, AspectJ utiliza as palavras reservadas `cflow` para capturar todos os pontos de junção que ocorrem a partir da execução de um ponto de junção passado como parâmetro e `cflowbelow`, cujo comportamento é semelhante ao anterior, excluindo-se porém a execução do próprio ponto de junção passado como parâmetro.

Para exemplificar a implementação de um aspecto, mostra-se na Figura 2.2 um exemplo onde é definido um primeiro conjunto de junção correspondente a todas as execuções de métodos públicos (linha 3), um segundo correspondente às execuções de métodos da classe `Logger` (linha 4) e um terceiro que define um conjunto composto pelo primeiro com exceção do segundo (linha 5). Em seguida, são definidos adendos que introduzem uma chamada de método para registro de *log* antes (linhas 7-9) e depois do terceiro conjunto de junção (linhas 11-13).

Apesar de flexível, o modelo de captura de pontos de junção de AspectJ apresenta limitações como, por exemplo, a impossibilidade de capturar leitura e escrita de variáveis locais e a execução de estruturas condicionais ou de repetição da linguagem Java. Uma das motivações desta dissertação é justamente discutir e propor alternativas para essas limitações.

2.1.3 Tipos de Interesses Transversais

Interesses transversais podem ser classificados em dois conjuntos: (1) homogêneos e (2) heterogêneos [CC04]. Interesses homogêneos apresentam uma lógica de implementação com características similares em todos os locais onde são implementados. Exemplos desse

```
 1: public aspect LoggingAspect {
 2:
 3:   pointcut publicMethods(): execution(public * *(..));
 4:   pointcut logObjectCalls(): execution(* Logger.*(..));
 5:   pointcut loggableCalls(): publicMethods() && !logObjectCalls();
 6:
 7:   before(): loggableCalls() {
 8:     Logger.logEntry(thisJoinPoint.getSignature().toString());
 9:   }
10:
11:   after(): loggableCalls() {
12:     Logger.logExit(thisJoinPoint.getSignature().toString());
13:   }
14: }
```

Figura 2.2: Exemplo de aspecto para registro de *log*. [TBBV04]

tipo de interesse incluem mecanismos de controle de transações, autenticação e autorização, dentre outros. Já os interesses heterogêneos apresentam comportamento que pode variar de um ponto para outro do sistema.

Interesses homogêneos se beneficiam diretamente de uma implementação orientada por aspectos, pois facilitam a implementação de conjuntos de junção que agrupam múltiplos pontos de junção. Sendo assim, tiram proveito dos recursos de quantificação característicos de AspectJ [FF00]. Por outro lado, interesses heterogêneos, em função de demandarem o emprego de códigos diferentes em pontos diferentes do sistema, podem dificultar a implementação de conjuntos de junção que consigam agrupar um conjunto significativo de pontos de junção.

2.2 Mineração de Aspectos

2.2.1 Técnicas para Mineração de Aspectos

Através do desenvolvimento de software orientado por aspectos é possível modularizar a implementação de interesses transversais que se encontram espalhados e entrelaçados com o código base de um sistema. No entanto, a identificação manual da ocorrência de tais interesses em sistemas legados pode ser uma tarefa difícil, trabalhosa e sujeita a erros. Esta dificuldade demanda técnicas e ferramentas para facilitar o processo de descoberta e localização de interesses transversais em sistemas legados ou até mesmo automatizar esse processo de mineração de aspectos (*aspect mining*) [AKT07].

Técnicas de mineração de aspectos têm como objetivo apresentar aspectos candidatos a partir de critérios de seleção definidos. A partir desses critérios, as técnicas de mineração avaliam o código fonte de um sistema buscando características que identifiquem os interesses transversais candidatos. A seguir são apresentadas as principais técnicas utilizadas para mineração de aspectos:

- **Mineração baseada em texto [HK01]:** pode ser utilizada quando existe uma padronização consistente para nomear tipos, métodos, atributos e classes. Por exemplo, a identificação de um interesse transversal cujo código segue um padrão que define um prefixo para todos os elementos que implementam esse interesse facilita tanto a identificação quanto a extração do mesmo para um aspecto. Quando não existe esse tipo de padronização, essa técnica de mineração tende a falhar ou ser pouco eficiente, pois uma parte da implementação do interesse transversal poderá não ser identificada. Em termos práticos, sistemas legados dificilmente apresentam um padrão de nomenclatura que possa facilitar a descoberta e localização de interesses transversais.

- **Mineração baseada em tipos [HK01]:** analisa a utilização de tipos no código de um sistema e não depende da padronização de nomes para os elementos que compõem esse código. A utilização dessa técnica depende do grau de modularização do sistema legado, pois é necessário que o interesse transversal esteja modularizado em uma ou mais classes para que suas ocorrências sejam devidamente identificadas. Particularmente, essa técnica não pode ser aplicada caso existam várias instâncias de uma mesma classe associadas a diferentes interesses. Por exemplo, em um sistema Java, onde apenas alguns botões pertencentes à classe `JButton` fazem parte de um determinado interesse transversal, não seria possível diferenciar esses botões dos demais através de mineração baseada em tipos.

- **Análise *Fan-in* [MDM04]:** a métrica *fan-in*, proposta por Henderson-Sellers, consiste em contar o número de chamadas de um determinado método realizadas em outros métodos [HS96]. Em 2004, Marin et al. propuseram a utilização dessa métrica para descobrir interesses transversais em sistemas legados [MDM04]. O algoritmo de mineração proposto consiste basicamente de três etapas: (1) calcular a métrica *fan-in* para todos os métodos do sistema; (2) filtrar os resultados eliminando métodos `get` (*acessors*), `set` (*mutators*) e utilitários e considerar apenas métodos cuja métrica superou um determinado valor; (3) analisar manualmente os métodos selecionados. No trabalho mencionado, os autores apresentaram um estudo de caso

onde um terço dos métodos selecionados através da métrica *fan-in* corresponderam efetivamente a interesses transversais do sistema.

- **Detecção de clones [SGP04]:** a duplicação de código em um sistema pode ser considerada um indicador de interesse transversal. Um exemplo de interesse transversal com esse tipo de comportamento é a implementação de um mecanismo de controle de transações, cujos blocos de comandos para abrir, confirmar e cancelar uma transação seguem sempre um mesmo padrão em todos os pontos onde esse interesse estiver presente.

- **Análise formal de conceitos em traços de execução:** a análise formal de conceitos (FCA) é um modelo matemático para análise estruturada de dados. Esse modelo procura estabelecer relacionamentos entre objetos e atributos através de uma tabela de contexto, produzindo conceitos. Assim, FCA permite identificar grupos significativos de elementos que possuem propriedades comuns. Em um trabalho realizado em 2004, Tonella e Ceccato propuseram a aplicação de FCA em traços de execução de programas para identificação de interesses transversais [TC04]. Através da execução de uma versão instrumentada de um sistema para um conjunto de casos de uso pré-determinado, são gerados traços de execução para que uma análise baseada em FCA possa identificar interesses transversais.

Cada técnica para mineração de aspectos se aplica melhor a um determinado tipo de interesse transversal. Portanto, as técnicas não são mutuamente exclusivas e as ferramentas para mineração de aspectos acabam utilizando mais de uma técnica para aumentar a sua eficiência e eficácia.

2.2.2 Ferramentas para Mineração de Aspectos

Ferramentas para mineração de aspectos são utilizadas para automatizar as técnicas de mineração. Uma ferramenta com esse propósito deve suportar uma análise quantitativa e qualitativa de entrelaçamento e espalhamento de código em sistemas legados de acordo com critérios de avaliação pré-estabelecidos. A seguir são descritas as principais ferramentas para mineração de aspectos:

- AMT: apresentada por Hannemann e Kiczales em 2001, foi uma das primeiras ferramentas para mineração de aspectos [HK01]. Utiliza as técnicas de mineração baseada em texto, para unificar padrões, e baseada em tipo, para identificar código

entrelaçado. A ferramenta possibilita realizar consultas a partir de expressões regulares ou de tipos utilizados no sistema e é composta por dois componentes: analisador e visualizador. O analisador (*AMTAnalyzer*) estende o compilador de AspectJ para extrair informações relevantes do código-fonte e armazená-las em um arquivo de resultados. O visualizador (*AMTVisualizer*) utiliza as informações armazenadas para representar graficamente o sistema.

- FINT: é um *plugin* para o ambiente Eclipse que foi desenvolvido para suportar a localização de interesses transversais com base na análise *fan-in* [MMD06]. Para computar a métrica *fan-in*, primeiro a ferramenta constrói a árvore de sintaxe abstrata para as classes selecionadas e depois cria um grafo contendo os métodos declarados nessas classes e os que fazem chamadas a estes em outras classes do sistema. A métrica *fan-in* é então calculada a partir desse grafo e o usuário da ferramenta pode aplicar filtros, como número mínimo de ocorrências, padrão de nomes e assinaturas de métodos, dentre outros.

- Dynamo: é uma ferramenta para identificação de interesses transversais existentes em sistemas Java utilizando análise dinâmica de código [TC04]. Traços de execução são gerados para os casos de uso que executam as principais funcionalidades de um sistema e, a partir da análise dos resultados dessas execuções e da aplicação da técnica de análise formal de conceitos (FCA), interesses transversais são detectados quando contribuem para a implementação de múltiplos casos de uso.

- PRISM: é um *plugin* para o ambiente Eclipse que utiliza técnicas de mineração baseadas em padrões léxicos e padrões de tipos [ZJ03a]. É composto basicamente por três abstrações: (1) impressões digitais dos aspectos (*Prism aspect fingerprint*) que representam os interesses transversais; (2) *footprint* de aspectos (*Prism aspect footprint*), que representam as localizações dos interesses transversais; (3) algoritmo de mineração (*Prism aspect mining algorithm*), que recebe impressões digitais como entrada e produz *footprints* como saída.

- FEAT: *Feature Exploration and Analysis Tool* é um *plugin* para o ambiente Eclipse que utiliza grafos de interesses (*concern graphs*) para localizar interesses transversais em um sistema. Um grafo de interesse abstrai detalhes da implementação do interesse armazenando seus elementos – classes, métodos e atributos – e respectivos relacionamentos e dependências. Como resultado, essa estrutura permite a navegação e manipulação da representação de um interesse em um nível mais alto de abstração.

2.3 Refatorações Orientadas por Aspectos

Refatoração consiste na melhoria do projeto interno de um sistema preservando o seu comportamento externo [FBB+99]. Refatorações orientadas por aspectos podem ser divididas em dois grupos: (1) refatorações para extração de aspectos, cujo objetivo é a modularização de interesses transversais, e (2) refatorações de aspectos, para facilitar o entendimento, manutenção e evolução dos mesmos. A seguir são descritos os principais catálogos para as refatorações correspondentes a esses grupos.

Monteiro e Fernandes descrevem um conjunto de refatorações para extração, reestruturação interna e generalização de aspectos [MF05, MF06]. Com esse propósito, eles apresentam um catálogo com três conjuntos de refatorações: (1) um conjunto de dez refatorações para eliminar problemas em código orientado por objetos relacionados à implementação de interesses transversais, (2) um conjunto de seis refatorações para reestruturação interna de aspectos, com o objetivo de reduzir duplicidade e facilitar o entendimento e a manutenção dos aspectos, e (3) um conjunto de onze refatorações para generalizar e especializar aspectos, com o objetivo de promover reutilização de código e a criação de super-aspectos. O primeiro conjunto pertence ao primeiro grupo de refatorações mencionado no parágrafo anterior (extração de aspectos), já o segundo e o terceiro conjuntos estão no segundo grupo (refatorações de aspectos). Algumas dessas refatorações são derivadas de refatorações orientadas por objetos tradicionais, adaptadas para programação orientada por aspectos. Exemplos dessas adaptações incluem extração de métodos, movimentação de atributos e métodos para superclasses, dentre outras. As Tabelas 2.1, 2.2 e 2.3 sintetizam os três conjuntos de refatorações propostos por Monteiro e Fernandes.

Cole e Borba descrevem um conjunto de leis de programação para auxiliar na formalização e automatização de processos de extração de aspectos [CB05]. As leis propostas se aplicam tanto ao processo de extração de aspectos quanto à reestruturação destes e têm como principal objetivo garantir que o comportamento do sistema seja preservado após as refatorações realizadas. Leis de programação são mais simples que refatorações e se aplicam a construções específicas de AspectJ. Cada lei é descrita através de seu objetivo, construção e pré-condições. A Tabela 2.4 apresenta um sumário das leis de programação propostas. Essas leis se mostraram eficientes na modularização de interesses transversais de dois sistemas reais, cujos interesses refatorados foram respectivamente distribuição e controle de concorrência. No entanto, segundo os autores, uma das principais dificuldades encontradas no processo de refatoração desses sistemas foi definir as pré-condições necessárias para aplicação das leis propostas.

Refatoração	Descrição
Alterar classe abstrata para interface	Possibilitar que subclasses implementem várias interfaces ao invés de herdar apenas de uma classe
Extrair funcionalidade para aspecto	Extrair para um aspecto todo código que implementa uma funcionalidade e que esteja espalhado por múltiplas classes
Extrair fragmento para adendo	Extrair para um adendo parte do código de um método que corresponde a um interesse transversal
Extrair classe interna para aspecto	Extrair para aspectos classes internas que correspondem a interesses transversais
Mover classe para aspecto	Mover para um aspecto classes que são utilizadas apenas pelo aspecto
Mover interface para aspecto	Mover para um aspecto interfaces que são utilizadas apenas pelo aspecto
Mover atributo para aspecto	Mover para um aspecto atributos relacionados a um interesse transversal implementado pelo aspecto
Mover método para aspecto	Mover para um aspecto métodos relacionados a um interesse transversal implementado pelo aspecto
Substituir `implements` por `declare parents`	Retirar de uma classe a cláusula `implements` relacionada a um interesse transversal e incluir a cláusula `declare parents` no aspecto correspondente a esse interesse
Transformar classe abstrata em aspecto e interface	Mover membros concretos de uma classe abstrata para um aspecto e transformar a classe restante em interface

Tabela 2.1: Refatorações para extração de interesses transversais [MF06]

Outro catálogo relacionado ao grupo de extração de aspectos foi proposto por Laddad [Lad03]. Como exemplo introdutório, foram apresentados os passos necessários para extrair de um sistema um conjunto de chamadas de métodos e modularizá-las em um aspecto. Em seguida, foi apresentado um conjunto de técnicas para refatoração de interesses transversais presentes em diversos tipos de aplicações. Essas técnicas incluem refatorações para extração de código de tratamento de exceções, para criação de *worker objects*, para criação de *wormholes*, para implementação de métodos de uma interface, para sobreposição de métodos através de adendos, dentre outras.

Por fim, com o objetivo de avaliar o processo de refatoração de software através de Decomposição Horizontal (HD) e programação orientada por aspectos, Godil e Jacobsen realizaram um estudo de caso no qual princípios de HD foram aplicados no Prevayler – um sistema gerenciador de banco de dados em memória principal – para modularização de seis interesses transversais: gerenciamento de *snapshots*, censura de transações, replicação, *logging* persistente, *clock* de transações e concorrência [GJ05].

HD é um conjunto de princípios para conduzir refatorações orientadas por aspectos

Refatoração	Descrição
Estender interfaces de marcação com declarações intertipos	Definir métodos abstratos através de declarações intertipos para que interfaces de marcação não dependam de conversão de tipos (*casting*) quando esses métodos forem utilizados
Utilizar interfaces de marcação para generalizar tipos	Substituir referências a tipos concretos por interfaces de marcação para possibilitar a extensão de aspectos
Introduzir proteção de aspectos	Implementar declarações de erros para utilização de membros públicos
Substituir declarações intertipos por estruturas de mapeamento	Substituir declarações intertipos que introduzem estados adicionais em classes por estruturas de dados internas ao aspecto que possam mapear o objeto alvo e o estado adicional
Substituir declarações intertipos de métodos por métodos do aspecto	Substituir declarações intertipos que introduzem métodos em classes por métodos do aspecto
Organizar o aspecto internamente	Melhorar a organização interna de um aspecto removendo, por exemplo, declarações intertipos duplicadas

Tabela 2.2: Refatorações para reestruturação interna de aspectos [MF06]

através de duas dimensões arquiteturais: dimensão vertical, que implementa um núcleo composto pelas funcionalidades essenciais de um sistema, e horizontal, correspondente a interesses transversais que implementam requisitos funcionais ou não-funcionais do sistema e que podem ser acoplados à arquitetura vertical de forma independente [ZJ04]. De acordo com os princípios de HD, utilizando o núcleo de uma aplicação como referência, uma funcionalidade é considerada transversal se sua semântica e sua implementação não estão restritas a um único componente desse núcleo. Para medir o nível de transversalidade de um interesse, foram considerados o número de métodos, classes, pacotes e aplicações através dos quais a implementação do interesse está espalhada. A refatoração dos aspectos selecionados foi realizada em duas etapas: implementação das funcionalidades dos aspectos e implementação da interação dos aspectos com o núcleo da aplicação. Todas as classes cujas funcionalidades implementam apenas os interesses transversais selecionados foram movidas para o pacote correspondente ao interesse e em seguida o respectivo código que se encontrava espalhado pela aplicação foi refatorado para um aspecto desse pacote. A modularização dos seis interesses transversais considerados no estudo de caso realizado permitiu a geração de 36 diferentes configurações para o Prevayler através de suas combinações.

Uma das contribuições do trabalho de Godil e Jacobsen foi a proposição de um catálogo com 22 padrões de refatoração, dos quais 17 foram resultantes do trabalho realizado e 5 de validações de refatorações já existentes. Exemplos dos padrões propostos incluem:

Refatoração	Descrição
Extrair super-aspectos	Agrupar aspectos com estrutura e comportamento similar em um super-aspecto
Generalizar adendos	Mover para super-aspectos adendos iguais presentes em vários sub-aspectos
Generalizar `declare parents`	Mover para super-aspectos cláusulas `declare parents` presentes em sub-aspectos
Generalizar declarações intertipos	Mover para super-aspectos declarações intertipos presentes em sub-aspectos
Generalizar interfaces de marcação	Mover para super-aspectos interfaces de marcação presentes em sub-aspectos
Generalizar conjuntos de junção	Mover para super-aspectos conjuntos de junção idênticos em sub-aspectos
Especializar adendos	Retirar de super-aspectos partes específicas de adendos e movê-las para os sub-aspectos que as utilizam
Especializar `declare parents`	Retirar de super-aspectos `declare parents` que não são relevantes para todos os sub-aspectos e movê-los para os sub-aspectos que os utilizam
Especializar declarações intertipos	Retirar de super-aspectos declarações intertipos que não são relevantes para todos os sub-aspectos e movê-las para os sub-aspectos que as utilizam
Especializar interfaces de marcação	Retirar de super-aspectos interfaces de marcação que não são relevantes para todos os sub-aspectos e movê-las para os sub-aspectos que as utilizam
Especializar conjuntos de junção	Retirar de super-aspectos conjuntos de junção que não são relevantes para todos os sub-aspectos e movê-los para os sub-aspectos que os utilizam

Tabela 2.3: Refatorações para generalização e especialização de aspectos [MF06]

criar aspecto correspondente a interesse transversal, aumentar a visibilidade de atributos ou métodos para torná-los acessíveis pelos aspectos, introduzir métodos auxiliares para interceptar pontos que não podem ser capturados por conjuntos de junção, dentre outros. Ou seja, as refatorações propostas fazem parte do grupo de extração de aspectos.

2.4 Ferramentas para Extração de Aspectos

Para extrair interesses transversais de aplicações legadas e modularizá-los em aspectos, são necessárias refatorações no código da aplicação. Esse processo de extração envolve basicamente duas etapas: mineração de aspectos, abordada na Seção 2.2, e refatorações orientadas por aspectos, abordada na Seção 2.3. Uma vez identificados por meio de um

Lei	Nome
1	Criar aspecto vazio
2	Tornar aspecto privilegiado
3	Criar adendo do tipo `before-execution`
4	Criar adendo do tipo `before-call`
5	Criar adendo do tipo `after-execution`
6	Criar adendo do tipo `after-call`
7	Criar adendo do tipo `after-execution` com retorno bem sucedido
8	Criar adendo do tipo `after-call` com retorno bem sucedido
9	Criar adendo do tipo `after-execution` com retorno por meio de exceção
10	Criar adendo do tipo `after-call` com retorno por meio de exceção
11	Criar adendo do tipo `around-execution`
12	Criar adendo do tipo `around-call`
13	Agrupar adendos
14	Remover parâmetro `this`
15	Remover parâmetro `target`
16	Remover parâmetro `args`
17	Capturar exceções suavizadas
18	Suavizar exceções
19	Remover exceções de cláusulas `throws`
20	Remover tratamento de exceções
21	Mover tratamento de exceções para aspecto
22	Mover atributo para aspecto
23	Mover método para aspecto
24	Mover declaração `implements` para aspecto
25	Mover declaração `extends` para aspecto
26	Extrair conjunto de junção nomeado
27	Usar conjunto de junção nomeado
28	Mover introdução de atributo para interface
29	Mover introdução de método para interface
30	Remover implementação de método

Tabela 2.4: Leis de programação [CB05]

processo de mineração de aspectos, os interesses transversais a serem refatorados podem se apresentar em múltiplos pontos do sistema. Diante dessa característica, o processo de extração manual de aspectos pode se tornar improdutivo, tedioso e sujeito a erros. Para minimizar esse problema, torna-se importante a utilização de ferramentas que possam automatizar total ou parcialmente o processo de extração de aspectos em sistemas legados. A seguir são descritas três ferramentas com esse propósito.

2.4.1 AOP-Migrator

A ferramenta AOP-Migrator[1], proposta por Binkley, Ceccato, Harman, Ricca e Tonella, tem como objetivo a migração de interesses transversais para aspectos [BCH+05, BCH+06]. Como pré-condição para sua utilização, é necessário que um processo de mineração de aspectos seja previamente realizado no sistema e que cada ocorrência de um interesse transversal esteja delimitada por comentários contendo anotações específicas. O processo de extração de aspectos consiste em percorrer todos os blocos de código previamente marcados no sistema e, com o envolvimento do desenvolvedor, mover o código para os aspectos correspondentes. Essa iteração é feita até que todos os blocos sejam removidos da aplicação.

A ferramenta implementa seis tipos de refatorações cujos mecanismos podem ser automatizados. Esse conjunto tem como objetivo cobrir a maior parte dos casos encontrados em sistemas reais. A seguir são descritas as refatorações realizadas pela ferramenta:

- **Extração no Início/Final de Método/Bloco de Exceção:** O código marcado se encontra no início ou no final de um método ou em um de seus blocos de tratamento de exceções.
- **Extração Antes/Após Chamada de Método:** O código marcado se encontra antes ou depois de uma chamada de método.
- **Extração Condicional:** Um comando condicional controla a execução de código marcado.
- **Extração Pré-retorno:** O código marcado se encontra imediatamente antes de um comando `return`.
- **Extração de Bloco:** O código marcado faz parte de um bloco de código que será totalmente extraído para um aspecto.
- **Extração de Tratamento de Exceção:** O código marcado corresponde a um bloco de tratamento de exceção.

O processo de extração de aspectos pela ferramenta ocorre em quatro etapas que são descritas a seguir:

- **Descoberta:** A ferramenta analisa um bloco de código marcado e tenta associá-lo às refatorações possíveis. Essa associação ocorre quando uma refatoração é aplicável

[1] `http://se.itc.it/aop-migrator`

ao bloco e pode ser utilizada para movê-lo para um aspecto. Todas as refatorações aplicáveis são listadas em ordem de aplicabilidade, definida de acordo com o impacto no código original e com o grau de complexidade e generalidade do aspecto a ser criado.

- **Seleção:** O usuário da ferramenta escolhe qual refatoração deseja aplicar, obedecendo ou não a ordem de prioridade sugerida. A ferramenta permite ao usuário avaliar as alternativas sugeridas apresentando uma simulação de como o código ficará após a aplicação da refatoração.

- **Transformação:** Quando a ferramenta não consegue associar nenhuma das refatorações suportadas a um bloco de código, é necessário realizar transformações no código orientado por objetos como, por exemplo, extração de métodos, movimentação de comandos ou qualquer outro tipo de intervenção manual no código original.

- **Refatoração:** O bloco de código marcado é removido da classe e introduzido em um adendo. De acordo com o mecanismo de refatoração selecionado, é gerado também um conjunto de junção apropriado para interceptar o ponto de junção correspondente ao bloco extraído. Antes de criar um novo adendo, a ferramenta verifica se já existe algum outro com o mesmo código. Caso isso ocorra, realiza-se apenas um agrupamento de conjuntos de junção.

Para avaliar a ferramenta, Binkley et al. realizaram quatro estudos de caso nos quais interesses transversais foram extraídos de sistemas reais. Os resultados dos estudos de caso comprovaram que é possível realizar apenas parte do processo de extração de aspectos de forma automatizada, pois é necessário a intervenção humana para selecionar a refatoração a ser aplicada em cada bloco de código marcado. Além disso, em várias situações, é necessário realizar transformações no código orientado por objetos para viabilizar as refatorações suportadas pela ferramenta. Segundo os autores, a fase mais crítica do processo de extração de aspectos foi a realização de tais transformações.

2.4.2 Ferramenta para refatoração baseada em papéis

A ferramenta proposta por Hannemann, Murphy e Kiczales apresenta uma abordagem de refatoração baseada em papéis [HMK05]. Interesses transversais são descritos em termos de papéis abstratos e as instruções para refatoração desses interesses são descritas em função desses papéis. Isso permite que as mesmas transformações sejam aplicadas a

interesses transversais com estrutura semelhante. Para aplicar uma refatoração com o apoio da ferramenta, o desenvolvedor deve mapear um subconjunto de papéis para elementos de programa e, em seguida, instruções de refatoração são aplicadas para manipular e modularizar os elementos correspondentes aos interesses transversais.

Os passos referentes ao fluxo de trabalho da ferramenta são descritos a seguir:

1. **Seleção:** O desenvolvedor seleciona a refatoração apropriada a partir de uma biblioteca de refatorações para interesses transversais. Cada refatoração inclui uma descrição abstrata para os interesses transversais aos quais ela se aplica e um conjunto de instruções para produzir uma implementação orientada por aspectos.

2. **Mapeamento:** A ferramenta auxilia o desenvolvedor a mapear papéis que descrevem o interesse transversal para elementos de programa.

3. **Planejamento:** Como a refatoração de um interesse transversal modifica várias partes de um sistema, a ferramenta realiza uma análise de impacto para identificar possíveis problemas no processo de refatoração como, por exemplo, duplicidade de nomes entre métodos, atributos, aspectos ou classes. Essa análise possibilita que o desenvolvedor planeje antecipadamente como solucionar esses problemas.

4. **Execução:** Após o planejamento da refatoração, a ferramenta transforma o código de acordo com as instruções de refatoração, incorporando as decisões tomadas na etapa de planejamento. Essa execução resulta na criação de aspectos correspondentes aos interesses transversais mapeados.

Para avaliar a abordagem e a ferramenta proposta foi realizado um estudo de caso no qual foram moduralizados seis instâncias de três padrões de projetos (*Observer, Singleton* e *Template Method*) presentes no sistema JHotDraw, um *framework* para edição gráfica. Como a abordagem baseada em papéis consiste em capturar a essência estrutural de um interesse transversal descrevendo de forma abstrata como sua implementação ocorre em um sistema, uma das principais dificuldades relatada pelos autores foi tratar as diversas variações que a implementação de determinados interesses pode apresentar em relação à sua definição abstrata. Tais variações podem fazer com que a ferramenta não reconheça todas as ocorrências desses interesses no código base de um sistema legado.

2.4.3 Ferramenta proposta por Anbalagan e Xie

A ferramenta proposta por Anbalagan e Xie vai além do processo específico de extração de aspectos, pois propõe uma abordagem automatizada que começa com a identificação

de aspectos candidatos, passa pela identificação e agrupamento de pontos de junção e termina com a geração de expressões correspondentes a conjuntos de junção, incluindo um mecanismo de testes para garantir a eficiência destes [AX07]. No entanto, a ferramenta não extrai o código dos aspectos.

A ferramenta é composta por quatro módulos principais:

- **Módulo de mineração de aspectos:** Utiliza a ferramenta de mineração de aspectos AMT (*Aspect Mining Tool* [HK01]) para identificar interesses transversais entrelaçados e espalhados pelo código fonte de um sistema.

- **Módulo identificador de pontos de junção:** A partir da lista de aspectos identificados, localiza suas ocorrências no código fonte da aplicação. Para cada ocorrência de um interesse transversal, é gerado um ponto de junção (usando AspectJ).

- **Módulo de agrupamento:** Recebe como entrada os interesses transversais candidatos e seus respectivos pontos de junção para identificar características comuns entre eles e organizá-los em grupos. Essa organização é baseada principalmente em uma análise dos padrões de nomes de métodos.

- **Módulo de inferência:** Recebe como entrada grupos de pontos de junção e gera conjuntos de junção em forma de expressões. Possui um identificador de designação para, por exemplo, distinguir execuções (`execution`) de chamadas (`call`) de métodos. Através da identificação das partes que compõem os nomes dos grupos de pontos de junção, são deduzidas expressões regulares que representam conjuntos de junção correspondentes aos pontos de junção. Esse módulo também possui um mecanismo de testes que verifica a robustez dos conjuntos de junção gerados.

Para avaliar a ferramenta foi realizado um estudo de caso com seis aplicações reais. De acordo com os resultados empíricos desse estudo, a ferramenta se mostrou eficiente no processo de simplificação de expressões correspondentes a conjuntos de junção. Entretanto, os interesses transversais identificados não foram caracterizados e, de acordo com o número de interesses apresentados para cada aplicação, pressupõe-se um baixo nível de granularidade no processo de mineração de aspectos utilizado. Por exemplo, para o sistema JHotDraw, que possui aproximadamente 28 KLOC, foram identificados 456 aspectos.

Segundo os autores da ferramenta, a atual versão não trata situações onde interesses transversais ocorrem em locais do código que não podem ser associados a pontos de junção capturados por AspectJ. Nesses casos, é necessário que o desenvolvedor refatore o código do sistema para permitir que a ferramenta identifique corretamente os pontos de junção.

2.5 Experiências no Uso de Aspectos

Desenvolvimento de Software Orientada por Aspectos (DSOA) já foi utilizado para modularizar interesses transversais em vários domínios de aplicações. Por exemplo, Colyer e Clement descrevem uma experiência de uso de aspectos em uma linha de produtos de *middleware* da IBM, incluindo interesses transversais homogêneos (*tracing*, *logging*, captura de falhas e monitoramento) e heterogêneos (suporte a EJB) [CC04]. Segundo os autores, o emprego de DSOA em linhas de produtos de *middleware* tem dois papéis principais: (1) prover consistência entre os membros de uma linha de produtos, o que permite capturar as políticas de um determinado interesse transversal e introduzi-las de forma consistente ao longo da linha de produtos, e (2) prover flexibilidade para configuração de linhas de produtos, pois funcionalidades que não estão devidamente modularizadas não podem ser facilmente retiradas ou inseridas no núcleo de uma linha de produtos construída usando técnicas tradicionais de programação orientada por objetos.

Ainda na área de *middleware*, Zhang e Jacobsen descrevem uma experiência de refatoração de interesses transversais presentes em ORBacus – um *middleware* para desenvolvimento de aplicações CORBA [ZJ03b]. Os interesses transversais refatorados incluem interface de invocação dinâmica, suporte a interceptadores portáveis, sincronização e *logging*, dentre outros. Segundo os autores, a arquitetura tipicamente usada em sistemas de *middleware* é uma das mais propícias para aplicação de DSOA, pois propicia um nível de modularidade dificilmente alcançado por técnicas tradicionais de programação. Comparando-se os resultados da experiência reportada com a versão original do ORBacus, registrou-se uma redução significativa da complexidade da arquitetura do ORB, preservando-se o seu comportamento e nível de desempenho.

Uma experiência no uso de aspectos em um sistema real baseada em AspectJ é descrita por Soares, Laureano e Borba [SLB02]. Nesse trabalho, foi utilizado o sistema *Health Watcher*, um sistema de informação Web para registro de reclamações sobre restaurantes e estabelecimentos alimentícios para que instituições de saúde possam investigá-los. O sistema foi desenvolvido em Java utilizando uma arquitetura em camadas e padrões de projetos. Apesar dessa arquitetura em camadas auxiliar na separação de interesses como gerenciamento de dados, regras de negócio, distribuição e apresentação, ela não consegue tratar completamente problemas como espalhamento e entrelaçamento de código. Em função disso, o sistema foi reestruturado com a implementação dos interesses transversais de persistência e distribuição através de aspectos.

Para modularizar o interesse de distribuição, foi extraído todo o código específico de Java RMI – *framework* Java para implementar invocações remotas de métodos – e

respectivos mecanismos de serialização e tratamento de exceções presentes no sistema. Em seguida, para reintroduzir distribuição na aplicação, foram implementados aspectos específicos para as classes que fazem parte do lado servidor e do lado cliente do sistema. O aspecto referente ao lado servidor da aplicação garante que os métodos da fachada do sistema possuam parâmetros e tipos de retorno serializáveis, conforme requerido por Java RMI. O aspecto cliente é responsável por capturar chamadas remotas e direcioná-las para a instância da fachada remota.

Para modularizar o interesse de persistência, o primeiro passo foi remover todo código de persistência do sistema, concentrado principalmente em coleções de dados e classes de persistência. Sem o código de persistência no sistema, é possível testar seus requisitos funcionais antes mesmo da implementação desse interesse. Em seguida, foram implementados aspectos e também classes e interfaces auxiliares para reintroduzir o interesse de persistência na aplicação junto com mecanismos para garantir sua consistência como, por exemplo, controle de conexões e transações, *caching* e sincronização.

A experiência realizada no sistema *Health Watcher* possibilitou a validação do uso da linguagem AspectJ para implementação de persistência e distribuição. No entanto, o estudo mostrou algumas dependências entre os interesses refatorados, relacionadas principalmente ao tratamento de exceções e sincronização, e sugeriu que atividades de projeto sejam realizadas de forma cuidadosa em DSOA para melhor tratar dependências e conflitos entre aspectos.

Experiências no uso de aspectos incluem também padrões de projetos utilizados no desenvolvimento de software orientado por objetos com o objetivo de implementá-los de forma modularizada, facilitando assim a extensão e manutenção de sistemas que utilizam esses padrões. Garcia et al. apresentam um estudo quantitativo, utilizando métricas para separação de interesses, acoplamento, coesão e tamanho, que compara a implementação orientada por aspectos com a orientada por objetos de diversos padrões de projetos [GSF+05]. A utilização de aspectos se mostrou eficiente em relação ao aumento do nível de separação de interesses para a maioria dos padrões de projeto considerados. No entanto, alguns padrões resultaram em maior acoplamento entre componentes, operações mais complexas e maior número de linhas de código.

Outra experiência relacionada a padrões de projeto, foi a descrição de um padrão para separação explícita entre o código de tratamento de erros e o código base de aplicações através de aspectos [FGR07]. A idéia básica desse padrão consiste em utilizar adendos para implementar tratamento de exceções e conjuntos de junção para associar esses adendos a diferentes partes do sistema com objetivo de facilitar a manutenção e a reutilização de código para tratamento de erros. Os autores constataram que os melhores resultados

foram obtidos quando o projeto do sistema em questão levou em consideração a utilização de aspectos. Esses resultados foram obtidos em função de uma maior exposição de pontos de junção, para facilitar a interceptação por AspectJ, e de uma preparação específica para que aspectos responsáveis pelo tratamento de exceções possam acessar informações de contexto da aplicação.

2.6 Considerações Finais

Neste capítulo, foi apresentada uma revisão da literatura diretamente relacionada ao desenvolvimento desta dissertação de mestrado. A revisão apresentada incluiu os seguintes temas: Desenvolvimento Orientado por Aspectos, com abordagem específica para a linguagem AspectJ, Mineração de Aspectos, Refatorações Orientadas por Aspectos, incluindo seus principais catálogos, Ferramentas para Extração de Aspectos e Experiências no Uso de Aspectos.

Nota-se, no entanto, que, apesar de alguns dos trabalhos referenciados citarem a necessidade de preparação prévia do código fonte de aplicações legadas de forma a habilitar a extração de aspectos, nenhum deles dá maior ênfase a essa tarefa de preparação, por exemplo, descrevendo, classificando e exemplificando as transformações requeridas. Além disso, como AspectJ apresenta algumas limitações e não é capaz de interceptar todos os pontos de execução de um sistema, a implementação de interesses transversais usando essa linguagem também pode requer transformações no código base de sistemas legados, isto é, sistemas cujo projeto inicial não levou em consideração a modularização de interesses transversais através de aspectos.

Capítulo 3

Transformações Orientadas por Objetos

Neste capítulo, são apresentadas as transformações orientadas por objetos mais comuns para extração de aspectos, alguns exemplos de uso dessas transformações em sistemas reais e uma descrição formal de cada transformação [MV07, MV08].

No contexto desta dissertação de mestrado, uma transformação de código é uma reestruturação do código de um sistema que preserva o seu comportamento e que viabiliza a aplicação, em um passo seguinte, de uma refatoração de interesses transversais. Como essas transformações não necessariamente aprimoram a organização interna do sistema, mas sim preparam o mesmo para um processo seguinte de separação de interesses, optou-se por chamá-las de transformações ao invés de refatorações. Essa distinção entre os dois termos é adotada também em outros trabalhos [BCH+05, BCH+06].

Nesta dissertação, são consideradas apenas transformações em código transversal, isto é, código que implementa interesses transversais, de sistemas desenvolvidos em Java. Considera-se ainda que o código alvo das transformações propostas já foi devidamente identificado, possivelmente com auxílio de uma ferramenta de mineração de aspectos. Por fim, assume-se que AspectJ é a linguagem que será empregada para modularização em aspectos do código resultante das transformações apresentadas.

A escolha de AspectJ se justifica por ser esta a linguagem para implementação de aspectos mais madura e utilizada atualmente. Por esta razão, as transformações orientadas por objetos propostas têm como principal propósito viabilizar a extração de aspectos com base no modelo dinâmico de pontos de junção definido por AspectJ. Esse modelo, conforme descrito no Capítulo 2, limita a captura de pontos de junção aos seguintes eventos principais da execução de um sistema Java: chamada ou execução de métodos, leitura ou modificação de atributos de classes, tratamento de exceções e inicialização estática de

classes.

Os exemplos usados para ilustrar as transformações catalogadas foram extraídos dos seguintes sistemas:

- JSpider[1]: um robô para *download* e análise de páginas *Web*, com cerca de catorze mil linhas de código. São usadas como exemplo transformações aplicadas nesse sistema por Binkley et al. para extração de aspectos responsáveis pela funcionalidade de *logging* [BCH+06].

- JHotDraw[2]: um *framework* para criação de editores de figuras geométricas, com mais de quarenta mil linhas de código. São usadas como exemplo transformações aplicadas nesse *framework* por Binkley et al. para posterior modularização em aspectos do interesse responsável pela funcionalidade de *undo* [BCH+05, BCH+06].

- JAccounting[3]: um sistema Web para controle de pedidos e pagamentos, com mais de onze mil linhas de código. São usadas como exemplo transformações aplicadas nesse sistema por Binkley et al. para viabilizar a extração de aspectos com código para controle de transações [BCH+06].

- Prevayler[4]: um sistema gerenciador de bancos de dados em memória principal, com cerca de duas mil linhas de código. Nesta dissertação são usadas como exemplos transformações aplicadas nesse sistema por Godil e Jacobsen para extração de aspectos destinados a modularizar os seguintes interesses transversais: gerenciamento de *snapshots*, censura de transações, replicação, *logging* persistente, *clock* de transações e concorrência [GJ05].

As transformações propostas se subdividem em dois grupos: movimentações de código e extração de métodos. Nas seções seguintes, descrevem-se esses grupos de transformações. Na Seção 3.1 apresentam-se as movimentações de código. Na Seção 3.2 apresentam-se as extrações de métodos. Por fim, na Seção 3.3 apresenta-se uma definição formal para cada tipo de transformação.

3.1 Movimentações de Código

Em geral, as transformações deste grupo têm como objetivo mover um código transversal para antes ou depois de um ponto que possa ser capturado por um descritor de

[1] `http://j-spider.sourceforge.net.`
[2] `http://www.jhotdraw.org.`
[3] `https://jaccounting.dev.java.net.`
[4] `http://www.prevayler.org.`

conjuntos de junção de AspectJ. Uma listagem das dezessete transformações incluídas neste grupo é apresentada na Tabela 3.1.

	Descrição
T1	Mover código para início de um método
T2	Mover código para final de um método
T3	Mover código para antes de um comando return
T4	Mover código para antes de uma chamada de método
T5	Mover código para depois de uma chamada de método
T6	Mover código para antes de uma chamada de construtor
T7	Mover código para depois de uma chamada de construtor
T8	Mover código para início de um bloco catch
T9	Mover código para final de um bloco catch
T10	Mover código para antes de uma leitura de campo
T11	Mover código para depois de uma leitura de campo
T12	Mover código para antes de uma escrita em campo
T13	Mover código para depois de uma escrita de campo
T14	Mover código para início do bloco de inicialização estática de uma classe
T15	Mover código para fim do bloco de inicialização estática de uma classe
T16	Agrupar código transversal em um único bloco de código
T17	Desagrupar código transversal entrelaçado em expressões

Tabela 3.1: Movimentações de código

Na Tabela 3.1, as duas primeiras transformações propõem a movimentação do código transversal para o início ou para o final de um método, de forma a facilitar sua modularização por meio de um conjunto de junção do tipo `execution`, associado a um adendo do tipo `before` ou `after returning`. Para exemplificar o uso da Transformação 1, mostra-se na Figura 3.1 um caso de aplicação dessa transformação na refatoração do sistema JAccounting. Nesse exemplo, o código transversal relativo à abertura de uma transação foi movido para o início do método. Com isso, habilita-se uma refatoração do tipo Extração no Início de Método [BCH+06]. De forma análoga, o uso da Transformação 2 habilita uma refatoração do tipo Extração no Final de Método [BCH+06].

A terceira transformação propõe a movimentação do código transversal para antes de um comando `return`, para também facilitar sua captura por meio de um adendo do tipo `after returning`. Entretanto, em métodos onde existem vários comandos `return`, não é possível realizar esse tipo de transformação em função da impossibilidade de distinguir entre esses pontos de junção.

As transformações de número 4 a 7 propõem a movimentação do código transversal para antes ou depois de uma chamada de método ou de construtor. O objetivo é viabilizar a modularização desse código por meio de um conjunto de junção do tipo `call`,

```
Account createInternalAccount(Session sess, ...) throws ... {
   Account account= null;
   net.sf.hibernate.Transaction tx= null;    // aspecto
   try {
      tx = sess.beginTransaction();          // aspecto
      ...
}
```

```
Account createInternalAccount(Session sess, ...) throws ... {
   net.sf.hibernate.Transaction tx= null;    // aspecto
   tx= sess.beginTransaction();              // aspecto
   Account account= null;
   try {
      ...
}
```

Figura 3.1: Código antes e após aplicação da Transformação 1 em método do sistema JAccounting. Comentários indicam código classificado como transversal.

associado a um adendo do tipo `before` ou `after returning`. Para exemplificar o uso da Transformação 5, mostra-se na Figura 3.2 um caso de aplicação dessa transformação na refatoração do sistema JAccounting. Nesse exemplo, o código transversal relativo à abertura de uma transação foi movido para após a chamada do método `openSession()`. Com isso, habilita-se uma refatoração do tipo Extração Após Chamada de Método [BCH+06]. De forma análoga, o uso da Transformação 7 habilita o mesmo tipo de refatoração, enquanto o uso das Transformações 4 e 6 habilitam refatorações do tipo Extração Antes de Chamada de Método [BCH+06].

As Transformações 8 e 9 determinam a movimentação de código para o início ou o final de um bloco tratador de exceções, visando sua modularização por meio de conjuntos de junção do tipo `handler`.

As transformações de número 10 a 13 propõem a movimentação do código transversal para antes ou depois de uma leitura ou atribuição em um campo de uma classe, viabilizando a modularização desse código por meio de conjuntos de junção do tipo `set` ou `get`. A aplicação dessas transformações ocorre de forma similar às transformações de número 4 a 7.

As Transformações 14 e 15 determinam a movimentação do código transversal para o início ou final do bloco de inicialização estática de uma classe, tendo como objetivo a sua extração mediante um conjunto de junção do tipo `staticinitialization`. A aplicação dessas transformações ocorre de forma similar às Transformações 1 e 2.

A transformação de número 16 propõe o agrupamento de blocos de códigos distintos em um único bloco, de forma a permitir a extração do bloco resultante para um único

```
protected String perform2() throws Exception {
   ...
   net.sf.hibernate.Transaction tx = null;
   Session sess = getSessionFactory().openSession();
   try {
      tx = sess.beginTransaction();          // aspecto
      if (formaction.equals("phone")) {
         ...
}
```

⇓

```
protected String perform2() throws Exception {
   ...
   net.sf.hibernate.Transaction tx = null;
   Session sess = getSessionFactory().openSession();
   tx = sess.beginTransaction();             // aspecto
   try {
      if (formaction.equals("phone")) {
         ...
}
```

Figura 3.2: Código antes e após aplicação da Transformação 5 em método do sistema JAccounting.

adendo. Após essa transformação, ao invés de duas, aplica-se apenas uma refatoração para extração de um conjunto de comandos que implementam um interesse transversal. Para exemplificar o uso da Transformação 16, mostra-se na Figura 3.3 um caso de aplicação dessa transformação na refatoração do sistema JHotDraw. Nesse exemplo, o último bloco de comandos relativos à funcionalidade de *undo* foi agrupado com o primeiro bloco. Com isso, faz-se necessária apenas uma refatoração do tipo Extração Antes de Chamada de Método para extração do interesse transversal. Essa transformação foi possível somente porque o comando que estava entre os dois blocos de comandos que implementam o interesse transversal não influencia na execução do segundo bloco.

Por fim, a Transformação 17 propõe a subdivisão de uma expressão contendo código transversal em duas subexpressões, uma com código transversal e outra sem. O objetivo é permitir a extração da expressão com código transversal para um aspecto. Na Figura 3.4 mostra-se um exemplo de aplicação da Transformação 17 na refatoração do sistema JHotDraw, onde a expressão de retorno do método foi particionada em duas subexpressões cujos resultados foram armazenados em variáveis locais. O objetivo foi restringir a uma única subexpressão o interesse relativo à funcionalidade de *undo* do sistema JHotDraw. Com isso, habilita-se uma refatoração do tipo Extração de Código *Pre-Return* [BCH+06].

```
public void execute() {
   ...
   if (selection != null) {
   // início do aspecto
      setUndoActivity(createUndoActivity());
      getUndoActivity().setAffectedFigures(
      (FigureEnumerator)selection.getData(StandardFigureSelection.TYPE));
      if (!getUndoActivity().getAffectedFigures().hasNextFigure()) {
         setUndoActivity(null);
         return;
      }
      Rectangle r = getBounds(getUndoActivity().getAffectedFigures());
   // fim do aspecto
      view().clearSelection();
   // início do aspecto
      FigureEnumeration fe =
         insertFigures(getUndoActivity().getAffectedFigures(),
         lastClick.x-r.x, lastClick.y-r.y);
      getUndoActivity().setAffectedFigures(fe);
   // fim do aspecto
      view().checkDamage();
   }
}
```

⇓

```
public void execute() {
   ...
   if (selection != null) {
   // início do aspecto
      setUndoActivity(createUndoActivity());
      getUndoActivity().setAffectedFigures(
      (FigureEnumerator)selection.getData(StandardFigureSelection.TYPE));
      if (!getUndoActivity().getAffectedFigures().hasNextFigure()) {
         setUndoActivity(null);
         return;
      }
      Rectangle r = getBounds(getUndoActivity().getAffectedFigures());
      FigureEnumeration fe =
         insertFigures(getUndoActivity().getAffectedFigures(),
         lastClick.x-r.x, lastClick.y-r.y);
      getUndoActivity().setAffectedFigures(fe);
   // fim do aspecto
      view().clearSelection();
      view().checkDamage();
   }
}
```

Figura 3.3: Código antes e após aplicação da Transformação 16 em método do sistema JHotDraw.

```
Tool createDragTracker(Figure f) {
   return new UndoableTool(
             new DragTracker(editor(), f));        // aspecto
}
```

⇓

```
Tool createDragTracker (Figure f) {
   DragTracker tmp1= new DragTracker(editor (), f);
   UndoableTool tmp2= new UndoableTool(tmp1);      // aspecto
   return tmp2;
}
```

Figura 3.4: Exemplo de aplicação da Transformação 17 em um método do sistema JHotDraw.

3.1.1 Pré-condições para Movimentações de Código

Recomenda-se que transformações envolvendo movimentações de código sejam as primeiras transformações investigadas quando inicia-se o processo de preparação do código fonte de um sistema para extração de aspectos. Contudo, nem sempre o código classificado como transversal pode ser movido conforme requerido pelas transformações propostas, pois existem pré-condições a serem atendidas para que uma movimentação de código possa ser realizada. Caso tais pré-condições não sejam respeitadas, o comportamento do sistema poderá ser alterado provocando erros ou fluxos de execução indesejados.

Para exemplificar as pré-condições para movimentação de código, supõe-se a seguir um exemplo da aplicação da Transformação 1 no método `foo()`, onde o comando `B`, originalmente executado após o comando `A`, é movido para o início do método:

`void foo() { A;B; ... }` ⇒ `void foo() { B;A; ... }`

Para que a Transformação 1 possa ser aplicada nesse exemplo, as seguintes pré-condições devem ser atendidas:

- `B` não deve alterar o estado que é acessado por `A`;
- `A` não deve alterar o estado que é requerido pela execução de `B`;
- `B` não deve alterar o fluxo de execução de forma a evitar a execução de `A` (por exemplo, ativando uma exceção ou executando um `return`);
- `A` e `B` não devem alterar o mesmo estado.

As pré-condições de cada uma das 17 movimentações de código catalogadas são detalhadas na Seção 3.3.

3.1.2 Transformações *Ad Hoc*

Quando as pré-condições para movimentação de código não são satisfeitas, o programador pode optar por um outro tipo de transformação manual classificada por Binkley et al. como transformações *ad hoc* [BCH+06]. Essas transformações têm como objetivo restaurar o estado e o fluxo de execução de um sistema após a aplicação de uma das transformações descritas nesta seção. Transformações *ad hoc* em geral são particulares para cada contexto de sistema e, por serem de difícil generalização, não são catalogadas nesta dissertação.

Para exemplificar o uso de transformações *ad hoc*, mostra-se na Figura 3.5 um exemplo de aplicação desse tipo de transformação na modularização do controle de transações no sistema JAccounting. Inicialmente, aplicou-se a Transformação 4, para mover o código de confirmação da transação (*commit*) para antes do método responsável pelo fechamento da sessão com o gerenciador de banco de dados. No entanto, pode-se observar que a aplicação da Transformação 4 altera o fluxo original de execução do sistema, pois, no código original, o `commit` é chamado apenas se não houver nenhuma exceção dentro do bloco `try`, mas, no código transformado, o `commit`, por estar agora no interior de um bloco `finally`, é invocado mesmo em caso de término da execução com uma exceção levantada. Para restabelecer o fluxo original de execução, o programador responsável pela transformação optou por introduzir uma variável local `tx` para indicar se o bloco `catch` foi executado (`tx==null`) ou não. Assim, o `commit` movimentado para o bloco `finally` somente deve ser chamado se `tx` for diferente de `null`. A aplicação dessa transformação *ad hoc* garante o comportamento original do sistema após a aplicação da Transformação 4.

3.2 Extrações de Métodos

Extração de método é uma refatoração orientada por objetos bastante conhecida, sendo aplicada para particionar um método extenso em métodos menores, o que pode facilitar o entendimento [FBB+99]. No caso específico de transformações para extração de interesses transversais, este tipo de reestruturação do código base é recomendado quando o código definido como transversal não se encontra nem antes nem depois de um ponto de junção, nem pode ser movido para tais posições, conforme sugerido pelas transformações de número 1 a 17. Nesse caso, a idéia consiste em extrair para um novo método parte do código não-transversal que ocorre antes ou após o código classificado como transversal. Com isso, cria-se um ponto de junção que pode ser capturado por meio de um conjunto de junção do tipo `call`.

```
try {
   ...
   tx.commit();                  // aspecto
}
catch (Exception e) {
   if (tx != null)
      tx.rollback();
   throw e;
}
finally {
   sess.close();
}
```

⇓

```
try {
   ...
}
catch (Exception e) {
   if (tx != null)
      tx.rollback();
   throw e;
}
finally {
   tx.commit();                  // aspecto
   sess.close();
}
```

⇓

```
try {
   ...
}
catch (Exception e) {
   if (tx != null) {
      tx.rollback();
      tx= null;
   }
   throw e;
}
finally {
   if (tx != null)
      tx.commit();               // aspecto
   sess.close();
}
```

Figura 3.5: Exemplo de aplicação da Transformação 4, seguida da aplicação de uma transformação *ad hoc*, no sistema JAccounting.

A Tabela 3.2 descreve as sete transformações incluídas nesse grupo.

	Descrição
T18	Extrair método com parte do código que antecede um código transversal
T19	Extrair método com parte do código que sucede um código transversal
T20	Extrair método com código que antecede e sucede um código transversal
T21	Extrair método contendo bloco de comandos `try-catch-finally`
T22	Extrair método contendo bloco de comandos `synchronized`
T23	Extrair método quando `withincode` não é capaz de definir ponto junção
T24	Extrair método com código usado em atribuição a variável local

Tabela 3.2: Extração de Métodos

As Transformações 18 e 19 sugerem converter em método parte do código que antecede ou sucede um código identificado como transversal. O código a ser extraído para um novo método pode ser composto por um bloco de comandos, um comando ou parte de uma expressão. Mostra-se na Figura 3.6 um exemplo de aplicação da Transformação 19 no sistema JSpider onde o valor atribuído a uma variável local, ou seja, apenas parte de uma expressão, foi extraído para um novo método. Após essa transformação, pode-se criar um conjunto de junção que capture a chamada do novo método associado a um adendo do tipo `before`.

Para exemplificar a extração de método com código correspondente a um bloco de comandos, mostra-se na Figura 3.7 um exemplo de aplicação da Transformação 19 no sistema JSpider, onde o bloco `else` é extraído para um novo método. Após essa transformação, pode-se criar um conjunto de junção que capture a chamada do novo método associado a um adendo do tipo `around` que, quando satisfeita a condição da cláusula `if`, executa o interesse transversal. Caso contrário, o método extraído é executado utilizando-se a cláusula `proceed`.

A Transformação 20 sugere converter em método um código presente entre dois códigos identificados como transversais, ou seja, um código que sucede o primeiro código transversal e, ao mesmo tempo, antecede o segundo. Após essa transformação, pode-se criar um conjunto de junção que capture a chamada do novo método e associá-lo a dois adendos: um do tipo `before` correspondente ao primeiro código identificado como transversal, e outro do tipo `after`, correspondente ao segundo.

A Transformação 21 recomenda a extração de um método contendo os blocos de comandos especificados em uma estrutura `try-catch-finally`. O objetivo no caso é permitir a modularização do código para tratamento de exceções por meio de um adendo do tipo `around`. Mostra-se na Figura 3.8 um exemplo de aplicação da Transformação 21 no sistema Prevayler para extração de uma estrutura `try-catch-finally` para um

```
public Throttle createThrottle(PropertySet props) {
  ...
  if (interval < INTERVAL_MIN) {
    log.warn("Throttle interval < " + INTERVAL_MIN +       // aspecto
      " ms is dangereous - set to minimum allowed of " +   // aspecto
      INTERVAL_MIN + " ms");                               // aspecto
    interval = INTERVAL_MIN;
  } ...
}
```

⇓

```
public Throttle createThrottle(PropertySet props) {
  ...
  if (interval < INTERVAL_MIN) {
    log.warn("Throttle interval < " + INTERVAL_MIN +       // aspecto
    " ms is dangereous - set to minimum allowed of " +     // aspecto
    INTERVAL_MIN + " ms");                                 // aspecto
    interval = setToMininum();
  } ...
}

private int setToMininum() {                          // método extraído
  return INTERVAL_MIN;
}
```

Figura 3.6: Exemplo de aplicação da Transformação 19 em parte de uma expressão no sistema JSpider.

novo método. Após a aplicação dessa transformação, habilita-se uma refatoração do tipo Extração de Tratamento de Exceção [BCH+06].

De forma similar, a Transformação 22 recomenda a extração de um método contendo o bloco de comandos especificado em um comando `synchronized`, a fim de auxiliar na modularização do código transversal de sincronização.

A Transformação 23 sugere extrair um método quando o designador `withincode` não é suficiente para definição de um ponto de junção. Essa transformação sugere então extrair o comando adjacente ao código correspondente ao interesse transversal que se deseja modularizar. Mostra-se na Figura 3.9 um exemplo de aplicação da Transformação 23 no sistema Prevayler para evitar a multiplicidade de pontos de junção correspondentes às chamadas do método `readObject()`.

A Transformação 24 é útil quando um aspecto precisa obter o valor de uma variável local do código base. Como AspectJ não classifica como pontos de junção atribuições a variáveis locais, a solução proposta consiste em extrair um método contendo código correspondente ao lado direito de comandos de atribuição a uma determinada variável. Um adendo do tipo `around` associado a chamadas desse método pode ser então usado

```
public Resource registerURL(URL url) {
   ...
   if (site == null) {
      log.warn("unable to register resource " + url +
               " because Site object wasn't found");    // aspecto
   }
   else {
      resource = new ResourceInternal(storage, id, site, url,
                                      new Date(), folder);
      spi.create(id, resource);
   }
   ...
}
```

```
public Resource registerURL(URL url) {
   ...
   if (site == null) {
      log.warn("unable to register resource " + url +
               " because Site object wasn't found");    // aspecto
   } else
      resource = addSite(url, folder, id, resource, site);
   ...
}
                                                        // método extraído
private ResourceInternal addSite(URL url, FolderInternal folder, int id,
        ResourceInternal resource, SiteInternal site) {
   resource = new ResourceInternal(storage, id, site, url,
                                   new Date(), folder);
   spi.create(id, resource);
   return resource;
}
```

Figura 3.7: Exemplo de aplicação da Transformação 19 em um bloco de comandos no sistema JSpider.

para capturar o valor da variável. Mostra-se na Figura 3.10, um exemplo de aplicação da Transformação 24 no sistema JSpider onde a atribuição a uma variável local é extraída para um método para que posteriormente possa ser utilizada pelo registro de *log* encapsulado pelo aspecto.

3.2.1 Limitações para Extração de Métodos

O código a ser extraído para um novo método pode apresentar características que dificultam a aplicação da técnica de extração de métodos, tais como: conter múltiplas atribuições a variáveis locais ou conter múltiplas saídas. No caso de múltiplas atribuições

```
void log(Transaction tx, Date execTime, Turn myTurn) {
   ...
   try {                                                  // aspecto
      myTurn.start();
      log.add(new TransactionTimestamp(tx, exeTime));
    }                                                     // aspecto
    catch (Exception e) { ... }                           // aspecto
    finally { ... }                                       // aspecto
    ...
}
```

⇓

```
void log(Transaction tx, Date execTime, Turn myTurn) {
   ...
   logHelper(tx, executionTime, myTurn);
   ...
}
                                                          // método extraído
void logHelper(Transaction tx, Date execTime, Turn myTurn) {
   try {                                                  // aspecto
      myTurn.start();
      log.add(new TransactionTimestamp(tx, execTime));
   }                                                      // aspecto
   catch (Exception e) { ... }                            // aspecto
   finally { ... }                                        // aspecto
}
```

Figura 3.8: Exemplo de aplicação da Transformação 21 no sistema Prevayler.

a variáveis locais, uma solução recomendada é a aplicação da refatoração Substituição de Método por Método de Objeto [FBB+99]. No caso de múltiplas saídas (por exemplo, por meio de um comando `return` interno ao fragmento de código a ser extraído), a solução pode ser o retorno de um *flag* que indica uma saída normal ou uma saída por meio de um comando específico. Normalmente, esses casos mais complexos de extração de métodos não são automatizados por ferramentas de apoio a refatoração, como, por exemplo, o *plugin* de refatoração do ambiente Eclipse.

Quando não é possível aplicar uma extração de método, uma outra possível solução é a criação de métodos vazios (*hook methods*) [KAB07]. O único propósito para criação de um método como esse é expor um ponto de junção. No entanto, a criação de métodos vazios não é recomendada e deve ser utilizada somente como última alternativa para extração de métodos.

```
void receiveTransactionFromServer()
                throws IOException,ClassNotFoundException {
   Object transactionCandidate= _fromServer.readObject();
   ...
   Date timestamp = (Date) _fromServer.readObject();
   _clock.advanceTo(timestamp);                        // aspecto
   ....
}
```

⇓

```
void receiveTransactionFromServer()
                throws IOException,ClassNotFoundException {
   Object transactionCandidate= _fromServer.readObject();
   ...
   Date timestamp = (Date) getTimeStamp();
   _clock.advanceTo(timestamp);                        // aspecto
   ...
}
                                                       // método extraído
Object getTimeStamp() throws IOException, ClassNotFoundException {
   return _fromServer.readObject();
}
```

Figura 3.9: Exemplo de aplicação da Transformação 23 no sistema Prevayler.

3.3 Definições Formais

Esta seção apresenta definições formais para as 24 transformações propostas nesta dissertação. A notação utilizada para formalizar as transformações foi baseada no conceito de regras "*push*" proposto por Harman et al. para construção de *slices* amorfos de programas (*amorphous slices*) [HHM+04]. Um *slice* é uma decomposição de um programa em partes menores que correspondem a um subconjunto específico do seu comportamento [Wei84]. Um *slice* amorfo representa uma decomposição de um programa com mais flexibilidade em relação a restrições de sintaxe. Tradicionalmente, na criação de *slices* com preservação de sintaxe, apenas a retirada de comandos pode ser aplicada no código base de uma aplicação. Já na construção de *slices* amorfos, outros tipos de transformações que reduzem o tamanho do *slice* são permitidos. No trabalho de Harman et al., regras "*push*" são transformações definidas para reduzir interdependências entre comandos.

Seguindo essa notação, propõe-se nesta dissertação que uma transformação orientada por objetos é descrita por uma regra com o seguinte formato:

$$\frac{A}{B \Rightarrow C}$$

onde A estipula as pré-condições requeridas pela transformação e B e C são fragmentos de

```
public SpiderContextImpl( ... ) {
   ...
   String proxyHost = DEFAULT_PROXY;
   ...
   System.setProperty("http.proxyHost", proxyHost);
   log.info ("Using proxy " + proxyHost);          // aspecto
   useProxyAuthentication =
      proxyProps.getBoolean(ConfigConstants.CONFIG_PROXY_AUTHENTICATE,
                            false);
   ...
}
```

⇓

```
public SpiderContextImpl( ... ) {
   ...
   String proxyHost = getDefaultProxy();
   ...
   System.setProperty("http.proxyHost", proxyHost);
   log.info ("Using proxy " + proxyHost);          // aspecto
   useProxyAuthentication =
      proxyProps.getBoolean(ConfigConstants.CONFIG_PROXY_AUTHENTICATE,
                            false);
   ...
}
private String getDefaultProxy() {                 // método extraído
   return DEFAULT_PROXY;
}
```

Figura 3.10: Exemplo de aplicação da Transformação 24 no sistema JSpider.

código. Essa regra deve ser interpretada da seguinte forma: se a condição A for atendida, então qualquer fragmento de código no padrão B pode ser transformado em um fragmento no padrão C.

Considerando que `st` representa um bloco de comandos, as definições propostas utilizam as seguintes funções auxiliares:

- **def**(`st`): retorna o conjunto de variáveis que podem ser definidas pela execução de `st` (incluindo variáveis locais, variáveis armazenadas no *heap* e variáveis estáticas).
- **ref**(`st`): retorna o conjunto de variáveis que podem ser referenciadas pela execução de `st`.
- **exit**(`st`): retorna verdadeiro se `st` inclui um comando `return` ou `throw`.
- **ret**(`st`): retorna verdadeiro se `st` inclui um comando `return`.

- **call**(`m`, `st`): retorna verdadeiro se `st` inclui uma chamada do método (ou construtor) `m`.
- **get**(`f`, `st`): retorna verdadeiro se `st` lê o valor do atributo `f` (`st` deve representar um único comando).
- **set**(`f`, `st`): retorna verdadeiro se `st` modifica o valor do atributo `f` (`st` deve representar um único comando).

Na definição proposta nesta seção, concatenação de dois blocos de comandos é representada pela simples justaposição de suas descrições. Por exemplo, **ret**($\texttt{st}_1$ $\texttt{st}_2$) retorna verdadeiro se $\texttt{st}_1$ ou $\texttt{st}_2$ incluir um comando `return`. Além disso, `st`[`exp`] expõe a expressão `exp` que é parte do comando `st`. Finalmente, as definições referentes às transformações de movimentações de código dependem da seguinte função:

$$\begin{aligned}\textbf{chg}(\texttt{st}_1, \texttt{st}_2) \equiv\ & \textbf{def}(\texttt{st}_1) \cap \textbf{ref}(\texttt{st}_2) = \emptyset \ \wedge\ \textbf{ref}(\texttt{st}_1) \cap \textbf{def}(\texttt{st}_2) = \emptyset \ \wedge \\ & \textbf{def}(\texttt{st}_1) \cap \textbf{def}(\texttt{st}_2) = \emptyset \ \wedge\ !\textbf{exit}(\texttt{st}_2)\end{aligned}$$

Essa função define quando a ordem dos blocos de comandos $\texttt{st}_1$ e $\texttt{st}_2$ pode ser trocada. A troca é permitida quando: $\texttt{st}_1$ não define nenhuma variável que é referenciada por $\texttt{st}_2$; $\texttt{st}_1$ não referencia nenhuma variável definida por $\texttt{st}_2$; $\texttt{st}_1$ e $\texttt{st}_2$ não definem nenhuma variável em comum; e $\texttt{st}_2$ não afeta a execução do programa impedindo a execução de $\texttt{st}_1$ (por exemplo, ativando uma exceção ou executando um comando `return`).

As Tabelas 3.11 e 3.12 apresentam as definições para as transformações correspondentes a movimentações de código (T1 a T17).

(T1) $$\frac{\mathbf{chg}(\mathtt{st}, \mathtt{cc})}{\mathtt{mtd}\,\{\mathtt{st}\ \mathtt{cc}\ \mathtt{st}'\} \Rightarrow \mathtt{mtd}\,\{\mathtt{cc}\ \mathtt{st}\ \mathtt{st}'\}}$$

(T2) $$\frac{\mathbf{chg}(\mathtt{cc}, \mathtt{st})}{\mathtt{mtd}\,\{\mathtt{st}'\ \mathtt{cc}\ \mathtt{st}\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st}'\ \mathtt{st}\ \mathtt{cc}\}}$$

(T3) $$\frac{\mathbf{chg}(\mathtt{cc}, \mathtt{st}_2),\ \mathbf{!ret}(\mathtt{st}_1\ \mathtt{st}_2\ \mathtt{st}_3)}{\mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{cc}\ \mathtt{st}_2\ \mathtt{return}\ \mathtt{st}_3\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{st}_2\ \mathtt{cc}\ \mathtt{return}\ \mathtt{st}_3\}}$$

(T4) $$\frac{\mathtt{st}_3 = \mathtt{m(p)}\ \mathtt{st}'_3,\ \mathbf{chg}(\mathtt{cc}, \mathtt{st}_2),\ \mathbf{!call}(\mathtt{m}, \mathtt{st}_1\ \mathtt{st}_2\ \mathtt{st}'_3)}{\mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{cc}\ \mathtt{st}_2\ \mathtt{st}_3\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{st}_2\ \mathtt{cc}\ \mathtt{st}_3\}}$$

(T5) $$\frac{\mathtt{st}_2 = \mathtt{st}'_2\ \mathtt{m(p)},\ \mathbf{chg}(\mathtt{cc}, \mathtt{st}_2),\ \mathbf{!call}(\mathtt{m}, \mathtt{st}_1\ \mathtt{st}'_2\ \mathtt{st}_3)}{\mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{cc}\ \mathtt{st}_2\ \mathtt{st}_3\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{st}_2\ \mathtt{cc}\ \mathtt{st}_3\}}$$

(T6) $$\frac{\mathtt{st}_3 = \mathtt{new\,T(p)}\ \mathtt{st}'_3,\ \mathbf{chg}(\mathtt{cc}, \mathtt{st}_2),\ \mathbf{!call}(\mathtt{new\,T}, \mathtt{st}_1\ \mathtt{st}_2\ \mathtt{st}'_3)}{\mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{cc}\ \mathtt{st}_2\ \mathtt{st}_3\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{st}_2\ \mathtt{cc}\ \mathtt{st}_3\}}$$

(T7) $$\frac{\mathtt{st}_2 = \mathtt{st}'_2\ \mathtt{new\,T(p)},\ \mathbf{chg}(\mathtt{cc}, \mathtt{st}_2),\ \mathbf{!call}(\mathtt{new\,T}, \mathtt{st}_1\ \mathtt{st}'_2\ \mathtt{st}_3)}{\mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{cc}\ \mathtt{st}_2\ \mathtt{st}_3\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{st}_2\ \mathtt{cc}\ \mathtt{st}_3\}}$$

(T8) $$\frac{\mathbf{chg}(\mathtt{st}, \mathtt{cc})}{\mathtt{catch(e)\{st\ cc\ st'\}} \Rightarrow \mathtt{catch(e)\{cc\ st\ st'\}}}$$

(T9) $$\frac{\mathbf{chg}(\mathtt{cc}, \mathtt{st})}{\mathtt{catch(e)\{st'\ cc\ st\}} \Rightarrow \mathtt{catch(e)\{st'\ st\ cc\}}}$$

onde `cc` são comandos que implementam interesses transversais; `mtd` é uma declaração de método; `m` é um nome de método; `p` é uma lista de parâmetros; `T` é um nome de classe.

Figura 3.11: Transformações T1-T9

A Tabela 3.13 apresenta as definições para as transformações correspondentes a extrações de métodos (T18 a T24). As regras para definição dessas transformações utilizam uma chave que delimita o código que deve ser extraído para o novo método, conforme ilustrado a seguir:

$$\mathtt{st}\ \underbrace{st_1\ st_2\ \ldots st_n}_{\mathtt{m}}\ \mathtt{st}' \Rightarrow \mathtt{st\ m(p)\ st}'$$

Essa notação indica que a transformação requer primeiro a extração dos blocos de

(T10) $$\frac{\mathtt{st_3} = \mathtt{st'_3}\,\mathtt{st''_3},\ \mathbf{get}(\mathtt{f}, \mathtt{st'_3}),\ \mathbf{!get}(\mathtt{f}, \mathtt{st_1}\ \mathtt{st_2}\ \mathtt{st''_3}),\ \mathbf{chg}(\mathtt{cc}, \mathtt{st_2})}{\mathtt{mtd}\,\{\mathtt{st_1}\ \mathtt{cc}\ \mathtt{st_2}\ \mathtt{st_3}\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st_1}\ \mathtt{st_2}\ \mathtt{cc}\ \mathtt{st_3}\}}$$

(T11) $$\frac{\mathtt{st_2} = \mathtt{st'_2}\,\mathtt{st''_2},\ \mathbf{get}(\mathtt{f}, \mathtt{st''_2}),\ \mathbf{!get}(\mathtt{f}, \mathtt{st_1}\ \mathtt{st'_2}\ \mathtt{st_3}),\ \mathbf{chg}(\mathtt{cc}, \mathtt{st_2})}{\mathtt{mtd}\,\{\mathtt{st_1}\ \mathtt{cc}\ \mathtt{st_2}\ \mathtt{st_3}\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st_1}\ \mathtt{st_2}\ \mathtt{cc}\ \mathtt{st_3}\}}$$

(T12) $$\frac{\mathtt{st_3} = \mathtt{st'_3}\,\mathtt{st''_3},\ \mathbf{set}(\mathtt{f}, \mathtt{st'_3}),\ \mathbf{!set}(\mathtt{f}, \mathtt{st_1}\ \mathtt{st_2}\ \mathtt{st''_3}),\ \mathbf{chg}(\mathtt{cc}, \mathtt{st_2})}{\mathtt{mtd}\,\{\mathtt{st_1}\ \mathtt{cc}\ \mathtt{st_2}\ \mathtt{st_3}\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st_1}\ \mathtt{st_2}\ \mathtt{cc}\ \mathtt{st_3}\}}$$

(T13) $$\frac{\mathtt{st_2} = \mathtt{st'_2}\,\mathtt{st''_2},\ \mathbf{set}(\mathtt{f}, \mathtt{st''_2}),\ \mathbf{!set}(\mathtt{f}, \mathtt{st_1}\ \mathtt{st'_2}\ \mathtt{st_3}),\ \mathbf{chg}(\mathtt{cc}, \mathtt{st_2})}{\mathtt{mtd}\,\{\mathtt{st_1}\ \mathtt{cc}\ \mathtt{st_2}\ \mathtt{st_3}\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st_1}\ \mathtt{st_2}\ \mathtt{cc}\ \mathtt{st_3}\}}$$

(T14) $$\frac{\mathbf{chg}(\mathtt{st}, \mathtt{cc})}{\mathtt{static}\,\{\mathtt{st}\ \mathtt{cc}\ \mathtt{st'}\} \Rightarrow \mathtt{static}\,\{\mathtt{cc}\ \mathtt{st}\ \mathtt{st'}\}}$$

(T15) $$\frac{\mathbf{chg}(\mathtt{cc}, \mathtt{st})}{\mathtt{static}\,\{\mathtt{st'}\ \mathtt{cc}\ \mathtt{st}\} \Rightarrow \mathtt{static}\,\{\mathtt{st'}\ \mathtt{st}\ \mathtt{cc}\}}$$

(T16) $$\frac{\mathbf{chg}(\mathtt{cc_1}, \mathtt{st_2})}{\mathtt{st_1}\ \mathtt{cc_1}\ \mathtt{st_2}\ \mathtt{cc_2}\ \mathtt{st_3} \Rightarrow \mathtt{st_1}\ \mathtt{st_2}\ \mathtt{cc_1}\ \mathtt{cc_2}\ \mathtt{st_3}}$$

$$\frac{\mathbf{chg}(\mathtt{st_2}, \mathtt{cc_2})}{\mathtt{st_1}\ \mathtt{cc_1}\ \mathtt{st_2}\ \mathtt{cc_2}\ \mathtt{st_3} \Rightarrow \mathtt{st_1}\ \mathtt{cc_1}\ \mathtt{cc_2}\ \mathtt{st_2}\ \mathtt{st_3}}$$

(T17) $$\frac{\mathtt{cc\ instanceof\ type(exp)}}{\mathtt{st[cc(exp)]} \Rightarrow \mathtt{t_1{=}exp;\ t_2{=}cc(t_1);\ st[t_2]}}$$

onde `cc` são comandos que implementam interesses transversais; `mtd` é uma declaração de método; `f` é um atributo de classe; `e` é uma expressão; $\mathtt{t_1}$ e $\mathtt{t_2}$ são variáveis locais.

Figura 3.12: Transformações T10-T17

comandos $\mathtt{st_1}\ \mathtt{st_2}\ \ldots \mathtt{st_n}$ para o método `m`. O método extraído pode então ser referenciado no lado direito da regra. Como é usual em refatorações para extração de métodos, o código extraído deve corresponder a um bloco de código com apenas uma entrada e uma saída.

(T18) $\texttt{st}_1 \underbrace{st}_{\texttt{m}} \texttt{ cc } \texttt{st}_2 \Rightarrow \texttt{st}_1 \texttt{ m(p) cc } \texttt{st}_2$

$\texttt{st}[\underbrace{exp}_{\texttt{m}}] \texttt{ cc st}' \Rightarrow \texttt{st}[\texttt{m(p)}] \texttt{ cc st}'$

(T19) $\texttt{st}_1 \texttt{ cc } \underbrace{st}_{\texttt{m}} \texttt{ st}_2 \Rightarrow \texttt{st}_1 \texttt{ cc m(p) st}_2$

$\texttt{st}' \texttt{ cc st}[\underbrace{exp}_{\texttt{m}}] \Rightarrow \texttt{st}' \texttt{ cc st}[\texttt{m(p)}]$

(T20) $\texttt{st}_1 \underbrace{st}_{\texttt{m}} \texttt{ cc } \underbrace{st'}_{\texttt{m}} \texttt{ st}_2 \Rightarrow \texttt{st}_1 \texttt{ cc m(p) st}_2$

$\texttt{st}_1 \underbrace{st}_{\texttt{m}} \texttt{ cc } \underbrace{st'}_{\texttt{m}} \texttt{ st}_2 \Rightarrow \texttt{st}_1 \texttt{ m(p) cc st}_2$

(T21) $\texttt{st}_1 \underbrace{\texttt{try}\{st\} \texttt{ catch}\{cc_1\} \texttt{ finally}\{cc_2\}}_{\texttt{m}} \texttt{ st}_2 \Rightarrow \texttt{st}_1 \texttt{ m(p) st}_2$

(T22) $\texttt{st}_1 \underbrace{\texttt{synchronized}\{st\}}_{\texttt{m}} \texttt{ st}_2 \Rightarrow \texttt{st}_1 \texttt{ m(p) st}_2$

(T23)
$$\frac{\textbf{call}(\texttt{m(p)}, \texttt{st}_1\, \texttt{st}_2)}{\texttt{mtd } \{\texttt{st}_1 \underbrace{m(p)}_{\texttt{m}'} \texttt{ cc st}_2\} \Rightarrow \texttt{mtd } \{\texttt{st}_1 \texttt{ m}'\texttt{(p}')\texttt{ cc st}_2\}}$$

$$\frac{\textbf{call}(\texttt{m(p)}, \texttt{st}_1\, \texttt{st}_2)}{\texttt{mtd } \{\texttt{st}_1 \texttt{ cc } \underbrace{m(p)}_{\texttt{m}'} \texttt{ st}_2\} \Rightarrow \texttt{mtd } \{\texttt{st}_1 \texttt{ cc m}'\texttt{(p}')\texttt{ st}_2\}}$$

(T24) $\texttt{st}_1 \texttt{ v} = \underbrace{st}_{\texttt{m}} \texttt{ st}_2 \Rightarrow \texttt{st}_1 \texttt{ v} = \texttt{m(p) st}_2$

onde `cc` são comandos que implementam interesses transversais; `mtd` é uma declaração de método; `m` ou `m'` correspondem ao nome do método extraído; `v` é uma variável local. Blocos de comandos extraídos para um novo método devem ter apenas uma entrada e uma saída.

Figura 3.13: Transformações T18-T24

3.4 Considerações Finais

Neste capítulo foram apresentados dois conjuntos de transformações orientadas por objetos para extração de aspectos: Movimentações de Código e Extrações de Métodos. As

transformações propostas foram descritas e algumas delas exemplificadas com aplicações em quatro sistemas reais: JSpider, JHotDraw, JAccounting e Prevayler. Foram apresentadas também pré-condições e algumas limitações para aplicação das transformações propostas. Por fim, foi apresentada uma definição formal para cada uma das transformações.

Pelos exemplos apresentados, verifica-se a importância da aplicação de transformações no código orientado por objetos de sistemas reais para viabilizar o processo de extração de aspectos. De acordo com o tamanho e a complexidade de um sistema a ser refatorado para extração de interesses transversais, a atividade de preparação do código através da aplicação de transformações orientadas por objetos pode se tornar trabalhosa e sujeita a erros.

Capítulo 4

Ferramenta para Mapeamento de Transformações

Neste capítulo, apresenta-se uma ferramenta que auxilia o programador a identificar transformações necessárias em um sistema legado antes de se iniciar um processo de refatoração de interesses transversais através de aspectos. Na Seção 4.1, apresentam-se os objetivos da ferramenta proposta. Na Seção 4.2, são detalhadas as principais funcionalidades da ferramenta e, na Seção 4.3, são descritos aspectos de implementação. Por fim, na Seção 4.4, relacionam-se casos onde a ferramenta pode ser aplicada e suas restrições.

4.1 Objetivos

As transformações apresentadas no Capítulo 3 desta dissertação, mesmo tendo sido formalizadas, são de difícil automatização e devem ser aplicadas manualmente pelo programador, pois dependem do conhecimento das regras de negócio do sistema em questão e de uma análise semântica do código a ser transformado para que o comportamento original do sistema seja preservado. Essa dificuldade está relacionada principalmente às movimentações de código, pois essas traformações requerem uma análise global de dependência de dados e controle do sistema. Para realizar essa análise, é necessário que seja aplicada a função de dependência descrita na Seção 3.3, que define quando a ordem de dois blocos de comandos pode ser trocada.

Antes que as transformações sejam aplicadas, é preciso identificar onde elas são necessárias e analisar se o esforço necessário para realizá-las é justificado pelos benefícios que a modularização de um determinado interesse transversal pode trazer para o sistema. Para auxiliar o programador no processo de identificação de transformações requeridas em aplicações Java para viabilizar refatorações de interesses transversais, utilizando-se

AspectJ, propõe-se nesta dissertação uma ferramenta que seja capaz de mapear parte dessas transformações. A ferramenta proposta foi implementada como um *plugin* do ambiente de desenvolvimento Eclipse.

4.2 Principais Funcionalidades

A ferramenta proposta é uma extensão do *plugin* *ConcernMapper*, proposto por Robillard para mapeamento de interesses transversais em aplicações Java [RWW05]. O *ConcernMapper* é uma ferramenta que permite organizar atributos e métodos correspondentes a interesses transversais em estruturas denominadas *Concerns*. Essa organização permite uma visão modularizada dos interesses transversais de um sistema sem que ele sofra qualquer tipo de alteração estrutural ou comportamental. Após o mapeamento ter sido realizado, é possível identificar, para os atributos e métodos que compõem um determinado *Concern*, todas as suas ocorrências nas classes do sistema. A utilização do *ConcernMapper* facilita a manutenção e evolução de interesses transversais. Além disso, o mapeamento de um determinado interesse transversal facilita a sua futura modularização por meio de aspectos.

Mostra-se na Figura 4.1 o mapeamento da funcionalidade de *logging* no sistema JSpider através do *ConcernMapper*. A estrutura apresentada inclui os elementos que compõem essa funcionalidade (parte esquerda da figura) e, na visão *Search* do Eclipse, são listadas todas as ocorrências do método `debug` no sistema (parte direita da figura).

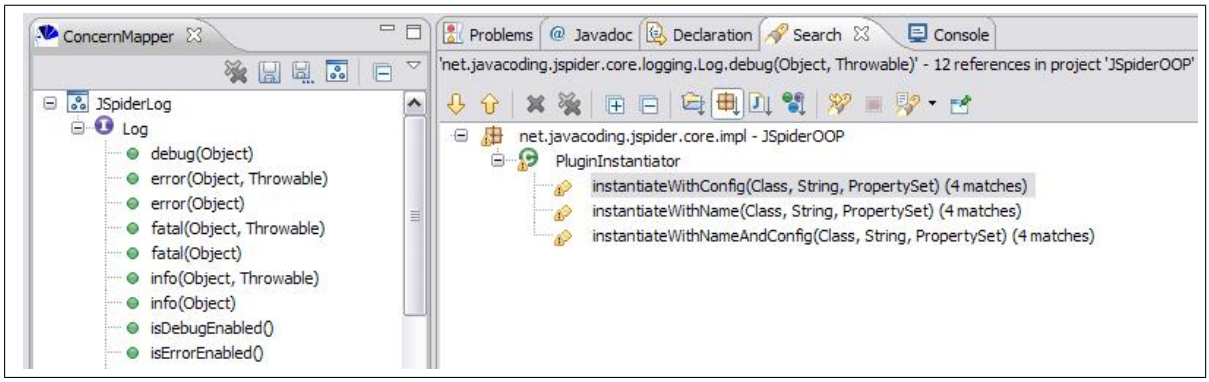

Figura 4.1: Mapeamento, através do *ConcernMapper*, da funcionalidade de *logging* no sistema JSpider.

Propõe-se neste trabalho um *plugin* para o Eclipse denominado *TransformationMapper*. Essa ferramenta, que estende o *ConcernMapper*, permite ao programador, após mapear os atributos e métodos que constituem interesses transversais, localizar as ocorrências desses elementos em um sistema e, para cada ocorrência, verificar se existe ou não a

necessidade de se aplicar uma das transformações orientadas por objetos descritas no Capítulo 3.

Mostra-se na Figura 4.2 o mapeamento da funcionalidade de *logging* no sistema JSpider através do *TransformationMapper*, incluindo os elementos que compõem essa funcionalidade e suas respectivas ocorrências no sistema. Mostra-se também, para cada ocorrência de um elemento, a necessidade ou não de uma transformação orientada por objetos. Nesse exemplo, é apresentada a ocorrência do método `Log.debug(Object)` em métodos da classe `EventDispatcherImpl`. No construtor dessa classe, por exemplo, foram identificadas cinco ocorrências do método mapeado como parte do interesse transversal, onde duas estão prontas para refatoração (linhas 53 e 78 da classe) e três requerem transformações orientadas por objetos antes de serem refatoradas (linhas 63, 64 e 65).

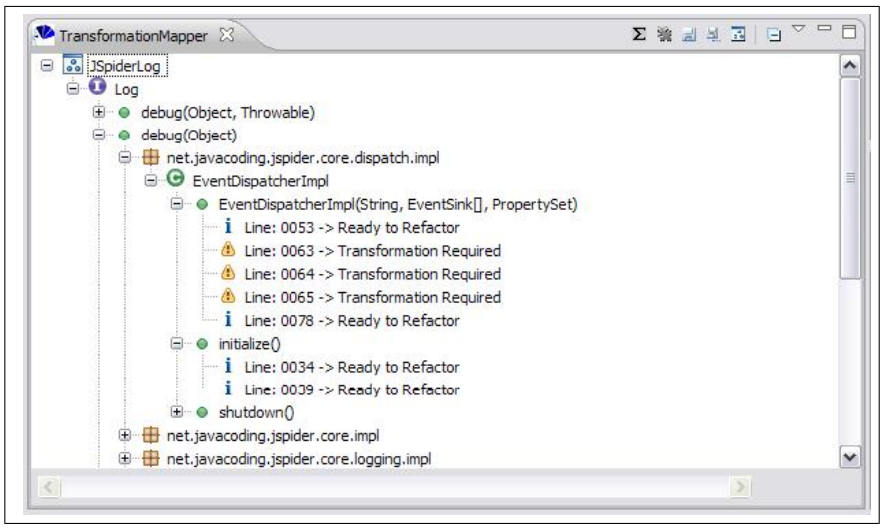

Figura 4.2: Mapeamento, através do *TransformationMapper*, da funcionalidade de *logging* no sistema JSpider.

Na implementação do *TransformationMapper* procurou-se manter as mesmas funcionalidades existentes no *ConcernMapper*. Para que um interesse transversal seja mapeado, é necessário que um novo *Concern* seja criado por meio do botão "*Create a New Concern*". Após essa operação, basta que os elementos correspondentes ao interesse sejam arrastados a partir da visão *Package Explorer* do Eclipse para o nó correspondente ao *Concern* criado. A cada mapeamento realizado, o *TransformationMapper* automaticamente insere na árvore todas as ocorrências do elemento mapeado e já informa se existe ou não a necessidade de uma transformação orientada por objetos. Conforme pode ser observado na Figura 4.3, o *TransformationMapper* mantém vínculos entre a árvore de *concerns* e o código-fonte de tais *concerns* no sistema, o que permite uma navegação interativa entre essas visões do Eclipse. Por exemplo, clicando-se em qualquer elemento da árvore de *con-*

cerns, o código-fonte correspondente é aberto. Os vínculos estabelecidos fazem com que todos os elementos que foram mapeados para um *concern* fiquem destacados em negrito na árvore do projeto exibida pela visão *Package Explorer*.

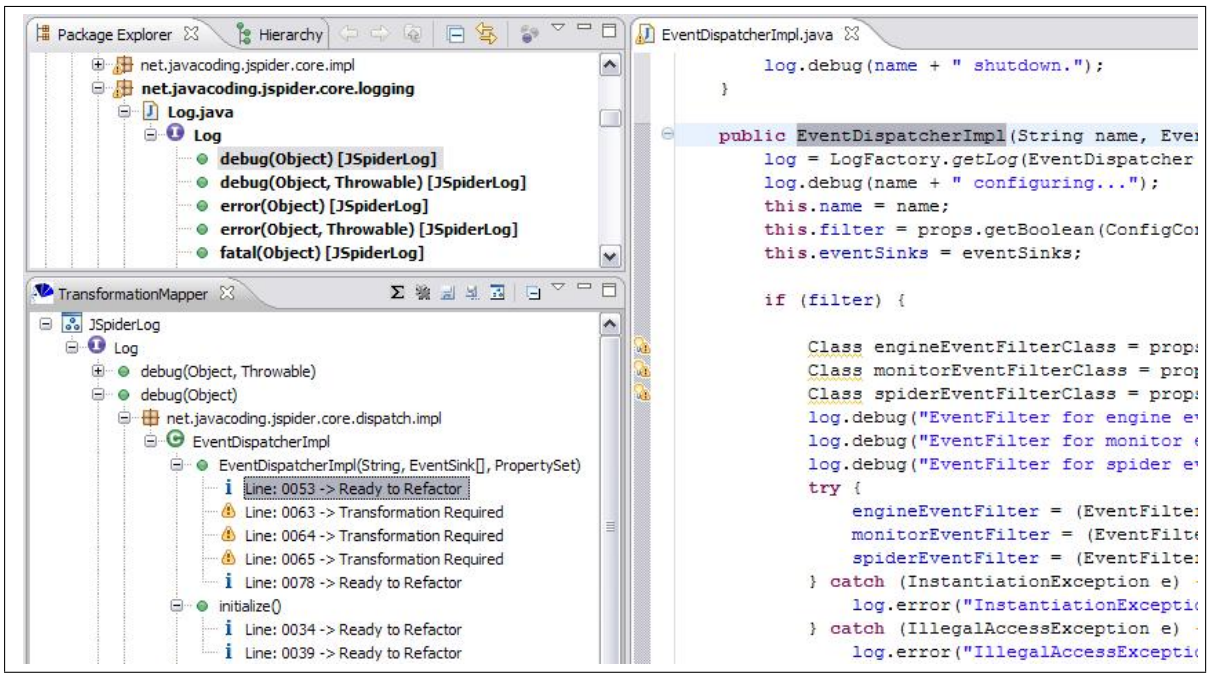

Figura 4.3: Vínculos estabelecido pelo *TransformationMapper* entre diferentes visões do Eclipse.

Com o objetivo de apresentar informações sobre a quantidade de transformações necessárias para que interesses transversais sejam refatorados, foi implementada a funcionalidade *Concern Statistics*, que é executada através do primeiro botão da barra de ferramentas do *plugin*. Mostra-se na Figura 4.4 os resultados obtidos na análise da funcionalidade de *logging* do sistema JSpider.

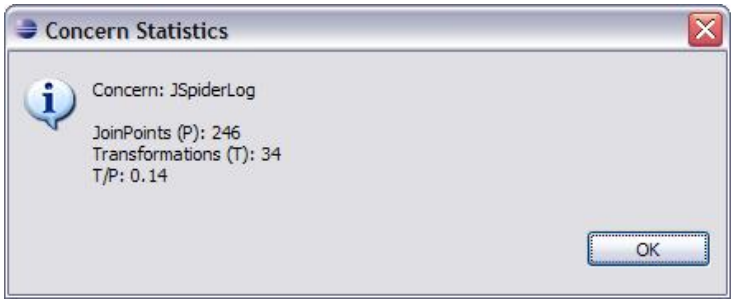

Figura 4.4: Estatísticas obtidas após mapeamento da funcionalidade de *logging* do sistema JSpider.

4.3 Implementação

Na implementação do *TransformationMapper* foram utilizadas duas importantes funcionalidades da plataforma Eclipse: o mecanismo de pesquisa e localização (*Search Engine*) e a árvore de sintaxe abstrata (*Abstract Syntax Tree*).

O mecanismo de pesquisa e localização é uma API do Eclipse para localizar declarações e referências de elementos em projetos Java. Para que essa localização seja realizada, é necessário definir um padrão de pesquisa (*search pattern*) e um escopo de pesquisa (*search scope*). A partir dessas definições, é executado um coletor de resultados denominado `SearchRequestor` e os resultados são armazenados em objetos do tipo `SearchMatch`. O *TransformationMapper* utiliza esse mecanismo de pesquisa para inserir em uma árvore, onde estão mapeados os elementos que compõem os interesses transversais de uma aplicação, todas as ocorrências desses elementos. Essas ocorrências incluem informações de pacotes, classes, interfaces, métodos e linhas de programa.

A árvore de sintaxe abstrata (AST) é uma estrutura de dados interna da plataforma Eclipse que permite ao programador realizar atividades como análise, construção, refatorações e correção de programas. A AST, através de um objeto do tipo `ASTParser`, mapeia um programa Java em uma árvore, onde cada nodo (`ASTNode`) corresponde a um elemento sintático deste programa. Essa estrutura possibilita a navegação e a manipulação dos elementos que fazem parte de um programa, bem como a inclusão de novos elementos. O *TransformationMapper* cria uma AST para cada classe onde existem ocorrências de elementos correspondentes a interesses transversais. A partir dessa criação, navegando-se pela AST da classe, é feita uma análise do ponto do programa onde o interesse transversal está implementado para determinar se existe ou não a necessidade de uma transformação orientada por objetos que viabilize sua extração para aspectos. Essa análise leva em consideração os comandos que precedem e sucedem o interesse e a posição dos mesmos no código do método.

O algoritmo implementado pelo *TransformationMapper* para localizar ocorrências de elementos que compõem interesses transversais e identificar a necessidade de transformações é composto pelas seguintes iterações:

1. Para cada elemento `E` mapeado, realiza-se uma pesquisa no projeto por meio do *Search Engine*. O objetivo é localizar todas as ocorrências de `E` no projeto.

2. Cada item do resultado da pesquisa é inserido na árvore do *TransformationMapper* em um nodo filho de `E`, seguindo a estrutura de pacote, classe, método e números das linhas correspondentes às ocorrências de `E`.

3. Para cada linha `L` correspondente a uma ocorrência de `E`, é chamada a função `Check(E,L)`, que avalia se `L` necessita ou não de uma transformação. Essa função é composta pelos seguintes passos:

   (a) Montar a AST da classe correspondente a `L`.

   (b) Verificar se existe um ponto de junção acima ou abaixo de `L`.

   (c) Se não existir, retornar informando que é necessário realizar uma transformação em `L`.

   (d) Se existir, verificar se o ponto de junção ocorre múltiplas vezes dentro do mesmo método.

   (e) Se ocorrer, retornar informando que é necessário realizar uma transformação em `L`.

   (f) Se não ocorrer, retornar informando que não é necessário realizar uma transformação em `L`.

Mostra-se na Figura 4.5 o Diagrama de Pacotes do *TransformationMapper* com os pacotes de classes que compõem a ferramenta e seus respectivos relacionamentos de dependência. Nesse diagrama são relacionados o Pacote `tm`, que agrupa as classes específicas do *TransformationMapper*, e os pacotes do *ConcernMapper* que foram estendidos por meio de alterações em classes já existentes ou inclusão de novas classes. Também são relacionados os pacotes do Eclipse que foram utilizados para implementar o mecanismo de pesquisa e localização (*Search Engine*) e a árvore de sintaxe abstrata (AST).

Para exemplificar o funcionamento do *TransformationMapper*, mostra-se na Figura 4.6 o código correspondente ao método `PluginFactory.createPlugins()` do sistema JSpider. Nesse exemplo, ocorrem dez chamadas ao método `Log.info(Object)`, que faz parte da funcionalidade de *logging*. Em seguida, na Figura 4.7, mostra-se como o *TransformationMapper* avaliou a necessidade de transformação para cada uma dessas dez chamadas.

Para justificar como o *TransformationMapper* avaliou a necessidade de cada transformação, mostra-se na Tabela 4.1 uma análise de cada uma das ocorrências do interesse transversal.

Mostra-se na Tabela 4.2 informações sobre o código fonte do *ConcernMapper* (código original) e também do *TransformationMapper*. Essa tabela tem como objetivo apresentar o percentual de crescimento do número de linhas de código e da estrutura interna da ferramenta após a implementação do *TransformationMapper*. Para levantamento dessas informações foi utilizado o *plugin* Eclipse *Project Statistics*[1].

[1] `http://www.eclipseplugincentral.com/Web_Links-index-req-viewlink-cid-249.html`.

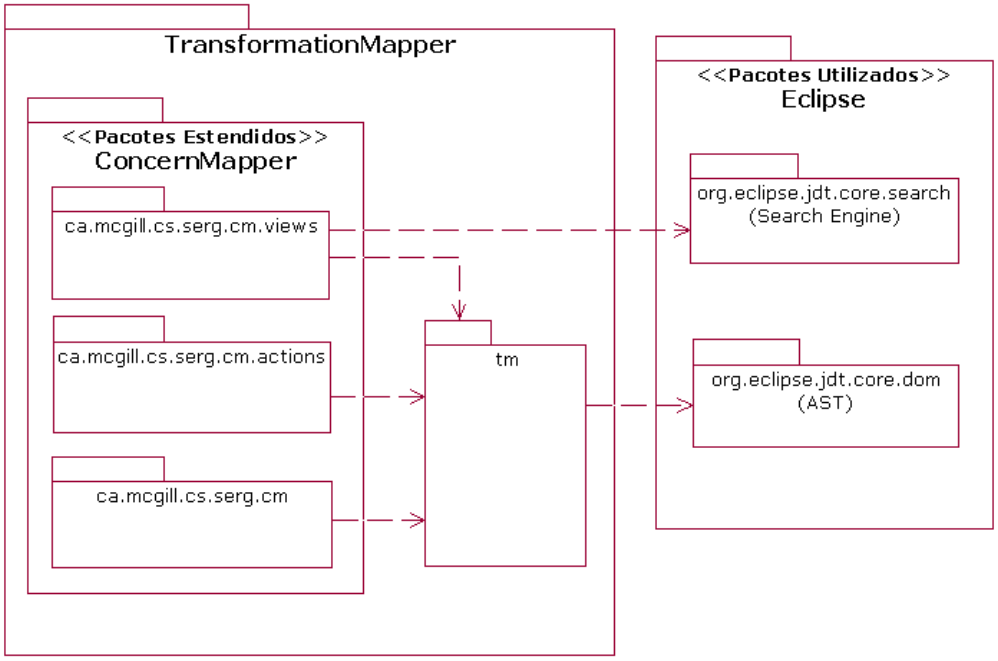

Figura 4.5: Diagrama de Pacotes do *TransformationMapper*.

4.4 Aplicabilidade e Restrições

Conforme descrito na Seção 2.1, a instrumentação de um programa Java por AspectJ está restrita a um conjunto de eventos da execução de um sistema. Sendo assim, quando um interesse transversal ocorre em um ponto que não pode ser interceptado por AspectJ, uma transformação orientada por objetos é requerida. O principal objetivo da ferramenta proposta é identificar se pontos do programa onde interesses transversais estão implementados podem ser diretamente associados a pontos de junção, conforme requerido por AspectJ.

A ferramenta apresentada nesta dissertação é capaz de identificar se o código correspondente a um interesse transversal pode ser associado a algum dos tipos de pontos de junção abaixo relacionados:

- Execução de métodos
- Chamada de métodos
- Execução de construtores
- Chamada de construtores

```
26:public Plugin[] createPlugins() {
     ...
31:  PropertySet props = ConfigurationFactory.getConfiguration(
                        ).getPluginsConfiguration();
     ...
33:  int pluginCount = pluginsProps.getInteger(
                       ConfigConstants.PLUGINS_COUNT, 0);
34:  log.info("Loading " + pluginCount + " plugins.");               // aspect
35:
36:  for (int i = 0; i < pluginCount; i++) {
37:    String pluginInstance = pluginsProps.getString( "" + (i+1) +
                               "." + ConfigConstants.PLUGINS_CONFIG, null);
38:    if (pluginInstance != null) {
39:       log.info("Loading plugin configuration '" +                // aspect
                  pluginInstance + "'...");
40:       PropertySet config = ConfigurationFactory.getConfiguration(
                               ).getPluginConfiguration(pluginInstance);
          ...
43:       if (pluginClass == null) {
44:          log.info("Plugin class '" + pluginConfig.getString(     // aspect
                    ConfigConstants.PLUGIN_CLASS, "") + "' not found");
45:       } else {
            ...
50:         if (filterConfig.getBoolean(
51:             ConfigConstants.PLUGIN_FILTER_ENABLED, false)) {
52:             log.info("Plugin uses local event filtering");       // aspect
53:             loadedPlugins.add(new PluginSocket(plugin, filterConfig));
54:         } else {
                                                                     // aspect
55:            log.info("Plugin not configured for local event filtering");
56:            loadedPlugins.add(plugin);
57:         }
58:
59:         log.info("Plugin Name    : " + plugin.getName());        // aspect
60:         log.info("Plugin Version : " + plugin.getVersion());     // aspect
61:         log.info("Plugin Vendor  : " + plugin.getVendor());      // aspect
62:       }
63:    } else {
64:       log.info("Plugin configuration '" + pluginInstance +       // aspect
                  "' couldn't be loaded");
65:    }
66:  }
67:
68:  log.info("Loaded " + loadedPlugins.size() + " plugins.");       // aspect
69:  return (Plugin[]) loadedPlugins.toArray(new Plugin[loadedPlugins.size()]);
70:}
```

Figura 4.6: Exemplo de método do sistema JSpider onde a funcionalidade de *logging* está presente.

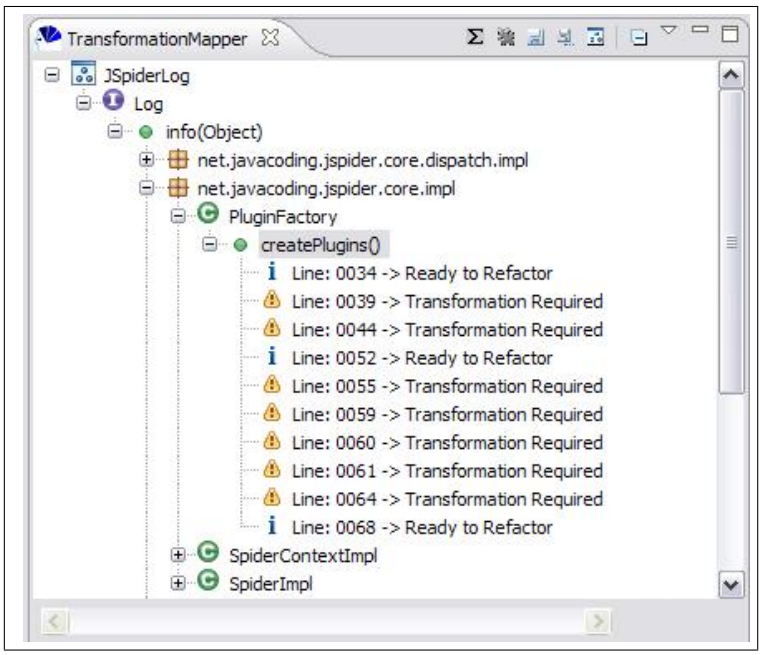

Figura 4.7: Resultado gerados pelo *TransformationMapper* (considerando como alvo o método da Figura 4.6).

- Leitura de atributos de classe
- Modificação de atributos de classe
- Execução de blocos de tratamentos de exceção

Caso a ferramenta identifique que não é possível associar um interesse transversal a um dos tipos de pontos de junção relacionados, ela indica que uma transformação orientada por objetos é requerida para viabilizar a extração do interesse.

A ferramenta consegue identificar também quando a implementação de um interesse transversal está associada um ponto de junção, mas este ocorre mais de uma vez dentro de um mesmo método. Por exemplo, quando existe uma chamada de método logo após um interesse transversal, mas esta chamada ocorre múltiplas vezes no corpo do método. O método apresentado na Figura 4.6 exemplifica essa situação nas linhas 39 e 56.

Limitações: As limitações da atual versão da ferramenta são descritas a seguir:

- A ferramenta não indica qual transformação deve ser aplicada. Essa indicação depende de uma análise semântica do código para que o comportamento original do sistema seja preservado. Recomenda-se que o programador verifique primeiro a possibilidade de realizar uma movimentação de código (Transformações 1 a 17). No

Linha	Transf.	Justificativa
34	Não	A chamada do método `getInteger` (linha 33) pode ser interceptada, criando-se um adendo do tipo `after-call`
39	Sim	O comando anterior não pode ser interceptado por AspectJ e a chamada dos métodos `getConfiguration` e `getPluginsConfiguration` no comando posterior (linha 40) ocorre mais de uma vez dentro do método (linha 31)
44	Sim	Nem o comando anterior nem o comando posterior podem ser interceptados por AspectJ
52	Não	A chamada do construtor da classe `PluginSocket` no comando posterior (linha 53) pode ser interceptada, criando-se um adendo do tipo `before-call`
55	Sim	O comando anterior não pode ser interceptado por AspectJ e a chamada do método `add` no comando posterior (linha 56) ocorre mais de uma vez dentro do método (linha 53)
59	Sim	Os comandos anterior e posterior ao bloco correspondente ao interesse transversal não podem ser interceptados por AspectJ
60	Sim	Os comandos anterior e posterior ao bloco correspondente ao interesse transversal não podem ser interceptados por AspectJ
61	Sim	Os comandos anterior e posterior ao bloco correspondente ao interesse transversal não podem ser interceptados por AspectJ
64	Sim	Os comandos anterior e posterior não podem ser interceptados por AspectJ
68	Não	O comando `return` posterior (linha 69) pode ser interceptado, criando-se um adendo do tipo `before returning`

Tabela 4.1: Justificativas para os resultados gerados pelo *TransformationMapper* (considerando como alvo o método da Figura 4.6)

	ConcernMapper	*TransformationMapper*
LOC	4576	5173 (+13%)
Classes	48	59 (+22,9%)
Atributos	110	125 (+13,6%)
Interfaces	4	4
Métodos	294	341 (+16%)

Tabela 4.2: Informações sobre o código fonte do *ConcernMapper* e do *TransformationMapper*

entanto, movimentações de código exigem que pré-condições sejam atendidas para sua aplicação, conforme descrito na Subseção 3.1.2. Quando não for possível aplicar uma movimentação de código, recomenda-se então uma extração de método (Transformações 18 a 24).

- Situações nas quais são requeridas as Transformações 14 e 15 não são identificadas pela ferramenta. Optou-se por deixar a avaliação de blocos de inicialização estática de classes para uma próxima versão da ferramenta, pois esse tipo de construção ocorre com baixa freqüência em sistemas Java.

- A ferramenta não identifica quando a Transformação 17 é requerida, pois, na versão atual, ela avalia apenas se existe um ponto de junção que pode ser associado ao interesse transversal imediatamente acima ou abaixo do ponto do programa onde ele ocorre. Portanto, a ferramenta não analisa a expressão onde um interesse transversal ocorre para verificar se ele está entrelaçado com código não transversal do sistema e precisa ser desagrupado.

- A necessidade de se aplicar as Transformações 21, 22 e 24 não é identificada pela ferramenta, pois tais transformações não têm como único objetivo a criação de pontos de junção.

- A ferramenta também não identifica a necessidade de transformações *ad hoc*, pois são particulares para cada contexto de sistema e de difícil generalização.

Acredita-se, no entanto, que a identificação das Transformações 14, 15, 17, 21, 22 e 24 possa ser incorporada em uma versão futura do *TransformationMapper*, utilizando parte das atuais rotinas e estruturas de dados da ferramenta para localização de pontos de junção.

4.5 Considerações Finais

Neste capítulo, foi apresentada uma ferramenta que é capaz de identificar transformações orientadas por objetos em sistemas Java para modularização de interesses transversais por meio de aspectos. A ferramenta é capaz de identificar a maior parte das transformações necessárias e assim facilitar o processo de extração de aspectos. Foram apresentados também exemplos de utilização da ferramenta e uma visão geral da sua implementação. Por fim, foram relacionadas situações nas quais a ferramenta se aplica e suas principais limitações.

Capítulo 5

Avaliação

Neste capítulo, apresenta-se uma avaliação da solução proposta por meio de quatro estudos de caso. Os resultados dessa avaliação foram obtidos com a identificação e quantificação do uso das transformações propostas nesta dissertação no processo de extração de aspectos dos sistemas JSpider, Prevayler, JHotDraw e JAccounting. Esses sistemas foram escolhidos por terem passado por processos de refatoração orientada por aspectos nos quais foram aplicadas transformações orientadas por objetos. Em trabalhos recentes foram apresentados resultados do uso dessas transformações nesses sistemas, os quais foram obtidos por meio de inspeção manual, isto é, sem o uso de uma ferramenta de apoio [MV07, MV08]. Neste capítulo, são apresentados os resultados dos trabalhos citados com algumas revisões e também os resultados obtidos com o uso do *TransformationMapper*. Nas Seções 5.1 a 5.4, apresentam-se as avaliações realizados em cada sistema e na Seção 5.5 são discutidos os resultados dessas avaliações.

5.1 JSpider

O sistema JSpider é um robô para *download* e análise de páginas *Web* com o objetivo de identificação de erros, verificação de *links* e mapeamento de estrutura de sites[1]. Nesta seção, são analisadas as transformações realizadas por Binkley et al. para extração de aspectos responsáveis pela funcionalidade de *logging* [BCH+06].

Relacionam-se na Tabela 5.1 as transformações realizadas no sistema JSpider. Os resultados apresentados foram obtidos por meio de inspeção manual e também por meio da ferramenta *TransformationMapper* (TM).

As diferenças entre os resultados obtidos manualmente e por meio do *Transformation-*

[1] `http://j-spider.sourceforge.net`.

	Manual (Qtd.)	**TM (Qtd.)**
Transf. 1	3	2
Transf. 18	6	2
Transf. 19	16	10
Transf. 20	3	2
Transf. 21	5	0
Transf. 23	1	1
Ad hoc	2	0
Total (T)	36	17

Tabela 5.1: Transformações no sistema JSpider

Mapper ocorrem devido a dois fatores: (1) limitações que a ferramenta apresenta na sua atual versão (por exemplo, não são identificadas transformações *ad hoc* e a Transformação 21) e (2) transformações que não eram necessárias para expor um ponto de junção que pudesse ser associado ao código correspondente à funcionalidade de *logging* e que mesmo assim foram realizadas pelos desenvolvedores dos aspectos.

Com a utilização do *TransformationMapper*, foram identificadas 12 transformações realizadas desnecessariamente no sistema JSpider, isto é, transformações aplicadas em pontos onde o código correspondente ao interesse transversal poderia ser associado a um ponto de junção na versão original do sistema. As diferenças decorrentes de transformações desnecessárias incluem uma Transformação 1, quatro Tranformações 18, seis Transformações 19 e uma Transformação 20.

Para ilustrar uma situação em que uma transformação foi aplicada desnecessariamente, mostra-se na Figura 5.1 um exemplo onde o método `exit(Exception)` é extraído com o código que sucede a chamada do método `log.fatal(Object, Throwable)`. Essa extração não é necessária, pois seria possível criar um conjunto de junção que capturasse a ativação da exceção `Exception` e associá-lo a um adendo do tipo `after` contendo a chamada do método `log.fatal(Object, Throwable)`.

Quando processos de extração de aspectos utilizando AspectJ são realizados de forma manual, é necessário que o programador conheça bem o modelo de pontos de junção de AspectJ para evitar transformações desnecessárias no código orientado por objetos. Além disso, ações repetitivas também podem induzir transformações desnecessárias. A identificação da ocorrência de 12 transformações desnecessárias nessa avaliação mostra que o uso da ferramenta *TransformationMapper* é vantajoso para evitar que alterações sejam feitas no código de um sistema sem que haja necessidade.

```
public synchronized void run() {
   ...
   try {
      ...
   } catch (Exception e) {
      log.fatal("PANIC! Task " + task +
                " threw an excpetion!", e);        // aspecto
      System.exit(1);
   }
   ...
}
```

⇓

```
public synchronized void run() {
   ...
   try {
      ...
   } catch (Exception e) {
      log.fatal("PANIC! Task " + task +
                " threw an excpetion!", e);        // aspecto
      exit(e);
   }
   ...
}

private void exit(Exception e) {           // método extraído
   System.exit(1);
}
```

Figura 5.1: Código antes e após aplicação da Transformação 19 em método do sistema JSpider.

5.2 Prevayler

O Prevayler é um sistema gerenciador de bancos de dados em memória principal que fornece uma biblioteca de persistência de objetos para sistemas Java[2]. São analisadas nesse sistema as transformações realizadas por Godil e Jacobsen para modularizar em aspectos o seguinte conjunto de interesses transversais: gerenciamento de *snapshots*, censura de transações, replicação, *logging* persistente, *clock* de transações e concorrência [GJ05].

Relacionam-se na Tabela 5.2 as transformações realizadas no sistema Prevayler. Os resultados apresentados foram obtidos por meio de inspeção manual e também por meio da ferramenta *TransformationMapper* (TM).

As diferenças entre os resultados obtidos manualmente e por meio do *TransformationMapper* ocorrem em decorrência das limitações que a ferramenta apresenta na atual

[2] `http://www.prevayler.org`.

	Manual (Qtd.)	TM (Qtd.)
Transf. 19	1	1
Transf. 22	1	0
Transf. 23	2	2
Transf. 24	2	0
Ad hoc	2	0
Total (T)	8	3

Tabela 5.2: Transformações no sistema Prevayler

versão onde não são identificadas transformações *ad hoc* e as Transformações 22 e 24.

5.3 JHotDraw

O sistema JHotDraw é um *framework* para criação de aplicações para edição de figuras geométricas[3]. São analisadas nesta seção as transformações aplicadas nesse *framework* por Binkley et al. para posterior modularização em aspectos do interesse responsável pela funcionalidade de *undo* [BCH+05, BCH+06].

Relacionam-se na Tabela 5.3 as transformações realizadas no sistema JHotDraw. Os resultados apresentados foram obtidos por meio de inspeção manual e também por meio da ferramenta *TransformationMapper* (TM).

	Manual (Qtd.)	TM (Qtd.)
Transf. 5	1	1
Transf. 16	1	0
Transf. 17	15	0
Transf. 18	1	0
Total (T)	18	1

Tabela 5.3: Transformações no sistema JHotDraw

As diferenças entre os resultados obtidos manualmente e por meio do *Transformation-Mapper* ocorrem devido a dois fatores: (1) limitações que a ferramenta apresenta na atual versão na qual não é identificada a Transformação 17 e (2) assim como ocorreu no sistema JSpider, foram identificadas duas transformações que não eram necessárias para expor um ponto de junção que pudesse ser associado ao código correspondente à funcionalidade de *undo*.

Para que a Transformação 17, que é a mais freqüente no sistema JHotDraw, possa ser identificada em uma versão futura da ferramenta, seria necessário que o programador

[3] `http://www.jhotdraw.org`.

informasse, no momento do mapeamento de um interesses transversal, quais elementos que compõem esse interesse precisam ser separados do código base da aplicação quando estiverem entrelaçados em uma expressão. Para exemplificar essa situação, mostra-se na Figura 5.2 um exemplo onde os construtores de `UndoableCommand` e `DeleteCommand` estão entrelaçados na expressão que fornece o retorno do método `createDeleteCommand()`. Nesse caso, seria necessário informar à ferramenta que o construtor de `UndoableCommand`, que faz parte da funcionalidade de *undo*, não poderia estar presente em expressões com outros tipos que não fazem parte dessa funcionalidade.

```
protected Command createDeleteCommand() {
   return new UndoableCommand(
          new DeleteCommand("Delete", editor()));          // aspecto
}
```

⇓

```
protected Command createDeleteCommand () {
   DeleteCommand tmp1 = new DeleteCommand ("Delete", editor ());
   UndoableCommand tmp2 = new UndoableCommand (tmp1);      // aspecto
   return tmp2;
}
```

Figura 5.2: Código antes e após aplicação da Transformação 17 em método do sistema JHotDraw.

Com a utilização do *TransformationMapper*, foram identificadas duas transformações realizadas desnecessariamente no sistema JHotDraw em pontos onde o código correspondente ao interesse transversal poderia ser associado a um ponto de junção na versão original do sistema. As diferenças decorrentes de transformações desnecessárias incluem uma Transformação 16 e uma Transformação 18.

5.4 JAccounting

O JAccounting é um sistema Web para controle de pedidos, pagamentos e faturamento[4]. Nesse sistema, são analisadas as transformações aplicadas por Binkley et al. para viabilizar a extração de aspectos com código para controle de transações [BCH+06].

Relacionam-se na Tabela 5.4 as transformações realizadas no sistema JAccounting. Os resultados apresentados foram obtidos por meio de inspeção manual e também por meio da ferramenta *TransformationMapper* (TM).

As diferenças entre os resultados obtidos manualmente e por meio do *TransformationMapper* ocorrem devido a dois fatores: (1) limitações que a ferramenta apresenta na

[4] `https://jaccounting.dev.java.net`.

	Manual (Qtd.)	**TM (Qtd.)**
Transf. 1	1	1
Transf. 4	15	9
Transf. 5	14	7
Ad hoc	30	0
Total (T)	60	17

Tabela 5.4: Transformações no sistema JAccounting

atual versão, onde não são identificadas transformações *ad hoc* e (2) transformações que não eram necessárias para expor um ponto de junção associado ao código correspondente ao controle de transações.

Apesar do *TransformationMapper* ter identificado a necessidade de transformações para expor pontos de junção que pudessem ser associados ao código correspondente ao controle de transações, nota-se que o objetivo principal dessas transformações foi tornar a implementação do interesse transversal mais homogênea, possibilitando assim a criação de apenas quatro conjuntos de junção para capturar quarenta e cinco pontos de junção no sistema. Todas as transformações não identificadas pelo *TransformationMapper* foram realizadas com esse propósito.

Para ilustrar o objetivo das transformações *ad hoc* que foram aplicadas no sistema JAccounting, mostra-se um exemplo na Figura 5.3. Nesse exemplo, mesmo o método `beginTransaction` ocorrendo imediatamente antes do método `load`, foi aplicada a Transformação 5 para que a abertura da transação passasse a ocorrer sempre logo após a abertura da sessão. No caso do método `commit`, mesmo ele ocorrendo imediatamente após o método `addMessage`, foi aplicada a Transformação 4 para que a confirmação da transação passasse a ocorrer sempre antes do encerramento da sessão. Além disso, também foram necessárias transformações *ad hoc* associadas aos métodos `rollback` e `commit` para que o estado de transação pudesse ser controlado através da variável local `tx`.

Contudo, em sua atual versão, o *TransformationMapper* não identifica a necessidade de transformações *ad hoc* no código OO de um sistema para homogeneizar a implementação de interesses transversais. O principal motivo é que tais transformações requerem um domínio detalhado da lógica do sistema, o qual normalmente apenas seus desenvolvedores possuem.

5.5 Discussão

Nesta seção é discutida a importância das transformações, como elas afetam interesses transversais homogêneos e heterogêneos e as principais dificuldades para automatização

```
protected String perform2() throws Exception {
   ...
   net.sf.hibernate.Transaction tx = null;
   Session sess = getSessionFactory().openSession();
   try {
      tx = sess.beginTransaction();                          // aspecto
      payment = (Payment) sess.load(Payment.class, id);
      ...
      addMessage("Payment has been deleted");
      tx.commit();                                           // aspecto
   } catch (Exception e) {
       if (tx != null)
          tx.rollback();                                     // aspecto
       throw e;
   } finally {
       sess.close();
   }
   ...
}
```

⇓

```
protected String perform2() throws Exception {
   ...
   net.sf.hibernate.Transaction tx = null;
   Session sess = getSessionFactory().openSession();
   tx = sess.beginTransaction();                             // aspecto
   try {
      payment = (Payment) sess.load(Payment.class, id);
      ...
      addMessage("Payment has been deleted");
   } catch (Exception e) {
       if (tx != null) {
          tx.rollback();                                     // aspecto
          tx = null;
       }
       throw e;
   } finally {
       if (tx != null) {
          tx.commit();                                       // aspecto
          tx = null;
       }
       sess.close();
   }
   ...
}
```

Figura 5.3: Código antes e após aplicação de transformações em método do sistema JAccounting.

das transformações propostas.

Alguns pontos a serem discutidos levam em consideração os resultados das avaliações realizadas consolidados na Tabela 5.5. Para cada sistema avaliado, mostra-se nessa tabela o total de transformações obtido por meio de inspeção manual. Mostra-se também o número de pontos de junção e de conjuntos de junção dos sistemas avaliados. Por último, a tabela apresenta as razões entre transformações por pontos de junção (T/P) e pontos de junção por conjuntos de junção (P/C).

	JSpider	**Prevayler**	**JHotDraw**	**JAccounting**
Total (T)	36	8	18	60
Join Points (P)	246	55	117	45
Pointcuts (C)	236	40	86	4
T/P	0.15	0.15	0.15	1.33
P/C	1.04	1.37	1.36	11.25

Tabela 5.5: Resultados das avaliações

5.5.1 Importância das Transformações

Transformações no código orientado por objetos constituem um passo fundamental para o sucesso de processos de extração de aspectos em sistemas legados. Nos sistemas com menor percentual de transformações em relação ao número de pontos de junção – JHotDraw, JSpider e Prevayler – elas foram responsáveis pela geração de 15% dos pontos de junção da versão orientada por aspectos do sistema. Ou seja, os resultados obtidos evidenciam que transformações no código OO são imprescindíveis para extração de aspectos nesses sistemas.

No sistema JAccounting, devido ao grande número de transformações *ad hoc* e ao objetivo de homogeneizar a implementação do controle de transações, o número de transformações foi 33% superior ao número de pontos de junção.

5.5.2 Interesses Transversais Homogêneos e Heterogêneos

O número de transformações por ponto de junção (T/P) dos sistemas JSpider, JHotDraw e Prevayler foi 0.15. Esse índice é bastante inferior ao do JAccounting (1.33). Quanto ao número de pontos de junção por conjuntos de junção (P/C), nos sistemas JSpider, JHotDraw e Prevayler, cada conjunto de junção captura em média respectivamente 1.04, 1.37 e 1.36 pontos de junção. Já no sistema JAccounting, cada conjunto de junção captura em média 11.25 pontos de junção. Acredita-se que esses fatos tenham relação direta com o tipo de interesse transversal modularizado.

Nos sistemas JHotDraw e JSpider, foram modularizados interesses transversais heterogêneos, isto é, com comportamento diferente em cada ponto de junção. Em decorrência disso, o número de transformações foi menor, já que nesse caso não há benefício em padronizar os pontos de junção onde será introduzido código transversal. Verifica-se também que os conjuntos de junção capturam um número muito pequeno de pontos de junção.

No JAccounting, modularizou-se um interesse transversal homogêneo, isto é, um interesse implementado por meio do mesmo código em diversos pontos do sistema [CC04]. Como interesses transversais homogêneos devem ser implementados de forma similar e consistente em todo sistema, justifica-se um grande número de transformações quando a implementação de tais interesses estiver presente em múltiplas classes e entrelaçada com o código base do sistema. Por outro lado, as transformações realizadas para tornar homogênea a implementação do interesse transversal nesse sistema melhoraram a qualidade do aspecto, possibilitando que cada conjunto de junção capture em média 11.25 pontos de junção.

5.5.3 Automatização de Transformações

A automatização de transformações orientadas por objetos é um desafio, mesmo quando os elementos que compõem um determinado interesse transversal já tenham sido previamente identificados por uma ferramenta de mineração de aspectos. A principal complexidade está na movimentação de código e nas transformações *ad hoc*.

Conforme descrito na Seção 3.1, para avaliar se as pré-condições para que a seqüência de comandos `A;B` possa ser refatorada para `B;A`, é necessário uma análise global do fluxo de dados e de controle no sistema. Além disso, a posição de um comando no código pode estar relacionada à lógica do interesse transversal que estiver sendo refatorado. Por exemplo, no sistema JSpider, a movimentação de comandos correspondentes à funcionalidade de *logging* pode tornar as mensagens incorretas ou menos precisas.

Transformações *ad hoc*, conforme sugerido pelo próprio nome, dependem do controle de fluxo de um programa e do estado do sistema que se deseja salvar para posteriormente ser recuperado. Por essa razão, torna-se bastante complexo definir uma seqüência de ações para automatizar esse tipo de transformação.

5.6 Considerações Finais

Neste capítulo, foram apresentados os resultados de quatro avaliações realizadas em sistemas reais que foram alvo de refatoração orientada por aspectos. Nessas avaliações, foram analisadas as transformações aplicadas nesses sistemas por meio da comparação entre as versões OO e AO de cada sistema e da utilização do *TransformationMapper* na versão OO. Por fim, foram discutidas as principais lições aprendidas com essas avaliações.

As avaliações realizadas mostraram que o *TransformationMapper*, mesmo ainda apresentando limitações na atual versão, é útil no processo de identificação de transformações requeridas no código base de sistemas legados para modularização de interesses transversais em aspectos. As avaliações também mostraram que a ferramenta pode evitar a aplicação de várias transformações desnecessárias.

As discussões deste capítulo abordaram a importância das transformações, como as transformações podem afetar interesses transversais homogêneos e heterogêneos e as dificuldades para que as transformações sejam automatizadas.

Capítulo 6

Conclusões

Com a consolidação de aspectos como o principal paradigma de programação para separação e modularização de interesses transversais, foram realizados diversos trabalhos de refatoração em sistemas legados para extração e modularização de tais interesses por meio de aspectos [Lad03, HMK05, MF06, BCH+05, BCH+06]. Porém, observa-se nesses trabalhos que nem sempre refatorações podem ser aplicadas de forma direta visando a extração de aspectos, pois linguagens orientadas por aspectos permitem a introdução de comportamento transversal apenas em pontos de junção, os quais correspondem a pontos pré-definidos da execução de um programa. No entanto, em se tratando de sistemas legados, não se pode garantir que o código que deve ser extraído para um aspecto ocorre exatamente antes, depois ou no lugar de um ponto de junção. Além disso, para que alguns tipos de refatorações possam ser aplicadas, também são necessárias transformações *ad hoc* no código de um programa para salvar ou restabelecer o fluxo de execução em pontos específicos.

Nesta dissertação, utilizou-se o termo *transformações* para diferenciar as tarefas que fazem parte do processo de preparação do código de um sistema para a extração de aspectos das que fazem parte do processo de refatoração de interesses transversais. Essa diferenciação é necessária, pois refatorações devem melhorar a qualidade do projeto e da organização interna do código de um sistema, preservando seu comportamento [FBB+99], enquanto a aplicação de transformações nem sempre promove esse tipo de melhoria.

A principal motivação para a pesquisa desenvolvida nesta dissertação surgiu a partir da observação de que trabalhos na área de refatoração de interesses transversais geralmente mencionam a necessidade de transformações no código-base de sistemas legados após a identificação de trechos de código com comportamento transversal e antes da aplicação de uma refatoração visando a extração destes para aspectos [MLWR01, CB05, BCH+06, MF06, LBL06, YR07]. No entanto, tais trabalhos não descrevem essas transformações de

forma detalhada, nem apresentam diretrizes que possam apoiar desenvolvedores no uso dessas transformações em sistemas legados.

6.1 Contribuições

As principais contribuições desta dissertação são relacionadas a seguir:

- Foi proposto um catálogo com 24 transformações dividas em dois conjuntos: Movimentações de Código (Transformações 1 a 17) e Extrações de Métodos (Transformações 18 a 24) [MV07, MV08]. Essas transformações foram descritas e o uso de algumas delas foi exemplificado em sistemas reais que passaram por processos de refatoração orientada por aspectos. Pelos exemplos apresentados, foi possível evidenciar a importância das transformações nos sistemas avaliados. Foram apresentadas pré-condições e algumas limitações para aplicação das transformações catalogadas. Para cada uma das transformações, também foram apresentadas definições formais.

- Foi implementada a ferramenta *TransformationMapper* que, a partir do mapeamento de elementos que compõem interesses transversais, é capaz de identificar a maior parte das transformações orientadas por objetos que devem ser aplicadas em sistemas Java para modularização desses interesses por meio de aspectos. Apresentou-se uma descrição do projeto dessa ferramenta, suas principais funcionalidades e exemplos práticos de sua utilização. Também foram relacionadas as situações nas quais a ferramenta se aplica e as principais limitações de sua atual implementação.

- Com o apoio do *TransformationMapper*, foram realizados estudos de caso envolvendo os sistemas JSpider, Prevayler, JHotDraw e JAccounting. Foram identificadas, discriminadas e quantificadas as transformações aplicadas nesses sistemas para viabilizar a extração de interesses transversais para aspectos. As principais lições aprendidas nesses estudos de caso são resumidas a seguir:

  - Os resultados obtidos evidenciam que transformações no código orientado por objetos são atividades imprescindíveis para extração de aspectos.
  - Interesses transversais homogêneos podem requerer um número maior de transformações se comparados a interesses heterogêneos. Tal fato se deve à necessidade de se utilizar transformações para tornar a implementação de interesses homogêneos similar e consistente em todo sistema.

– A partir de inspeção manual no código de sistemas legados, desenvolvedores podem realizar transformações desnecessárias. O uso de uma ferramenta como o *TransformationMapper* pode evitar essas transformações.

– A aplicação de transformações de forma automatizada é uma tarefa de alta complexidade, principalmente por demandar movimentações de código e transformações *ad hoc*. Tal complexidade se deve à necessidade de uma análise global do fluxo de dados e de controle no sistema.

6.2 Comparação com Outros Trabalhos

Binkley et al. apresentam uma abordagem semi-automática para migração de sistemas orientados por objetos para o paradigma orientado por aspectos [BCH+06]. Nessa abordagem, seis tipos de refatorações foram implementadas na ferramenta *AOP-Migrator*. Os autores apresentam ainda os resultados de quatro estudos de caso envolvendo os sistemas JAccounting, JSpider, JHotDraw – também utilizados nos estudos de caso desta dissertação – e PetStore[1]. Os autores descrevem a necessidade de se realizar uma preparação do código orientado por objetos como atividade complementar às refatorações para extração de aspectos. As atividades relacionadas a essa preparação, assim como no presente trabalho, também são denominadas *transformações*. No entanto, as transformações mencionadas no trabalho não são discriminadas, exemplificadas e formalizadas. Esta dissertação propõe uma abordagem mais detalhada para descrever transformações orientadas por objetos, além de uma ferramenta de apoio para auxiliar na identificação destas.

Monteiro e Fernandes apresentam um catálogo com três conjuntos de refatorações para extração, reestruturação interna e generalização de aspectos [MF05, MF06]. Segundo os autores, nem sempre as refatorações propostas nesse catálogo podem ser aplicadas de forma direta visando a extração de aspectos, mesmo quando o interesse transversal já tenha sido corretamente identificado e delimitado no código fonte de um sistema. Porém, as atividades de preparação do código orientado por objetos necessárias para que as refatorações possam ser aplicadas não são detalhadas nos trabalhos desses autores.

Cole e Borba apresentam um conjunto de leis de programação para auxiliar na formalização e automatização de processos de extração e reestruturação de aspectos [CB05]. Os autores citam a definição de pré-condições para aplicação das leis propostas como uma das principais dificuldades encontradas no processo de refatoração dos sistemas que

[1] `http://java.sun.com/developer/releases/petstore/`

foram avaliados. Quando tais pré-condições não são satisfeitas, deve-se transformar o código orientado por objetos, mas, assim como nos trabalhos de Monteiro e Fernandes, essas transformações são mencionadas de forma abreviada.

Em [YR07], Yuen e Robillard abordam a lacuna existente entre processos de mineração de aspectos e refatorações orientadas por aspectos. Segundo os autores, apesar das técnicas de mineração de aspectos serem capazes de identificar interesses transversais em sistemas legados, apenas um pequeno conjunto dos elementos que compõem tais interesses podem ser diretamente extraídos para aspectos. Uma das razões para essa limitação é que linguagens orientadas por aspectos, como AspectJ, capturam apenas um pequeno conjunto de pontos da execução de um programa. Para solucionar esse problema, os autores propõem que refatorações orientadas por aspectos sejam precedidas por um estágio inicial de refatorações orientadas por objetos, sem descrevê-las de forma detalhada. Basicamente, o trabalho de Yuen e Robillard foi uma das fontes de inspiração para o desenvolvimento desta dissertação.

6.3 Trabalhos Futuros

Com o objetivo de aumentar o número de transformações que a ferramenta *TransformationMapper* é capaz de identificar, pretende-se investigar a possibilidade de incorporar a ela a capacidade de detectar as Transformações 14, 15, 17, 21, 22 e 24. Planeja-se incorporar essa capacidade estendendo as atuais rotinas e estruturas de dados da ferramenta para localização de pontos de junção.

Pretende-se também realizar novas avaliações utilizando sistemas reais para melhor validar o catálogo de transformações proposto e uso do *TransformationMapper* no processo de modularização de interesses transversais por meio de aspectos. Essas avaliações poderão sugerir novas transformações para o catálogo, além de novas funcionalidades para o *TransformationMapper*.

Adicionalmente, pretende-se investigar métricas que possam ser aplicadas sobre o mapeamento criado pela ferramenta *TransformationMapper*. Com base nessas métricas pretende-se avaliar propriedades dos aspectos a serem extraídos a partir dos elementos que compõem os interesses mapeados. Resultados preliminares relacionados a essa investigação são descritos em [MMSV07].

Bibliografia

[AKT07] Kim Mens Andy Kellens and Paolo Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-Oriented Software Development*, 4:145–164, 2007.

[AX07] Prasanth Anbalagan and Tao Xie. Automated inference of pointcuts in aspect-oriented refactoring. In *29th International Conference on Software Engineering (ICSE)*, pages 127–136, 2007.

[BCH+05] David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Automated refactoring of object oriented code into aspects. In *21st IEEE International Conference on Software Maintenance (ICSM)*, pages 27–36, 2005.

[BCH+06] David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions Software Engineering*, 32(9):698–717, 2006.

[CB05] Leonardo Cole and Paulo Borba. Deriving refactorings for AspectJ. In *4th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 123–134, 2005.

[CC04] Adrian Colyer and Andrew Clement. Large-scale AOSD for middleware. In *3rd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 56–65, 2004.

[CMM+05] Mariano Ceccato, Marius Marin, Kim Mens, Leon Moonen, Paolo Tonella, and Tom Tourwé. A qualitative comparison of three aspect mining techniques. In *13th International Workshop on Program Comprehension (IWPC)*, pages 13–22, 2005.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[FF00] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *OOSPLA Workshop on Advanced Separation of Concerns*, pages 21–35, 2000.

[FGR07] Fernando Castor Filho, Alessandro Garcia, and Cecília Mary F. Rubira. Error handling as an aspect. In *2nd Workshop on Best Practices in Applying Aspect-Oriented Software Development (BPAOSD)*, pages 1–6, 2007.

[GJ05] Irum Godil and Hans-Arno Jacobsen. Horizontal decomposition of Prevayler. In *15th Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 83–100, 2005.

[GL03] Joseph D. Gradecki and Nicholas Lesieck. *Mastering AspectJ: Aspect-Oriented Programming in Java*. Wiley, 2003.

[GSF+05] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing design patterns with aspects: a quantitative study. In *4th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 3–14, 2005.

[HHM+04] Mark Harman, Lin Hu, Malcolm Munro, Xingyuan Zhang, Dave Binkley, Sebastian Danicic, Mohammed Daoudi, and Lahcen Ouarbya. Syntax-directed amorphous slicing. *Automated Software Engineering*, 11(1):27–61, 2004.

[HK01] Jan Hannemann and Gregor Kiczales. Overcoming the prevalent decomposition in legacy code. In *ICSE Workshop on Advanced Separation of Concerns*, pages 1–5, 2001.

[HMK05] Jan Hannemann, Gail C. Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In *4th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 135–146, 2005.

[HS96] Brian Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1996.

[KAB07] Christian Kastner, Sven Apel, and Don Batory. A case study implementing features using AspectJ. In *11th International Software Product Line Conference (SPLC)*, pages 223–232, 2007.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *15th European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *11th European Conference Object-Oriented Programming (ECOOP)*, volume 1241 of LNCS, pages 220–242, 1997.

[Lad03] Ramnivas Laddad. *Aspect-Oriented Refactoring*. TheServerSide.com, 2003.

[LBL06] Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *28th International Conference on Software Engineering (ICSE)*, pages 112–121, 2006.

[MDM04] Marius Marin, Arie van Deursen, and Leon Moonen. Identifying aspects using fan-in analysis. In *11th Working Conference on Reverse Engineering (WCRE)*, pages 132–141, 2004.

[MF03] Miguel P. Monteiro and João M. Fernandes. Some thoughts on refactoring objects to aspects. In *VIII Jornadas de Ingeniería de Software y Bases de Datos (JISBD)*, pages 1–6, 2003.

[MF05] Miguel P. Monteiro and João M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *4th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 111–122, 2005.

[MF06] Miguel P. Monteiro and João M. Fernandes. Towards a catalogue of refactorings and code smells for AspectJ. *Transactions on Aspect-Oriented Software Development*, 3880:214–258, 2006.

[MLWR01] Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard. Separating features in source code: an exploratory study. In *23rd International Conference on Software Engineering (ICSE)*, pages 275–284, 2001.

[MMD06] Marius Marin, Leon Moonen, and Arie van Deursen. FINT: Tool support for aspect mining. In *13th Working Conference on Reverse Engineering (WCRE)*, pages 299–300, 2006.

[MMSV07] Marcelo Nassau Malta, Tarik Melo, Leonardo Humberto Guimarães Silva, and Marco Túlio O. Valente. Um estudo sobre o emprego de quantificação em sistemas orientados por aspectos: Resultados preliminares. In *I Latin American Workshop on Aspect-Oriented Software Development (LA-WASP)*, pages 163–168, 2007.

[MV07] Marcelo Nassau Malta and Marco Túlio O. Valente. Transformações de código para extração de aspectos. In *I Latin American Workshop on Aspect-Oriented Software Development (LA-WASP)*, pages 93–104, 2007.

[MV08] Marcelo Nassau Malta and Marco Túlio O. Valente. Object-oriented transformations for extracting aspects. *Information and Software Technology*, 2008. (in press).

[RWW05] Martin P. Robillard and Frédéric Weigand-Warr. ConcernMapper: simple view-based separation of scattered concerns. In *OOPSLA Workshop on Eclipse Technology eXchange (ETX)*, pages 65–69, 2005.

[SGP04] David Shepherd, Emily Gibson, and Lori L. Pollock. Design and evaluation of an automated aspect mining tool. In *International Conference on Software Engineering Research and Practice (SERP)*, pages 601–607, 2004.

[SLB02] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *17th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 174–190, 2002.

[TBBV04] Fábio Tirelo, Roberto da Silva Bigonha, Mariza Andrade da Silva Bigonha, and Marco Túlio de Oliveira Valente. Desenvolvimento de software orientado por aspectos. In *XXIII Jornada de Atualização em Informática (JAI)*, volume 2, pages 57–96, 2004.

[TC04] Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *11th Working Conference on Reverse Engineering (WCRE)*, pages 112–121, 2004.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[YR07] Isaac Yuen and Martin P. Robillard. Bridging the gap between aspect mining and refactoring. In *3rd AOSD Workshop on Linking Aspect Technology and Evolution*, pages 1–6, 2007.

[ZJ03a] Charles Zhang and Hans-Arno Jacobsen. A prism for research in software modularization through aspect mining. Technical communication, Middleware Systems Research Group, University of Toronto, 2003.

[ZJ03b] Charles Zhang and Hans-Arno Jacobsen. Refactoring middleware with aspects. *IEEE Transactions Parallel and Distributed Systems*, 14(11):1058–1073, 2003.

[ZJ04] Charles Zhang and Hans-Arno Jacobsen. Resolving feature convolution in middleware systems. In *19th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 188–205, 2004.