Transformações de Código para Extração de Aspectos

Marcelo Nassau Malta e Marco Túlio de Oliveira Valente

Instituto de Informática
Pontifícia Universidade Católica de Minas Gerais
{nassau,mtov}@pucminas.br

Abstract. *In the migration of object-oriented systems towards the aspect technology, after locating fragments of code that presents a crosscutting behavior and before extracting such code to aspects, some transformations must be applied in the base program. Such transformations aim to associate crosscutting code to points of the base program that can be captured using the pointcut descriptor model of an aspect-oriented language. In this paper, we present a catalog of transformations that can be applied to object-oriented systems, in order to prepare such systems to aspect extraction. Moreover, we have conducted an experimental study in order to evaluate the demand for the proposed transformations in systems that have been the target of crosscutting concerns refactoring.*

Resumo. *Em processos de migração de sistemas legados em direção ao uso de aspectos, após a identificação de trechos de código com comportamento transversal e antes da aplicação de uma refatoração visando a extração destes para aspectos, pode ser necessária uma transformação no programa base. Essa transformação tem como objetivo viabilizar a associação de tais códigos a pontos passíveis de captura usando o modelo para descrição de conjuntos de junção de uma linguagem orientada por aspectos. Neste artigo, apresenta-se um catálogo das principais transformações que podem ser aplicadas em sistemas orientados por objetos, visando a preparação destes para extração de aspectos. Adicionalmente, apresenta-se um estudo empírico para avaliação dessas transformações em sistemas legados que passaram por refatorações para a extração de interesses transversais.*

1 Introdução

Na última década, aspectos consolidaram-se como o principal paradigma de programação quando se trata de separação e modularização de interesses transversais. Em decorrência dessa consolidação, diversos trabalhos foram desenvolvidos em duas áreas: mineração de aspectos [1] e refatoração de software orientada por aspectos [8, 7, 10, 11]. Mineração de aspectos tem como um dos seus principais objetivos a identificação de interesses transversais em código orientado por objetos. Normalmente, a implementação de tais interesses ocorre de forma entrelaçada e espalhada, o que pode prejudicar o entendimento, evolução e manutenção dos mesmos. Uma vez identificados os interesses transversais presentes no código fonte de um sistema, técnicas de refatoração orientada por aspectos são usadas para modularizar em aspectos a implementação de interesses transversais. 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 [6].

No entanto, nem sempre refatorações podem ser aplicadas de forma direta visando a extração de aspectos, mesmo quando o interesse transversal já foi corretamente identificado no código fonte de um sistema [10, 2, 3]. 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 esses pontos incluem chamadas e execução de métodos e construtores, leitura e atribuição em atributos de classes, execução de tratadores de exceção, dentre outros. Contudo, em se tratando de um sistema legado, 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, pode ser necessária uma *transformação* no programa base, de forma a associar esse código a um ponto do programa passível de captura usando os recursos para descrição de conjuntos de junção de uma linguagem orientada por aspectos [2, 3].

Em geral, trabalhos na área de refatoração de interesses transversais mencionam a necessidade de tais transformações de forma abreviada, sem um esforço mais aprofundado de descrição, classificação e quantificação das transformações requeridas. Por exemplo, Monteiro em recentes trabalhos sobre catálogos de refatorações orientadas por aspectos [10, 11] não investiga de forma mais apurada a necessidade de refatorar o código para expor pontos de junção requeridos por AspectJ. Binkley et al., em um trabalho que descreve uma versão mais elaborada da ferramenta AOP-Migrator [3], declaram que 40% das refatorações aplicadas nos estudos de caso consistiram em transformações no código fonte dos sistemas estudados. Porém, apesar de classificarem e quantificarem as transformações, estas não são descritas de forma detalhada.

O foco deste artigo é a investigação das transformações que são necessárias no código fonte de um sistema legado após a identificação, mas antes da extração de aspectos. Basicamente, o artigo possui duas contribuições principais: (1) uma descrição das principais transformações que podem ser aplicadas em sistemas orientados por objetos, visando a incorporação de aspectos; (2) um estudo empírico sobre o uso dessas transformações em três sistemas que já foram objeto de refatorações orientadas por aspectos. Acredita-se que o estudo realizado possa ser útil em processos de migração de sistemas legados para sistemas orientados por aspectos, bem como para projetistas e usuários de ferramentas de refatoração automática de interesses transversais, na medida em que contribui para descrever e quantificar uma tarefa essencial ao funcionamento dessas ferramentas.

O restante deste artigo encontra-se organizado conforme descrito neste parágrafo. Na Seção 2, apresentam-se as transformações orientadas por objetos mais comuns para extração de aspectos, bem como alguns exemplos de uso dessas transformações em sistemas reais. A Seção 3 descreve os principais resultados de um estudo sobre a aplicação e a relevância dessas transformações em sistemas que já foram migrados para a tecnologia de aspectos. A Seção 4 discute trabalhos relacionados e a Seção 5 apresenta as conclusões do artigo.

2 Transformações de Código

No contexto deste artigo, 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 (e não de refatorações). Essa distinção entre os dois termos é adotada também em outros trabalhos [2, 3].

Neste artigo, serão consideradas apenas transformações em código transversal – código que implementa interesses transversais – de sistemas Java que encontra-se espalhado e entrelaçado em classes de sistemas implementados nessa linguagem. 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.

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* [2].

- 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* [2].

- 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 [3].

As transformações propostas se subdividem em dois grupos: movimentações de código e extração de métodos. Nas subseções seguintes, descrevem-se esses grupos de transformações.

2.1 Movimentações de Código

Em geral, as transformações deste grupo têm como objetivo mover um código transversal identificado para antes ou depois de um ponto que possa ser capturado por um descritor de conjuntos de junção de AspectJ. Uma listagem das dezessete transformações incluídas neste grupo é apresentada na Tabela 1.

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

	Descrição
1	Mover código para início de um método
2	Mover código para final de um método
3	Mover código para antes de um comando return
4	Mover código para antes de uma chamada de método
5	Mover código para depois de uma chamada de método
6	Mover código para antes de uma chamada de construtor
7	Mover código para depois de uma chamada de construtor
8	Mover código para início de um bloco catch
9	Mover código para final de um bloco catch
10	Mover código para antes de uma leitura de campo
11	Mover código para depois de uma leitura de campo
12	Mover código para antes de uma escrita em campo
13	Mover código para depois de uma escrita de campo
14	Mover código para início do bloco de inicialização estática de uma classe
15	Mover código para fim do bloco de inicialização estática de uma classe
16	Agrupar código transversal em um único bloco de código
17	Desagrupar código transversal entrelaçado em expressões

Tabela 1. Movimentações de código

Na Tabela 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`. 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`. 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`, associado a um adendo do tipo `before` ou `after returning`. 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`. 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`.

Por fim, 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 adendo. Já 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.

Exemplos: O exemplo mostrado na Figura 1 ilustra um caso de aplicação da Transformação 1 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 de Código Inicial de Método [3]. Na Figura 2, 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* no sistema JHotDraw. Com isso, habilita-se uma refatoração do tipo Extração de Código *Pre-Return* [3].

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

⇓

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

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

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

⇓

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

Figura 2. Exemplo de aplicação da Transformação 17 em um método do sistema JHotDraw

Limitações: 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. Suponha, por exemplo, a seguinte aplicação da Transformação 1:

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

As seguintes condições devem ser satisfeitas para que essa transformação possa ser aplicada: (i) `B` não deve alterar o estado que é acessado por `A`; (ii) `A` não deve alterar o estado que é requerido pela execução de `B`; (iii) `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`).

Quando essas condições não são satisfeitas, o programador pode optar por transformações classificadas neste artigo como de segunda ordem. 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 (as quais são classificadas como de primeira ordem). Transformações de segunda ordem em geral são transformações *ad hoc*, isto é, particulares para cada contexto de sistema e, por serem de difícil generalização, não são catalogadas neste artigo.

Na Figura 3 mostra-se um exemplo de aplicação de transformações de segunda ordem 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 bancos de dados. No entanto, pode-se observar que essa transformação de primeira ordem altera o fluxo original de execução do sistema: no código original, o `commit` é chamado apenas se não houver nenhuma exceção dentro do bloco `try`; 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, lançou-se mão da 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`.

```
try { ...
   tx.commit();                    // aspect
}
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();                    // aspect
   sess.close();
}
```

⇓

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

Figura 3. Exemplo de aplicação da Transformação 4, seguida da aplicação de uma transformação de segunda ordem, no sistema JAccounting

2.2 Extração de Método

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 [6]. 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 um ponto de junção, conforme sugerido pelas Transformações 1 a 17. Nesse caso, a idéia consiste em transformar parte do código não-transversal que ocorre antes ou após o código classificado como transversal em um método. Com isso, cria-se um ponto de junção que pode ser capturado por meio de um conjunto de junção do tipo `call`.

A Tabela 2 descreve as seis transformações incluídas neste grupo.

	Descrição
18	Extrair método com parte do código que antecede um código transversal
19	Extrair método com parte do código que sucede um código transversal
20	Extrair método com código que antecede e sucede um código transversal
21	Extrair método contendo bloco de comandos `try-catch-finally`
22	Extrair método contendo bloco de comandos `synchronized`
23	Extrair método quando `withincode` não é capaz de definir ponto junção
24	Extrair método com código usado em atribuição a variável local

Tabela 2. Extração de Métodos

Descrição: As Transformações 18, 19 e 20 sugerem converter em método parte do código que antecede e/ou sucede um código identificado como transversal. A Transformação 21 recomenda a extração de um método contendo os blocos de comandos especificados em um `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`. De forma similar, a Transformação 22 recomenda a extração de um método contendo o bloco de comandos especificado em um comando `synchronized`, de forma a 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. Suponha, por exemplo, o seguinte método (onde CCC designa um código transversal e NCCC um código não-transversal):

```
void foo() {
    ...
    bar(); CCC;        // aspect
    ...
    bar(); NCCC        // non aspect
}
```

Nesse exemplo, o designador `withincode` não é suficiente para delimitar apenas as chamadas de `bar` feitas em `foo` antes de um código transversal. A Transformação 23 sugere então extrair um método com a primeira chamada de `bar` e com o código transversal que a sucede.

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 que sempre será utilizado no lado direito de comandos de atribuição a uma determinada variável. Um adendo do tipo `around` associado a chamadas desse método pode ser então usado para capturar o valor da variável.

Exemplo: Mostra-se na Figura 4 um exemplo de aplicação da Transformação 19 no sistema JSpider. Após essa transformação, pode-se criar um conjunto de junção que capture a chamada do novo método associado a um adendo do tipo `before`.

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

⇓

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

private int setToMininum() {
  return INTERVAL_MIN;
}
```

Figura 4. Exemplo de aplicação da Transformação 19 para extração de método com parte do código que sucede um código transversal no sistema JSpider

Limitações: O código a ser extraído 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 a variáveis locais, uma solução recomendada é a aplicação da refatoração Substituição de Método por Método de Objeto [6]. 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.

3 Estudo de Caso

Apresentam-se nesta seção os resultados de um estudo realizado com o intuito de quantificar e discriminar as transformações realizadas no processo de extração de aspectos dos sistemas JSpider, JHotDraw e JAccounting. A Tabela 3 resume as transformações aplicadas nesses sistemas.

Para permitir a extração de aspectos responsáveis pela abertura, confirmação e cancelamento de transações no sistema JAccounting, foram necessárias 60 transformações para modularização desses três subinteresses, sendo que nos códigos para confirmação e cancelamento de transações foram necessárias a aplicação de transformações de segunda ordem. Ao todo, essas transformações foram responsáveis por gerar 45 pontos de junção no programa final (isto é, em média foram realizadas 1.33 transformações por ponto de junção). No entanto, o resultado final – após a aplicação das transformações e da criação de aspectos para modularização dos interesses mencionados – foi bastante satisfatório. Por exemplo, o grau de quantificação dos conjuntos de junção criados foi de 11.25, isto é, cada conjunto de junção captura em média 11.25 pontos de junção.

No sistema JHotDraw, foram aplicadas 18 transformações, sendo que 15 delas foram para desagrupar código transversal entrelaçado em expressões (Transformação 17). Já no JSpider, foram aplicadas 36 transformações, sendo que 31 delas corresponderam a extrações de métodos (Transformações 18, 19, 20, 22 e 23).

JSpider	
	Quant
Transf. 1	3
Transf. 18	4
Transf. 19	18
Transf. 20	3
Transf. 21	2
Transf. 23	1
Transf. 24	3
2a ordem	2
Total (T)	36
Pts. junção (P)	236
Cjs. junção (C)	236
T/P	0.15
P/C	1
Introduções	37

JHotDraw	
	Quant
Transf. 5	1
Transf. 16	1
Transf. 17	15
Transf. 18	1
2a ordem	0
Total (T)	18
Pts. junção (P)	86
Cjs. junção (C)	86
T/P	0.21
P/C	1
Introduções	56

JAccounting	
	Quant
Transf. 1	1
Transf. 4	15
Transf. 5	14
2a ordem	30
Total (T)	60
Pts. junção (P)	45
Cjs. junção (C)	4
T/P	1.33
P/C	11.25
Introduções	0

Tabela 3. Transformações nos sistemas JSpider, JHotDraw e JAccounting

O número de transformações por ponto de junção dos sistemas JSpider e JHotDraw varia de 0.15 a 0.21. Esses índices são bastante inferiores àquele do JAccounting. Acredita-se que este fato tenha relação direta com o tipo de interesse transversal modularizado. No JAccounting, modularizou-se um interesse transversal homogêneo, isto é, um interesse implementado por meio do mesmo código em diversos pontos do sistema [5]. Com isso, transformações foram extensivamente usadas para mover o código associado ao interesse para os mesmos pontos de junção nos métodos onde controle de transações

era requisitado. Como benefício, foi possível gerar conjuntos de junção com alto grau de quantificação. Já nos sistemas JHotDraw e JSpider, foram modularizados interesses transversais presentes no código de forma heterogênea, isto é, com comportamento diferente em cada ponto de junção. Em decorrência, o número de transformações foi menor, já que nesse caso não há benefício em padronizar os pontos de junção onde serão introduzidos código transversal em função da variação do código de um ponto para outro. Por outro lado, o grau de quantificação dos aspectos extraídos foi menor e foram empregadas um número bem maior de declarações intertipos.

Lições Aprendidas: As lições aprendidas no desenvolvimento deste estudo de caso são resumidas a seguir:

- Transformações no código orientado por objetos constituem um passo fundamental para o sucesso de processos de extração de aspectos de sistemas legados. No sistema com menor percentual de transformações em relação ao número de pontos de junção – JSpider – elas foram responsáveis pela geração de 15% dos pontos de junção da versão orientada por aspectos do sistema.

- As transformações descritas neste trabalho são mais freqüentes na refatoração de interesses transversais homogêneos, já que nesse caso é importante que o interesse seja implementado de forma consistente em todo o sistema. Deve-se ressaltar ainda que interesses homogêneos se beneficiam diretamente de uma implementação orientada por aspectos, pois tiram grande proveito dos recursos de quantificação característicos dessa tecnologia. Evidência desse fato é que, no estudo realizado, o sistema no qual aspectos proporcionaram os melhores resultados – JAccounting – foi exatamente aquele com o maior número de transformações. Essa evidência explica também a aparente contradição presente nos estudos de casos realizados pelos desenvolvedores da ferramenta AOP-Migrator, mencionada na Seção 1. No primeiro estudo, incluindo apenas o sistema JHotDraw, constatou-se que mais de 80% das refatorações não demandaram transformações de código [2]. Já no segundo, que além do sistema JHotDraw, incluiu também os sistemas JavaPetStore, JAccounting e JSpider, afirma-se que o número de transformações alcançou 40% do total de refatorações [3]. Porém, a justificativa para essa aparente contradição parece ser clara: o interesse refatorado no JHotDraw é heterogêneo, enquanto o interesse refatorado no JAccounting é homogêneo (e, portanto, demanda mais transformações).

- Mesmo que o código transversal seja corretamente identificado por um minerador de aspectos, acredita-se que seja uma tarefa complexa o desenvolvimento de uma ferramenta capaz de preparar automaticamente esse código, de forma que ele possa funcionar como entrada de uma terceira ferramenta de suporte a refatorações de interesses transversais. Um dos maiores entraves para essa automatização são as transformações de segunda ordem, as quais são dependentes do fluxo de controle e dos dados do programa alvo. Em alguns sistemas, essas transformações são aplicadas com grande freqüência. Por exemplo, no sistema JAccounting foi realizado o mesmo número de transformações de primeira e de segunda ordem.

4 Trabalhos Relacionados

Trabalhos relacionados com esta pesquisa podem ser classificados em dois grupos: ferramentas para extração de aspectos e catálogos de refatorações orientadas por aspectos. Descrevem-se a seguir as principais pesquisas desenvolvidas em cada um desses grupos.

Ferramentas para Extração de Aspectos: A motivação para realização deste trabalho surgiu de uma experiência de uso da ferramenta AOP-Migrator, a qual disponibiliza suporte a um conjunto pré-definido de refatorações para extração de aspectos [2, 3]. Essa experiência mostrou que com freqüência não é possível aplicar as refatorações implementadas por AOP-Migrator, já que as mesmas pressupõem pré-condições que nem sempre o código base de um sistema consegue atender. Necessita-se então de transformações no código base, conforme os próprios desenvolvedores de AOP-Migrator sugerem. Comparado com o trabalho e as sugestões desses desenvolvedores, considera-se que a principal contribuição deste artigo consiste em catalogar e quantificar o emprego dessas transformações. Procurou-se ainda analisar e correlacionar a necessidade de transformações com o tipo de interesse a ser refatorado (homogêneo ou heterogêneo) e com propriedades dos aspectos a serem extraídos (tais como grau de quantificação).

Catálogos de refatorações orientadas por aspectos: Monteiro e Fernandes descrevem um conjunto de refatorações para extração, reestruturação interna e generalização de aspectos [9, 10, 11]. Outro catálogo para extração de aspectos bastante utilizado foi proposto por Laddad [8]. Hannemann e Kiczales propõem uma abordagem diferente para refatorações orientadas por aspectos, que inclui a especificação de um conjunto de participantes abstratos e do mapeamento dos mesmos para componentes de um programa base [7]. Cole e Borba descrevem um conjunto de leis para auxiliar na formalização e, eventualmente, automatização de processos de extração de aspectos [4]. Os autores reconhecem que a maioria das leis propostas requerem pré-condições específicas e que a determinação de tais pré-condições constituiu a tarefa mais difícil da pesquisa realizada. Contudo, conforme afirmado na Seção 1, nenhum dos catálogos de refatorações de aspectos mencionados inclui uma investigação mais aprofundada sobre a preparação prévia que deve ser realizada no código base de um sistema antes de se iniciar a aplicação das refatorações propostas.

5 Conclusões

Neste artigo, procurou-se: (1) catalogar e descrever as principais transformações orientadas por objetos que podem ser aplicadas em sistemas legados, de forma a viabilizar a extração de aspectos; (2) exemplificar e quantificar o emprego dessas transformações em sistemas legados para os quais já foram disponibilizadas versões orientadas por aspectos e correlacionar as mesmas com propriedades do interesse refatorado e do aspecto extraído do código. Para atender ao primeiro objetivo, foram definidas 24 transformações, distribuídas em dois grupos (Movimentação de Código e Extração de Métodos). Para algumas transformações, foram identificados exemplos reais de aplicação. Para atender ao segundo objetivo, realizou-se um estudo empírico envolvendo três sistemas. As principais lições aprendidas neste estudo foram discutidas na Seção 3.

Como trabalho futuro, pretende-se elaborar uma descrição mais formal das transformações propostas. Pretende-se também desenvolver uma ferramenta que possa receber como entrada informações geradas por ferramentas de mineração de aspectos e apresentar as transformações de código necessárias para um processo de refatoração orientado por aspectos em sistemas legados. Com o apoio dessa ferramenta pretende-se ampliar o estudo de caso realizado, investigando o emprego de transformações de código em outros sistemas para os quais já foram disponibilizadas versões orientadas por aspectos. Com isso, espera-se descobrir novas transformações e também novas correlações entre transformações, interesses transversais e aspectos.

Agradecimentos: Este trabalho foi desenvolvido como parte de um projeto de pesquisa financiado pela FAPEMIG (processo CEX-817/05 - Edital Universal 2005). Gostaríamos de agradecer a David Binkley e a Mariano Ceccato pela disponibilização da versão orientada por aspectos dos sistemas JAccounting, JSpider e JHotDraw.

Referências

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

[2] 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.

[3] 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.

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

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

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

[7] 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.

[8] Ramnivas Laddad. Aspect-oriented refactoring. TheServerSide.com, 2003.

[9] 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)*, 2003.

[10] 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.

[11] 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.