Em direção a uma abordagem para separação de interesses por meio de Mineração de Aspectos e Refactoring

Vinicius C. Garcia*, Daniel Lucrédio†, Antonio F. do Prado
GOES – Grupo de Engenharia de Software
Departamento de Computação – Universidade Federal de São Carlos
Caixa Postal 676 – São Carlos, SP, Brasil
(vinicius, lucredio, prado)@dc.ufscar.br

and

Eduardo K. Piveta, Luiz C. Zancanella
Centro Tecnológico – CTC – Universidade Federal de Santa Catarina
Caixa Postal 1212 – Florianópolis, SC, Brasil
(kessler, zancanella)@inf.ufsc.br

and

Alexandre Alvaro, Eduardo S. de Almeida
Centro de Informática – Universidade Federal de Pernambuco
Caixa Postal 7851 – Recife, PE, Brasil
aa2, esa2(@cin.ufpe.br)

Abstract

This paper presents an approach to aid migration from object-oriented code, written in *Java*, to a combination of objects and aspects, using *AspectJ*. This approach supports the use of aspect mining, in order to identify possible crosscutting concerns to be implemented as aspects. The concerns, previously identified, are extracted from object-oriented code throw refactorings and encapsulated into aspects to obtain the new aspect oriented code. We present in this paper a collection of manual aspect-oriented refactorings to extract crosscutting concerns from object-oriented code.

Keywords: AspectJ, Aspect Mining, Refactoring, Aspect-Oriented Software Development

Resumo

Este artigo apresenta uma abordagem para auxiliar na migração de código orientado a objetos, escrito em *Java*, para uma combinação de objetos e aspectos usando *AspectJ*. A abordagem se apóia no uso de mineração de aspectos, de forma a identificar possíveis interesses multi-dimensionais a serem implementados como aspectos. Os interesses, previamente identificados, são extraídos do código orientado a objetos por meio de *refactorings* e encapsulados em aspectos para obter o novo código orientado a aspectos. É apresentado neste artigo uma coleção de *refactorings* orientados a aspectos para extrair interesses multi-dimensionais do código orientado a objetos.

Palavras-chaves: AspectJ, Mineração de Aspectos, *Refactoring*, Desenvolvimento de Software orientado a Aspectos

*Financiado pela Fundação de Amparo à Pesquisa do Estado da Bahia (FAPESB) - Brazil
†Financiado pela Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) - Brazil

1 Introdução

Diversas empresas são compelidas a migrar seus sistemas legados para novas linguagens ou procurar por novas formas para melhorar seus sistemas de software existentes. Isto normalmente é feito para reduzir custos de manutenção, aumentar a velocidade de desenvolvimento e melhorar a legibilidade dos sistemas.

Metodologias recentes de desenvolvimento de software utilizam a Orientação a Objetos (OO) em busca de um maior nível de reutilização, o que leva a um desenvolvimento de software mais rápido e a sistemas de melhor qualidade, facilitando futuras manutenções. O software desenvolvido usando a Programação Orientada a Objetos (POO) é mais fácil de manter porque oferece mecanismos para reduzir o acoplamento e aumentar a coesão dos módulos, melhorando a manutenibilidade.

Porém, mesmo utilizando uma metodologia OO e decompondo o sistema em um conjunto de objetos individuais, cada um representando uma funcionalidade específica, ainda existem funcionalidades que ficarão espalhadas por diferentes objetos. Isso acontece porque o paradigma OO possui algumas limitações [1], como, por exemplo, o entrelaçamento e o espalhamento de código com diferentes propósitos. Estas limitações levaram a diversas propostas na tentativa de melhorar a modularização das aplicações. Padrões de projeto [6] podem ser utilizados para atenuar essas limitações, porém não são suficientes.

O Desenvolvimento de Software Orientado a Aspectos (DSOA) [2] é um novo paradigma para o desenvolvimento de software que procura aumentar a modularidade, reduzindo os problemas com entrelaçamento e espalhamento de código, tornando a estrutura do software mais clara e de mais fácil manutenção [2]. O DSOA pode ser aplicado para a implementação de vários interesses multi-dimensionais (*crosscutting concerns*) por meio de uma unidade modular chamada de aspecto. Como exemplo destes interesses pode-se citar: interação entre objetos, segurança, persistência, distribuição e tratamento de exceções. Outro conceito introduzido pelo DSOA são os pontos de combinação (*join points*), que especificam os elementos que serão modificados pelos aspectos. Os aspectos encapsulam a implementação dos interesses multi-dimensionais e se referem a um conjunto de pontos de combinação.

Na literatura pode-se encontrar diversas propostas [4, 11, 15] para suportar a identificação de aspectos em estágios iniciais do processo de desenvolvimento de software. Embora estes aspectos a serem identificados possam ser encaixados como *"late aspects"* (ao contrário dos aspectos na fase de requisitos, denominados de *"early aspects"*), eles tornam-se necessários na definição de requisitos quando na migração de um sistema legado ou na adoção de aspectos em um projeto em andamento.

O objetivo de migrar sistemas OO para orientado a aspectos (OA) inclui, entre outros benefícios, a melhora no entendimento do sistema aumentando assim a manutenibilidade e dando uma maior garantia para a sua evolução. A identificação dos candidatos a aspectos requer primeiramente uma clara idéia de quais aspectos procura-se encontrar. Assim, um estudo sobre os aspectos genéricos ou específicos de um domínio, é um pré-requisito para a mineração de aspectos. Esse estudo pode-se iniciar a partir dos interesses multi-dimensionais como, por exemplo, a busca por pontos de combinação estáticos como, por exemplo, chamada e execução de métodos.

Em paralelo, *Refactoring* [3] é uma técnica utilizada para reestruturar código OO de uma forma disciplinada. A intenção do *refactoring* é aprimorar a a legibilidade e a compreensão do código OO. A maioria das tecnicas de *refactoring* se propõem a aumentar a modularidade e eliminar a redundância do código. Considerando que as mesmas vantagens são obtidas por meio do DSOA, parece ser natural aplicá-los no mesmo processo de desenvolvimento. O benefício de usar ambas as abordagens é considerável. *Refactoring* pode ajudar a reestruturar o código OO segundo o paradigma OA. Neste contexto, torna-se viável migrar sistemas OO para OA.

Neste artigo é proposta a utilização de técnicas de DSOA combinadas com Mineração de Aspectos e *Refactorings* para possibilitar a evolução de sistemas OO por meio da separação de interesses multi-dimensionais em aspectos. Sendo assim, o artigo está organizado da seguinte forma: na seção 2 faz-se uma apresentação sobre os conceitos do Desenvolvimento de Software Orientado a Aspectos. Na seção 3 mostram-se os avanços nas pesquisas sobre Mineração de Aspectos e os principais trabalhos nesta área. Na seção 4, além dos conceitos, definem-se alguns *refactorings* para auxiliar na extração de interesses multi-dimensionais de códigos OO. E, na seção 5, apresentam-se as considerações finais acerca dos resultados obtidos.

2 Desenvolvimento de Software Orientado a Aspectos

Na Engenharia de Software, decompor um sistema em pequenas partes é uma forma essencial de reduzir a complexidade e garantir a sua evolução. Esta decomposição resulta na *"separação de interesses"* (do inglês *separation of concerns*) e facilita o trabalho paralelo, a especialização da equipe de desenvolvimento, a localização de pontos que devem sofrer modificação e consequentemente auxilia na manutenção, testes sistemáticos e na garantia da qualidade [4].

Infelizmente, algumas funcionalidades existentes nos sistemas de software, como tratamento de erros ou segurança, são inerentemente difíceis de se decompor e isolar, reduzindo assim a legibilidade e a manutenibilidade destes sistemas.

Existem diversas formas de contornar estes problemas usando técnicas OO como a utilização de padrões de projetos [6]. Para poucas classes e/ou poucos objetos isto funcionaria adequadamente. Entretanto, a medida que o número de objetos aumenta, cresce também a quantidade de memória, do computador, necessária e o número explícito de associações entre os decoradores e os objetos decorados.

O Desenvolvimento de Software Orientado a Aspectos (DSOA), surgiu nos anos noventa como um paradigma direcionado a implementar interesses multi-dimensionais(*crosscutting concerns*) ou mais especificamente aspectos, por meio de técnicas de geração de código para combinar (*weaver*) aspectos na lógica da aplicação [2]. A separação dos interesses multi-dimensionais proporcionada pelo DSOA busca resolver os problemas de espalhamento e entrelaçamento de código encontrados em sistemas OO.

De acordo com os princípios da OA, aspectos modificam componentes de software através de mecanismos estáticos e mecanismos dinâmicos. Os mecanismos estáticos preocupam-se com a adição de estado e comportamento nas classes, enquanto que os mecanismos dinâmicos modificam a semântica destas em tempo de execução. Estes aspectos são implementados como módulos separados, de forma que fique transparente a maneira como os aspectos agem sobre os componentes funcionais, e como estes o fazem [5].

Para a definição e implementação destes mecanismos, uma abordagem OA normalmente provê, em adição às linguagens tradicionais (OO ou estruturadas) uma linguagem para a definição de aspectos e um combinador de aspectos (*aspect weaver*).

AspectJ[1] é uma linguagem de programação OA de apoio ao DSOA por meio de novas construções para implementar os interesses multi-dimensionais que utiliza a linguagem *Java* como base. Além dos mecanismos relativos a OO (classes, métodos, atributos etc) também existem mecanismos relacionados a implementação de aspectos, tais como: pontos de corte (*pointcuts*), pontos de combinação (*joinpoints*), *advices* e introduções (*introductions*).

Atualmente, como o DSOA está entrando em uma fase de inovações, novos desafios vão surgindo enquanto a tecnologia vai sendo adotada e estendida. As versões das linguagens do DSOA como *AspectJ*, as contribuições de diversos grupos de pesquisa[2] e a recente integração com servidores de aplicação como *JBOSS*[3] e *BEA's WebLogic*[4] demonstram o crescimento da popularidade do DSOA.

3 Mineração de Aspectos

Várias preocupações relativas ao uso do DSOA são levantadas, como, por exemplo, o risco de obter um código mais entrelaçado, conhecido como *"código spaghetti"*, pela não utilização correta dos conceitos da OA e o uso indiscriminado de abordagens *ad-hoc* para o projeto de sistemas, pelo fato de não haver um padrão definido para expressar a modelagem de sistemas OA. Estas preocupações levam a seguinte questão: *Quando o DSOA é necessário e quando somente a POO resolve o problema ?.*

Neste contexto, nós acreditamos que a mineração de aspectos pode ajudar a resolver esta questão.

Técnicas de mineração de software ajudam a encontrar informações valiosas no código de um sistema, tornando estas informações disponíveis aos engenheiros de software envolvidos na evolução daquele sistema. Um bom exemplo de mineração de software é a *extração de regras de negócio.*

A mineração de software é apoiada por técnicas de exploração de software [7]. Ela envolve três passos: (1) coletar dados do código fonte, (2) inferir conhecimento com base na abstração dos dados coletados, e (3) apresentar a informação usando, por exemplo, hipertexto ou outros recursos para visualização.

Para realizar a mineração de aspectos existe a necessidade de fazer um *parsing* dos programas OO e verificar os locais nos quais existam códigos duplicados, difusos ou referentes a diversas preocupações de projeto.

Considere uma implementação do padrão de projetos *Observer* utilizando a linguagem *Java* como um exemplo de mineração utilizando pontos de combinação, no qual a classe *Termometer* faz o papel de *Observer* e a classe *Celcius* faz o papel de *Subject.*

A classe *Observer*:

```
1   public abstract class Termometer{
2      private Subject subject = null;
3      private Celcius tempSource;
4      // getter and setter methods
5      public abstract void drawTemperature();
```

[1] http://eclipse.org/aspectj
[2] http://aosd.net
[3] http://www.jboss.org
[4] http://www.bea.org

```
6       public void update() { drawTemperature(); }
7   }
```

A classe *Subject*:

```
  import java.util.Vector;
1   public class Celcius implements Subject{
2     private double degrees;
3     private Vector observers = new Vector();
4     public Object getData() { return this; }
5     public double getDegrees(){ return degrees; }
6     public void setDegrees(double aDegrees){
7       degrees = aDegrees;
8       for (int i=0;i<getObservers().size();i++){ ((Observer)getObservers().elementAt(i)).update(); }
9     }
10    public void add(Observer obs) {
11      observers.addElement(obs);
12      obs.setSubject(this);
13    }
14    public void remove(Observer obs) {
15      observers.removeElement(obs);
16      obs.setSubject(null);
17    }
18    public Vector getObservers(){ return observers; }
19    Celcius(double aDegrees){ setDegrees(aDegrees); }
20  }
```

Um observador concreto, chamado de *FahrenheitTermometer*:

```
1  public class FahrenheitTermometer extends Termometer{
2    public void drawTemperature(){
3      System.out.println("Temperature in Fahrenheit:" + (1.8 * getTempSource().getDegrees())+32); }
4  }
```

E um programa de teste:

```
1  public class TestTemperatures{
2    public static void main(String a[]){
3      Celcius c = new Celcius(10);
4      Termometer termo = new CelciusTermometer();
5      Termometer termo1 = new FahrenheitTermometer();
6      termo.setTempSource(c);
7      termo1.setTempSource(c);
8      c.setDegrees(100);
9      c.add(termo);
10     c.setDegrees(200);
11     c.setDegrees(300);
12     c.add(termo1);
13     c.setDegrees(400);
14     c.setDegrees(500);
15   }
16 }
```

Uma solução seria a ferramenta de mineração verificar a ocorrência de pontos de combinação estáticos e, a partir de então, gerar aspectos de forma a rastrear estas ocorrências. Logo, para cada classe selecionada pelo usuário um aspecto é gerado, bem como pontos de corte e *advices* para interceptar chamadas, execuções de métodos, leitura e escrita de atributos ou quaisquer informações selecionadas pelo usuário. O código a seguir mostra um aspecto gerado para a classe *Celcius* sempre que uma chamada ao método `setDegrees()` é realizada.

```
1  aspect MiningCelciusSetDegrees{
2    pointcut stateChanges(Subject s): target(s) && call(void Celcius.setDegrees(..));
3      after(Subject s): stateChanges(s){
4        System.out.println("State changed at" + thisJoinPoint.getSourceLocation());
5      }
6  }
```

Ao combinar e rodar o programa `TestTemperatures` é informada na saída padrão a localização dos pontos de combinação especificados, conforme o código abaixo. Esta saída pode ser configurada de forma a gerar arquivos a serem analisados pela ferramenta de mineração. Neste exemplo, a chamada ao método `setDegrees` têm grandes chances de ser uma das chamadas que encontra-se mais espalhada pela aplicação. O engenheiro pode tentar extrair as chamadas a este método para um aspecto de forma a visualizar as mudanças de projeto e implementação resultantes desta decisão de projeto.

```
State changed at Celcius.java:34
State changed at TestTemperatures.java:13
Temperature in Celcius:200.0
State changed at TestTemperatures.java:18
Temperature in Celcius:300.0
State changed at TestTemperatures.java:19
Temperature in Celcius:400.0
Temperature in Fahrenheit:720.032
State changed at TestTemperatures.java:22
Temperature in Celcius:500.0
Temperature in Fahrenheit:900.032
State changed at TestTemperatures.java:23
```

As transformações requeridas para a geração de aspectos usando os resultados da mineração podem ser consideradas similares a *refactorings*, de forma a obter um projeto melhor. Uma *refactoring* denominada *Extract Aspect* pode ser definida de forma a tentar extrair estes fragmentos em uma única abstração.

As ferramentas existentes de mineração de aspectos [8, 9, 11] foram desenvolvidas com o objetivo de encontrar termos comuns no léxico (mineração textual) e utilizando expressões regulares baseadas em tipos. Entretanto, existem problemas com estas abordagens, que serão discutidos a seguir.

3.1 Aspect Mining Tool

A ferramenta denominada *Aspect Mining Tool* (AMT) [8] é um aplicativo que realiza a analise do código-fonte existente na tentativa de identificar e extrair código relacionado a interesses que estão espalhados através das classes do sistema. Para este trabalho, foi utilizada a versão 0.6a da ferramenta[5].

De acordo com [8], duas formas de mineração de interesses podem ser utilizadas para encontrar elementos que pertencem a um mesmo interesse multi-dimensional:

Mineração baseada em texto: A mineração de dados baseada em texto procura por padrões no código fonte utilizando os nomes de classes, métodos e atributos como base. Ela é interessante caso sejam utilizadas convenções de nomeação para os elementos do sistema. A não utilização de um padrão para tal pode dificultar no processo de identificação de interesses multi-dimensionais.

Mineração baseada em tipos: A mineração através de tipos procura pela ocorrência dos diversos tipos (i.e. classes) definidos no aplicativo a ser analisado. Ela permite encontrar pontos em um sistema nos quais o acoplamento e a coesão possam deixar a desejar. Um único módulo que utiliza alguns poucos tipos é candidato a possuir uma coesão alta e baixo acoplamento, por exemplo. A mineração baseada em tipos funciona melhor que a mineração baseada em texto quando não existem convenções de nomeação. Ela não é muito interessante quando instâncias de um mesmo tipo são utilizadas para diferentes propósitos.

Apesar dos interesses selecionados estarem dispersos nas classes da aplicação, isto não significa que sejam potenciais aspectos. Isto leva a consideração de que uma forma interessante de realizar a mineração é a utilização dos pontos de combinação disponíveis nas linguagens OA mais utilizadas. Logo, além da procura por texto e por tipo, é válida a procura através de chamadas e execuções a métodos (ou padrões de métodos), leitura e escrita de atributos, utilização de construtores etc.

Outras limitações incluem, por exemplo, o fato de que os relacionamentos de herança não são levados em consideração na mineração baseada em tipos e que não é utilizado nenhum mecanismo de inferência para sugerir ao usuário problemas de modularização. Ela apenas informa as ocorrências dos tipos ou de padrões de texto dentro das classes. Uma funcionalidade interessante seria a extração desses trechos de código em um aspecto.

3.2 FEAT

A ferramenta FEAT (*Feature Exploration and Analysis Tool*) tem por objetivo auxiliar na identificação de interesses multi-dimensionais. Para isso ela possibilita a criação de grafos de interesses [9], os quais permitem a definição de um conjunto de interesses formados por classes e fragmentos de código que são adicionados a medida que o usuário interage com o sistema. Ela é implementada como um *plugin* para o *eclipse*[6] e permite a utilização das ferramentas do ambiente enquanto tenta-se melhorar a modularização do sistema. Para esta seção, foi utilizada a versão 2.5.0 da ferramenta[7].

[5] `http://www.cs.ubc.ca/~jan/amt/`
[6] `http://eclipse.org`
[7] `http://www.cs.ubc.ca/labs/spl/projects/feat/`

A ferramenta é utilizada basicamente para localizar, descrever e analisar um interesse em um sistema implementado em *Java*. No entanto, nada impede que os conceitos utilizados pela ferramenta não possam ser estendidos para outras linguagens. É possível explicitar dependências entre os elementos em cada interesse, bem como comparar fragmentos presentes em mais de um interesse.

A idéia é produzir e analisar descrições de código que implementam interesses multidimensionais em código. O uso dos grafos de interesses permite a utilização de uma representação abstrata que pode ser mapeada para código. A vantagem desta abordagem é a utilização de uma representação de mais alto nível do que o código-fonte da aplicação. As representações geradas durante o processo de identificação e análise de interesses podem ser utilizadas como base para possíveis *refactorings* visando encapsular estes interesses em construções de uma linguagem OA.

A ferramenta utiliza *bytecodes* para extrair os relacionamentos estruturais dos elementos dos programas: classes, métodos e atributos. Para isso, o *Jikes Bytecode Toolkit* (da IBM) é utilizado para manipulação destes elementos em tempo de execução.

Entretanto, existem diversas limitações encontradas na ferramenta, que podem dificultar a sua utilização e adoção. A primeira delas refere-se aos conceitos utilizados. Ao invés de basear-se em conceitos conhecidos através da utilização de linguagens OA, o uso da ferramenta exige o domínio de uma série de conceitos adicionais [10], dificultando sua utilização.

Embora pressuponha-se que o usuário esteja ciente dos conceitos relacionados a ferramenta, seu uso não ocorre de maneira intuitiva, sendo necessários diversos passos de forma a ter resultados úteis para a análise dos interesses da aplicação. Melhorias poderiam ser realizadas de forma a utilizar as relações de interesse disponíveis nos pontos de combinação das linguages OA, como por exemplo o modelo de pontos de combinação de *AspectJ*. Tal melhoria pode facilitar na adoção em escala da ferramenta, bem como sua integração com os *plugins* do próprio *AspectJ*.

Em relação a usabilidade da FEAT, existem alguns pontos a desejar: (a) sempre que o usuário desejar anotar um trecho de código e adicioná-lo as relações de interesses existentes, é necessário mudar de perspectiva no ambiente do *eclipse*, (b) os elementos dos interesses não podem ser movimentados de um interesse para outro.

Outro fato importante é que a busca de elementos ocorre através da sintaxe dos elementos e não da sua semântica associada. Os problemas que isto pode acarretar são semelhantes aos tratados na seção anterior.

3.3 JQuery

A ferramenta *JQuery* [11] é um navegador de código *Java* desenvolvido como um *plugin* do *eclipse*. A linguagem é baseada em consultas, permitindo que o usuário defina o que deseja visualizar. Estas consultas são escritas através de predicados lógicos, semelhante a outras linguagens do paradigma lógico de programação (como *Prolog*). Nesta seção, foi utilizada a versão 2.0b da ferramenta[8].

A linguagem de consulta [12] é baseada em uma linguagem de programação lógica chamada *TyRuBa*[9], que é implementada em *Java*. O usuário pode escolher entre escrever suas próprias consultas ou utilizar uma das consultas existentes na ferramenta. As consultas trabalham por meio da manipulação de uma tabela de fatos baseada na árvore sintática abstrata do espaço de trabalho selecionado pelo usuário. Esta manipulação é feita utilizando os recursos de *parsing* do próprio *eclipse* (usando a *Java Development Tools* - JDT).

Um dos problemas relacionados a *JQuery* é dificuldade de utilização. Por exemplo, o predicado:

```
method(?M), name(?M,?name), child(?C,?M),
package(?C,?P), re_match(/^a/,?name)
```

retorna uma árvore contendo os métodos (dentro de suas respectivas classes) que começam com o caractere "a". Apesar de poder representar, de maneira sucinta, predicados como estes, é difícil elaborar estes predicados. Para tal, uma experiência de programação lógica é desejável de forma a realizar consultas mais complexas.

Em certos casos, é mais simples utilizar um protocolo de meta-objetos para obter informações sobre a estrutura de uma aplicação, de um pacote ou de um conjunto de classes. Dependendo da representação utilizada, outras linguagens de consulta seriam mais simples de serem utilizadas. Se fosse utilizada uma representação em XML do código fonte da aplicação na qual estão sendo realizadas as consultas, poderia ser utilizada *XPath* ou *XQuery*, permitindo a realização de consultas (*queries*) complexas de maneira mais simples e padronizada, utilizando uma linguagem de ampla utilização.

Apesar disto, após aprender como escrever suas próprias consultas o usuário pode beneficiar-se da possibilidade da criação de predicados arbitrários. Outro ponto importante é a existência de um conjunto de consultas pré-definidas, o que pode auxiliar na tarefa de aprendizado da linguagem de consulta.

[8] http://www.cs.ubc.ca/labs/spl/projects/jquery/
[9] http://www.cs.ubc.ca/labs/spl/projects/tyruba/

3.4 AspectJ Plugins

Um dos pontos importantes para a ampla adoção de *AspectJ* como a linguagem OA mais utilizada é a preocupação com o suporte ferramental necessário para que a linguagem possa ser efetivamente usada pela indústria de desenvolvimento de software [13, 14]. Inicialmente o suporte existia para diversos ambientes, como: *JBuilder*, *Net Beans*, *EMacs* etc. Atualmente, o foco principal é no desenvolvimento de *plugins* para o *eclipse*, um ambiente de desenvolvimento da IBM.

Estes *plugins* permitem ao usuário o desenvolvimento de software orientado a aspectos de forma semelhante ao que já é feito para projetos OO. Algumas funcionalidades providas pelos *plugins* abrangem:

- *Highlight* de palavras-chave de *AspectJ* no editor *Java*;
- Assistentes para criação de projetos usando *AspectJ* e para a criação de aspectos, análogos aos existentes para a criação de projetos OO e para a criação de classes, respectivamente;
- Visualização gráfica (através de um *Outline*) dos pontos de atuação e dos *advices* de um aspecto, bem como a visualização dos pontos de combinação que podem ser estaticamente determináveis e são afetados por um *advice*;
- Suporte a bibliotecas de aspectos (através de arquivos .jar) e de combinação de aspectos binária (usando manipulação de bytecodes com BCEL) e de múltiplos arquivos de build (usando a ferramenta *Ant* desenvolvida pelo *Apache Group*);
- Visualizador gráfico que permite verificar o grau de dispersão dos interesses multi-dimensionais ao longo das classes da aplicação; e
- Suporte a depuração de aspectos (além da depuração normal de classes provida pelo *eclipse*)

Uma das funcionalidades que auxiliam na utilização da linguagem *AspectJ* é permitir, através dos *plugins*, a visualização de quais métodos são afetados pelos *advices* definidos em um aspecto. Ao utilizar os *plugins* de *Aspectj*, têm-se informações relativas aos aspectos e aos pontos nos quais os *advices* terão efeito no código da aplicação.

O contrário também é possível. Ao examinar uma classe, podem ser visualizados em que locais os aspectos existentes na aplicação afetam estas classes. A partir de um menu de contexto, têm-se informações relativas ao aspecto e ao *advice* que afetam a classe a ser explorada.

Junto com estes *plugins*, existe ainda um visualizador que permite ao usuário constatar a dispersão do interesse sendo implementado como um aspecto.

Contudo, ainda existem alguns pontos fracos encontrados nos *plugins*. O primeiro deles remete a funcionalidade de *sintax highlight*, a qual ainda não funciona adequadamente em conjunto com os arquivos *Java*. Outra funcionalidade que perde um pouco de seu poder é o *code completion*, o qual não funciona no mesmo nível de qualidade encontrado ao manipular arquivos *Java* no ambiente. Apesar de bastante útil, o visualizador carece de melhorias em sua apresentação e otimização, de forma a melhorar a interação com o usuário.

Outro ponto a ser levado em consideração, quando na utilização dos *plugins*, diz respeito a utilização de *refactorings* nas classes afetadas por aspectos. Como os pontos de combinação em *AspectJ* podem ser especificados através de coringas ou de padrões de nomes, as *refactorings* do ambiente podem modificar os pontos de combinação afetados pelos aspectos da aplicação, tanto removendo quanto adicionando novos pontos (dependendo da *refactoring* utilizada).

Existem ainda restrições referentes a depuração: *breakpoints* podem ser colocados somente em classes (apesar de poder executar o código de aspectos usando execução passo a passo). Outra limitação, fruto da inexistência de suporte a *JSR 45*[10] é que não é possível atualmente percorrer um *advice* do tipo *around* usando execução passo a passo.

4 *Refactorings*

A utilização de *refactorings* na fase de desenvolvimento e na manutenção de software é uma prática que há muito tempo vem sendo utilizada por programadores, que sempre procuraram *"limpar"* seu código para torná-lo mais legível e para otimizar o desempenho.

O objetivo das técnicas de *refactoring* é realizar mudanças no código concluído, e operacional, para torná-lo mais legível e passível de modificações e/ou melhorar qualidades não funcionais do sistema, como, por exemplo, a performance. Para isto, a *refactoring* se apóia em técnicas bem planejadas com o objetivo

[10] *Java Specification Requests #45: Debugging Support for Other Languages,* `http://www.jcp.org/en/jsr/detail?id=45`

de diminuir o risco da inserção de erros no sistema. Desta forma, a *refactoring* pode ser definido como *"um conjunto de técnicas bem planejadas que tem o objetivo de melhorar a capacidade do sistema de se adaptar a novas necessidades e acomodar novas funcionalidades"* [3].

Refactorings são sistematicamente organizadas em catálogos, de um modo análogo aos padrões de projeto [6]. A aplicação de *refactorings* é aconselhada quando são identificados no código de um sistema trechos, denominados *"code smells"*, que requerem melhorias.

Refactoring e mineração de aspectos devem ser definidos e aplicados com algumas metas em mente como, por exemplo, na melhora da manutenibilidade do código e/ou da sua legibilidade ou extensibilidade. Conseqüentemente, antes de executar a transformação do código fonte que isolaria os candidatos a aspectos do código e o encapsularia de acordo com o paradigma OA, o engenheiro de software precisa avaliar o valor dessa migração com respeito a estas metas.

4.1 *Refactorings* para o paradigma OA

O engenheiro de software utiliza os *refactorings* para extrair os interesses que estão espalhados pelo sistema, dispersos por métodos e classes. Em algumas pesquisas [15, 16], os *refactorings* existentes são adaptados para realizar a organização de código OO, extraindo aspectos.

Os *refactorings* apresentados aqui são propostos a partir da experiência dos autores em DSOA e na realização de estudos de caso. É importante ressaltar que estes *refactorings* não abrangem todos os problemas e nem pretendem ser a solução completa para a extração de interesses multi-dimensionais do código OO, mas representam o primeiro passo para a criação de um catálogo com este objetivo.

Para facilitar a compreensão na especificação dos *refactorings* foi escolhido o mesmo formato utilizado por Fowler em [3]. Será utilizado também, assim como Fowler, um único código no exemplo de aplicação dos *refactorings*. A especificação compreende os seguintes elementos: o nome do *refactoring*; uma descrição da situação na qual ele é necessário e o que ele propõe; uma descrição da ação recomendada; pré-condições para sua aplicação, quando existir; o mecanismo de aplicação do *refactoring*; e, um código de exemplo.

O primeiro *refactoring* apresentado é o principal e está em um nível de abstração maior que os demais. Todos os outros *refactorings* relacionados com a extração de elementos ou de trechos de código do sistema OO estão inseridos neste primeiro aspecto.

Extract Aspect

Situação

Existe um interesse espalhado e entrelaçado pelo código de diversas classes e métodos do sistema. Pretende-se separar este interesse do código primário do sistema.

Ação Recomendada

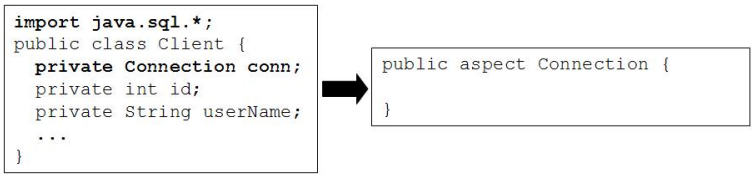
Extrair todo o código relacionado ao interesse das classes e métodos do sistema para um aspecto.

Mecanismo

- Criar um aspecto, levando em consideração o pacote em que o aspecto será criado. Caso necessário utilizar o comando *"import"*.
- Mover para o aspecto os atributos relativos ao interesse por meio do *Extract Field*. É necessário que os atributos, inicialmente, sejam declarados como públicos no aspecto até o novo código pós-aplicação deste *refactoring* ser compilado e testado.
- Mover para o aspecto os métodos relativos ao interesse por meio do *Extract Method*.
- Mover para o aspecto, como *advices*, todo trecho de código relativo ao interesse, que não seja um método por meio do *Extract Advice*.
- Modificar para *"private"* todas as declarações de membros do aspecto que devem ser visíveis somente no aspecto.
- Compilar o código e testar.

Exemplo

Como este *refactoring* está num nível de abstração maior que os demais, o exemplo irá ilustrar apenas a criação do aspecto para implementar um determinado interesse multi-dimensional, no caso o interesse de persistência.



Extract Field

Situação

Um atributo de uma classe está relacionado a um determinado interesse, e este está implementado ou foi extraído para um aspecto.

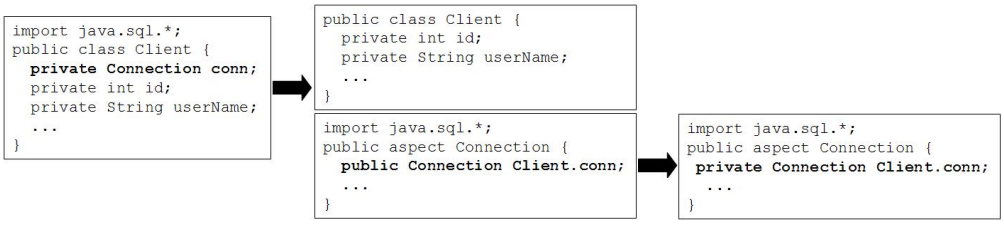
Ação Recomendada

Mover o atributo para o aspecto que implementa o interesse, como uma declaração de introdução.

Mecanismo

- Se o atributo for público, considerar o uso do *Encapsulate Field* ([3], p.206) antes da aplicação deste *refactoring*.
- Mover a declaração do atributo da classe para o aspecto, incluindo a declaração de valor inicial, caso exista.
- Adicionar o nome da classe e o "." antes do nome do atributo na declaração de introdução.
- Analisar se um novo *"import"* deve ser declarado no aspecto.
- Mudar para público a visibilidade do atributo. O valor pode voltar a ser privado assim que todo código referente ao atributo for extraído para o aspecto. Se for necessário manter o código relacionado ao atributo na classe, considerar o uso do *Self Encapsulate Field* ([3], p.146).
- Checar todos os pontos de corte com a declaração *within()* que devem ser atualizados após a aplicação do *refactoring*.
- Compilar o código e testar.

Exemplo



Extract Method

Situação

O método de uma classe está relacionado a um determinado interesse, e este está implementado, ou foi extraído para um aspecto.

Ação Recomendada

Mover o método para o aspecto que implementa o interesse, como uma declaração de introdução.

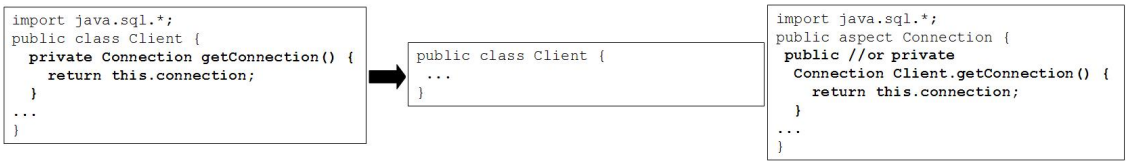
Pré-condições

Em [17], Monteiro especifica algumas pré-condições para a aplicação deste *refactoring*, principalmente se o método for público.

Mecanismo

- Mover a declaração do método da classe para o aspecto.
- Adicionar o nome da classe e o "." antes do nome do método na declaração de introdução.
- Se a visibilidade do método não for pública, mudar temporariamente para pública. Após todo código relativo ao método ser extraído para o aspecto, retornar à visibilidade ao valor original.
- Analisar se um novo *"import"* deve ser declarado no aspecto.
- Checar todos os pontos de corte com a declaração *within()* que devem ser atualizados após a aplicação do *refactoring*.
- Compilar o código e testar.

Exemplo



Extract Advice

Situação

Um trecho de código de um método está relacionado a um interesse e precisa ser extraído para o aspecto que implementa este interesse.

Ação Recomendada

Criar um ponto de corte que capture o ponto de combinação relacionado ao trecho de código, e mover o trecho de código para um *advice* apropriado.

Pré-condições

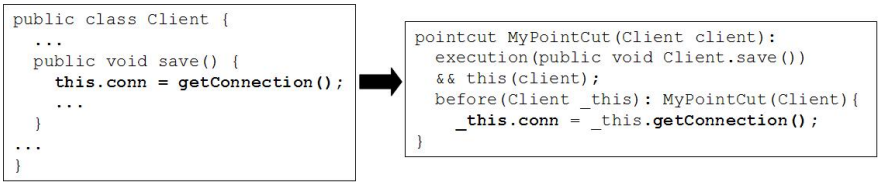
Antes de fazer a extração dos trechos de código da classe para o *advice* no aspecto, deve ser feita uma cuidadosa análise do corpo do método, para especificar o ponto de corte que irá capturar o(s) ponto(s) de combinação adequado(s). Caso o código primário não ofereça nenhum ponto de combinação adequado, outros *refactorings* podem ser aplicados para corrigir este problema. Mais detalhes sobre as pré-condições deste *refactoring* podem ser encontradas em [16] e [17] que especificam *refactorings* semelhantes a este.

Mecanismo

- Criar um ponto de corte que capture o conjunto de pontos de combinação. Se o ponto de corte já existir, então deve-se estendê-lo para incluir o ponto de combinação relacionado ao trecho extraído.
- Verificar se o ponto de corte captura todo contexto requerido pelo trecho de código e se o trecho extraído faz referência a declarações de *"this"* ou *"super"*. Os casos mais comuns inluem o uso da declaração de ponto de corte *target()* combinado com *call()*, ou o uso de *this()* combinado com *execution()*, *set()* ou *get()* [17].

- Criar o *advice* adequado para o ponto de corte, sem implementação do corpo.
- Mover o código a ser extraído para o corpo do *advice*.
- Adicionar código necessário para implementar o contexto do *advice*.
- Substitur referências às próprias variáveis *"this"* pela variável capturada no contexto do ponto de combinação.
- Analisar o código extraído em busca de referências a variáveis locais no escopo do método, incluindo parâmetros e variáveis locias. Declarações a quaisquer variáveis temporárias, usadas somente no trecho extraído, podem ser substituídas no corpo do *advice*.
- Compilar o código e testar.

Exemplo



Rename Pointcut

Situação

O nome do ponto de corte não revela seu propósito.

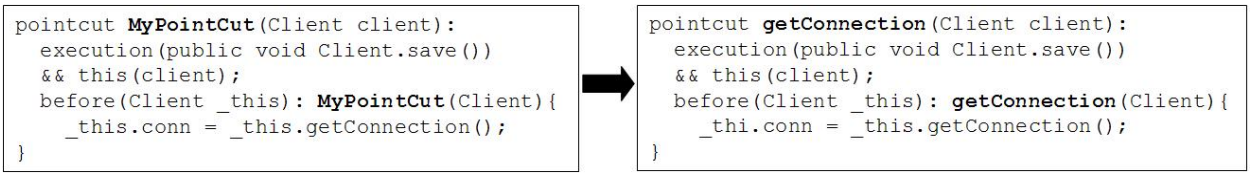
Ação Recomendada

Mudar o nome do ponto de corte.

Mecanismo

- Criar um novo ponto de corte com o novo nome. Copiar o código dos *advices* do ponto de corte com o nome antigo, para o novo ponto de corte e fazer as alterações adequadas, como, por exemplo, atualizar o nome do ponto de corte nos *advices*.
- Checar se existe algum código relacionado ao ponto de corte. Caso exista, modificar as referências ao antigo nome, para o novo.
- Remover o antigo ponto de corte.
- Compilar o código e testar.

Exemplo



Outros *refactorings*

Outros *refactorings* podem ser criados para solucionar problemas não previstos. Os trabalhos de Monteiro e Fernandes [17] e o de Hanenberg et al. [16] especificam alguns *refactorings* semelhantes e outros que complementam os que foram apresentados.

5 Conclusões

Pesquisas em Mineração de Aspectos preocupam-se com o desenvolvimento de conceitos, princípios, métodos e ferramentas para apoiar a identificação de aspectos em sistemas OO, e com o seu subsequente *refactoring* para sistemas OA.

Neste artigo foi apresentado o estado da arte na Mineração de Aspecto, indicando promissoras direções nesta área de pesquisa. As principais ferramentas existentes foram analisadas, identificando as vantagens e as desvantagens. Os resultados desta análise podem ser utilizados como base para a implementação de uma nova ferramenta ou técnica de Mineração de Aspectos em sistemas OO.

Técnicas de *refactoring* foram adaptadas para apoiar na extração de aspectos de códigos OO. Neste artigo não foi possível descrever todos os *refactorings* criados pelos autores, porém os mais importantes foram apresentados. O objetivo é fornecer um catálogo *online*, semelhante ao criado por Fowler[11], para apoiar os processos de migração de sistemas OO para OA.

O próximo passo é estudar formas de se automatizar a aplicação de alguns dos *refactorings* propostos por meio de transformações de software.

Referências

[1] Ossher, H. and Tarr, P. Using subject-oriented programming to overcome common problems in object-oriented software development/evolution. *Proceedings of 21st International Conference on Software Engineering (ICSE'99)*, 1999.

[2] Kiczales, G. et a. Aspect-Oriented Programming. *Proceedings of the 11st European Conference Object-Oriented Programming (ECOOP'97)*, 1997,

[3] Fowler, M. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[4] Deursen, A. van et al. Aspect Mining and Refactoring. *Proceedings of the First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE03). Held in conjunction with WCRE'2003*, 2003.

[5] Filman, R. E. and Friedman, D. P. Aspect-Oriented Programming is Quantification and Obliviousness. *Workshop on Advanced Separation of Concerns (OOPSLA'2000)*, 2000.

[6] Gamma, E. et al. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] Moonen, L. Exploring Software Systems. *Proceedings of the 19th International Conference on Software Maintenance (ICSM'2003)*. IEEE Computer Society Press, 2003.

[8] Hannemann, J. and Kiczales, G. Overcoming the Prevalent Decomposition in Legacy Code. *Workshop on Advanced Separation of Concerns in Software Engineering (ICSE'2001)*, 2001.

[9] Robillard, M. P. and Murphy, G. C. Capturing Concern Descriptions During Program Navigation. *Workshop on Tools for Aspect-Oriented Software Development (OOPSLA 2002)*, 2002.

[10] Robillard, M. P. *Representing Concerns in Source Code. Ph.D. Thesis*. Department of Computer Science, University of British Columbia. November 2003.

[11] Janzen, D. and Volder, K. De. Navigating and Querying Code Without Getting Lost. *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'2004)*, 2004.

[12] Volder, K. De et al. Logic Meta Programming as a Tool for Separation of Concerns. *Workshop on Aspects and Dimensions of Concerns (ECOOP'2000)*, 2000.

[13] Kersten, M. AO Tools: State of the (AspectJ) Art and Open Problems. *Workshop on Tools for Aspect-Oriented Software Development (OOPSLA'2002)*, 2002.

[14] Kersten, M. Tool Requirements for Commercial Development with AspectJ. *AOSD Workshop on Commercialization of AOSD Technology*, 2003.

[15] Iwamoto, M. and Zhao, J. Refactoring Aspect-Oriented Programs. *The 4th AOSD Modeling With UML Workshop*, 2003.

[16] Hanenberg, S. et al. Refactoring of Aspect-Oriented Software. *Net.Object Days 2003*, 2003.

[17] Monteiro, M. P. and Fernandes, J. M. Object-to-Aspect Refactorings For Feature Extraction. *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'2004)*, ACM Press, 2004.

[11] `http://www.refactoring.com/catalog/index.html`