Bad Smells em Sistemas Orientados a Aspectos

Eduardo Kessler Piveta[1] **, Marcelo Hecht**[1] **,**
Marcelo Soares Pimenta[1] **, Roberto Tom Price**[1]

[1]Instituto de Informática
Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500 - Campus do Vale - Bloco IV
Bairro Agronomia - Porto Alegre - RS -Brasil
CEP 91501-970 Caixa Postal: 15064

epiveta@inf.ufrgs.br, mhecht@gmail.com, mpimenta@inf.ufrgs.br, tomprice@inf.ufrgs.br

Abstract. *This paper adapts a collection of bad smells that may occur in object-oriented software to the context of aspect-oriented software. This adaptation is done by describing the problems related to each bad smell when it appears in aspects and proposing a set of refactorings to help on the task of removing those bad smells.*

Resumo. *Este artigo adapta um conjunto de* bad smells *que podem ocorrer em sistemas orientados a objetos para o contexto de sistemas orientados a aspectos. Esta adaptação é feita de forma a descrever os problemas que cada* bad smell *acarreta ao estar presente em aspectos e propor o uso de um conjunto de refatorações para auxiliar na remoção destes.*

1. Introdução

Alguns problemas com desenvolvimento de software orientado a objetos surgem quando existem interesses de uma aplicação que são difíceis de modularizar ao serem empregados os mecanismos de abstração mais comumente utilizados, tais como: classes, métodos e atributos [Lopes 1997].

Normalmente, estes interesses ao serem modelados e implementados espalham-se ao longo das abstrações de um sistema, desviando-as de suas responsabilidades principais, dificultando o reuso e comprometendo sua legibilidade [Czarnecki and Eisenecker 2000]. Exemplos de interesses com estas características incluem: mecanismos para tratamento de exceções, sincronização, restrições de tempo real, concorrência, distribuição de objetos e persistência, dentre outros.

O desenvolvimento de software orientado a aspectos [Kiczales et al. 1997] pode auxiliar a separação de interesses (*separation of concerns*) em situações nas quais existam interesses que se encontram espalhados ao longo das abstrações da aplicação e entrelaçados com a funcionalidade básica destas. Esta melhoria na modularização dos interesses da aplicação é possível devido a existência de novos mecanismos de abstração e composição que facilitam a modularização destes interesses multi-dimensionais [Tarr et al. 1999], [Ossher and Tarr 2001], tais como: aspectos, conjuntos de junção (*pointcuts*), declarações inter-tipos (*inter-type declarations*) etc. [1].

[1]As traduções utilizadas neste artigo seguem as recomendações definidas no WASP 2004 - 1º Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos. Traduções disponíveis em http://twiki.im.ufba.br/bin/view/AOSDbr/WebHome

Embora o uso de aspectos possa auxiliar na modularização de interesses desta natureza, sua adição pode introduzir problemas específicos do uso de aspectos ou similares aos que podem ser encontrados em sistemas orientados a objetos, tais como: trechos de código que foram abandonados em um módulo e não estão mais sendo usados, duplicações de código, classes com poucas responsabilidades ou atribuições em demasia [Fowler et al. 2000].

Estes problemas normalmente dificultam o reuso em todos as fases de um processo de desenvolvimento [Boehm and Sullivan 2000] e podem ser minimizados através da identificação de seus sintomas e da remoção das causas destes problemas. Estes sintomas (denominados de *bad smells* por Fowler [Fowler et al. 2000]) podem ser vistos como sinais ou alertas de que problemas existem no software [Elssamadisy and Schalliol 2002] e podem ser corrigidos através da aplicação de refatorações, definidas para cada *bad smell*.

Embora existam catálogos e descrições para *bad smells* em sistemas orientados a objetos (vide [Fowler et al. 2000], [M.P. Monteiro 2005]), a detecção destes em sistemas orientados a aspectos ainda não é bastante explorada. Monteiro [M.P. Monteiro 2005], por exemplo, discute *bad smells* que podem aparecer em sistemas orientados a objetos, indicando oportunidades de refatorações para extração de código de objetos para aspectos, sem entretanto, discutir extensivamente *bad smells* específicos para aspectos.

Este artigo objetiva adaptar um conjunto de *bad smells* que podem ocorrer em sistemas orientados a objetos (descritos inicialmente em [Fowler et al. 2000]) para o contexto de sistemas orientados a aspectos. Esta adaptação é feita de forma a (i) descrever os problemas que cada *bad smell* acarreta ao estar presente em aspectos e (ii) propor o uso de um conjunto de refatorações para auxiliar na remoção destes *bad smells*.

Desta forma, para cada *bad smell* descrito, são encontradas informações referentes a definição do problema indicado pelo *bad smell*, refatorações que podem ser usadas para minimizar os efeitos deste, bem como exemplos da ocorrência deste usando a linguagem *AspectJ* [Kiczales et al. 2001].

O restante deste artigo é estruturado da seguinte forma: na seção 2 são descritos os principais conceitos relacionados ao desenvolvimento de software orientado a aspectos, e na seção 3 a linguagem de aspectos mais utilizada atualmente (*AspectJ*). Esta foi escolhida, além de sua popularidade, pelo fato de diversas outras linguagens utilizarem seu modelo de pontos de junção (*join points*).

Na seção 4 são descritas refatorações e em especial refatorações em sistemas na presença de aspectos. São mostradas refatorações para programas orientados a aspectos, as quais serão utilizadas nas tarefas de remoção de *bad smells*.

A seguir, na seção 5, são definidos e detalhados diversos tipos de *bad smells*, bem como refatorações que podem ser utilizadas para minimizar o impacto destes. Por fim, nas seções 6 e 7, são descritas a relação deste trabalho com trabalhos anteriores e algumas considerações acerca deste trabalho e de trabalhos futuros.

2. Desenvolvimento de Software Orientado a Aspectos

Existem requisitos que afetam diversas classes de maneira sistemática, modificando a semântica destas e/ou o desempenho da aplicação, tais como: mecanismos de persistência, depuração, distribuição etc. A implementação destas normalmente é feita de

forma tal que fragmentos de código correspondentes a estas são encontrados dispersos ao longo de diversas classes da aplicação. Da mesma forma, atributos e métodos para tratar destas propriedades aparecem em diversas classes do sistema de forma direta (inseridos na classe) ou indireta (através de associações) [Kiczales et al. 1997].

A implementação destes requisitos utilizando técnicas orientadas a objetos pode levar a problemas, quanto ao entendimento da funcionalidade das classes, ao reuso e a manutenibilidade [Lopes 1997]. A idéia da orientação a aspectos é possibilitar ao desenvolvedor a utilização de um novo mecanismo de abstração para separar os componentes funcionais da aplicação destes requisitos.

Este novo mecanismo de abstração é denominado de *aspecto*. Os aspectos modificam classes, funções e/ou outros aspectos através de mecanismos estáticos e mecanismos dinâmicos. Os mecanismos estáticos preocupam-se com a adição de estado e comportamento às classes, enquanto que os mecanismos dinâmicos modificam a semântica destas em tempo de execução.

Considere um exemplo: a implementação de um mecanismo de *logging* de forma a obter informações posteriores em caso de problemas de execução ou tentativas de acesso indevidas. Para tal, foi utilizada uma figura representando as classes do *Apache TomCat* (um *container* JSP do *Apache Group*) [Hilsdale and Kiczales 2001]. Nas classes deste, foram marcadas todas as chamadas a código relativo a *logging*. Na figura 1, as classes são representadas como linhas brancas e o código de *logging* grifado.

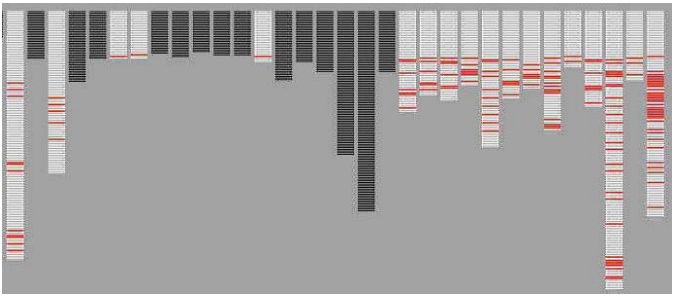

Figura 1. *Logging* em *org.apache.tomcat - AspectJ Tutorial*

A partir desta visão global, pode-se inferir o quão difícil seria uma modificação nas políticas de *logging* da aplicação. A troca do *framework* de *logging* ou a configuração para implantação (de forma a remover entradas de testes) podem ser atividades trabalhosas, dado que o código de *logging* encontra-se espalhado através das classes da aplicação e misturado (*tangled*) com o código básico desta.

A idéia de orientação a aspectos é separar interesses multi-dimensionais em entidades de primeira ordem (*first class entities*), denominadas aspectos. Estes aspectos especificam que locais do sistema que serão afetados e como isto será conduzido.

O objetivo principal do desenvolvimento de software orientado a aspectos é auxiliar na tarefa de separar interesses multi-dimensionais, usando mecanismos de abstração e composição permitindo especificar e combinar os diferentes módulos de uma aplicação.

O desenvolvimento de software orientado a aspectos estende outras técnicas (orientada a objetos, estruturada, funcional, etc.) que não oferecem abstrações para lidar especificamente com interesses multi-dimensionais [Kiczales et al. 1997].

3. AspectJ

AspectJ é uma linguagem orientada a aspectos que utiliza a linguagem *Java* como base. Além dos mecanismos relativos a orientação a objetos (classes, métodos, atributos etc) existem mecanismos relacionados a implementação de aspectos, tais como: aspectos (*aspects*), conjuntos de junção (*pointcuts*), pontos de junção (*joinpoints*), adendos (*advices*) e declarações inter-tipos (*inter-type declarations*).

3.1. Pontos de junção

Pontos de junção são pontos bem definidos no fluxo de execução de um programa. Exemplos de pontos de junção: chamadas e execuções de métodos e construtores, leitura e escrita de atributos, inicializações etc. Considere uma classe *Conta*, contendo o método *sacar* para retiradas de dinheiro e um atributo *saldo* contendo o saldo da conta.

```
1   public class Conta{
2       double saldo;
3       public void sacar(double valor){
4           saldo -= valor;
5       }
6       public static void main(){
7           Conta c = new Conta();
8           c.saldo = 100;
9           c.sacar(50);
10      }
11  }
```

Neste contexto, pontos de junção seriam, por exemplo, a execução do método sacar (linha 4), sua chamada na linha 9, a leitura/escrita do atributo saldo (linha 4 e linha 8), a criação de um objeto da classe *Conta* (linha 7) e assim sucessivamente. Em *AspectJ*, existem elementos sintáticos que permitem definir pontos de junção de acordo com os pontos em que se deseja afetar através de um aspecto.

3.2. Conjuntos de junção

Conjuntos de junção agrupam pontos de junção através da definição de um predicado que, quando satisfeito, faz com que as ações a ele associadas sejam executadas. Existem diversos elementos que podem ser utilizados para a definição destes. Estes pontos de junção ainda podem ser compostos através de operadores lógicos, denotando *e*, *ou* e *não* lógicos (&&, || e ! respectivamente).

Os conjuntos de junção podem ser nomeados e podem receber parâmetros. Estes representam os argumentos que são recebidos pelo ponto de junção, o objeto que está recebendo a mensagem para que um método seja executado, por exemplo. Estes argumentos podem ser inspecionados e posteriormente modificados.

Por exemplo, caso o desenvolvedor queira definir que um aspecto afete todas as chamadas ao método sacar da classe *Conta*, ele poderia definir um conjunto de junção da seguinte forma `call(void Conta.sacar(double))`. Ainda, poderia definir que este aspecto afeta a criação de objetos

`initialization(public Conta.new(..))`. A linguagem de expressões de conjuntos de junção é bastante poderosa, podendo capturar pontos tanto na estrutura estática quanto na estrutura dinâmica das classes do sistema.

3.3. Adendos

Adendos são ações normalmente associadas a um conjunto de junção. Estas ações podem ocorrer antes de um dado ponto de junção, após ou ao invés deste (denotando-se através de tipos diversos de construções). Adendos posteriores (do tipo *after*) possuem ainda duas variações. A primeira delas é executada sempre após a execução com sucesso do código associado ao ponto de junção e a segunda ocorre caso uma exceção seja lançada no decorrer da execução do ponto de junção associado.

3.4. Aspectos

Em *AspectJ*, existe uma nova abstração chamada de aspecto e declarada através da palavra reservada *aspect*. Aspectos são similares à classes em diversos sentidos: aspectos podem conter atributos, métodos e implementar interfaces. Entretanto, diferentemente das classes, eles não podem ser instanciados e o mecanismo definido para herança entre aspectos é de certa forma limitado (apenas aspectos abstratos ou classes podem ser estendidos). Aspectos unem conjuntos de junção, pontos de junção, declarações inter-tipos e adendos em uma única abstração.

4. Refatorações em Sistemas Orientados a Aspectos

Refatorações são transformações de código fonte utilizadas para melhorar o projeto de um software sem modificar o comportamento da aplicação. Estas são normalmente descritas através de (a) um nome (descrevendo a refatoração), (b) um contexto no qual esta pode ou deve ser aplicada, (c) um conjunto de passos bem definidos para sua execução e (d) um exemplo ou conjunto de exemplos demonstrando como a refatoração ocorre [Fowler et al. 2000]. Opcionalmente, podem existir figuras mostrando a transformação. A estrutura utilizada para descrever refatorações é bastante semelhante a utilizada por padrões de projeto.

As refatorações são verificadas através de um contrato, normalmente especificado através de um conjunto de pré e pós condições, na tentativa de preservar o comportamento observável da aplicação. Por exemplo, sempre que uma refatoração de renomeação de método (*rename method*) for executada, deve ser garantido que todas as referências a este método também sejam renomeadas. Normalmente, este processo é auxiliado por ferramentas de refatoração e apoiado por um conjunto de testes unitários, facilitando que os efeitos da refatoração sejam desfeitos caso necessário.

Fowler [Fowler et al. 2000] define diversas refatorações utilizadas em sistemas orientados a objetos. Dentre estas existem refatorações para a renomeação de classes, atributos e métodos, encapsulamento de escrita e leitura de atributos, movimentação de membros de uma classe entre a hierarquia de classes etc.

No contexto de sistemas orientados a aspectos, também existe a necessidade de refatorações que permitam a manipulação de código na presença de aspectos. Mais especificamente, refatorações que tratem de sistemas orientados a aspectos devem possibilitar

(a) mover código implementado em classes para aspectos, (b) manipular código de aspectos para aspectos e (c) mover código de aspectos para classes.

Algumas refatorações foram propostas de forma a possibilitar a manipulação de código na presença de aspectos (vide [Garcia et al. 2004, Hanenberg et al. 2003, Iwamoto and Zhao 2003, Monteiro and ao M. Fernandes 2004, M.P. Monteiro 2005]). Estas refatorações auxiliam nas atividades de remoção de *bad smells* em aspectos. A seguir, na tabela 1, são listadas algumas refatorações orientadas a objetos e orientadas a aspectos usadas neste trabalho com este propósito. Para cada refatoração, é descrito seu nome, sua motivação, a solução proporcionada e a fonte de consulta utilizada.

Refatoração	**Descrição**	**Fonte**
Extract Method	Remove um trecho de código para um novo método	[Fowler et al. 2000]
Combine Pointcut	Mescla os predicados de vários conjuntos de junção	[Iwamoto and Zhao 2003]
Pull Up Field	Move um atributo para uma das super-classes da classe atual	[Fowler et al. 2000]
Pull Up Method	Move um método para uma das super-classes da classe atual	[Fowler et al. 2000]
Pull Up Advice	Move um adendo para uma das super-classes ou super-aspectos do aspecto atual	[Garcia et al. 2004]
Pull Up Pointcut	Move um conjunto de junção para uma das super-classes ou super-aspectos do aspecto atual	[Garcia et al. 2004]
Pull Up Inter-Type Declaration	Move uma declaração inter-tipos para uma das super-classes ou super-aspectos do aspecto atual	[Garcia et al. 2004]
Extract Pointcut	Extrai uma definição de um conjunto de junção de um adendo	[Iwamoto and Zhao 2003]
Extract Class	Cria uma nova classe e move atributos e métodos relevantes da classe antiga para a nova classe	[Fowler et al. 2000]
Extract Aspect	Extrai um aspecto a partir de código constante em classes	[M.P. Monteiro 2005]
Extract Sub-Aspect	Cria um sub-aspecto contendo um sub-conjunto de funcionalidades	-
Collapse Aspect Hierarchy	Une uma hierarquia de aspectos	[Garcia et al. 2004]
Inline Aspect	Insere o código de aspectos nas classes que ele afeta	-
Remove Advice Parameter	Remove um parâmetro de um adendo	-
Rename Aspect	Renomeia um aspecto e todas as referências a este	[Hanenberg et al. 2003]
Rename Pointcut Definition	Renomeia um conjunto de junção e todas as definições a este	[Garcia et al. 2004]
Move Pointcut	Move um conjunto de junção de uma classe/aspecto para outra classe/aspecto	-

Tabela 1. Refatorações para Sistemas Orientados a Aspectos

5. Bad smells em Sistemas Orientados a Aspectos

Bad smells são propostos por Fowler [Fowler et al. 2000] de forma a possibilitar a identificação de problemas em artefatos de software pré-existentes. Isto é feito através da sugestão de possíveis sintomas que podem aparecer nestes artefatos, indicando áreas que normalmente podem ser melhoradas, através de refatorações. O uso de refatorações possibilita que estes sintomas sejam minimizados e excluídos, atacando as causas dos problemas.

Nesta seção, é descrito um conjunto de *bad smells* que podem ocorrer em sistemas orientados a objetos (descritos inicialmente em [Fowler et al. 2000]) para o contexto de

sistemas orientados a aspectos. Esta adaptação é feita de forma a (i) descrever os problemas que cada *bad smell* acarreta ao estar presente em aspectos, e (ii) propor o uso de um conjunto de refatorações para auxiliar na remoção destes *bad smells*. Os exemplos utilizados nesta seção foram extraídos e modificados das demonstrações constantes no AspectJ, versão 1.1 [Hilsdale and Kiczales 2001].

Cada *bad smell* é detalhado através de (a) uma definição do problema indicado pelo *bad smell*, (b) refatorações que podem ser usadas para minimizar e excluir os efeitos deste e (c) um exemplo da ocorrência do *bad smell*.

5.1. Duplicação de Código

Diminuir a quantidade de duplicação de código é uma das motivações do desenvolvimento de software orientado a aspectos. Ao prover mecanismos de abstração para a modularização de interesses multi-dimensionais, a tendência é que a duplicação seja reduzida, dado que interesses antes espalhados ao longo das abstrações de uma aplicação podem ser encapsulados em um único aspecto ou em um conjunto pequeno de aspectos.

Caso a duplicação de código ocorra em diferentes adendos de um mesmo aspecto, o código repetido deve ser extraido através da refatoração *Extract Method* [Fowler et al. 2000]. O uso desta refatoração possibilita que ambos os adendos chamem o método extraido ao invés de conter a funcionalidade deste duplicada. Ainda é possível, caso o código dos adendos seja idêntico, a combinação dos conjuntos de junção, removendo o adendo duplicado. Para isto, a refatoração *Combine Pointcut* [Iwamoto and Zhao 2003] deve ser utilizada.

Caso a duplicação ocorra em aspectos que possuem a mesma super-classe ou super-aspecto através do uso do mecanismo de herança, a estrutura duplicada deve ser movida para níveis acima na hierarquia. Possíveis estruturas duplicadas: atributos, métodos, adendos, conjuntos de junção, declarações inter-tipos. Para que estas duplicações possam ser removidas, as seguintes refatorações podem ser aplicadas (respectivamente): *Pull Up Field* [Fowler et al. 2000], *Pull Up Method* [Fowler et al. 2000], *Pull Up Advice* [Garcia et al. 2004], *Pull Up Pointcut* [Garcia et al. 2004] e *Pull Up Inter-Type Declaration* [Garcia et al. 2004].

Considere um aspecto responsável por implementar mecanismos de rastreamento (*tracing*) nos momentos da construção de objetos e da execução de quaisquer métodos de determinada classe. Este aspecto, denominado de *Trace*, possui adendos para mostrar o estado dos pontos de junção antes e após a ocorrência destes. Isto ocorre sempre que o predicado definido no conjunto de junção é satisfeito.

```
1   abstract aspect Trace {
2       abstract pointcut myClass(Object obj);
3       pointcut myConstructor(Object obj): myClass(obj) && execution(new(..));
4       pointcut myMethod(Object obj): myClass(obj) && execution(* *(..))
5          && !execution(String toString());
6       before(Object obj): myConstructor(obj) {
7          traceEntry("" + thisJoinPointStaticPart.getSignature(), obj);
8       }
9       after(Object obj): myConstructor(obj) {
10         traceExit("" + thisJoinPointStaticPart.getSignature(), obj);
11      }
12      before(Object obj): myMethod(obj) {
13         traceEntry("" + thisJoinPointStaticPart.getSignature(), obj);
```

```
14        }
15        after(Object obj): myMethod(obj) {
16            traceExit("" + thisJoinPointStaticPart.getSignature(), obj);
17        }
18    }
```

Deve ser observado que o código dos adendos relacionados ao conjunto de junção *myConstructor* (adendo nas linhas 6-8 e adendo nas linhas 9-11) é idêntico ao dos adendos associados a *myMethod* (linhas 12-14 e 15-17). Esta duplicação pode ser removida através da união dos predicados definidos nas linhas 3 e 4. O novo predicado associado aos conjuntos de junção seria algo como `myConstructor(obj) || myMethod(obj)`, o que possibilitaria a remoção dos adendos duplicados.

5.2. Mudanças Divergentes

Outro *bad smell* que pode ser identificado, ocorre quando a definição de vários conjuntos de junção são praticamente iguais, só mudando os modificadores ou pequenas partes do predicado. Toda vez que um conjunto de junção é modificado, o mesmo deve ser feito em todos os outros. O uso de uma refatoração de extração (*Extract Pointcut* [Iwamoto and Zhao 2003]) resolveria o problema, definindo semântica para o trecho do conjunto de junção que repete-se ao longo dos conjuntos.

Considere um trecho de um aspecto denominado *Debug*, o qual faz parte de um exemplo chamado *Space War* (um jogo de naves e asteróides) [Hilsdale and Kiczales 2001]. Este aspecto é responsável por manter e mostrar informações de depuração.

```
1   aspect Debug {
2       pointcut allConstructorsCut():
3           call((spacewar.* && !(Debug+ || InfoWin+)).new(..));
4       pointcut allInitializationsCut():
5           initialization((spacewar.* && !(Debug+ || InfoWin+)).new(..));
6       pointcut allMethodsCut():
7           execution(* (spacewar.* && !(Debug+ || InfoWin+)).*(..));
8   }
```

Pode ser observada na definição dos conjuntos de junção que parte do predicado definindo os pontos afetados repete-se, o que faz com que cada vez que parte dos conjuntos é modificada, devem ser modificados diversos outros conjuntos no mesmo aspecto.

5.3. Definição Anônima de Conjunto de Junção

Como adendos não possuem nomes, as vezes é necessário recorrer a descrição contida no conjunto de junção para obter uma idéia dos pontos afetados pelo adendo. O uso da definição do conjunto de junção diretamente no adendo pode diminuir a legibilidade e ocultar a intenção deste.

Para definir claramente a intenção de um conjunto de junção, deve ser definido um nome para o conjunto e este utilizado em quaisquer adendos que afetem os pontos. A refatoração denominada *Extract Pointcut* [Iwamoto and Zhao 2003] pode ser utilizada para extrair definições de conjuntos de junção declaradas diretamente no adendo.

Considerando-se outro trecho do aspecto de depuração *Debug*, existe um conjunto de junção associado ao adendo (linhas 3-4). Extrair a definição do conjunto de junção permite uma melhor compreensão dos pontos afetados.

```
1  aspect Debug {
2     after(Ship ship, SpaceObject obj) returning :
3        call(void Ship.handleCollision(SpaceObject))
4        && target(ship) && args(obj) { ... }
5  }
```

Após a extração, fica mais simples inferir a que tipo de pontos de junção o adendo está associado. O uso de um conjunto de junção nomeado possibilita que seja definida semântica ao conjunto de pontos afetados, facilitando a comunicação e a clareza do aspecto.

```
1  aspect Debug {
2     pointcut collision(Ship ship, SpaceObject obj):
3        call(void Ship.handleCollision(SpaceObject))
4        && target(ship) && args(obj);
5     after(Ship ship, SpaceObject obj) returning : collision (ship, obj){ ... }
6 }
```

5.4. Aspecto Extenso

Quando um aspecto tenta lidar com mais de um interesse, este deve ser dividido em tantos aspectos quantos houverem interesses. Normalmente isto ocorre com a definição de adendos com propósitos diferentes ou outros tipos de estruturas, como atributos, declarações inter-tipo etc.

Caso os membros do aspecto que correspondam a interesses diferentes possam existir em uma classe, deve ser utilizada a refatoração *Extract Class* [Fowler et al. 2000]. Caso estes membros sejam estruturas exclusivas de aspectos deve ser aplicada a refatoração *Extract Aspect* [M.P. Monteiro 2005].

Quando os diferentes interesses podem ser separados através de herança, dada a afinidade dos interesses tratados, pode ser utilizada a refatoração *Extract Sub-Aspect* [2].

Considere um exemplo no qual o aspecto *Debug* define adendos preocupados com diferentes tarefas simultaneamente. Este aspecto modifica a interface com o usuário (linhas 2-3), altera o conteúdo do registro (linhas 4-6), preocupa-se com informações relativas a colisões de naves do exemplo (linhas 7-9), dentre outras preocupações omitidas neste exemplo. Embora todas essas características estejam relacionadas com a depuração do sistema, elas poderiam ser divididas em vários aspectos, cada um englobando uma diferente perspectiva de depuração.

```
1  aspect Debug {
2     after() returning (SWFrame frame):
3        call(SWFrame+.new(..)) { ... }
4     after(Registry registry) returning :
5        target(registry) && (call( void register(..)) ||
6           call(void unregister(..))) { ... }
7     after(Ship ship, SpaceObject obj) returning :
8        call(void Ship.handleCollision(SpaceObject))
9        && target(ship) && args(obj) { ... }
10 }
```

Outra possibilidade seria fazer com que os diferentes aspectos extraídos herdassem de uma mesma super-classe, ou super-aspecto, que no caso poderia ser o mesmo

[2]Não definida anteriormente. Esta refatoração seria o equivalente no contexto de aspectos a uma refatoração de extração de subclasse

aspecto *Debug*.

No exemplo a seguir, foi criada uma sub-classe de *Debug* contendo um adendo responsável por manipular a depuração de colisões de naves do exemplo (linhas 2-4). O uso de um aspecto separada possibilita que o desenvolvedor preocupe-se com parte das responsabilidades de depuração, no caso somente os aspectos relacionados a colisões. Esta separação possibilita também que o aspecto de depuração (*Debug*) seja mais facilmente reutilizado, já que este possui apenas os mecanismos básicos de depuração.

```
1  aspect Collision extends Debug{
2     after(Ship ship, SpaceObject obj) returning :
3        call(void Ship.handleCollision(SpaceObject))
4        && target(ship) && args(obj) { ... }
5  }
```

5.5. Aspecto com poucas responsabilidades

Este *bad smell*, inicialmente definido em [M.P. Monteiro 2005] e estendido nesta seção, ocorre quando um aspecto tem poucas responsabilidades e sua eliminação poderia proporcionar benefícios na etapa de manutenção. Muitas vezes esta diminuição de responsabilidade está relacionada a refatorações anteriores ou uma modificação que foi adicionada em função de alterações que foram planejadas/previstas mas que não ocorreram. A medida em que refatorações são efetuadas, certas classes e/ou aspectos podem perder responsabilidades.

Caso as responsabilidades de um aspecto não sejam suficientes para justificar sua existência, a refatoração *Collapse Aspect Hierarchy* [Garcia et al. 2004] pode ser utilizada. Esta refatoração visa reduzir a árvore de hierarquia, movendo membros de um aspecto para suas sub-classes ou das sub-classes para o super-aspecto. Aspectos vazios podem ser removidos com o uso de *Inline Aspect* [3].

Considere um aspecto responsável por implementar rastreamento das chamadas a métodos em uma aplicação denominado *TraceMyClasses*. Este aspecto apenas define quais pontos da aplicação devem ser afetados pelo rastreamento. A menos que o aspecto *Trace* seja utilizado por mais sub-aspectos ou faça parte de uma biblioteca reutilizável de aspectos, não existe a necessidade de estender o aspecto *Trace* apenas para definir os pontos afetados (linha 2). Isto pode ser feito diretamente no super-aspecto.

```
1   public aspect TraceMyClasses extends Trace {
2      pointcut myClass(): within(TwoDShape) || within(Circle) || within(Square);
3   }
```

5.6. Generalidade Especulativa

Algumas vezes classes e aspectos são criados para lidar com requisitos futuros. Como aspectos tornam mais simples postergar algumas decisões do projeto, é possível remover as funcionalidades que não são usadas de alguma maneira.

De forma a remover parâmetros que não estão sendo usados no adendo, usar a refatoração *Remove Advice Parameter*[4]. Usar as refatorações *Collapse Aspect Hierarchy* [Garcia et al. 2004] e *Inline Aspect* para remover aspectos que não estão sendo

[3]Esta refatoração não é definida anteriormente. Deve ser considerada como o equivalente a outras refatorações de *Inline*
[4]Esta refatoração não é definida anteriormente

usados. Aspectos e conjuntos de junção com nomes genéricos podem ser renomeados para a semântica atual usando as refatorações *Rename Aspect* [Hanenberg et al. 2003] ou *Rename Pointcut* [Garcia et al. 2004].

Considere um exemplo em que um aspecto obtém informações sobre pontos de junção e o objeto passado. Este aspecto possui dois conjuntos de junção definidos: *demoExecs* e *goCut* (linhas 2 e 3). Embora *demoExecs* seja compreensível, poderia ser substituído por um nome mais significativo, como *allDemoMethods*. O conjunto de junção *goCut* não possui nenhum significado. Ele poderia ser substituído por um nome mais relevante.

```
1   aspect GetInfo {
2      pointcut goCut(): cflow(this(Demo) && execution(void go()));
3      pointcut demoExecs(): within(Demo) && execution(* *(..));
4      Object around(): demoExecs() && !execution(* go()) && goCut() {
5         println("Intercepted message: " +
6            thisJoinPointStaticPart.getSignature().getName());
7         println("in class: " +
8            thisJoinPointStaticPart.getSignature().
9            getDeclaringType().getName());
10        printParameters(thisJoinPoint);
11        return proceed();
12     }
13  }
```

5.7. Feature Envy

Em *AspectJ* é permitido usar conjuntos de junção em classes. Caso este conjunto de junção seja utilizado por apenas um aspecto, é interessante que este seja movido para o aspecto que o utiliza. Este mesmo problema pode ocorrer em classes, em situações tais que um método de uma classe faz referência aos atributos e métodos de outra classe ao invés de referenciar os membros da classe que o contém.

Uma refatoração de movimentação pode ser utilizada, de forma a movimentar o conjunto de junção da classe para o aspecto. Esta refatoração, denominada de *Move Pointcut* [Hanenberg et al. 2003], pode ser utilizada também para mover conjuntos de junção entre aspectos.

Considere uma classe denominada *Ship*, a qual implementa uma nave espacial no exemplo *Space War* [Hilsdale and Kiczales 2001]. Esta classe contém uma definição de um conjunto de junção (linha 2-4) que é utilizada apenas no aspecto *EnsureShipIsAlive*.

```
1   class Ship extends SpaceObject {
2      pointcut helmCommandsCut(Ship ship): target(ship) &&
3         (call(void rotate(int))|| call(void thrust(boolean)) ||
4         call(void fire()));
5   }
```

```
1   aspect EnsureShipIsAlive {
2      void around (Ship ship): Ship.helmCommandsCut(ship) {
3         if (ship.isAlive()) { proceed(ship); }
4      }
5   }
```

Ao mover a definição do conjunto de junção para o aspecto, diminui-se o acoplamento entre a classe e o aspecto e aumenta-se a coesão do aspecto.

5.8. Introdução de Método Abstrato

Aspectos podem ser utilizados de forma a adicionar estado e comportamento à classes existentes. Isto é feito através de um mecanismo chamado de declaração inter-tipos (inter-type declaration). Este mecanismo permite que métodos e/ou atributos sejam inseridos nas classes afetadas pelo aspecto. Entretanto, o uso desta funcionalidade pode causar problemas caso sejam inseridos métodos abstratos nas classes da aplicação.

Esta introdução força o desenvolvedor a prover implementações concretas para os métodos introduzidos em todas as classes afetadas e/ou sub-classes. Esta dependência aumenta de forma desnecessária o acoplamento entre o aspecto e as classes afetadas.

A introdução de métodos abstratos através de uma declaração inter-tipos (*inter-type declaration*) deve ser evitada, pois implica que cada vez que uma sub-classe de uma classe afetada é criada, implementações para os métodos abstratos inseridos pelo aspecto devem ser providas.

No exemplo a seguir, é definido um aspecto de cobrança de ligações denominado *Billing* [Hilsdale and Kiczales 2001], o qual cobra chamadas telefônicas de acordo com o tipo de ligação realizada e o tempo que o usuário permanece na ligação. Na linha 2, é introduzido um método abstrato na classe *Connection* responsável pela determinação da taxa a ser aplicada de acordo com o tipo da ligação. Este método é chamado de *callRate*.

A seguir, nas linhas 3 e 4, são adicionadas implementações do método *callRate* nas sub-classes diretas de *Connection*, denominadas de *LongDistance* e *Local*.

```
1   public aspect Billing {
2       public abstract long Connection.callRate();
3       public long LongDistance.callRate() { return 10; }
4       public long Local.callRate() { return 3; }
5       after(Connection conn): Timing.endTiming(conn) {
6           long time = Timing.aspectOf().getTimer(conn).getTime();
7           long rate = conn.callRate();
8           getPayer(conn).addCharge(rate * time);
9       }
10  }
```

Considere que o usuário adicione uma nova sub-classe de *Connection* denominada *International*. Problemas vão ocorrer dado que esta nova classe não implementa o método *callRate*. Logo, o usuário da classe deve estar ciente de quais aspectos afetam o código em questão, e então adicionar métodos ao aspecto. Esta dependência é desnecessária e aumenta a complexidade da solução utilizada.

5.9. Bad smells e refatorações

A seguir, sumarizando a discussão anterior, é apresentada uma tabela sintetizando os diferentes tipos de *bad smells* encontrados em sistemas na presença de aspectos, bem como um conjunto de possíveis refatorações a serem usadas de forma a melhorar o sistema em observação.

6. Trabalhos Relacionados

Existem algumas ferramentas relacionadas a identificação de interesses multi-dimensionais em código orientado a objetos. Os pontos identificados por estas ferramentas fornecem subsídios para a identificação e posterior extração de aspectos por meio de

Bad smell	**Refatoração**
Duplicação de Código	Extract Method Combine Pointcut Pull Up Field Pull Up Method Pull Up Advice Pull Up Pointcut Pull Up Inter-Type Declaration
Mudanças Divergentes	Extract Pointcut
Definição Anônima de Conjunto de Junção	Extract Pointcut
Aspecto Extenso	Extract Class Extract Aspect Extract Sub-Aspect
Definição de Conjunto de Junção Extensa	Extract Pointcut
Aspecto com poucas responsabilidades	Collapse Aspect Hierarchy Inline Aspect
Generalidade Especulativa	Remove Advice Parameter Collapse Aspect Hierarchy Inline Aspect Rename Aspect Rename Pointcut Definition
Feature Envy	Move Pointcut
Introdução de Método Abstrato	-

Tabela 2. Bad smells e refatorações

refatorações. Segundo Hannemann [Hannemann and Kiczales 2001], as ferramentas que auxiliam na tarefa de encontrar potenciais aspectos normalmente tem por objetivo encontrar termos comuns no léxico (mineração textual) e em expressões regulares baseadas em tipos. Exemplos de ferramentas que implementam estas características são:

Aspect Mining Tool [Hannemann and Kiczales 2001] é um aplicativo que realiza análise de código-fonte existente, na tentativa de identificar e extrair código relacionado a interesses que estão espalhados através das classes do sistema.

FEAT tem por objetivo auxiliar na identificação de interesses multi-dimensionais. Para isso ela possibilita a criação de grafos de interesses ([Robillard and Murphy 2002], [Murphy et al. 2001]), 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. Pode-se visualizar, por exemplo, todos os locais onde o construtor de uma classe é chamado, em que classes e locais um atributo é lido ou escrito, todos os membros de uma classe e assim por diante.

JQuery [Janzen and Volder 2003], [De Volder 1998] é um navegador de código *Java* desenvolvido como um *plug-in* 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).

Após a identificação, o engenheiro de software pode utilizar refatorações para extrair os interesses que estão espalhados pelo sistema, dispersos por métodos, classes e pacotes através de refatorações. Algumas propostas adaptam refatorações orientadas a objetos existentes para o contexto de aspectos.

Em Iwamoto [Iwamoto and Zhao 2003] são apresentadas análises de diversas refatorações de [Fowler et al. 2000], concluindo que poucas delas podem ser usadas em código com aspectos sem modificações. Iwamoto ainda define diversas refatorações no contexto de aspectos, incluindo refatorações de extração (*Extract Advice*, *Extract Pointcut*), manipulação (*Pull Up Per Target*, *Move to Introduction*) e inserção (*Introduce Call Pointcut*, *Introduce Around Advice*).

Hanenberg [Hanenberg et al. 2003] discute a relação entre refatorações orientadas a aspectos e refatorações orientadas a objetos, demonstrando os conflitos encontrados e provendo mecanismos para resolvê-los em *AspectJ*. Também são introduzidas novas refatorações para auxiliar na migração de software orientado a objetos para código orientado a aspecto, bem como para reestruturar código de aspectos. Dentre estas incluem-se: *Extract Advice* (também definida por Iwamoto [Iwamoto and Zhao 2003]), *Extract Introduction* e *Separate Pointcut*.

Garcia [Garcia et al. 2004] descreve refatorações para a manipulação de interesses multi-dimensionais através de uma série de refatorações inter-relacionadas. Algumas refatorações definidas pelo autor visam a manipulação especificamente de código de aspectos, como: *Rename Pointcut*, *Collapse Aspect Hierarchy* e *Collapse Pointcut Definition*.

Monteiro [Monteiro and ao M. Fernandes 2004, M.P. Monteiro 2005], por sua vez, apresenta uma coleção de refatorações na forma de um catálogo, de forma a auxiliar na extração de aspectos de código legado. O autor revisita *bad smells* na presença de aspectos, que não os apresentados neste artigo e define um *bad smell* específico de aspectos.

As propostas definidas nesta seção enfocam principalmente nas questões de identificação de aspectos e refatoração de sistemas na presença de aspectos. Destas, apenas uma discute a questão de *bad smells* em sistemas orientados a aspectos. Esta discussão (encontrada em [M.P. Monteiro 2005]) concentra-se em apenas um *bad smell* para sistemas orientados a aspectos e outros para sistemas orientados a objetos.

7. Considerações Finais e Trabalhos Futuros

A relação entre *bad smells* e refatorações é bastante importante, dado que a identificação de *bad smells* fornece indícios de possíveis problemas no projeto e/ou na codificação da aplicação. Estes indícios podem ser vistos como oportunidades para refatoração, as quais podem aumentar a qualidade do código analisado. Este trabalho descreve *bad smells* que podem ocorrer em sistemas orientados a aspectos e refatorações que podem ser utilizadas para a remoção destes *bad smells*. Ele estende outros trabalhos que visam a definição de *bad smells* para código orientado a objetos [Fowler et al. 2000] e de *bad smells* que indiquem o uso de aspectos para sua eliminação [M.P. Monteiro 2005].

Para isto foram utilizadas as refatorações definidas nos trabalhos relacionados descritos na seção anterior, de forma a prover subsídios para a remoção de bad smells. Das propostas anteriores, apenas Monteiro [M.P. Monteiro 2005] discute a questão de *bad smells* para aspectos. Entretanto, esta discussão resume-se a *bad smells* que ocorrem em sistemas orientados a objetos e um *bad smell* para sistemas orientados a aspectos.

As principais contribuições deste trabalho compreendem na adaptação e discussão detalhada de diversos *bad smells* para sistemas orientados a aspectos e a recomendação de

mecanismos para a remoção destes *bad smells* através da aplicação de refatorações. São discutidos ainda exemplos da ocorrência destes *bad smells* no contexto de uma linguagem orientada a aspectos (no caso *AspectJ*).

Embora os *bad smells* discutidos neste artigo sejam expressos de forma a descrever sintomas em uma linguagem específica, estes podem ser adaptados de forma a serem identificados e removidos de outras linguagens orientadas a aspectos que não sigam o modelo proposto por *AspectJ*. Ainda, *bad smells* que ocorrem na utilização de novas características da linguagem (como o suporte a Java 5 e anotações) não foram ainda explorados.

Trabalhos futuros visam definir de forma mais aprofundada a identificação de *bad smells* auxiliada por ferramentas. Esta identificação consiste na definição de predicados que devem ser satisfeitos, e no uso de métricas para auxiliar na identificação de sintomas. Outra área a ser explorada compreende na verificação de modificações nos atributos de qualidade dos sistemas antes e depois da aplicação de refatorações, de forma a possibilitar a mensuração de projetos alternativos para um mesmo problema.

Referências

Boehm, B. W. and Sullivan, K. J. (2000). Software economics: a roadmap. In *ICSE - Future of SE Track*, pages 319–343.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston.

De Volder, K. (1998). Aspect-oriented logic meta programming. In *Workshop on Aspect Oriented Programming (ECOOP 1998)*.

Elssamadisy, A. and Schalliol, G. (2002). Recognizing and responding to bad smells in extreme programming. In *Proceedings of the 24th International conference on Software Engineering*.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (2000). *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley.

Garcia, V. C., Piveta, E. K., Lucrédio, D., Alvaro, A., de Almeida, E. S., do Prado, A. F., and Zancanella, L. C. (2004). Manipulating crosscutting concerns. *4th Latin American Conference on Patterns Languages of Programming (SugarLoafPlop 2004)*.

Hanenberg, S., Oberschulte, C., and Unland, R. (2003). Refactoring of aspect-oriented software. In *Net.Object Days 2003*.

Hannemann, J. and Kiczales, G. (2001). Overcoming the prevalent decomposition in legacy code. In *Workshop on Advanced Separation of Concerns in Software Engineering (ICSE 2001)*.

Hilsdale, E. and Kiczales, G. (2001). Aspect-oriented programming with AspectJ. Proceedings of the 16th Object Oriented Programming Systems Languages and Applications (OOPSLA'2001). *Tutorial*. Tampa Bay, FL, USA.

Iwamoto, M. and Zhao, J. (2003). Refactoring aspect-oriented programs. In *The 4th AOSD Modeling With UML Workshop, UML'2003*.

Janzen, D. and Volder, K. D. (2003). Navigating and querying code without getting lost. In Aksit, M., editor, *Proc. 2nd Int Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 178–187. ACM Press.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In Knudsen, J. L., editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin. Springer-Verlag.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag.

Knuth, D. E. (1984). *The TEX Book*. Addison-Wesley, 15th edition.

Lopes, C. V. (1997). *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University.

Monteiro, M. P. and ao M. Fernandes, J. (2004). Object-to-aspect refactorings for feature extraction. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'2004)*. ACM Press.

M.P. Monteiro, J. F. (2005). Towards a catalog of aspect-oriented refactorings. In Mezini, M., editor, *Proc. 4th, Int Conf. on Aspect-Oriented Software Development (AOSD-2005)*. ACM Press.

Murphy, G. C., Lai, A., Walker, R. J., and Robillard, M. P. (2001). Separating features in source code: An exploratory study. In *Proc. 23rd Int'l Conf. Software Engineering*, pages 275–284. IEEE Computer Society.

Ossher, H. and Tarr, P. (2001). The shape of things to come: Using multi-dimensional separation of concerns with Hyper/J to (re)shape evolving software. *Comm. ACM*, 44(10):43–50.

Robillard, M. P. and Murphy, G. C. (2002). Capturing concern descriptions during program navigation. In *Workshop on Tools for Aspect-Oriented Software Development (OOPSLA 2002)*.

Tarr, P., Ossher, H., Harrison, W., and Sutton Jr., S. M. (1999). N degrees of separation: Multi-dimensional separation of concerns. In *Proc. 21st Int'l Conf. Software Engineering (ICSE'1999)*, pages 107 – 119. IEEE Computer Society Press.