Controlando a Evolução de Sistemas Orientados por Aspectos por meio de Aspect-Aware Interfaces

Leonardo Humberto Guimarães Silva, Marco Túlio de Oliveira Valente

Programa de Pós-graduação em Informática
Pontifícia Universidade Católica de Minas Gerais
{leonardosilva,mtov}@pucminas.br

Abstract. *In this paper, we propose the use of aspect-aware interfaces (AAIs) not just as a documentation resource but also to manage the evolution of aspect-oriented systems. The key idea is that AAIs can be generated in a semi-automatic and incremental way during the weaving process of aspect-oriented systems. Moreover, we propose to check AAIs in order to detect two unwanted situations that might occur when an aspect oriented system evolves: the unintended join point capture and the accidental join point miss.*

Resumo. *Neste artigo, apresenta-se uma proposta de uso de aspect-aware interfaces (AAIs) não apenas como um documento de apoio a raciocínio modular, mas também como uma abstração útil para controlar e gerenciar a evolução de um sistema orientado por aspectos. A idéia é computar a AAI de um sistema de forma semi-automática, interativa e incremental, sempre que um processo de weaving for requisitado. Adicionalmente, propõe-se que AAIs passem a ser verificadas, de forma a detectar duas situações indesejáveis que podem ocorrer quando de uma evolução no código de um sistema orientado por aspectos: uma captura acidental de um ponto de junção ou uma falha na captura de um ponto de junção.*

1 Introdução

Programação orientada por aspectos estende paradigmas tradicionais de programação, como procedimental ou orientado por objetos, com novas abstrações destinadas a modularizar a implementação de interesses transversais (*crosscutting concerns*) [7]. Por exemplo, em AspectJ – a linguagem de programação orientada por aspectos mais utilizada atualmente – aspectos podem declarar conjuntos de junção (*pointcuts*), os quais especificam de forma precisa pontos bem definidos da execução de um programa Java, os quais são chamados de pontos de junção (*joinpoints*). *Advices* correspondem a métodos anônimos que são implicitamente chamados antes, depois ou no lugar de pontos de junção [6]. Em essência, aspectos foram propostos para incrementar os níveis de reuso, para facilitar o entendimento e para diminuir o esforço demandado por manutenções em sistemas orientados por objetos.

No entanto, aspectos também podem comprometer a implementação e evolução em separado de módulos de um sistema, isto é, podem impactar um dos principais benefícios conquistados ao longo de décadas por técnicas e conceitos de programação modular [11]. O principal motivo é que conjuntos de junção utilizam nomes e expressões regulares para definir pontos de junção de um programa alvo. Se por um lado isso viabiliza quantificação, por outro torna os conjuntos de junção sensíveis a mudanças nos

nomes e nas abstrações do programa alvo. Por exemplo, uma simples alteração no nome de um método pode, como efeito colateral, remover ou inserir pontos de junção em um dado conjunto de junção. Como um programa é normalmente implementado sem conhecimento dos aspectos que atuarão sobre o mesmo (propriedade conhecida como *obliviousness*), uma alteração como a mencionada pode, de forma silenciosa, habilitar a execução de *advices* indesejáveis ou então desabilitar a execução de *advices* necessários ao correto funcionamento de um sistema. Em outras palavras, o modelo de definição de conjuntos de junção de linguagens como AspectJ, a fim de conciliar quantificação e *obliviousness*, acaba por produzir um importante efeito colateral: a criação de um forte acoplamento entre conjuntos de junção e a estrutura do programa alvo. Este problema já foi investigado por outros pesquisadores, sendo usualmente chamado de problema dos conjuntos de junção frágeis (*fragile pointcut problem*) [9, 13, 4, 5].

Neste artigo, apresenta-se uma solução para o problema dos conjuntos de junção frágeis de AspectJ. A solução apresentada é baseada no conceito de *aspect-aware interfaces* (AAIs), proposto originalmente por Kiczales e Mezini [8]. Basicamente, uma *aspect-aware interface* estende uma interface tradicional de linguagens orientadas por objetos com informações sobre os *advices* que são habilitados pelos seus métodos. AAIs foram originalmente propostas para tratar um outro problema típico de orientação por aspectos: a impossibilidade de se raciocinar localmente sobre um determinado módulo, visto que aspectos podem modificar o comportamento de qualquer ponto de junção do mesmo.

A principal contribuição deste artigo consiste em uma proposta de uso de *aspect-aware interfaces* não apenas como um documento de apoio a raciocínio modular, mas também como uma abstração útil para controlar e gerenciar a evolução de um sistema orientado por aspectos. Descreve-se no artigo uma ferramenta capaz de computar a AAI de um sistema de forma semi-automática, interativa e incremental, sempre que um processo de *weaving* for requisitado. Ou seja, a geração de uma AAI passa a ser parte integrante do processo de desenvolvimento de um sistema orientado por aspectos, da mesma forma que são as tarefas de compilação e *weaving*. Adicionalmente, propõe-se que AAIs passem a ser verificadas, de forma a detectar duas situações indesejáveis que podem ocorrer quando de uma evolução no código de um sistema orientado por aspectos: uma captura acidental de pontos de junção ou uma falha na captura de pontos de junção. Estas duas situações são definidoras do problema dos conjuntos de junção frágeis [4]. No entanto, por ser baseada em AAIs, a solução proposta não pretende estabelecer regras que viabilizem o desenvolvimento modular e independente de aspectos.

O restante deste artigo está estruturado conforme descrito a seguir. Na Seção 2, caracteriza-se e exemplifica-se o problema dos conjuntos de junção frágeis. Na Seção 3, apresenta-se o conceito de *aspet-aware interfaces*, tal como proposto originalmente para viabilizar raciocínio modular na presença de aspectos. A Seção 4 apresenta a solução proposta neste artigo e uma ferramenta desenvolvida para controlar e gerenciar a evolução de sistemas orientados por aspectos. A Seção 5 apresenta um estudo de caso envolvendo uma seqüência de pequenas manutenções em um sistema hipotético. Na Seção 6, apresenta-se uma avaliação da solução proposta. A Seção 7 discute trabalhos relacionados e a Seção 8 conclui o artigo.

2 Conjuntos de Junção Frágeis

Tradicionalmente, em orientação por objetos, o acoplamento entre o código cliente e o código que implementa um módulo (ou classe) é controlado por meio de interfaces. O cliente adquire o direito de usar os métodos de uma interface e a classe assume o compromisso de implementar tais métodos. Caso a assinatura dos métodos de uma interface seja alterada, esta alteração impacta tanto os clientes da interface, como as classes que implementam a mesma. Além disso, este impacto é verificado pelo compilador, que notifica os clientes sobre a necessidade de utilizar os métodos com suas novas assinaturas e as classes implementadoras sobre a necessidade de prover uma implementação para os novos métodos.

No caso de orientação por aspectos, é interessante observar que aspectos podem ser vistos como clientes do programa base [12, 10]. Isto é, embora aspectos complementem o programa base com código para implementação modular de requisitos transversais, este código não é explicitamente chamado pelo programa base. Em vez disso, para seu correto funcionamento, aspectos pressupõem que o programa base disponibiliza determinados pontos de junção, os quais funcionam como ganchos para chamada implícita de *advices*[1]. No entanto, esta dependência entre aspectos e programa base não é documentada nem verificada por um contrato (ou interface). Com isso, estabelecem-se as condições para ocorrência do problema dos conjuntos de junção frágeis.

Mais especificamente, o problema dos conjuntos de junção frágeis em sistemas orientados por aspectos se manifesta quando, em razão de uma modificação no programa base, conjuntos de junção *silenciosamente* passam a capturar pontos de junção indesejados ou deixam de capturar determinados pontos de junção essenciais ao correto funcionamento de um sistema [4]. Assim, de acordo com esta definição, um conjunto de junção é caracterizado como frágil por dois motivos: (i) por não ser robusto a alterações no programa base; (ii) por ter sua semântica silenciosamente alterada em função de uma manutenção no programa base.

Para caracterizar melhor este problema, suponha o sistema descrito pelo diagrama de classes da Figura 1, usado com freqüência para ilustrar o uso de aspectos e de conceitos relacionados com esta tecnologia. Suponha ainda o seguinte aspecto, usado para capturar o requisito transversal que define quando o desenho de figuras deve ser atualizado no `Display` do sistema:

```
aspect UpdateSignaling {
   pointcut change(): execution(void Figure+.set*(..))
      || execution(void Figure+.moveBy(int, int));
   after() returning: change() {
     Display.update();
   }
}
```

[1]Esta inversão de responsabilidades constitui uma diferença fundamental entre orientação por objetos e orientação por aspectos. Em orientação por objetos, uma classe cliente explicitamente invoca métodos implementados por uma classe servidora. Em orientação por aspectos, cabe a um aspecto – que, do ponto de vista desta explicação, constitui um cliente – implementar métodos que serão implicitamente chamados pelo código base.

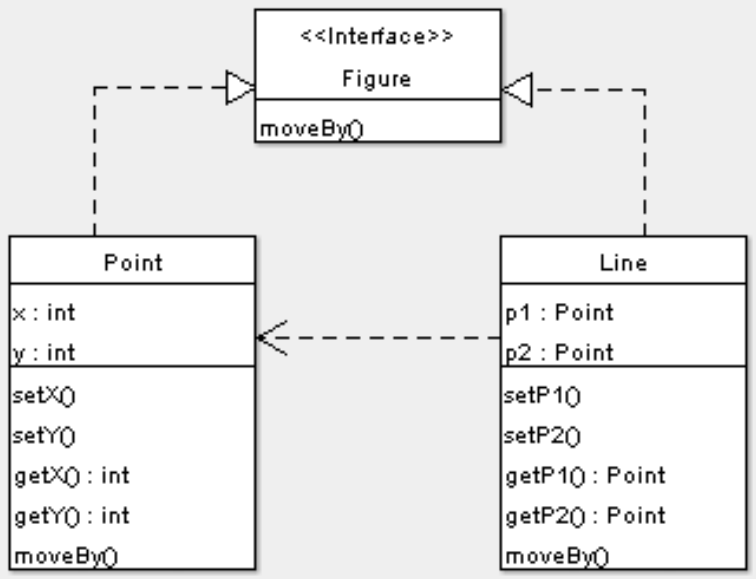

Figura 1. Programa base

Situações características de conjuntos de junção frágeis: Supondo o sistema da Figura 1 como programa base, descreve-se a seguir as duas principais situações nas quais a natureza frágil dos conjuntos de junção se manifesta:

- **Captura acidental de pontos de junção:** Suponha que um novo campo, chamado `date`, seja introduzido na classe `Line` para armazenar a data e a hora em que foi realizada a última alteração em um objeto desta classe. Para isso, acrescentam-se também na classe `Line` métodos `setDate` e `getDate`. Após esta alteração no programa base, o conjunto de junção `change` do aspecto mostrado irá incluir um novo ponto de junção, correspondente à execução do método `setDate`. No entanto, esta captura é acidental, pois viola a premissa básica de definição deste conjunto de junção, segundo a qual o mesmo somente deve capturar a execução de métodos que alteram propriedades visuais de uma figura. Além de acidental, esta captura é também silenciosa, visto que nem o desenvolvedor da classe, nem o desenvolvedor do aspecto são notificados a respeito da mesma.

- **Falha na captura de pontos de junção:** Suponha que uma alteração no programa base renomeie os métodos `setX` e `setY` da classe `Point` para, respectivamente, `changeX` e `changeY`. Com isso, o conjunto de junção `change` não irá mais capturar a execução dos métodos renomeados. No entanto, para o correto funcionamento do sistema, eles deveriam continuar sendo capturados, pois efetuam modificações em propriedades visuais de um ponto. Além disso, esta falha de captura também é silenciosa.

Os dois problemas mencionados, além de ocorrerem na especificação de pontos de junção por meio dos designadores `call` e `execution`, podem ocorrer também quando da especificação de pontos de junção via designadores `set` e `get`. Neste caso, eles ocorrem quando adicionam-se ou removem-se campos em classes do programa base.

No caso dos designadores `handler`, `within`, `staticinitialization`, `this` e `target`, que requerem a especificação de uma classe como parâmetro, não existe fragilidade silenciosa, visto que se, por exemplo, a classe informada não existir no programa base, um `warning` é emitido pelo `weaver` de AspectJ[2]. O mesmo ocorre com métodos especificados como parâmetros de um designador do tipo `withincode`. Quando o operador + é usado para indicar um tipo e seus subtipos, pode ocorrer de uma evolução no programa incluir um novo subtipo, que passará a ser capturado. No entanto, este fato não caracteriza uma captura acidental de pontos de junção, pois convencionalmente em orientação por objetos um subtipo sempre pode ser usado no lugar de seu supertipo. No caso de declarações intertipos, também não existe fragilidade silenciosa, pois caso um dos tipos especificados não exista no programa base, o *weaver* emite uma mensagem de erro.

3 Aspect-Aware Interfaces (AAIs)

Em sistemas orientados por aspectos implementados em AspectJ, AAIs podem ser definidas para classes e interfaces. Basicamente, a AAI associada a uma classe lista a assinatura de todos os métodos desta classe, incluindo construtores, independentemente de serem públicos ou não. Uma AAI lista também a declaração de todos os campos da classe (públicos e não-públicos). Interfaces convencionais de Java também possuem uma AAI. No caso de interfaces, AAIs listam os métodos das mesmas. Tanto no caso de interface ou de classe, para cada método definido em uma AAI são acrescentadas informações sobre os possíveis *advices* que são implicitamente executados quando tais métodos são chamados. Esta definição inclui o tipo do *advice* (`after`, `before` ou `around`), o nome do aspecto ao qual ele pertence e o nome do conjunto de junção associado ao mesmo[3].

Na Figura 2, mostram-se as AAIs geradas para as interfaces e classes do exemplo motivador. Para ser claro, suponha a seguinte entrada da AAI associada à classe `Line`:

```
void setP1(Point): after returning UpdateSignaling.change();
```

Esta entrada informa que no aspecto `UpdateSignaling` existe um *advice* do tipo `after returning` associado ao conjunto de junção `change`. Este *advice* captura chamadas ou execuções do método `setP1` da classe `Line`.

No caso de pontos de junção cuja captura somente é decidida em tempo de execução, como aqueles capturados por meio de `cflow` ou `if`, adota-se uma solução conservadora na geração das respectivas AAIs. Basicamente, em um conjunto de junção definido como PC_{static} && $PC_{dynamic}$, onde PC_{static} e $PC_{dynamic}$ denotam respectivamente descritores de conjuntos de junção estáticos e dinâmicos, assume-se na geração de AAIs que $PC_{dynamic}$ é sempre verdade. Com isso, se existe a possibilidade de um *advice* ser disparado em função da chamada de um método, este *advice* é incluído na AAI deste método.

A proposta original de Kiczales e Mezini contempla também a definição de AAIs para aspectos. Basicamente, a AAI associada a um aspecto possui uma entrada para cada

[2]No caso, o *weaver* considerado é o AJDT 1.4.1.

[3]Como *advices* são métodos anônimos, a solução adotada para identificá-los em uma AAI consiste em fornecer informações sobre o seu tipo e o nome do conjunto de junção ao qual ele está vinculado.

```
interface Figure
  void moveBy(int, int): after returning UpdateSignaling.change();

Point implements Figure
  int x;
  int y;
  void setY(int): after returning UpdateSignaling.change();
  void setX(int): after returning UpdateSignaling.change();
  void moveBy(int, int): after returning UpdateSignaling.change();

Line implements Figure
  Point P1;
  Point P2;
  void setP1(Point): after returning UpdateSignaling.change();
  void setP2(Point): after returning UpdateSignaling.change();
  void moveBy(int,int): after returning UpdateSignaling.change();
```

Figura 2. Exemplos de AAIs

um de seus *advices*. Nesta entrada, além do tipo do *advice*, informam-se também os pontos de junção do programa sobre o qual ele se aplica. Ou seja, AAIs associadas a aspectos disponibilizam informação inversa àquela provida por AAIs associadas a classes. No entanto, estas AAIs não são usadas na solução para o problema dos conjuntos de junção frágeis de AspectJ que será apresentada neste artigo. Por esse motivo, elas não foram incluídas na Figura 2. A solução proposta também não contempla aspectos que interceptam pontos de junção de outros aspectos. Mais informações sobre AAIs e a aplicação das mesmas no suporte a raciocínio modular em sistemas orientados por aspectos podem ser obtidas em [8].

4 Solução Proposta

Em orientação por aspectos, a interface de um módulo é determinada quando a configuração completa das classes e aspectos do sistema a que este módulo pertence for conhecida [8]. No entanto, este fato não necessariamente implica que uma ferramenta para geração de *aspect-aware interfaces* somente possa fazê-lo quando a configuração completa do sistema for conhecida (isto é, quando o sistema estiver pronto e, portanto, quando interfaces não sejam mais necessárias)[4].

De forma diferente daquela usualmente concebida para geração de AAIs, descreve-se neste artigo uma ferramenta para geração incremental de tais interfaces, chamada `aiicheck`[5]. Esta ferramenta foi implementada como um *plug in* do ambiente Eclipse. À medida que o sistema base sofre manutenções ou evoluções, propõe-se que o programador tenha que autorizar explicitamente quaisquer atualizações das informações sobre

[4]Esta crítica improcedente a AAIs consta, por exemplo, do recente ensaio sobre AOP de autoria de Steimann [12, pág. 145].

[5]Detalhes sobre a implementação da ferramenta `aaicheck` foram omitidos deste artigo devido a limitações de espaço.

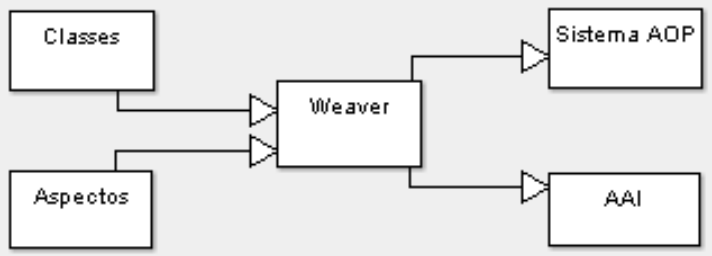

Figura 3. Geração de AAIs

pontos de junção e *advices* contidas em AAIs. Com isso, criam-se as condições para que este programador seja notificado das duas situações definidoras do problema dos conjuntos de junção frágeis de AspectJ, mencionadas na Seção 2.

Geração de AAIs: A proposta é que a geração de AAIs seja incorporada ao *weaver* de um ambiente de desenvolvimento orientado por aspectos, conforme ilustrado na Figura 3. Como o *weaver* realiza o espalhamento do código dos *advices* em pontos de junção de um programa base, ele possui as informações necessárias para gerar uma AAI. Além disso, como o *weaver* é sempre chamado após a compilação das classes e aspectos de um sistema, garante-se que novas versões de AAIs serão sempre geradas quando ocorrer uma alteração no programa base.

Atualização de AAIs: Novas versões de uma AAI – contendo informações atualizadas sobre pontos de junção de um programa base – podem ser criadas tanto em decorrência de manutenções realizadas nos aspectos como nas classes deste programa.

Manutenções realizadas em aspectos podem levar à inclusão de novas informações em uma AAI (quando a manutenção faz com que um novo ponto de junção do programa seja capturado) ou à remoção de informações contidas em uma AAI (quando a manutenção faz com que um ponto de junção anteriormente capturado não seja mais). Em ambos os casos, o programador que realizou a manutenção – e que, portanto, chamou o *weaver* para gerar a nova versão do sistema e de sua AAI – deve explicitamente aprovar a inclusão ou a remoção de informações na AAI gerada. Ou seja, na solução proposta, a geração de AAIs é semi-automática e interativa, já que o programador tem que aprovar as diferenças detectadas entre uma AAI e sua versão anterior. A exigência desta aprovação assegura que todas as informações sobre aspectos acrescentadas ou removidas de uma AAI tenham sido validadas pelo desenvolvedor do sistema. Em última instância, o objetivo é proibir alterações silenciosas em uma AAI, as quais constituem a origem do problema dos conjuntos de junção frágeis. Assim, na solução proposta, AAIs não podem ser modificadas sem explícita aprovação do desenvolvedor responsável pela evolução do sistema, mesmo que esta evolução tenha ocorrido em aspectos. Para facilitar, na solução proposta, aprovações podem ser feitas em bloco (ou seja, aprovam-se todas as alterações em uma única operação).

Se o programador aprovar a captura de um novo ponto de junção, então significa que esta captura é requerida e, portanto, consiste em um benefício proporcionado pelos recursos de quantificação de AspectJ. Por outro lado, caso esta captura seja acidental, o

Manutenção		Situação	aaicheck	Desenvolvedor
1	+ `setDate`	Captura acidental	Detecta	Refazer manutenção
2	- `setX`	Falha captura	Detecta	Refazer manutenção
3	+ `setColor`	Captura adicional	Detecta	Aprovar introdução AAI
4	- `setColor`	Eliminação captura	Detecta	Aprovar remoção AAI
5	+ `changeColor`	Captura pendente	Não detecta	–

Tabela 1. Exemplos de manutenção no código base e comportamento da solução proposta

programador certamente não irá aprová-la. Ou seja, neste caso, a solução proposta detecta uma das situações decorrentes do problema dos conjuntos de junção frágeis: a captura acidental de pontos de junção. Uma vez que a captura é classificada como acidental, restam duas alternativas ao programador responsável pela manutenção em questão: ou ele modifica novamente o código base ou então ele comunica ao desenvolvedor de aspectos que a manutenção para ser finalizada requer uma atualização nos aspectos do sistema.

De forma semelhante, se o programador aprovar uma falha na captura de um ponto de junção, então significa que sua captura não é mais requerida na nova versão do sistema. Por outro lado, caso esta falha não seja planejada, o programador certamente não irá aprová-la. Neste caso, a solução proposta detecta a segunda situação característica do problema dos conjuntos de junção frágeis: a falha na captura de um ponto de junção. Uma vez detectada esta falha, restam duas alternativas ao programador responsável pela manutenção em questão: ou ele volta atrás e mantém os nomes originais dos métodos ou então ele comunica ao desenvolvedor de aspectos que a manutenção para ser finalizada requer uma atualização nos aspectos sob sua responsabilidade.

5 Estudo de Caso

Nesta seção, apresenta-se um estudo de caso pequeno, mas que cobre os principais impactos em aspectos decorrentes de manutenções no código base de um sistema. Suponha que as seguintes manutenções sejam realizadas na classe `Point` do código base (após cada manutenção disponibiliza-se uma nova versão do sistema):

1. Acrescenta-se o método `setDate`;
2. Renomeia-se o método `setX`;
3. Acrescenta-se o método `setColor` (e o campo `color`);
4. Remove-se o método `setColor` (e o campo `color`);
5. Acrescenta-se o método `changeColor` (e o campo `color`);

A Tabela 1 resume o comportamento da ferramenta `aaicheck` após a compilação de cada uma das manutenções descritas.

As duas primeiras manutenções, conforme descrito na Seção 2, dão origem respectivamente a situações definidas como Captura Acidental e Falha na Captura. Nestes casos, cabe ao programador refazer a manutenção, de forma a restaurar a semântica original dos conjuntos de junção afetados. As Manutenções 3 e 4 consistem, respectivamente, em uma ampliação e redução desejável do grau de quantificação de um conjunto de junção. Nestes casos, cabe ao programador classificar a mudança detectada como desejável e, em seguida, autorizar a atualização da AAI corrente do sistema (conforme

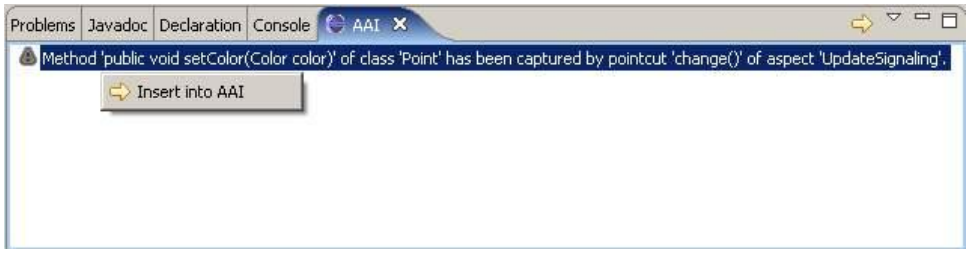

Figura 4. Aprovação de captura adicional de ponto de junção

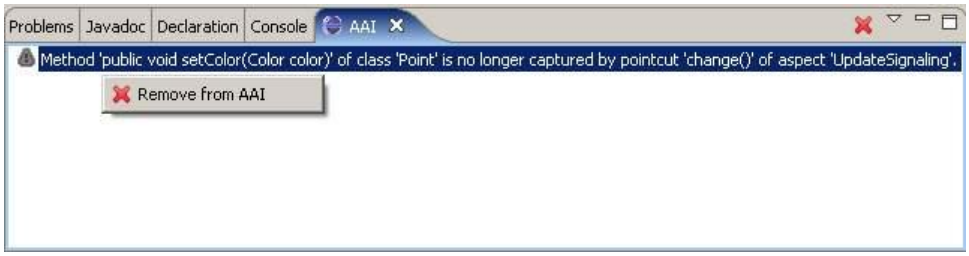

Figura 5. Aprovação de eliminação de captura de ponto de junção

mostrado nas Figuras 4 e 5). Por fim, a ferramenta `aaicheck` não é capaz de detectar a inserção de novos pontos de junção no programa base que deveriam mas que efetivamente não são capturados por um determinado conjunto de junção. É o caso, por exemplo, da Manutenção 5. Após esta manutenção, execuções do método `changeColor` deveriam ser capturadas pelo conjunto de junção `change`, mas efetivamente não são porque o nome do método adicionado não segue as convenções de nomes pressupostas por este conjunto de junção. Segundo estas convenções, o método deveria se chamar `setColor` e não `changeColor`.

6 Avaliação

Considera-se que a solução descrita neste artigo apresenta os seguintes benefícios:

- A solução notifica um programador sobre capturas acidentais de pontos de junção e sobre falhas na captura de pontos de junção, quando de uma manutenção em um sistema orientado por aspectos. Neste sentido, considera-se que a solução proposta oferece um tratamento para o problema dos conjuntos de junção frágeis de AspectJ.

- A solução é compatível com o atual modelo de captura de pontos de junção de AspectJ, isto é, a mesma pode ser usada em qualquer sistema orientado por aspectos que esteja sendo desenvolvido atualmente.

- A solução inclui uma ferramenta para geração semi-automática de AAIs. Ressalte-se que AAIs, além de constituírem um instrumento útil para detecção e validação de fragilidades em conjuntos de junção, podem ser usadas também no apoio a raciocínio modular (isto é, em outra propriedade fundamental para o sucesso e produtividade de tarefas relacionadas com a evolução de sistemas).

A solução, no entanto, não é capaz de capturar novos pontos de junção adicionados a um programa que não seguem as convenções de nome pressupostas por um aspecto

(conforme é o caso do método `changeColor` do estudo de caso apresentado na Seção 5). No entanto, considera-se que esta deficiência é inerente ao modelo de captura de pontos de junção de AspectJ, o qual é baseado exclusivamente em nomes. Qualquer solução para este problema requer inevitavelmente uma alteração ou extensão deste modelo.

7 Trabalhos Relacionados

Descreve-se a seguir outros esforços de pesquisa que também procuram fornecer soluções para o problema dos conjuntos de junção frágeis:

PCDiff: A ferramenta PCDiff [9, 13] compara duas versões orientadas por aspectos de um sistema, indicando diferenças entre os conjuntos de junção das mesmas. Estas diferenças podem incluir novos pontos de junção que são capturados ou então pontos de junção que deixaram de ser capturados por conjuntos de junção da nova versão do sistema. Portanto, a solução implementada por PCDiff, chamada por seus autores de análise de delta de conjuntos de junção (*pointcut delta analysis*), é semelhante à solução proposta neste artigo. No entanto, a solução implementada pela ferramenta `aaicheck` apresenta duas contribuições importantes:

- A ferramenta `aaicheck` não se limita a informar diferenças entre duas compilações de um sistema, como ocorre com PCDiff. Na solução proposta, o responsável por uma manutenção em um sistema orientado por aspectos é forçado a analisar as diferenças detectadas, informando se as mesmas são planejadas ou acidentais.

- Na ferramenta `aaicheck`, *aspect-aware interfaces* constituem uma espécie de "memória persistente" das alterações nos conjuntos de junção de um sistema que foram aprovadas pelo seu desenvolvedor. Isto é, se uma captura de um novo ponto de junção ou uma falha na captura de um ponto de junção não são decorrentes da fragilidade dos conjuntos de junção, as mesmas são aprovadas pelo desenvolvedor e automaticamente registradas na AAI corrente do sistema. Evita-se assim que esta diferença seja novamente detectada em uma execução subseqüente da ferramenta.

A partir da versão 1.2.1, a ferramenta AJDT [2] inclui uma funcionalidade denominada (*crosscutting comparison*) que compara duas versões de um sistema, apresentando as diferenças em uma janela própria no Eclipse. Essencialmente, essa nova funcionalidade do AJDT possui as mesmas características da ferramenta PCDiff.

Crosscutting interfaces (XPIs): Conforme proposto por Sullivan e colegas, uma interface transversal define um conjunto de regras de projeto (*design rules*) que desenvolvedores do código base devem seguir e nas quais projetistas de aspectos podem se basear [3, 14]. Em essência, a idéia é separar aspectos de detalhes de implementação do código base de um sistema, permitindo que tanto aspectos quanto classes possam evoluir de forma independente, desde que respeitem as regras de projeto especificadas em uma XPI. Na proposta mais recente de implementação de XPIs, estas regras são expressas em AspectJ.

Portanto, o projeto de uma XPI consiste em identificar e especificar propriedades estruturais de um programa base. Como conseqüência, quando comparadas com AAIs, acredita-se que XPIs tendem a detectar com mais facilidade casos de métodos que não obedecem às convenções de nomes implícitas na definição de conjuntos de junção, como é o caso do método `changeColor`, citado no final da Seção 5. Por outro lado, AAIs têm a vantagem de explicitar e congelar os pontos de junção do programa base que possuem *advices* associados e de controlar alterações nos mesmos. Por isso, acredita-se que AAIs tendem a lidar melhor com capturas acidentais de pontos de junção. Em resumo, uma linha interessante de trabalho futuro consiste em agrupar conceitos de XPIs e AAIs em uma mesma interface.

Model-based Pointcuts: Soluções baseadas em modelos propõem uma alteração no nível de abstração para declaração de pontos de junção, os quais passam a fazer parte de modelos conceituais [4, 1]. Assim, nestas soluções, o problema de conjuntos de junção frágeis não se manifesta em relação a estruturas sintáticas de um programa, mas sim em modelos (como diagramas de classes, de seqüência etc). Em geral, essas soluções apóiam-se na idéia de que é mais simples identificar problemas relativos aos conjuntos de junção frágeis por meio da análise de modelos conceituais de um sistema, do que através da análise do código fonte. Por outro lado, elas pressupõem uma perfeita sincronia entre modelos e código-fonte durante todo o ciclo de vida de um sistema, o que normalmente não se verifica na maioria dos projetos de desenvolvimento de *software*.

8 Conclusões

Neste trabalho, apresentou-se uma solução para o problema dos conjuntos de junção frágeis que: (1) não requer modificações ou extensões na linguagem para definição de conjuntos de junção de AspectJ; (2) consegue identificar capturas acidentais e falhas na captura de conjuntos de junção. A solução proposta é suportada por meio de uma ferramenta incorporada ao ambiente de desenvolvimento mais utilizado por programadores de AspectJ. Esta ferramenta se responsabiliza por gerar de forma semi-automática *aspect-aware interfaces*, as quais desempenham papel chave na solução proposta. Como benefício extra, as AAIs geradas – além de apoiarem a solução dada ao problema dos conjuntos de junção frágeis de AspectJ – podem ser usadas para suportar raciocínio modular.

Como trabalho futuro pretende-se investigar uma solução que combine recursos de AAIs com aqueles de *crosscutting interfaces* (XPIs). Pretende-se ainda realizar estudos de casos mais extensos, complexos e/ou variados, que permitam avaliar com mais precisão os benefícios da solução proposta, principalmente em termos de escalabilidade e em relação ao grau de participação requerido dos desenvolvedores nas tarefas de aprovação de capturas adicionais e de falhas na captura de pontos de junção.

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 Tarik Rocha pelo apoio na implementação da ferramenta `aaicheck`.

Referências

[1] Walter Cazzola, Sonia Pini, and Massimo Ancona. Design-based pointcuts robustness against software evolution. In *ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, pages 35–45, 2006.

[2] Eclipse Foundation. http://www.eclipse.org/ajdt/. Última visita: agosto de 2007.

[3] William G. Griswold, Kevin J. Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006.

[4] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *20th European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 501–525. Springer Verlag, 2006.

[5] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. A model-driven pointcut language for more robust pointcuts. In *AOSD Workshop on Software Engineering Properties of Languages for Aspect Technology*, 2006.

[6] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *15th European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–355. Springer Verlag, 2001.

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

[8] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *27th International Conference on Software Engineering (ICSE)*, pages 49–58, 2005.

[9] Christian Koppen and Maximilian Störzer. PCDiff: Attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software (EIWAS)*, September 2004.

[10] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In *19th European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240. Springer-Verlag, 2005.

[11] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[12] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *21st Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 481–497, 2006.

[13] Maximilian Störzer and Jürgen Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *21st IEEE International Conference on Software Maintenance (ICSM)*, pages 653–656, 2005.

[14] Kevin J. Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *13th International Symposium on Foundations of Software Engineering (FSE)*, pages 166–175, 2005.