UM PERFIL PARA MODELAGEM DE SOFTWARE ORIENTADO A ASPECTOS, COM USO DE ANOTAÇÕES

Thiago Gottardi[1]
Valter Vieira de Camargo [2]
Fábio Lúcio Meira[3]

RESUMO
O paradigma Orientado a Aspectos foi criado para tratar problemas identificados na orientação a Objetos, adicionando vários conceitos à intenção de melhorar a modularidade e reuso das aplicações. Estes conceitos são complexos para representar como modelos gráficos e muitos perfis de modelagem necessitam de extensões para tornarem mais capazes de representar novos conceitos. Aderindo a esses esforços, neste trabalho há uma proposta de extensão, permitindo que um perfil leve de modelagem represente uso de metadados de anotação com aspectos, assim como exemplos de uso desta proposta e do conceito de anotações para definir pontos de junção, que se tornou uma boa prática para Desenvolvimento de Softwares Orientados a Aspectos, incentivando prover a nova proposta.
PALAVRAS-CHAVE: 1. AspectJ 2. Modelagem 3. Anotações

ABSTRACT
The Aspect Oriented paradigm was created to deal with problems identified on Object Oriented softwares, adding several concepts in order to improve application modularity and reuse. These concepts are complex to represent as graphical models and many modeling profiles are in need of extensions to become more capable of representing newer concepts still being defined as more techniques are being described. Joining these efforts, this work contains an extension proposal allowing a lightweight modeling profile to represent use of annotation metadata with aspects, as well as usage examples for the new profile and for the annotation concept to define pointcuts, which is now known as a new Aspect Oriented Software Development good practice, encouraging the efforts to provide a new proposal.
KEYWORDS: 1. AspectJ 2. Modeling 3. Annotations

INTRODUÇÃO

[1] Graduado em Ciência da Computação (Centro Universitário Eurípides de Marília - UNIVEM/ Marília-SP). Estudante. E-mail: gottardi@univem.edu.br

[2] Doutor (ICMC-USP/ São Carlos-SP) e Mestre (UFSCAR/ São Carlos-SP) em Engenharia de Software. Graduado em Processamento de Dados (Faculdade de Tecnologia de Taquaritinga/ Taquaritinga-SP). Professor da UFSCAR (São Carlos-SP). E-mail: valter@dc.ufscar.br

[3] Mestre (UFSCAR/ São Carlos-SP) e Graduado (Escola de Engenharia de Piracicaba/ Piracicaba-SP) em Ciência da Computação. Professor do Centro Universitário Eurípides de Marília - UNIVEM (Marília-SP). E-mail: fabioluciomeira@gmail.com

Softwares de computador são amplamente utilizados nos dias atuais. Este produto engloba programas que podem ser computados por qualquer máquina, documentos físicos ou virtuais, dados que combinem números e texto, que também podem representar imagens, vídeo e áudio. Sua produção pode se tornar uma tarefa complexa, levando à criação da engenharia de software, que é a aplicação das técnicas de engenharia que estabelecem ideais para melhor desenvolver um software (PRESSMAN, 2001).

Desde a criação da engenharia de software, foram sugeridas melhorias para o desenvolvimento de software, provendo meios de projetar software, reutilização de código (a escrita de um programa) e facilidade de manutenção (PRESSMAN, 2001).

Também é fato que várias destas qualidades já eram desejadas desde antes da criação desta engenharia em consequência de sua constante necessidade. Ao longo dos anos, novos meios de programação e diagramação de software foram providos, de modo a aprimorar estas qualidades, assim como apoiar a produção de softwares de maior porte (PRESSMAN, 2001).

O desenvolvimento de um programa de computador se utiliza de um paradigma de programação, que são conceitos usados para abstrair um problema de modo a atingir uma programação correspondente (SEBESTA, 2002). O processo empregado para o desenvolvimento possui algumas dependências, conforme o paradigma utilizado.

Porém, neste momento, não há conhecimento de um paradigma definitivo e perfeito para definir, estruturar e implementar softwares de qualquer necessidade; por este motivo, a constante aprimoração é enfatizada, mesmo tratando de paradigmas que já foram considerados superiores nestes aspectos, mas que podem apresentar falhas que causam problemas no longo prazo (SEBESTA, 2002).

Uma ocorrência que demonstra a afirmação anterior é a descoberta de falhas no paradigma OO (Orientação a Objetos), paradigma muito aplicado para abstrair objetos encontrados no mundo real. As falhas identificadas dificultam o re-uso e a manutenção do código escrito conforme este paradigma. Assim, um novo paradigma nomeado de POA (Programação Orientada a Aspectos) foi criado para tratar destes problemas (KICZALES *et al.*, 1997).

Em consequência das diferenças deste paradigma com relação ao paradigma orientado a objetos, meios de modelagem de softwares utilizados para diagramar projetos de softwares orientados a objetos não são suficientes para representar softwares

orientados a aspectos.

Após a criação da Programação Orientada a Aspectos (POA) (Kiczales *et al.*, 1997), vários pesquisadores começaram a investigar seu impacto nas fases preliminares do desenvolvimento de software, como por exemplo, elicitação de requisitos, análise e planejamento, projeto arquitetural e propuseram perfis de modelagem (Evermann, 2007) (Aldawud, Elrad e Bader, 2003) (Pawlak *et al.*, 2002) (Han, Kniesel e Cremers, 2005) (Georg e France, 2002) (Mostefaoui e Vachon, 2006) (Suzuki e Yamamoto, 1999) (Groher e Baumgarth, 2004) (Stein, Hanenberg e Unland, 2002) para a mais aceita linguagem de modelagem orientada a objetos, a UML (*Unified Modeling Language*) (Booch, Rumbaugh e Jacobson, 2006) (Object Management Group, 2009) com a intenção de permitir diagramação de software baseado neste paradigma, dando suporte ao seu projeto e visualização.

Porém, não foi encontrado um perfil que incluísse o conceito de uso de anotações, que possui uma participação diferente em orientação a aspectos e vem sendo densamente utilizado, principalmente com a linguagem AspectJ, a primeira possuir este conceito implementado.

Pelo fato destas restrições para desenvolver softwares orientados a aspectos, grandes desenvolvedores costumam optar por não correr risco a utilizar este paradigma e podem continuar a ter prejuízos com os problemas nos quais poderiam ser tratados pela nova proposta. Este problema justifica a necessidade de prover melhor infra-estrutura a DSOA (Desenvolvimento de Softwares Orientados a Aspectos) (ALDAWUD *et al.*, 2001).

O objetivo deste trabalho é aprimorar a representatividade de conceitos oriundos da programação orientada a aspectos, provendo um meio de modelagem fácil de utilizar e implementar dentro do menor tempo possível, facilitando assim seu uso para desenvolver softwares mais complexos e assim sua aceitação, visto que suas melhorias em re-usabilidade e manutenção já são reconhecidas.

Existem propostas de meios de modelagem para o paradigma de Orientação a Aspectos, porém foi identificado em trabalhos anteriores (GOTTARDI E CAMARGO, 2008) que grande número destas propostas ou são preliminares e restritas ou possuem inconsistências que podem dificultar uso e necessitam de correções ou atualizações para que sejam consideradas para projetos mais exigentes.

Neste trabalho, foi proposto um novo perfil capaz de permitir uso de anotações para definição dos conjuntos de junção, tendo como base um perfil anterior

que permitia representar um conjunto menor de conceitos.

Por fim, a notação proposta foi avaliada para determinar se esta é capaz de representar corretamente conceitos novos sem afetar negativamente anteriores e realizar casos de teste com softwares simples utilizando a notação de modelagem proposta.

O desenvolvimento deste artigo é dividido em: POA (Programação Orientada a Aspectos), Linguagem AspectJ, Linguagem de Modelagem Orientada a Objetos, Modelagem Orientada a Aspectos, Perfil de Evermann, Conceito de Anotações, Modelagem de Anotações, Anotações e Orientação a Aspectos, Perfil Proposto, Estudo de Caso, Outras Capacidades do Perfil e por fim, Conclusões e Referências.

1 POA (PROGRAMAÇÃO ORIENTADA A ASPECTOS)

Programação orientada a aspectos é um paradigma de programação que estende a Programação Orientada a Objetos e que tem como objetivo modularizar de forma mais adequada um sistema de software, fornecendo abstrações que permitem implementar um sistema separando os interesses-base dos transversais (KICZALES *et al.*, 1997).

Um interesse de software é um requisito qualquer que este sistema deve cumprir. Quando um interesse representa uma parte funcional, ele é chamado de "base". Porém, se um interesse é dependente da parte base para operar e implementá-lo com técnicas tradicionais de programação e se existir, no mínimo, uma parte de sua implementação misturada dentro da parte base, este interesse é chamado de interesse "transversal".

A preocupação que incentivou a criação do paradigma é o fato de que a implementação de interesses transversais gera dependência entre a parte base e a parte transversal, o que dificulta a manutenção e deixa a parte base, a mais importante do sistema, mais propícia a modificações.

A dificuldade de manutenção de um interesse transversal é ocasionada por duas categorias: espalhamento e o entrelaçamento de código. O espalhamento ocorre quando o código de um interesse encontra-se espalhado entre vários módulos do sistema ao invés de ser encapsulado em uma construção determinada. O entrelaçamento ocorre quando o código de um determinado interesse encontra-se misturado com o código de outro interesse dentro de um mesmo módulo. Exemplificando o entrelaçamento, em um único método, podem existir linhas de código escritas para implementar interesses diferentes. Exemplos de interesses

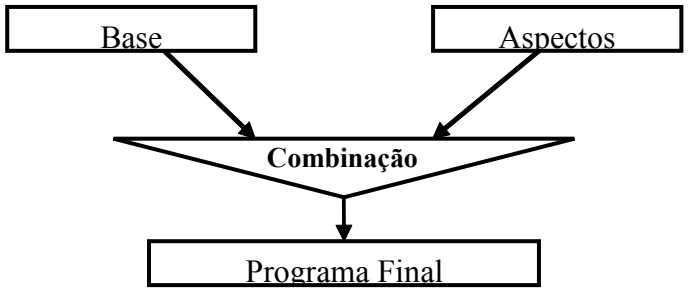
transversais são: Persistência, Distribuição, *Logging*, *Tracing* e Criptografia.

A Orientação a Aspectos foi criada de modo a estender a Orientação a Objetos, isso foi feito para tratar problemas do segundo paradigma e manter suas qualidades. Em códigos orientados a Objeto, este problema é mais evidente do que em códigos de paradigma funcional, por exemplo, porque orientação a objetos divide o sistema em classes e estas classes possuem comportamentos associados aos seus atributos; tais comportamentos podem representar requisitos diferentes.

No paradigma funcional, as funções podem ser separadas tanto como o dado em que operam quanto em sua categoria de funcionalidade, mas ainda estão sujeitas a este problema quando as chamadas são feitas em torno do código base. O problema causado pela dependência entre implementação de requisitos funcionais e implementação de requisitos não funcionais é que torna o núcleo mais importante do software sucetível a sofrer modificações quando partes possivelmente menos importantes são modificadas.

A Orientação a Aspectos foi criada com conceitos que visam a tratar o entrelaçamento e espalhamento de código. Tais conceitos eliminam o entrelaçamento e o espalhamento ao permitir construções que separem interesses do código fonte para que sejam combinados somente na geração do programa final. Na Figura 1, há um fluxograma ilustrando o processo.

Figura 1 – Construção de um programa Orientado a Aspectos.



Baseado em: Kiczales *et al.*, 2001

A POA possui duas características fundamentais: a inconsciência e a quantificação. A inconsciência se refere ao código base que pode ser escrito sem nenhum conhecimento do interesse transversal que possa afetá-lo, evitando assim qualquer dependência para o

interesse base (ELRAD *et al.*, 2001). A quantificação consiste em modularizar declarações unitárias que afetam muitos pontos de um sistema.

Graças a todas estas qualidades, a POA permite melhor modularização de softwares. Possuindo habilidade com este paradigma, um desenvolvedor é capaz de criar softwares de melhor compreensão, maior capacidade de reutilização e maior facilidade de manutenção.

Linguagens de programação orientadas a aspectos já estão disponíveis, sendo a mais popular e completa a AspectJ, que possui várias referências como Kiczales et al. (2001), AspectJ Team (2003), Kiselev (2002), Laddad (2003); outros exemplos são AspectC++ Spinczyk, Gal e Schröder-Preikschat (2002) e AspectH Andrade, Santos e Borba (2004).

2 LINGUAGEM ASPECTJ

AspectJ (KICZALES et al., 2001) é uma linguagem orientada a aspectos baseada na Linguagem Java, sendo completamente retro-compatível, ou seja, códigos-fonte e classes compiladas Java são compatíveis também com AspectJ. Além disso, esta linguagem é a mais utilizada, atualmente, para implementar softwares orientados a aspectos. Por ser condizente com os conceitos da POA (Programação Orientada a Aspectos), com AspectJ é possível separar interesses transversais do base, além de possuir vários conceitos como pontos de junção, conjuntos de junção, adendos, declarações inter-tipos e aspectos.

Pontos de junção são pontos dentro da execução de um código, como por exemplo: criação de objetos, invocação de métodos, acesso a campos e lançamento de exceções. Um conjunto de junção é uma expressão lógica que é capaz de selecionar um conjunto de pontos de junção.

Conjuntos de junção mais utilizados em AspectJ:

- Conjuntos de Junção por tipos usados:
  - Argumentos: "*args*" para selecionar uso de parâmetros com tais tipos;
  - Origem deste Objeto: "*this*" para selecionar objetos que iniciaram uma mensagem inter-objeto com tais tipos;
  - Objeto Alvo: "*target*" para selecionar um objeto que recebeu uma mensagem inter-objeto com tais tipos.
- Conjuntos de Junção por métodos usados:
  - Chamada: "*call*" para selecionar a ocorrência de

uma chamada ao método descrito (o contexto está no objeto que chamou o método especificado);

- Execução: "execution" para selecionar a ocorrência de uma execução ao método descrito (o contexto está sempre no objeto que executa o método especificado);

- Conjuntos de Junção para Atributos:

- Leitura de valor: "get" para selecionar ocorrência de leitura do atributo especificado;

- Alteração de valor: "set" para selecionar a ocorrência de alteração do atributo especificado.

Além da lista acima, é possível criar composições de conjuntos de junção, utilizando operadores "*&&*" para conjunção, "||" para disjunção e "*!*" para negação, seguindo a estrutura de expressões lógicas da linguagens Java, pelo fato de estender sua definição.

Um adendo é um trecho de código que pode ser declarado para ser executado quando a execução pontos de junção ocorre, para isso, ele é ligado a um conjunto de junção. Em AspectJ, um adendo é classificado quanto ao momento em que é executado, em comparação à ocorrência do ponto de junção. São, portanto, três tipos básicos:

- *Before* (Antes): Adendo é executado logo antes da ocorrência da execução do ponto de junção;

- *Around* (Durante): Adendo é executado durante a ocorrência da execução do ponto de junção;

- *After* (Após): Adendo é executado logo após a ocorrência da execução do ponto de junção.

Declarações inter-tipo permitem adicionar atributos, métodos e definições de herança e implementação de interfaces de qualquer classe (ou interface, quando aplicável) a partir de um aspecto externo.

Um aspecto de AspectJ é uma especialização de classe do Java e pode possuir conjuntos de junção, adendos e declarações inter-tipo. Aspectos podem, com esses conceitos, modificar a estrutura de um programa de duas formas: estática e dinâmica. A modificação estática (também chamada de entrecorte estático) é feita via declarações inter-tipo, visto que podem modificar a estrutura de classes do programa, e a modificação dinâmica é feita via adendos que executam na ocorrência de pontos de junção, capturados por conjuntos de junção.

A única grande diferença entre declaração de um aspecto e uma classe em AspectJ é a substituição de "class" para "aspect"; exceto isso, aspectos podem possuir adendos e instanciação diferente, o que não é necessário de ser introduzido por este texto. Um aspecto concreto não é instanciado por programação imperativa, mas automaticamente pela ocorrência de uma regra de sua instanciação. Deste modo, é comum exigir que aspectos concretos possuam construtores sem nenhum parâmetro.

Pelo fato de AspectJ ser a linguagem do paradigma mais completa até o momento, esta é muitas vezes utilizada como base para propostas de modelagem, como em Evermann (2007) e Han, Kniesel e Cremers (2005).

Para demonstrar a proposta deste trabalho, códigos AspectJ foram criados. Não são mostrados neste artigo, mas estão disponíveis em conjunto com a versão completa da monografia para apresentar o uso do perfil proposto neste trabalho.

3 MODELAGEM ORIENTADA A OBJETOS

Uma linguagem muito utilizada para modelar software orientado a objeto é a UML (*Unified Modeling Language*). Essa linguagem de modelagem suporta todas as fases de desenvolvimento presentes em processos de desenvolvimento de software orientado a objeto, como o RUP (*Rational Unified Process*), provendo diagramas que documentam e especificam informações pertinentes para facilitar o desenvolvimento destes softwares (BOOCH, RUMBAUGH, E JACOBSON, 2006).

A UML não só permite especificar software de modo gráfico, como também é especificada de modo gráfico. O modelo que define a semântica da UML é chamado de metamodelo, incluindo todos os tipos de elementos possíveis a modelos e suas relações. Tais definições de elementos são feitas em meta-classes, que são classes presentes em nível de metamodelo. As definições formais e metamodelo da UML estão disponíveis no site da OMG (Object Management Group) (OMG, 2006) e é chamado de MOF (Meta Object Facility). Por exemplo, existem as meta-classes "*Class*", "*Property*" e "*Operation*", que definem em nível de modelo, propriedades de *Class* (Classe), *Property* (Atributo) e *Operation* (Operação), que são úteis para usos posteriores.

São importantes as generalizações presentes no metamodelo, como, por exemplo: a meta-classe *Class* herda da meta-classe *Classifier*, a meta-classe *Property* herdada da meta-classe *StructuralFeature*, além dessas, outra generalização importante é a meta-classe

Operation herda de "*BehavioralFeature"* e útil para criação do perfil proposto, que será apresentado no próximo capítulo. As meta-classes *Class*, *Property* e *Operation* também possuem atributos e papéis nos relacionamentos. Em nível de modelo, essas características são representadas de maneiras diferentes. Por exemplo, a meta-classe *Class* em nível de modelo se torna uma classe que pode conter atributos e métodos.

Entre chaves existem restrições, que são códigos anexados para aumentar a semântica do modelo, são regras que não são possíveis de definir somente com uso de modelagem.

Em alguns casos, as metaclasses do metamodelo UML são limitadas para representar alguns conceitos de um domínio específico, linguagem, tecnologia, ou até outro paradigma. Para sanar este problema, a UML permite criação de perfis de modelagem, que é uma capacidade de estender o metamodelo adicionando, se necessário, restrições e conceitos via estereótipos, etiquetas valoradas de estereótipos e restrições em código. O uso de estereótipos é comum e a própria UML já tráz alguns estereótipos como "create" e "destroy", utilizados para denotar operações que definem construtores e destrutores, respectivamente.

Estereótipos são aplicados a instâncias das metaclasses as quais estendem, assim, uma classe em nível de modelo e podem receber um ou mais estereótipos que estendem a metaclasse "*Class*" ou qualquer uma de suas generalizações, como "*Classifier*" e "*Element*". O elemento que recebe estereótipos apresenta no diagrama em nível de modelo o nome do estereótipo entre os símbolos de "maior maior" "<<" e "menor menor" ">>" (como por exemplo, <<create>>). Estereótipos podem possuir etiquetas valoradas e restrições. Restrições são expressões lógicas que devem ser mantidas verdadeiras para que a semântica do perfil se mantenha correta (BOOCH *et al.*, 2000) e são codificados em OCL (Object Constraint Language) (OMG, 2006) para aumentar a consistência semântica do perfil, evitando uso errôneo que não é possível de evitar usando-se apenas de definição gráfica.

4 MODELAGEM ORIENTADA A ASPECTOS

A modelagem de software é muito utilizada na fase de projeto, porém ainda se encontra carente para o paradigma orientado a aspectos.

Pelo fato do paradigma de Orientação a Aspectos ser baseado no paradigma Orientado a Objetos, suas propostas de modelagem foram feitas como extensões aos

meios de modelagem utilizados na orientação a objetos.

Dentro desses esforços foram propostos vários perfis para o desenvolvimento de softwares orientados a aspectos como Evermann (2007), Aldawud, Elrad e Bader (2001), Aldawud, Elrad e Bader (2003), Pawlak *et al.* (2002), Han, Kniesel e Cremers (2005), Georg e France (2002), Mostefaoui e Vachon (2006), Suzuki e Yamamoto (1999), Groher e Baumgarth (2004) e Stein, Hanenberg e Unland (2002). Estas propostas foram feitas para procurar a melhor representação gráfica dos conceitos da orientação a aspectos, para obter a melhor forma de modelagem. Deste modo, foram extremamente conceituais, algumas vezes deixando de lado a consistência dos conceitos representados como também ausência de algumas representações, o que levou a criar uma versão aprimorada de um dos perfis em trabalhos passados (GOTTARDI E CAMARGO, 2008).

5 PERFIL DE EVERMANN

O perfil de Evermann foi selecionado para ser estendido na proposta deste trabalho. A escolha se deu pelo fato de que a maior parte das implementações encontradas são conceituais ou complexas demais, como em Zakaria, *et al.* (2002), Han, *et al*. (2005) e Mostefaoui e Vachon (2006).

A proposta de Evermann (2007) se difere das demais por ser uma proposta leve (*lightweight*) para projeto detalhado e por prover grande quantidade de propriedades da linguagem AspectJ, a mais conhecida do paradigma OA (Orientada a Aspectos) devido ao tempo investido no seu desenvolvimento, ainda em atividade (no momento da publicação deste estudo), que proveu grande quantidade de propriedades desta linguagem.

Segundo Evermann (2007), um problema em propor novas meta-classes é a dificuldade de uso com ferramentas já existentes, muitas vezes necessitando construir novas ou esperar a aprovação da OMG (*Object Management Group*) para que se torne disponível.

O embasamento em AspectJ é benéfico porque o diagrama de classes em nível detalhado é próximo da implementação; o AspectJ possui novos conceitos que ainda não foram padronizados, visto que não existiam na época da criação dos últimos documentos formalizadores do paradigma, como a ontologia de DSOA (BERG, CONEJERO E CHITCHYAN, 2005).

Na Figura 2 está a representação gráfica do perfil proposto por Evermann, após ser reimplementado para uso neste trabalho. Um perfil em si é representado por

um pacote estereotipado ("*profile*"). Dentro do perfil, há um conjunto de estereótipos e enumerações que servem de definição de conceitos do AspectJ.

Cada estereótipo concreto pode ser utilizado em nível de modelo, aplicando à instância da metaclasse seu nome de estereótipo e suas etiquetas valoradas próprias e herdadas (incluindo associações). Todo estereótipo estende uma ou mais metaclasses do metamodelo UML, podendo ser representado graficamente tanto como um relacionamento de extensão, a seta de centro preenchido, tanto como escrito entre colchetes ([]), modo usado na Figura 2. Por exemplo, o estereótipo "*Aspect*" possui especificado "[*Class*]", indicando que este estereótipo estende a metaclasse Class do metamodelo da UML.

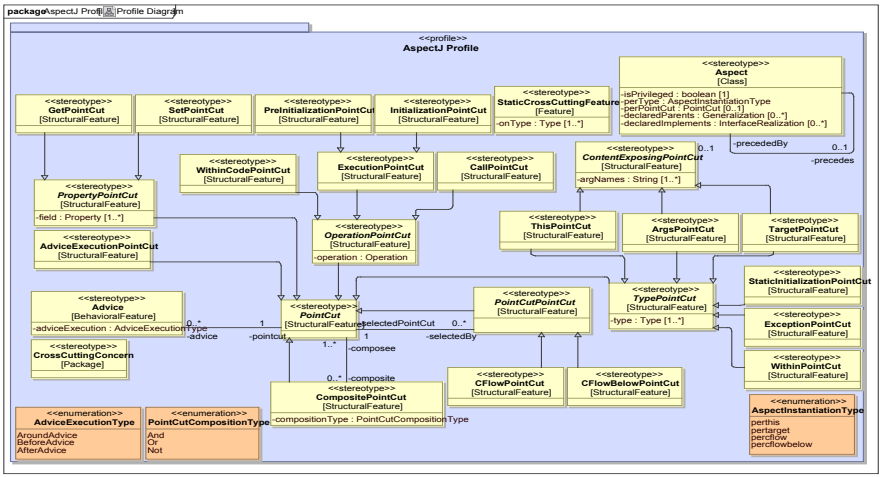
Pelo fato dos estereótipos serem redefinidos em melhores especificações, sua explicação pode ser agrupada pelos seguintes estereótipos principais:

- *Aspect*: Extensão da meta-classe "*Class*" e define aspectos. Um aspecto pode ter em seus compartimentos conjuntos de junção, adendos e definição de entrecorte estático, seu relacionamento com estes estereótipos se dá através de atributos definidos na meta-classe "*Class*", que também traz definições que correspondem tanto a aspectos quanto a classes convencionais, devido à sua similaridade. Aspectos podem definir generalizações e implementações externas de interfaces e classes, definir seu privilégio e ordem de precedência conforme regras de concorrência de aspectos no AspectJ.
- *Advice*: Extensão da metaclasse "*BehavioralFeature*" e define adendos. Isso se dá pelo fato de que adendos são similares ao comportamento de uma classe. De modo mais simplificado, "*BehavioralFeature*" é uma meta-classe superior a "*Operation*", que define operações de classes. "*Aspect*" não precisa ser associado a "*Advice*" pelo fato desta capacidade ser permitida às classes, cuja meta-classe o estereótipo "*Aspect*" estende. O adendo possui uma etiqueta valorada, originalmente no formato de associação, que o liga ao conjunto de junção e uma que define seu tipo de execução: "*AroundAdvice*", "*BeforeAdvice*" e "*AfterAdvice*" para adendos durante, antes e depois, respectivamente.
- *CrossCuttingConcern*: Extensão da meta-classe "*Package*" e define interesse transversal. O estereótipo

mais simples do perfil serve para marcar pacotes que agrupam aspectos que tratam de um interesse transversal. Porém, seu uso é opcional e não é definido em AspectJ.

- *PointCut:* Extensão da meta-classe StructuralFeature e define conjuntos de junção. O estereótipo "*PointCut*" é abstrato, assim, o estereótipo não pode ser aplicado aos atributos de um aspecto no nível de modelo, apenas para suas subclasses concretas como "*CallPointCut*" ou "*ExecutionPointCut*", que definem conjuntos de junção de operações, "*GetPointCut*" e "*SetPointCut*" que definem conjuntos de junção de atributos, entre outros. Além disso, "*CompositePointCut*" agrupa conjuntos de junção em um único para assim ser utilizado pelos adendos.
- *Static Crosscutting Features*: Extensão da meta-classe "*Feature*" e define entrecorte estático. Isso permite aos aspectos introduzirem atributos e métodos em outras classes existentes.

Figura 2 – Diagrama concebido ao reimplementar a proposta de Evermann



Adaptado de: Evermann, 2007

Enfim, além destas qualidades e grande quantidade de conceitos encontrados na linguagem mais utilizada no paradigma, por ser um perfil leve, é possível implementá-lo em ferramentas de modelagem da UML já existentes. Assim, é possível modelar diagramas de classe de softwares orientados a aspectos, que é o tipo de diagrama mais necessário para projetar e documentar softwares em nível detalhado, o nível mais similar à estrutura de codificação final, como será mostrado a seguir.

Na Figura 3, há um modelo que utiliza o perfil de Evermann. Este software é baseado

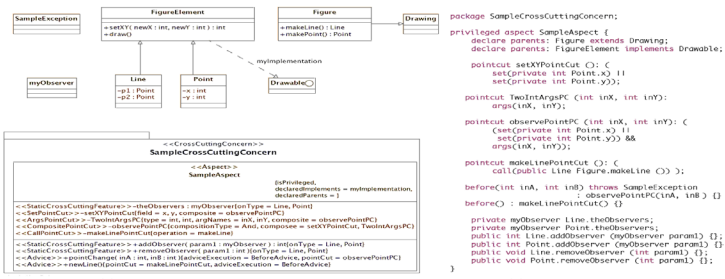
no padrão *Observer* de Gamma *et al.* (1995), segundo o qual um conjunto de objetos deve ser notificado quando outros são alterados. O interesse transversal é a notificação destes objetos observadores e é modelado dentro de um pacote com estereótipo "*<<CrossCuttingConcern>>*" e os aspectos desse interesse transversal são classes com o estereótipo "*<<Aspect>>*". O atributo "*isPrivileged*" do estereótipo "*Aspect*" se torna uma etiqueta do aspecto no modelo. Nesse exemplo, o aspecto insere uma generalização por meio da etiqueta "*declaredParents*" e uma implementação por meio da etiqueta "*declaredImplements*". Estes relacionamentos são modelados no modelo, porém o aspecto é declarado como responsável por essas inserções. Como a meta-classe "*Generalization"* não é uma subclasse da meta-classe "*NamedElement"*, nenhum nome é mostrado no valor da etiqueta "*declaredParents*", mesmo assim, é definida para a ferramenta a relação existente que é capaz de identificá-la.

Conjuntos de junção são atributos dos aspectos, já que são estereótipos de "*StructuralFeature*". Como por exemplo, o conjunto de junção "*setXYPointCut()*" de "*SampleAspect*" possui marcado o estereótipo como "*SetPointCut*". A etiqueta "*property"* herdada de "*PropertyPointCut*" se torna disponível ao atributo marcado como "*SetPointcut*" para listar as propriedades selecionadas por este ponto de junção.

"*StaticCrosscuttingFeatures*" são aplicáveis a qualquer elemento contido numa classe (métodos e atributos), que servem para declarar atributos ou métodos em outras classes dentro de aspectos. A etiqueta "*onType*" do estereótipo "*StaticCrosscuttingFeature*" é utilizada para definir a qual classe ou interface inserir o elemento. Por exemplo, o aspecto introduz uma operação pública em "*addObserver(myObserver)*" do tipo "*int*" nos tipos "*Line*" e "*Point*" e um atributo privado "*theObservers*" do tipo "*myObserver*" nos tipos "*Line*" e "*Point*".

Adendos são modelados como operações estereotipadas como "*<<Advice>>*". O estereótipo "*Advice*" é associado ao estereótipo "*Pointcut*". Assim, cada adendo no aspecto afeta um conjunto de junção especificado com o valor da etiqueta "*pointCut*".

Figura 3 – Exemplo utilizando o metamodelo de Everma



Fonte: Evermann, 2007

6 CONTEITO DE ANOTAÇÕES

Anotações, na linguagem Java, são especializações de interfaces e são declaradas em páginas de código Java diferindo de interfaces apenas por possuir na declaração um sinal de arroba (@) antes de simplesmente “interface”. Anotações não possuem operações implicitamente abstratas, possuem membros que representam etiquetas utilizáveis para determinada anotação.

Seu uso é recomendado para marcar elementos dentro de uma aplicação para descrever de modo conciso sua participação dentro do sistema. Nomes de anotações são geralmente adjetivos ou definições de suas características dentro do sistema como por exemplo: “Transacional” , “Protegido”, “Processo de Negócios”.

Anotações estão presentes no Java desde a terceira edição de sua especificação (GOSLING, JOY E STEELE, 2005), sendo a primeira implementação desta a versão “1.5”. As anotações podem ser aplicadas a vários elementos, como a métodos e pacotes, e não só a classes que as realizam, sendo que um elemento pode receber várias anotações.

As etiquetas devem possuir tipos básicos Java ou String, sendo também aceitos vetores desses tipos. Na Figura 4, há um exemplo de uma anotação válida com dois membros e na Figura 5, há um exemplo de sua utilização.

Figura 4 – Exemplo de anotação em Java 1.5 ou posterior

```
public @interface Bibliografia {

public String nome() default “Gottardi”;
public int paginas();
}
```

Figura 5 – Exemplo de uso da anotação anterior

```
@Bibliografia(paginas=10)
public class Exemplo {...
```

Deste modo, são declaradas e utilizadas as anotações, geralmente sem alterar o resultado do código gerado pelo compilador, exceções são quando anotações são utilizadas

para passar definições de compilação ao compilador, ao utilizar anotações pré-definidas. Porém, neste trabalho, serão utilizadas anotações genéricas, descritas nos capítulos a seguir.

7 MODELAGEM DE ANOTAÇÕES

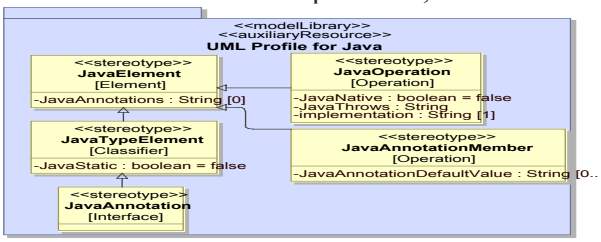
Na Figura 6, está parte do perfil UML para Java, para criar modelos de software específicos para esta linguagem.

Este perfil foi criado a partir de definições encontradas na linguagem Java (GOSLING, JOY E STEELE, 2005) e no MagicDraw (No Magic inc., 2008), uma ferramenta comercial de UML produzida por uma empresa integrante do Object Management Group, responsável pela Unified Modeling Language.

A importância desse perfil se dá pelo fato de representar anotações em nível de modelo e será utilizado em conjunto com a proposta deste trabalho, que será apresentada posteriormente.

Porém, existe uma restrição no perfil mostrado, identificada ao implementar a proposta deste trabalho: não é permitida a redefinição dos valores dos membros anotados para cada elemento. A solução para esse problema está no próximo capítulo.

Figura 6 – Parte do metamodelo para Java, definindo anotações



Adaptado de: No Magic inc., 2009.

8 ANOTAÇÕES E ORIENTAÇÃO A ASPECTOS

A aplicação de anotações na OA (Orientação a Aspectos) vem sendo incentivada (LADDAD, 2005) pelo fato de as anotações proverem melhor desacoplamento dos aspectos com as classes base, além de estar se tornando uma boa prática para o DSOA.

O ponto central beneficiado pela utilização desse conceito é um novo meio de capturar pontos de junção (LADDAD, 2005). Anotações permitem marcar declarações e estas declarações marcadas podem ser utilizadas para capturar pontos de junção.

Em alguns casos, usa-se de parte do nome da operação como expressão de conjunto de junção, algo permitido em AspectJ, por exemplo, ao utilizar um framework com nomes padronizados; todas as chamadas para certas ações podem possuir nomes significativos. Como no exemplo de RMI (*Remote Method Invocation*), presente no Java, que possui a maior parte das classes com nome iniciando em "*Remote*" e lançam exceções "*RemoteException*".

Porém, em outros casos, não é trivial capturar elementos que definem operações transacionais ou controle de concorrência que operam num nível que exige autenticação ou autorização, pois raramente suas assinaturas podem ser padronizadas para definir concisamente seus comportamentos para o sistema. Nesse caso, anotações podem prover uma boa solução.

Outro ponto importante de acordo com Laddad (2005), é que anotações podem prover nível multidimensional de tratamento, ou seja, podem tratar em níveis diferenciados, autenticação, transação, mesmo que seja de um único elemento, basta que o elemento de interesse receba múltiplas anotações.

Mais exemplos sobre implementações de softwares orientados a aspectos utilizando anotações para estes tratamentos estão na seção de Estudo de Caso.

9 PERFIL BASE DA PROPOSTA DESTE TRABALHO

O perfil base utilizado para a proposta contida neste trabalho ampara-se na versão aprimorada do perfil de Evermann (2007), conforme proposta em Gottardi e Camargo (2008).

Partindo de trabalhos passados, inicialmente foi reimplementado o perfil, conforme visível no diagrama da Figura 2. Para isto, foi utilizada a ferramenta MagicDraw UML (No Magic Inc., 2008), em seguida, foi possível testá-lo e foram identificadas as melhores opções para seu uso e inconsistências que não permitiam algumas propriedades válidas a serem modeladas (GOTTARDI E CAMARGO, 2008). Com isso, foram propostas melhorias para este perfil e assim a versão melhorada é a utilizada neste trabalho.

Entre as propriedades mantidas das versões deste perfil, estão as enumerações,

que são tipos que possuem valores listados e são utilizadas para definir tipos usados no AspectJ: os tipos de adendos ("*Advice*"/"antes", "depois" e "durante"), tipos de composição de conjuntos de junção ("*Pointcut*"/"não", "ou" e "e") e tipos de instanciação de aspectos, como "*per this*" e "*per target*".

Muitos dos estereótipos como "*StructuralFeature*", "*BehavioralFeature*", "*CompositePointCut*" e "*Advice*" foram mantidos inalterados e o uso geral do perfil sofreu poucas alterações. As principais correções, as visíveis pelo digrama, foram realizadas para tratar do problema em que o perfil original não permitia modelar alguns conceitos permitidos e algumas inconsistências as quais não eram válidas dentro da POA e DSOA.

As alterações diretas do perfil incluem a alteração do estereótipo "*PointCut*" de abstrato para concreto (para permitir modelagem de conjuntos de junção sem pontos de junção definidos) e a remoção do literal "*Not*" como tipo de composição, sendo esta função substituída para um atributo "*isComplemented*" de "*CompositePointCut*", pois anteriormente era possível definir composições não unárias com o operador de negação/complemento. Em trabalhos passados também foram propostas restrições OCL (Object Constraint Language – Linguagem de Restrição de Objetos) para evitar que certos erros semânticos sejam permitidos pelo perfil, aqueles que não são válidos para a linguagem AspectJ. Como as restrições são extremamente complexas e não aumentam a representatividade do perfil, não faz parte do escopo deste trabalho explicá-las e estendê-las.

Partindo deste perfil corrigido, o trabalho continua para aumentar sua capacidade de representação, entre outras melhorias que serão descritas no subcapítulo intitulado "Perfil Proposto".

10 PERFIL PROPOSTO

Neste trabalho, foi proposto um perfil UML para representar softwares orientados a aspectos com uso de anotações, conforme é possível em AspectJ. Este perfil foi baseado em um perfil analisado anteriormente que possuía inconsistências, além de não ser capaz de representar anotações.

Inicialmente, é apresentada uma extensão do perfil "*UML Profile for Java*", feita especificamente para ser integrada ao perfil "*AspectJ Profile*" proposto nesta seção, mostrado no diagrama na Figura 7.

O perfil "*UML Profile for Java*" possuía deficiência para representar redefinição de rótulos (ou etiquetas valoradas) de anotação, algo possível tanto em Java

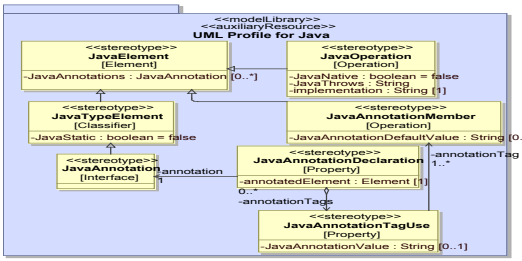
quanto em AspectJ. A solução escolhida para solucionar este problema foi criar os estereótipos "*JavaAnnotationDeclaration*", que representa a declaração genérica da anotação de um "*JavaElement*". Esta declaração agrega "*JavaAnnotationTagUse*", sendo que cada instância deste estereótipo representa uma redefinição de uma etiqueta de anotação, ou seja, "*JavaAnnotationMember*".

Esse processo também auxilia a permitir o uso de anotações a partir de aspectos. Basta que um aspecto possua uma propriedade marcada com "*JavaAnnotationDeclaration*", o que será demonstrado nos estudos de caso.

Para facilitar a definição externa de uso de anotações por aspectos, o atributo de estereótipo "*JavaAnnotationDeclaration*", anteriormente do tipo "*JavaElement*" foi alterado para tipo "*Element*", assim elementos a serem anotados não necessitam receber o estereótipo "*JavaElement*", provendo total inconsciência ao modelo base. Na Figura 7, há uma simplificação em relação à versão anterior, também utilizando as correções.

A proposta deste trabalho foi realizada a partir de uma versão corrigida e aprimorada do perfil de Evermann (2007) proposta em trabalhos passados, (GOTTARDI E CAMARGO (2008). O perfil de Evermann foi concebido de modo a permitir modelagem estática de aspectos (diagrama de classes com aspectos) seguindo as regras da linguagem AspectJ.

Figura 7 – Versão Corrigida e Simplificada de Parte do Perfil Java

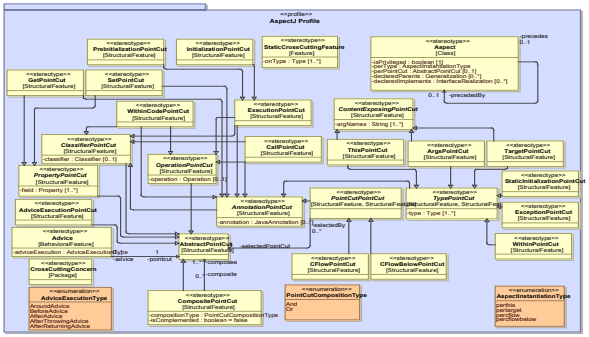


.

Na Figura 8, observa-se o diagrama do perfil proposto neste trabalho. É possível identificar que este é uma evolução do perfil melhorado por trabalhos anteriores, descrito anteriormente na Figura 2.

Também é importante notar que existem novos estereótipos criados para especificação de conjuntos de junção para uso de anotações como mais adições que permitem expressões por "*Classifier*", como por exemplo, capturar operações ou usos de atributos membros de uma determinada classe ou interface.

Figura 8 – Diagrama do perfil proposto dentro deste trabalho.



Alterações em relação ao anterior vão de sugestões simples de aprimoramento, como a mudança de nome do estereótipo "*PointCut*" para "*AbstractPointCut*", pois agora este é concreto em nível de metamodelo para representar os conjuntos de junção abstratos. Anteriormente como era definido como abstrato, não era possível criar conjuntos de junção sem pontos de junção definidos em nível de modelo. Acredita-se que a mudança de nome facilitará seu entendimento. Foram adicionados os tipos especializados do adendo "*after*" (após): "*after throwing*" (após lançamento de exceção) e "*after returning*" (após retorno normal). Foi adicionado o estereótipo "*ClassifierPointCut*" para capturar pontos de junção através de suas classes proprietárias.

Dentro das mudanças para prover representação de anotações, lembrando que o perfil deve ser utilizado em conjunto com o perfil de anotações Java, tendo como base "*ClassifierPointCut*", o estereótipo "*AnnotationPointCut*" permite representar expressões que envolvam elementos anotados.

Por fim, foi adicionado o rótulo "*declaredAnnotations*" para selecionar elementos anotados, para representar declaração das anotações do elemento via aspectos.

Maiores detalhes de uso do perfil proposto estão disponíveis no capítulo de estudo de caso, onde o perfil é utilizado para permitir a modelagem de software orientado a aspectos que utilize anotações para definir pontos de junção e também declaração de anotações através de aspectos. Além disso, softwares orientados a aspectos que tiram proveito deste conceito estarão visíveis graficamente graças ao perfil proposto, dando a possibilidade de avaliar sua eficácia para auxiliar o projeto das pequenas aplicações do estudo de caso.

11 ESTUDO DE CASO

Para averiguar se o perfil proposto é capaz de representar um software orientado a aspecto com uso de anotações, foram necessários estudos de caso modelados e implementados

com o auxilio do perfil proposto. Foram realizados três estudos de casos que fazem uso de anotações, com diagramas de classes que foram feitos graças ao perfil estendido. Por fim, o código correspondente foi escrito em AspectJ e testado. Porém, neste artigo, apenas um deles está presente.

Os estudos de caso exemplificam tanto usos tradicionais da orientação a aspectos para tratamento de interesses de pontos de junção facilmente identificáveis, como também, exemplos descritos como benéficos para uso de anotações.

Com o uso do perfil proposto é possível representar graficamente as anotações, suas declarações externas e definições de entrecorte. Isso não era possível em nenhuma proposta anterior referenciada por este trabalho.

O estudo de caso apresentado neste artigo é o de tratamento de segurança. Implementação de segurança via aspectos já é muito comum por sua característica transversal que causa grande repetição de código para testar se o usuário está autenticado em vários formulários de aplicações.

Existem propostas de Frameworks Transversais de Segurança (Camargo e Masiero, 2005). Com esta abordagem, o software base é produzido sem preocupação de exigência de autenticação. Um framework para tratar este interesse transversal é importado e por fim, o framework é ligado aos pontos interessantes à segurança, encontradas no software base, o que é chamado de instanciação. Para tanto, partes do software que devem ser protegidas de usuários não autorizados, recebem tratamento para redirecionar o controle para a autenticação. Esta autenticação pode ser bem genérica, como por exemplo, de formulário que peça uma senha a até uso de biometria: o software base, no máximo, só precisa saber a identificação do usuário, e não como e nem quando ele é autenticado.

Neste trabalho, foi implementado o uso de autenticação por aspectos. Para atenuar a maior complexidade deste estudo de caso, não foram utilizados todos os conceitos propostos como ideais para um Framework Transversal de Segurança, mas provê uma solução considerável para remover a dependência do programa para com a autenticação fazendo uso de aspectos com auxilio de anotações. Isso pode facilitar a instanciação, se aproveitar do conceito de anotações corretamente.

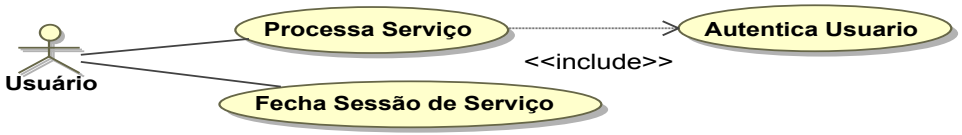
Inicialmente, existe uma biblioteca OO que possui tipos (classes e interfaces) e implementações necessárias para as mais variadas autenticações. Neste exemplo, todas as credenciais são definidas como vetores de "*String*" e os validadores verificam se esta

credencial está contida em um "*Vector*", ou seja, para avaliar se a credencial provida é válida. O validador ainda é capaz de invocar um método de uma interface que deve ser implementado para exibir mensagem ou qualquer outra ação que dependa do resultado da autenticação. No exemplo apenas um validador é provido, que verifica usuário e senha ignorando diferenciação de maiúsculas e minúsculas.

Um software que necessita de autenticação de usuário está é desenvolvido com o uso da biblioteca acima. Seu diagrama de caso de uso pode ser representado como o ilustrado na Figura 9.

Ao observar o diagrama da Figura 9, é visível que o caso de uso "Autentica Usuário" não é iniciado diretamente pelo usuário. Isso é um fator que sugere que este caso de uso não é um interesse base da aplicação. Para tratar esse requisito, foi escolhido o uso de OA. Como isso foi realizado é apresentado posteriormente.

Figura 9 – Possível diagrama de caso de uso do software-exemplo.

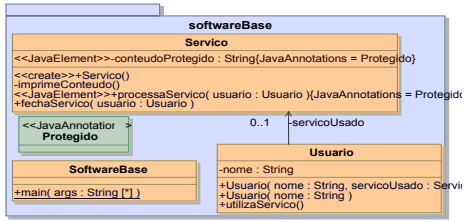


O núcleo de funcionalidade de software é arquiteturado e implementado sem a inclusão do caso de uso de requisito transversal. É sugerida neste trabalho a criação de uma anotação para cada caso de uso de requisito transversal. Nomes ideais para anotações (Laddad, 2005) especificam adjetivos que abstraem o que deve ser feito ao elemento (no exemplo), ou que define a natureza do comportamento (como no caso anterior).

Portanto, o software base pode ser arquiteturado da seguinte forma, confome representa o diagrama da Figura 10. A classe "Serviço" contém operações destinadas ao serviço, que são chamadas por um usuário, que é representado pela classe Usuário.

A biblioteca orientada a objeto não pode ser utilizada deste modo para que prover ao software base um tratamento completamente externo de modo orientado a aspecto, este é um caso de discrepância de paradigmas e isto deve ser tratado com algum tipo de mapeamento.

Figura 10 – Diagrama de classes do software base



Para tratar esta discrepância de paradigmas entre a biblioteca OO e o tratamento de segurança OA, um simples framework OA é criado, que serve de mapeamento entre a biblioteca de autenticação à instanciação do novo framework. Na Figura 11 está o diagrama de classes da biblioteca, descrita anteriormente, sendo mapaeada pelo framework OA. A função do framework é prover métodos necessários para chamar a biblioteca, que será herdada por um aspecto abstrato, aplicando tais métodos a um conjunto de junção abstrato que representa uma ocorrência que deve ser protegida. O adendo deste aspecto invoca a autenticação para permitir ou não a execução do ponto entrecortado.

Deste modo, a biblioteca OO pode ser utilizada com qualquer software orientado a aspectos, basta instanciá-la, tarefa que será apresentada em seguida.

Para que o framework seja ligado ao software base, é necessária a camada de instanciação, que é responsável por ligar a interface credencial ao Usuário, implementar os novos métodos abstratos e por fim, definir o conjunto de junção abstrato “servicoAlvo”. Para tanto, um aspecto é criado estendendo o aspecto abstrato. Além disso, são criadas implementações para as mensagens acarretadas pelos validadores. Diagrama visível na Figura 12.

Figura 11 – Framework mapeando a biblioteca de autenticação.

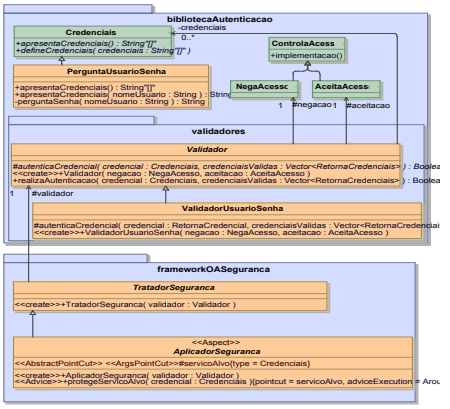
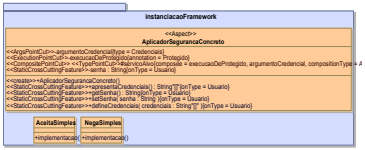


Figura 12 – Diagrama representando a instanciação do framework



12 OUTRAS CAPACIDADES DO PERFIL

São qualidades descritas anteriormente, mas não demonstradas nos estudos de caso: existem outras capacidades úteis que são permitidas tanto pelo perfil como em AspectJ. Definição de uso de anotação via aspectos, o que permite utilização do conceito de inconsciência; todo elemento Java pode ser anotado, ou seja, não é restrito às operações, mas também é possível incluir atributos, classes, pacotes e até outras anotações; os conjuntos de junção aplicáveis à anotação não se restringem a chamada e execução. Também incluem captura de elementos tipados, como uso de argumentos ("*args*"), este objeto ("*this*"), objeto alvo ("*target*"), exceções ("*exception*"), definições ("*set*"), acessos ("*get*"), entre outros.

13 ATIVIDADES REALIZADAS DURANTE O TRABALHO

Durante esse trabalho foi realizada uma revisão bibliográfica para obter conhecimento acerca do paradigma orientado a aspectos e de uma linguagem do mesmo paradigma. Em seguida, foram identificadas propostas existentes para permitir modelagem de software orientado a aspecto.

A proposta de Evermann (2007) foi selecionada e foi implementada na ferramenta de modelagem MagicDraw UML (No Magic inc., 2008). Após a implementação, foi possível modelar alguns sistemas orientados a aspectos, avaliar a fundo a proposta, identificando alguns problemas e melhorias.

Foi criada uma versão aprimorada do perfil de Evermann, sendo mais fiel ao paradigma e à linguagem de programação no qual foi embasado. Foram adicionados novos conceitos já presentes em AspectJ, possibilitando criação de novos modelos. O conceito mais importante dentre as contribuições é o uso de anotações.

Para adicionar o conceito de anotações ao perfil, foi realizada uma nova revisão bibliografica, na qual foi possível identificar boas práticas que justificam uso das anotações

com o paradigma (Laddad, 2005). Outro perfil que representa anotações em orientação a objeto foi importado para ser adaptado para permitir as novas abstrações relacionadas com o conceito no contexto da orientação a aspectos.

Por fim, como estudo de caso, sistemas foram implementados com o auxilio dos modelos até obter a codificação final, para averiguar se os conceitos necessários são visíveis no diagrama. Dentro dos estudos de caso, foi sugerida a criação de anotações para representarem casos de uso que representem interesses transversais, para anotarem os seus pontos de inclusão. Artefatos produzidos estão disponíveis com o autor.

CONSIDERAÇÕES FINAIS

Este trabalho apresentou uma proposta de perfil UML leve derivada do trabalho de Evermann (2007) com uso de correções propostas em trabalhos passados (Gottardi e Camargo, 2008) para modelagem de softwares orientados a aspectos.

Após a especificação do perfil proposto, foram realizados estudos de caso com modelos feitos através do perfil especificado e que não seriam possíveis de modelar utilizando os outros perfis apresentados.

Após sucessos apresentados neste e em outros trabalhos, supõe-se que modelagem de aspectos é um fato viável de ser atingido, apenas exige esforços para atualizações, adição de conceitos e união das propostas compatíveis para criar uma modelagem completa para o paradigma, unindo modelagem estática e dinâmica.

Uma sugestão para trabalhos futuros é avaliar se o perfil realmente é suficiente para projetos de maior porte e se possível melhorá-lo unindo com perfis que definem estereótipos para outros tipos de diagramas, não apenas diagramas de classes.

REFERÊNCIAS BIBLIOGRÁFICAS

ALDAWUD, O.; ELRAD, T.; BADER, A.. **UML profile for aspect-oriented software development**. Em: Proceedings of the AOM workshop at AOSD 2003. 2003.

ANDRADE, C.A.R.; SANTOS, A.L.M.; BORBA, P.H.M.. **AspectH**: Uma Extensão Orientada a Aspectos de Haskell. Em: WASP' 04 - workshop do 18º SBES . Brasília, Brasil. 2004.

ASPECTJ TEAM. **The AspectJ(TM) Programming Guide**. Palo Alto Research Center, inc. http://www.eclipse.org/aspectj/doc/released/progguide/index.html. Acessado em 11 de Agosto de 2009, 2003.

BERG, K.; CONEJERO, J.; CHITCHYAN, R.. **AOSD ontology 1.0**: public ontology of aspect orientation. Em: Common Foundation for AOSD. 2005.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I.. **UML**: Guia do Usuário. 2.ed. Rio de Janeiro: Elsevier,. 2006.

CAMARGO, V.V.; MASIERO, P.C.. **Frameworks Orientados a Aspectos**. Em: Anais do 19º Simpósio Brasileiro de Engenharia de Software. Uberlândia, MG, Brasil. Outubro de 2005.

EVERMANN, J.. **A Meta-Level Specification and Profile for AspectJ in UML**. Em: AOSD 2007. Victoria University Wellington, Wellington, New Zealand. 2007.

ELRAD, T. et al. **Discussing Aspects of AOP**. Communications of the ACM, v. 44, n. 10, p. 33-38, 2001.

FUENTES, L.; SÁNCHEZ, P.. **Towards Executable Aspect-Oriented UML Models**. In: Workshop AOM´07. Vancouver, 2007.

GAMMA, E. et al. **Design Patterns**: Elements of Reusable Object-Oriented Software. Addison-Wesley. Boston, Estados Unidos da América. 688p. 1995.

GEORG, G.; FRANCE, R.. **UML Aspect Specification Using Role Models**. Em: Proc. of the 8th International Conference on Object-Oriented. 2002.

GOSLING, J.; JOY, B.; STEELE, G.. **Java(TM) Language Specification**: The Third Edition. Addison-Wesley. 3.ed. Boston, Estados Unidos da América. 688p. Junho de 2005.

GOTTARDI, T.; CAMARGO, V.V.. **Melhorias em um perfil UML para Desenvolvimento de Software Orientado a Aspectos**. Em: WASP' 08 - workshop do 22º SBES. Campinas, São Paulo, Brasil. Outubro de 2008.

GROHER, I.; BAUMGARTH, T.. **Aspect-Orientation from Design to Code**. Munique, Alemanha. 2004.

HAN Y.; KNIESEL, G.; CREMERS, A.B.. **Towards Visual AspectJ by a Meta Model and Modeling Notation**. Em: AOSD - AOM 2005. 2005.

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; Maeda, C.; Lopes, C.; Loingtier, J.M.. **Aspect-oriented programming**. Em: ECOOP'97. LNCS. (1997)220-242. 1997.

KICZALES, Gregor *et al.* **Aspect–oriented programming with AspectJ**. In *OOPSLA* (Object-Oriented Programming, Systems, Languages and Applications)*'01, Tutorial*, Tampa FL. 2001.

KISELEV, I.. **Aspect-Oriented Programming with AspectJ**. Sams. 1.ed. 288p. Julho de 2002.

LADDAD, R.. **AOP and Metadata**: A perfect Match. AOP@WORK. IBM. Abril de 2005.

LADDAD, R.. **AspectJ in Action**: Practical Aspect-Oriented Programming. Manning Publications. 1.ed. 512p. Julho de 2003.

MOSTEFAOUI , F.; VACHON, J.. **Formalization of an aspect-oriented modeling approach**. Em: Proceedings of Formal Methods 2006. Hamilton, ON. 2006.

NO MAGIC INC.. **MagicDraw UML**. http://www.magicdraw.com/. Acessado em 30 de Abril de 2008, 2008.

OBJECT MANAGEMENT GROUP. **Unified Modeling Language**: Version 2.0 Meta Object Facility Specification. Disponível em http://www.omg.org/spec/MOF/2.0/. Acesso em: 17 de Julho de 2009, 2006.

OBJECT MANAGEMENT GROUP. **Unified Modeling Language**: Version 2.2 Superstructure Specification. Disponível em http://www.omg.org/cgi-bin/doc?formal/09-02-02. Acesso em: 17 de Julho de 2009, 2009.

PAWLAK, R. et al. **A UML notation for aspect-oriented software design**. Em: Proceedings of the AOM with UML work shop at AOSD. 2002.

PRESSMAN, R.S.. **Software engineering**: a practitioner's approach. McGraw-Hill. 5 ed. Estados Unidos da América: McGraw-Hi, 2001.

SEBESTA, R.. **Conceitos de linguagens de programação**. 4 ed. Porto Alegre: Bookman, 2002.

SPINCZYK, O.; Gal, A.; SCHRÖDER-PREIKSCHAT, W.. **AspectC++**: an aspect-oriented extension to the C++ programming language. Em: Proceedings of the Fortieth International Conference on Tools P. Sydney, Australia. 2002.

STEIN, D.; HANENBERG, S.; UNLAND, R.. An UML-based Aspect-Oriented Design Notation For AspectJ. 2002.

SUZUKI, J.; YAMAMOTO, Y.. **Extending UML with aspects**. Em: Proceedings of the third ECOOP Aspect-Oriented Programming Work. 1999.

ZAKARIA, A.A.; HOSNY H.; ZEID, A. **A UML Extension for Modelling Aspect-Oriented Systems**. Em: Proceedings of Workshop of Aspect Oriented Modeling with UML of Aspect Oriented Software Development Conference (AOSD), 2002.