CrossMDA2: Uma Abordagem Baseada em Modelos para Gerência de Evolução de *Pointcuts*

Victor H. Fernandes, Flávia C. Delicato, Paulo F. Pires
Thais Batista, Uirá Kulesza

Departamento de Informática e Matemática Aplicada – Universidade Federal do Rio Grande do Norte (UFRN) – Natal – RN – Brazil

victorhcf@gmail.com, Flavia.delicato@dimap.ufrn.br,
paulo.pires@dimap.ufrn.br, thais@dimap.ufrn.br, uira@dimap.ufrn.br

Resumo. *Este artigo propõe uma abordagem baseada em modelos para gerência da evolução de pointcuts em aplicações orientada a aspectos. A abordagem é centrada na definição de pointcuts com base em visões conceituais. Na abordagem os pointcuts são especificados como parte de visões do modelo conceitual e o processo é apoiado por uma ferramentas baseada em Model-Driven Architecture (MDA). Como prova de conceito, o artigo apresenta duas versões de um estudo de caso, configurando um cenário de evolução que mostra como o uso de pointcuts baseados em visões conceituais auxiliam na detecção e minimização da fragilidade de pointcuts. Para avaliação da proposta é usada o Goal/Question/Metric (GQM) juntamente com métricas para análise da eficiência na definição dos pointcuts.*

1. Introdução

O Desenvolvimento Orientado a Aspectos (DSOA) [Wiley et al. 2001] tem se destacado como uma abordagem que provê alto grau de modularização dos sistemas de software e favorece o reuso de artefatos desses sistemas (componentes, classes, etc). DSOA adiciona uma nova abstração, o *aspecto*, que visa modularizar características transversais de sistemas, promovendo a sua decomposição horizontal por meio de sua estruturação em modelo base e modelo de aspectos. O modelo base representa os comportamentos que podem ser facilmente isolados em unidades tradicionais de decomposição (como classses) enquanto o modelo de aspectos representa os comportamentos que se espalham por vários pontos do sistema. Os pontos de execução do modelo base interceptados pelos aspectos denominam-se *join points* e são definidos por meio de de *pointcuts*, expressões que classificam *join points* com características comuns. Para que tais pontos de interceptação sejam definidos, é necessário que expressões de *pointcut* referenciem tanto o aspecto quanto o modelo base [Wiley et al. 2001]. Da mesma forma com que a maioria dos paradigmas de programação conta com referências simbólicas, por exemplo, chamando uma função por meio de seu nome, DSOA conta com referências às propriedades estruturais ou comportamentais da aplicação por meio das definições dos *pointcuts*. Mais precisamente, os *pointcuts* impõem regras de design (do inglês *design rules*) [Sullivan et al. 2005] que os desenvolvedores do modelo base precisam conhecer plenamente para evitar que uma evolução no modelo base faça com que algum *join point* seja capturado acidentalmente ou um *joint point* não seja capturado quando necessário. Estas regras de design surgem do fato que as definições de *pointcuts* especificam regras estruturais ou comportamentais as quais o modelo base deve estar de acordo para que os *pointcuts* funcionem corretamente. Essa abordagem pode resultar em um grande

acoplamento entre os elementos dos modelos base e de aspecto [Koppen e Stoerzer 2004]. Por esse motivo, a orientação a aspectos demonstra fragilidade diante da evolução do software, uma vez que se o modelo base evoluir, todas as definições de *pointcuts* de cada aspecto acopladas a tal modelo, podem sofrer impacto e, portanto, devem ser revisadas para garantir consistência. Na literatura esse problema é denominado de fragilidade de *pointcuts* [Kellens et al. 2006a].

Existem formas de amenizar o problema da fragilidade de *pointcuts* fazendo uso, por exemplo, de linguagens de definição de *pointcuts* mais expressivas [Suzuki & Yamamoto 1999] ou utilizando definição de *pointcuts* baseada em comportamento [Ostermann 2005]. Porém, estas técnicas apenas amenizam o problema e, além disso, não fornecem meios para verificar se a evolução do software introduziu alguma falha devido a algum novo código incompatível com as definições dos *pointcuts*. Desta forma, a verificação tem que ser feita manualmente, ou seja, a cada evolução do sistema os desenvolvedores precisam revisar cada *pointcut* juntamente com o código da evolução para garantir que não houve uma violação. Em sistemas grandes, nos quais há uma grande quantidade de membros da equipe e uma grande quantidade de código, é improvável que todos os desenvolvedores conheçam todas as regras de design existentes. Fazer esse controle manualmente é um risco muito grande e na prática fatalmente irá ocorrer violação de alguma regra de design durante uma evolução do sistema.

Estudos recentes [Kellens et al. 2006b], [Batista e Vieira 2007] exploram a especificação de *pointcuts* baseados em modelos conceituais, os quais referenciam um modelo conceitual do software ao invés de referenciar diretamente o modelo base. O modelo conceitual é uma abstração de mais alto nível dos conceitos presentes no modelo de negócios do sistema. As abstrações existentes no modelo conceitual são criadas considerando características estruturais e comportamentais do modelo de negócios. Tal abordagem eleva o grau de abstração do relacionamento criado entre o modelo base e o de aspectos. Os modelos conceituais, por serem representações mais abstratas do modelo de negócios, possuem uma maior estabilidade que o modelo de negócios em si. Conseqüentemente, os efeitos da evolução do modelo de negócios sobre os *pointcuts* são minimizados, já que estes são definidos em termos do modelo conceitual e não de instâncias especificas desse modelo, como tipicamente acontece em DSOA tradicional. Além disso, abordagens de *pointcuts* baseadas em modelos provêem meios automatizados para documentar e verificar a consistência entre o modelo conceitual e o de negócio.

Apesar das abordagens de especificação de *pointcuts* baseados em modelos conceituais fornecerem mecanismos que favorecem a evolução de sistemas orientados a aspectos, as propostas atuais [Kellens et al. 2006b], [Batista e Vieira 2007] fornecem soluções em que as abstrações que povoam o modelo conceitual são definidas no nível de código, não sendo aplicáveis às fases iniciais do desenvolvimento de software. Como consequência, essas abordagens não definem processos de desenvolvimento que incluam formalmente as fases de construção do modelo conceitual e, posteriormente, de validação do modelo base quando este evolui. Além disso, como as abstrações do modelo conceitual são definidas em baixo nível (código), a sua especificação e gerência tende a ser mais complexa quando comparado com a manipulação de construções no nível de modelo.

Nesse contexto, este trabalho propõe elevar o nível de abstração da especificação de *pointcuts* baseados em modelos. Para atingir tal objetivo, propomos o uso de técnicas de desenvolvimento dirigido a modelos (do inglês, *Model Driven Development* - MDD) na construção e evolução de sistemas orientados a aspectos por meio de uma nova

instanciação da abordagem de *pointcuts* orientados a modelos [Kellens et al. 2006b], tendo como base os conceitos e técnicas da *Model Driven Architecture* (MDA) [MDA 2008]. A abordagem proposta é denominada de *pointcuts baseados em visões conceituais,* pois as definições dos *pointcuts* são criadas com base em visões existentes no modelo conceitual. Ao integrar a MDA com a abordagem de modelos conceituais buscamos oferecer um processo de desenvolvimento que inclui a especificação e manipulação do modelo conceitual em todas as fases do ciclo de vida do software. Essa proposta é realizada por meio de uma extensão do arcabouço CrossMDA [Alves et al. 2006], denominada *CrossMDA2*, que provê suporte para a gerência do processo MDA e ainda executa a verificação automática do sincronismo entre o modelo de negócio e o modelo conceitual. O CrossMDA é um arcabouço criado para se trabalhar com DSOA em uma abordagem MDA, com o objetivo de prover separação de interesses na dimensão vertical, por meio da transformação entre modelos, e dimensão horizontal, por meio do uso de aspectos. Em sua versão inicial o CrossMDA permitia apenas a definição de *pointcuts* usando a abordagem tradicional e, conseqüentemente, criava um forte acoplamento entre as definições de *pointcuts* e o modelo base.

Este artigo está estruturado da seguinte forma: na Seção 2 é descrita a abordagem proposta no trabalho. Na Secão 3 é apresentado um estudo de caso desenvolvido para ilustrar a abordagem e analisar os seus benefícios. Os trabalhos relacionados são descritos na Seção 4. A Seção 5 apresenta as conclusões do trabalho.

2. Especificação de *Pointcuts* baseados em Visões Conceituais

Este trabalho propõe uma instanciação do conceito de *pointcuts* baseados em modelos, denominada *pointcuts* baseados em visões conceituais. As visões conceituais formalizam regras de *design* existentes no modelo de negócios. Elas descrevem conceitos existentes que são do interesse do desenvolvedor, agrupando entidades do modelo (classes, métodos, etc) que compartilhem propriedades estruturais e/ou comportamentais comuns. Tais classificações (agrupamentos) são especificadas por meio de expressões escritas na linguagem OCL (*Object Contraint Language*) [OCL 2008] que, além de ser uma linguagem capaz de expressar classificações e validações sobre elementos de modelos UML, é uma linguagem padrão definida pela OMG [OMG 2008], tendo compiladores disponíveis em diversas ferramentas de modelagem existentes. As visões conceituais são importantes pois, além de fazer com que a definição dos *pointcuts* dependa de uma estrutura mais abstrata e, conseqüentemente, menos frágil no que se refere a evolução, elas fornecem um mecanismo que permite a validação automática das regras de *design* existentes. Na presente proposta, o modelo conceitual consiste em um conjunto de visões conceituais em que cada visão define formalmente uma regra de *design*, possibilitando assim que as definições dos *pointcuts* sejam feitas com base nessas visões.

Neste trabalho, a criação e manipulação do modelo conceitual, no contexto do desenvolvimento de sistemas OA é gerenciada por um processo dirigido por modelos. Para tanto, foi realizada uma extensão do arcabouço CrossMDA, alinhando-o com a proposta de desenvolvimento de *pointcuts* baseados em modelos. Na Seção 2.1 serão abordadas as principais modificações realizadas no CrossMDA, bem como no processo de desenvolvimento. A Seção 2.2 apresenta detalhes dos metamodelos e modelos propostos.

2.1. O processo CrossMDA2

O CrossMDA [Alves et al. 2006] é um arcabouço que integra DSOA em uma abordagem MDA, com o intuito de prover separação de interesses nas dimensões vertical e

horizontal. Ele permite a definição dos conceitos transversais de forma independente do modelo de negócios. No CrossMDA, tanto o modelo que representa características transversais quanto o modelo de negócios são independentes de plataforma (PIM - *Platform-Independent Model*). O CrossMDA realiza o *weaving* desses modelos por meio de uma transformação, resultando em um modelo no qual as características transversais interceptam as características de negócio. Após o *weaving,* o modelo resultante pode ser submetido a uma nova transformação que o instancie em uma plataforma específica obtendo um artefato executável. Os princípios e funcionamento básicos do CrossMDA foram mantidos neste trabalho que adicionou novos artefatos de software e novas transformações.

Os atores que desempenham atividades no fluxo de trabalho do CrossMDA2 são: (i) *Arquiteto de aspectos* – responsável por criar e manter os modelos de aspectos; (ii) *Arquiteto de negócios* – responsável por conhecer as regras de negócio da aplicação e criar um modelo de negócio que represente estas regras, sendo tal modelo criado independente de plataforma e devendo conter apenas os interesses não transversais; (iii) *Arquiteto de sistemas* – representa o usuário responsável por conhecer as regras de design da aplicação e criar o modelo conceitual que formalize estas regras; também é responsável por selecionar quais os modelos conceituais, de negócios e de aspectos que serão usados no processo de desenvolvimento, assim como realizar o mapeamento entre eles.

O processo do CrossMDA2 é ilustrado na Figura 1. O processo inicia (fase 1) com a criação de três modelos que contêm diferentes visões do software, descritos a seguir.

i. O modelo de aspectos é uma representação abstrata independente de plataforma (PIM) dos interesses transversais. Tais interesses são modelados usando o metamodelo UML adicionado de um *profile* especifico do CrossMDA2 no qual aspectos são modelados como classes anotadas com o estereótipo *aspect* [Stein 2002] e organizadas em pacotes. No CrossMDA2 um pacote de aspecto é uma entidade que agrega aspectos relacionados, ou seja, aspectos que tratam da mesma categoria de interesses.

ii. O modelo de negócios é uma representação do negócio da aplicação, e de forma semelhante ao de aspectos, independente de plataforma. Para construí-lo é utilizado o metamodelo UML. O processo CrossMDA2 não impõe nenhuma restrição na especificação do modelo de negócios, podendo ser utilizado qualquer elemento UML válido para representar as entidades e relacionamentos do modelo.

iii. O modelo conceitual é uma representação abstrata das características estruturais da aplicação. Ele formaliza as regras de *design* que deverão ser seguidas durante todo o desenvolvimento e evolução do sistema no nível de modelagem. Além disso, contém restrições e relacionamentos entre os elementos do modelo de negócios de modo a tornar possível realizar uma validação automática para verificar a sua corretude com relação ao modelo de negócios. O modelo conceitual é construído com base no metamodelo conceitual descrito na Seção 3.

Ainda na fase 1, há a seleção dos modelos no CrossMDA2. Após a seleção segue a fase de validação dos modelos (fase 2), uma fase nova na qual é realizada a validação das restrições definidas no modelo conceitual junto ao modelo de negócio com o objetivo de identificar possíveis inconsistências que possam ter sido adicionadas ao modelo de negócio. Caso não ocorram falhas durante a validação, o fluxo continua normalmente na fase 3. Caso ocorram falhas, elas serão exibidas em uma interface visual de forma que o

arquiteto consiga identificar o problema. Com os problemas identificados o fluxo deve ser reiniciado desde a fase 1, os modelos de negócio e/ou conceitual devem ser revisados para resolver os problemas identificados e o processo inicia novamente. Após a fase de validação tem-se a atividade de mapeamento (fase 3) e, em seguida, o *weaving* dos modelos (fase 4). Como esta fases estão presentes também na versão inicial do CrossMDA, elas não serão detalhadas aqui, maiores informações em [Alves et al. 2006].

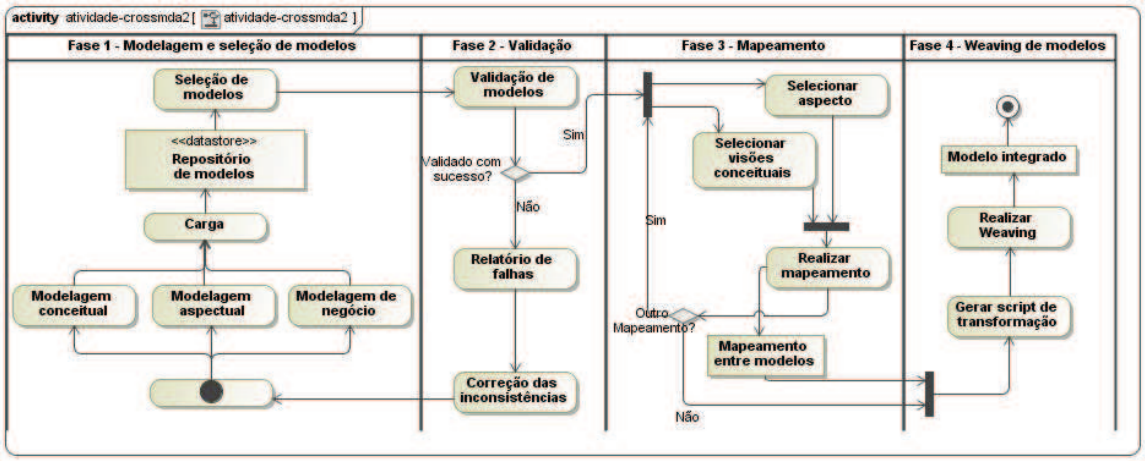

Figura 1 - Processo CrossMDA2

2.2. Modelos e Metamodelos Utilizados no processo CrossMDA2

Como estamos usando uma abordagem MDA, é necessário conhecer as quatro camadas usadas na estruturação MDA: (i) a camada M0 é a **instância do modelo** de dados e objetos utilizados que representam os dados que se deseja descrever; (ii) a camada M1 compreende o **modelo que descreve** os dados da camada M0, por exemplo, esquemas relacionais, modelos UML e, no nosso caso, o modelo conceitual, de negócio e de aspectos; (iii) a camada M2 contém o **metamodelo** que define a estrutura e semântica do modelo do nível M1, por exemplo, o metamodelo da UML e o metamodelo conceitual definido na Figura 2; (iv) a camada M3 compreende a descrição da estrutura e semântica do metametamodelo, em outras palavras, é uma linguagem abstrata para definição de diferentes tipos de metadados, por exemplo, o meta-metamodelo MOF [OMG 2008] que é responsável pela definição das entidades como classes, atributos e operações.

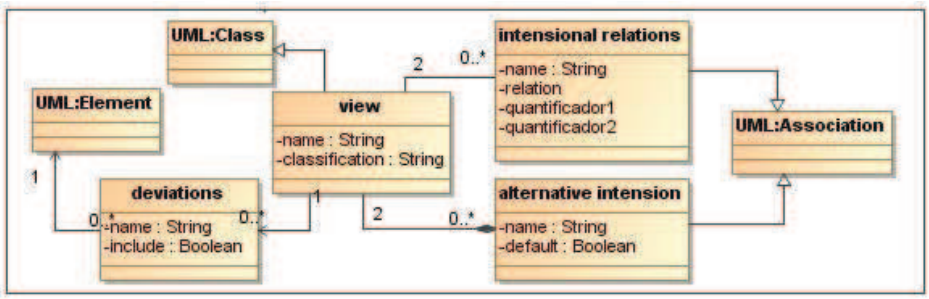

Figura 2 - Metamodelo conceitual

Sendo assim, a definição do modelo conceitual é realizada a partir de um metamodelo especificamente concebido para criação de modelos conceituais (Figura 2). No metamodelo definem-se as visões (*views*) como tendo uma classificação (*classification*) que é uma expressão em OCL. Cada visão pode ter ainda desvios e

relações intensionais[1] (*intensional relationship*) ou alternativas (*alternative intensions*), descritas posteriormente.

Como exemplo, para modelar o conceito "todos os métodos que realizam atualização da base de dados" pode ser criada uma visão conceitual chamada *Métodos de atualização* que classifica todos os métodos que estão marcados com o estereótipo *@updateMethod*. A expressão definida na Figura 3 classifica as entidades representadas por *self* que sejam do tipo *método* (*Operation*) e que tenham aplicado o estereótipo *@updateMethod*.

```
self.oclIsKindOf(Operation) and self.oclAsType(Operation)
        .getAppliedStereotype( 'CrossMDABusinessProfile::updateMethod')->notEmpty()
```

Figura 3 - Exemplo de *pointcut* baseado em visões conceituais

Como pode-se observar no metamodelo (Figura 2), as visões possuem uma descrição e podem possuir vários desvios (*deviations*). Um desvio é uma inclusão ou exclusão explicita na visão de um elemento do modelo de negócio. Por exemplo, se uma classificação diz que todas as classes que estão no pacote de *DAO* representam classes que implementam o padrão *DAO* (*Data Access Object*) [Gamma et al 1995], não seria possível ter uma fábrica de *DAOs* no mesmo pacote, visto que a entidade fábrica não implementa o padrão *DAO*. Para possibilitar que a fábrica permaneça no pacote *DAO*, ela deve ser indicada explicitamente como uma exceção à visão que classifica os *DAOs*, por meio de um desvio. Assim, a visão que classifica os *DAOs* permaneceria válida mesmo tendo uma entidade, neste caso a fábrica de *DAOs*, que não se enquadra na sua classificação, visto que esta exceção está identificada por meio do desvio.

No caso de evolução no sistema de software pode ocorrer que as visões conceituais tornem-se inconsistentes, acarretando na captura ou perda acidental de algum novo elemento, fato que é semelhante ao problema da fragilidade de *pointcuts*. Por exemplo, no caso de um sistema que faz uso da visão conceitual definida na Figura 3 evoluir e ser adicionado um método que realiza alteração do banco de dados, porém que não está marcado com o estereótipo @updateMethod, esse método não seria corretamente classificado por esta visão. Para evitar esse problema, as visões conceituais possuem, além das classificações, um conjunto de relacionamentos que impõem relações e restrições entre as visões de forma que as mesmas possam ser verificadas para garantir que nenhuma inconsistência será adicionada ao modelo. Existem dois tipos de relacionamentos entre as visões conceituais: relações alternativas e relações intensionais, que foram definidas em (Kellens et al. 2006b) na sua proposta de definição de *pointcuts*.

As relações alternativas definem que uma visão é alternativa, ou equivalente, a outra visão. Ou seja, uma visão A é alternativa a visão B se os elementos do modelo de negócio capturados pela visão A são exatamente iguais aos elementos do modelo de negócio capturados pela visão B. Caso isso não ocorra, deve ser identificada uma inconsistência. Já as relações intensionais definem relações binárias entre as visões. As relações binárias são definidas conforme uma fórmula lógica apresentada em (Kellens et al. 2006b). Um exemplo desse tipo de relação é apresentado na Figura 4. Esta relação diz que para todo elemento classificado na visão que define os Métodos de Consulta existe um elemento que se enquadra na visão que classifica os *DAOs* tal que o elemento da visão *Métodos de Consulta* é implementado pelo elemento da visão *DAO*.

[1] O termo intensional neste artigo aparece escrito com "s" pois remete a idéia de intensão.

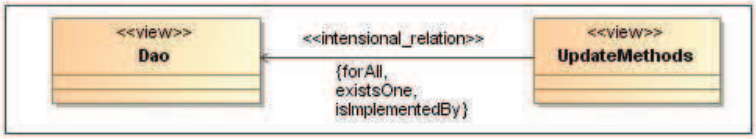

Figura 4 - Exemplo de relação intensional

Toda relação intensional verifica alguma propriedade entre elementos de dois conjuntos: visão1 e visão2. Caso necessário, o arquiteto pode ainda adicionar novas relações. Para tal, ele deve criar a relação e uma classe que implemente a interface *IntensionalRelationValidator* e dentro do método *validate* realizar a verificação.

Como visto, a proposta de *pointcuts* baseados em visões conceituais é uma nova instanciação da abordagem de *pointcuts* baseados em modelos inserida em um contexto MDA com o objetivo de alcançar o máximo de reuso de artefatos de software e ainda obter um sistema que possibilita a evolução por meio das classificações existentes nas visões conceituais. A abordagem é suportada pelo *CrossMDA2*, responsável por auxiliar na execução do fluxo do processo MDA e também por realizar a verificação automática da consistência entre o modelo conceitual e de negócio.

3. Estudo de caso: Sistema Acadêmico

Nesta Seção será apresentado um estudo de caso baseado em um sistema real, implementado como prova de conceito da abordagem proposta. Nele serão tratados problemas encontrados em um cenário real de desenvolvimento e será mostrado como o uso dos *pointcuts* baseados em visões conceituais auxiliou a detecção e eliminação da ocorrência do problema da fragilidade de *pointcuts*. O sistema é baseado no SIGAA (*Sistema Integrado de Gestão de Atividades Acadêmicas*), um software de gestão de atividades acadêmicas, desenvolvido e usado pela UFRN [SIGAA 2009]. Para o estudo de caso foram extraídos alguns componentes da arquitetura de duas versões diferentes do SIGAA identificadas por *release* 1 e *release* 2. O SIGAA é dividido em quatro camadas principais: visão, controle, negócio (ou serviço) e integração (persistência de dados).

3.1. Descrição do Estudo de Caso

Para ilustrar e avaliar a solução proposta foi conduzido um estudo de caso com as seguintes etapas:

1. Três aspectos foram selecionados para serem aplicados ao SIGAA.
2. Os aspectos selecionados foram aplicados a *release* 1 utilizando a abordagem de *pointcuts* baseados em visões conceituais.
3. Em seguida os mesmos aspectos foram também aplicados a *release* 2, inclusive usando as mesmas definições de *pointcuts da release 1*. Na seqüência, a validação das restrições definidas no modelo conceitual foi executada para identificar possíveis inconsistências adicionadas devido a evolução.
4. Por fim, foi realizada a análise do impacto da evolução do *release 1* para o *release 2* nas definições de *pointcuts* usadas, baseadas nas visões conceituais e, em seguida, comparado o impacto da evolução obtida usando a técnica de *pointcuts* baseados em visões conceituais com a técnica utilizada baseada em visões intensionais proposta por Kellens (Kellens et al. 2006b].

Os aspectos escolhidos para o estudo de caso foram transação, *logging* e autenticação. O aspecto de transação foi especificado como sendo aplicado às classes que executam métodos de negócio, identificadas como classes de serviço ou ainda classes de negócio. O aspecto de *logging* deve interceptar todos os métodos que atualizam o banco

de dados com o objetivo de registrar qualquer alteração de persistência no banco. O aspecto de autenticação intercepta qualquer método de negócio ou dos controladores para verificar se o usuário encontra-se autenticado. Na Figura 5 encontra-se um diagrama da arquitetura utilizada no estudo de caso, ilustrando seus diferentes componentes.

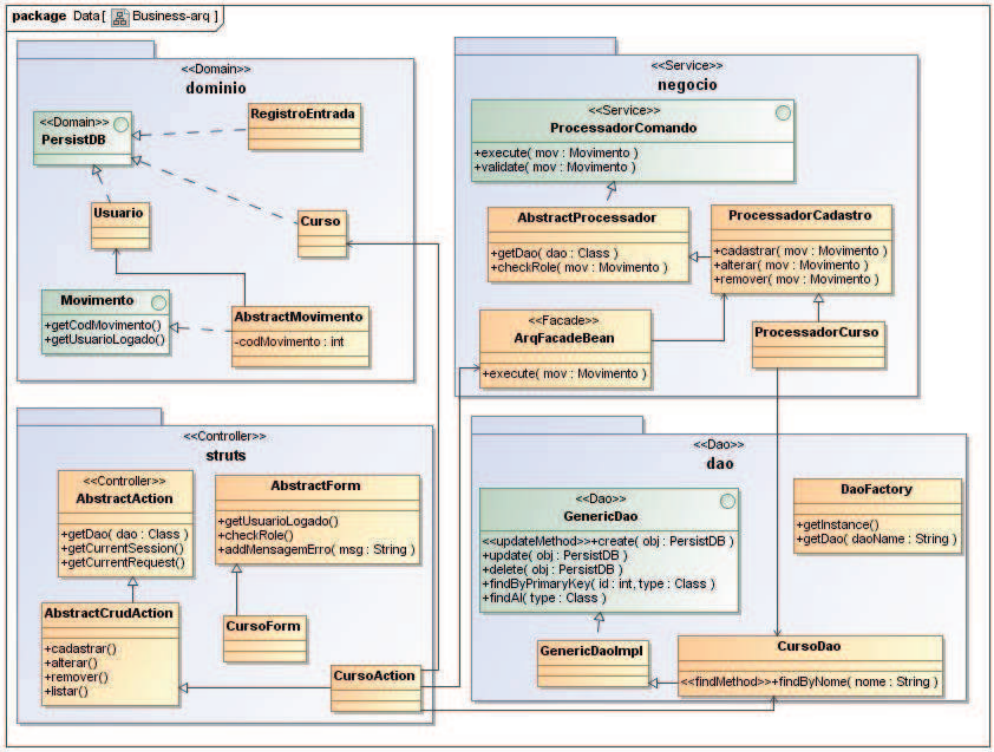

Figura 5 - Modelo de negócio - *Release* 1

3.2. Especificação e Implementação Inicial do Sistema – *Release* 1

Na *release* 1 foram selecionados alguns componentes da arquitetura do SIGAA desenvolvidos pela equipe do SIGAA em Janeiro de 2006, e então, a partir desses componentes foi construído, pelo arquiteto de negócios, um modelo de negócio (fase 1 do CrossMDA, Figura 1) para ser usado como entrada no processo do CrossMDA2. Esse modelo pode ser visualizado na Figura 5 em que podemos perceber as camadas lógicas da aplicação divididas em quatro pacotes: negócio, domínio, struts e dao. No pacote de negócios estão as classes que usam o padrão *EJBCommand* [Gamma et al 1995] e também uma *fachada* (*ArqFacadeBean*) de acesso, da qual todas as chamadas aos processadores de comando são realizadas. O pacote *DAO,* usa o padrão de projeto *DAO* [Alur D. et al. 2003] e contém os objetos que fazem acesso ao SGBD. Todos os *DAOs* do sistema herdam de *GenericDaoImpl*, uma classe que contém a implementação de vários métodos comuns aos *DAOs*. Além disso, é usado o padrão *Factory* [Gamma et al 1995] por meio da entidade *DaoFactory* para criar instâncias dos *DAOs*. No pacote controlador estão os componentes que fazem recepção e encaminhamento das requisições web. Como nesta *release* é utilizado o *Struts* como framework web no sistema, há uma entidade abstrata - *AbstractAction* - que é uma instanciação do padrão *Layer Supertype* [Alur D et al 2003] e implementa as funcionalidades comuns a todos os controladores do Struts. Assim, todos os controladores do sistema herdam de *AbstractAction*.

O CrossMDA2 tem como uma das entradas o modelo conceitual, selecionado pelo arquiteto na fase 1 do processo. A Figura 6 mostra algumas visões e classificações do modelo conceitual especificado para a *release* 1. Esse modelo é uma instância do

metamodelo descrito na Figura 2. Os estereótipos utilizados nesse modelo foram definidos em um *profile* UML especificamente criado para este fim. Algumas das visões conceituais definidas no modelo conceitual são: (i) *controladores* – visão que classifica todas as classes que estão na hierarquia de alguma entidade marcada com o estereótipo @controller (*controller*) e todas as classes que estão no pacote marcado com o estereótipo @controller (*controller2*); (ii) *domínio* – que representa as classes que estão na hierarquia de alguma entidade marcada com o estereótipo @domain (*domain*) e todas as classes que estão no pacote marcado com @domain (*domain2*); (iii) *métodos de consulta* – visão que classifica todos os métodos marcados com o estereótipo @findMethod (*findMethod*); e (iv) *métodos de atualização* – visão que classifica todos os métodos marcados com @updateMethod (*updateMethod*). Observe que quando um mesmo conceito é representado por duas visões diferentes, estas são alternativas.

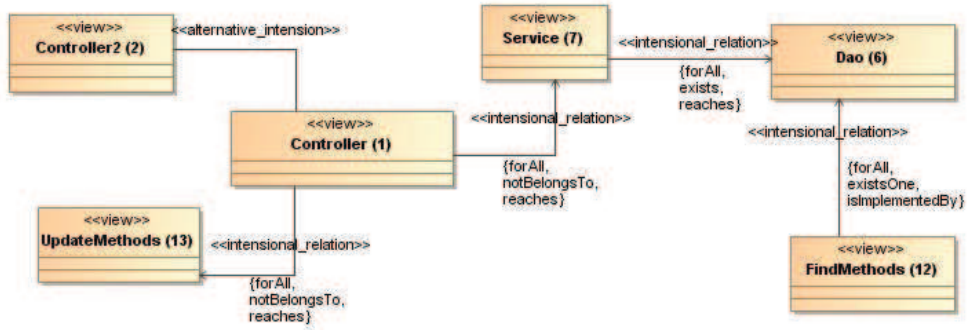

Figura 6 – Parte do Modelo conceitual - *Release* 1

Foi necessário fazer o uso dos desvios para garantir a integridade das visões. No caso das classes de serviço, por exemplo, a classe *ArqFacadeBean* é a fachada da camada de negócio e, por isso, encontra-se no pacote de serviço, porém não é um processador de comando e não está na hierarquia de uma classe marcada com o estereótipo @service, portanto, ela teve que ser registrada como um desvio da visão que classifica os processadores, de forma a manter o modelo consistente. Na Figura 6, as entidades são representadas com o estereótipo *view* e os relacionamentos entre essas classes representam as restrições existentes entre as visões. Como visto, há dois tipos de restrições: as relações intensionais e as alternativas. As relações intensionais são marcadas com o estereótipo *intensional_relation*, já as relações alternativas são marcadas com o estereótipo *alternative_intension*. Dois exemplos de relações intensionais especificadas para o sistema são: (i) para todo controlador não existe um processador tal que o controlador referencie o processador; restrição que permite identificar se algum controlador acessa um processador sem ser por meio da fachada, e (ii) para todo método de atualização existe um *dao* tal que o método de ação é implementado pelo *dao*. Esta restrição vai garantir que nenhum método de atualização do banco de dados seja implementado fora da camada *dao*.

Por fim, há o modelo de aspectos, criado na fase 1 do CrossMDA2, e contendo os três aspectos usados na implementação do estudo de caso. O aspecto de *Logging* é responsável por registrar qualquer alteração que seja realizada no banco de dados. O aspecto *Auth* é responsável por garantir que apenas usuários autenticados terão acesso aos métodos definidos na camada de controladores quanto de negócio. O aspecto *Persistence* é responsável por injetar transação em todo o contexto de negócio, garantindo que toda operação invocada na camada de negócio seja transacional.

3.3. Evolução do Sistema – *Release* 2

Para a *release* 2 foram extraídos componentes da arquitetura desenvolvidos pela equipe do SIGAA no mês de janeiro de 2008, dois anos após a primeira *release*. O modelo de negócio extraído pode ser visto na Figura 7. Tanto o modelo conceitual quanto o modelo de aspectos não sofreram alterações, sendo os mesmos usados na *release* 1. As principais alterações realizadas no modelo de negócios que tiveram impacto na porção da arquitetura usada no nosso estudo de caso são: (i) foi adicionado o framework JSF ao projeto, portanto, foi criada uma nova hierarquia de classes na camada de controladores; (ii) foram adicionadas operações de negócio que os usuários não precisam estar logados para realizar; (iii) foram criadas classes que realizam operações temporariamente e fazem atualização do banco de dados, porém não são processadores; (iv) foram adicionadas classes de domínio que não são gerenciadas pelo *hibernate* por fazer parte de outro banco de dados, portanto, a manipulação destas entidades acontece de forma diferente das outras. Após a extração do modelo de negócio para a *release* 2, todos os modelos foram processados pelo CrossMDA2 sendo executada a validação entre o modelo conceitual e o de negócio. Essa validação acusou a ocorrência de inconsistências no modelo de negócio. A evolução de um software pode acarretar em inconsistências no mesmo, devido a falhas humanas, tais como, por exemplo: (i) a adição de uma classe que não segue a regra de *design* definida para a mesma; ou (ii) ainda devido a uma nova situação que não existia no momento da construção do sistema e, conseqüentemente, não possui uma regra de *design* que a defina, como é o caso do uso de uma nova tecnologia na camada de controle. As inconsistências encontradas a partir do uso da abordagem CrossMDA2, constam na Tabela 1, assim como as respectivas soluções para eliminá-las. Após a solução das inconsistências apresentadas o processo foi reiniciado e finalizado sem problemas, com todos os aspectos sendo aplicados nos devidos lugares.

Tabela 1 - Evoluções e Impactos da *release* 2

Evolução	Descrição	Impacto	Solução
Adicionado a tecnologia JSF no projeto.	O projeto foi iniciado utilizando Struts e depois passou a utilizar o JSF, alterando a estrutura e hierarquia dos controladores.	O o aspecto que trata de autenticação não iria capturar os controladores que estivessem na hierarquia do controlador JSF.	Para solucionar esse problema bastou marcar a classe *AbstractMBean* com o estériótipo Controller.
Inscrição para seleção da pós.	Até então só havia acesso a páginas na qual o usuário encontrava-se logado no sistema. Porém, foi adicionada a inscrição em processos seletivos on-line que é realizada por usuários não autenticados.	Nesse caso o aspecto de autenticação não iria permitir o acesso a esta funcionalidade devido ao usuário não estar logado no sistema.	Como é uma situação nova teria que ser criado um estereótipo *@withoutAuthentication* e os controladores que permitem acesso sem autenticação teriam que ser marcados com esse estereótipo, como é o caso do controlador *InscricaoSelecaoMBean*.
Criação de tarefas automáticas.	Também foi adicionada ao projeto a execução de tarefas automáticas, que realizam processamento sobre os dados do banco de dados, as entidades que realizam esse tipo de tarefa ficam no pacote *timer*.	Estas classes podem realizar alterações no banco de dados e por isso devem estar em um contexto transacional, sendo que por não se enquadrar na definição de serviço elas não eram capturadas pelo aspecto de transação.	Para solucionar esse problema bastou enquadrar estas entidades no conceito de serviço, para isto foi necessário marcar o pacote timer e a classe *CalcularHistoricoThread* com o estereótipo *service*.
Adição de classe de domínio não mapeada pelo hibernate.	A classe Bolsa é uma classe utilizada para fazer uma integração com outro banco de dados. Portanto, ao ser adicionada no projeto a classe Bolsa não pôde ser mapeada pelo hibernate, não estando, por exemplo, na hierarquia de PersistDB.	Como esta classe não é mapeada pelo hibernate os métodos utilizados para realizar a persistência das informações não serão os mesmos das outras classes, que herdam os métodos definidos em *GenericDao*. Desta forma o aspecto de *logging* não registraria as alterações de banco realizadas com esta entidade.	Para solucionar esse problema, bastou marcar os métodos que realizam a persistência dos dados desta entidade com o estereótipo *updateMethod*. Desta forma, ele passou a ser capturado pelo aspecto de *logging*.

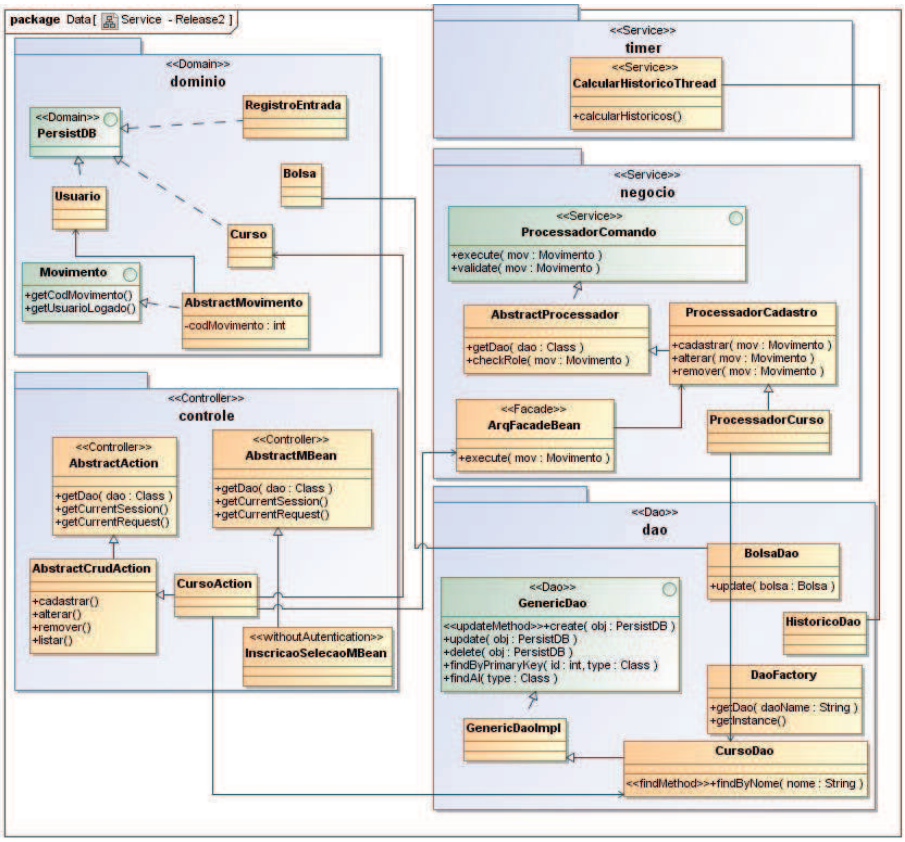

Figura 7 - Modelo de negócio - *Release* 2

3.4. Análise e Discussão dos Resultados

Para analisar os resultados obtidos foi usada a abordagem *Goal/Question/Metric* (GQM) [Basili et al. 1994] em conjunto com a métrica para análise da eficiência na definição dos *pointcuts* proposta por Khatchadourian [Khatchadourian 2008]. GQM é uma abordagem orientada a metas e utilizada em engenharia de software para avaliação de produtos e processos de software. GQM parte do princípio de que toda coleta dos dados deve ser baseada num fundamento lógico, em um objetivo ou meta, que é documentado explicitamente. O primeiro passo nessa abordagem é definir metas a serem alcançadas na avaliação. Após a identificação das metas, um plano GQM é elaborado para cada meta selecionada. O plano consiste, para cada meta, em um conjunto de questões quantificáveis que especificam as medidas adequadas para sua avaliação [Basili et al. 1994]. As questões identificam a informação necessária para atingir a meta e as medidas definem operacionalmente os dados a serem coletados para responder as perguntas.

Para analisar os analisar o estudo de caso foi definido as duas metas exibidas na Tabela 2. As métricas usadas para analisar a primeira meta foram propostas em [Khatchadourian 2008] e apresentam quantitativamente a eficiência da atividade de validação no que diz respeito à detecção das possíveis inconsistências adicionadas ao software. Estas métricas foram aplicadas em dois momentos: (i) inicialmente foi aplicada após a evolução do software e antes de realizar qual tipo de alteração (T1); (ii) em seguida as mesmas métricas foram aplicadas no software no instante após a execução da validação e resolução das incoerências detectadas (T2). Com isso observamos que após a resolução das inconsistências conseguimos obter uma confiabilidade bem significativa, nesse estudo de caso alcançamos uma confiabilidade de 100%, porém em um ambiente

real é possível que ainda haja alguma inconsistência não detectável pela validação do modelo conceitual, isto dependerá das restrições definidas no modelo conceitual.

Tabela 2 - Avaliação dos resultados utilizando GQM

Meta 1: Analisar a abordagem de *pointcuts* baseados em visões conceituais com relação a eficiência na detecção de inconsistências no modelo de negócios durante a evolução do software.			
		T1	T2
Questão 1.1: Qual o nível de precisão das definições dos *pointcuts*?	Métrica: Número de *join points* corretamente capturados (TP – *True Positives*)	9	11
	Métrica: Número de *join points* erroneamente capturados (FP – *False Positives*)	3	0
	Métrica: Número de *join points* erroneamente não capturados (FN – *False Negatives*)	5	0
	Métrica: Número de *join points* corretamente não capturados (TN – *True Negatives*)	20	20
Questão 1.2: Qual a proporção de *join points* corretamente identificados pelos *pointcuts*?	Métrica: Recall = TP / (TP + FN)	0,64	1
Questão 1.3: Qual a proporção dos *join points* corretamente não identificados pelos *pointcuts*?	Métrica: Fall-Out = FP / (FP + TN)	0,13	0
Questão 1.4: Qual a eficiência da validação na detecção de inconsistências?	Métrica: Confidence = 1 – FallOut/Recall	0,79	1
Meta 2: Analisar utilização dos *pointcuts* baseados em visões conceituais com a finalidade de identificar o custo de reparação das inconsistências detectadas a fim de sincronizar o modelo de negócios e conceitual.			
Questão 2.1: Qual o percentual de situações em que foi necessário alterar *pointcuts* para solucionar as inconsistências?	Métrica 2.1: Percentagem de casos em que foi necessário alterar a definição do *pointcut* para solucionar a inconsistência? (quantidade de alterações nas PCD / quantidade de inconsistências existentes) * 100	Resp. 2.1: 0 / 4 * 100 = **0%**	
Questão 2.2: Qual o percentual de situações em que foi necessário alterar o modelo de negócios para solucionar as inconsistências?	Métrica 2.2: Percentagem de casos em que foi necessário alterar a estrutura do modelo de negócios para solucionar a inconsistência (quantidade de alterações no modelo de negócios / quantidade de inconsistências existentes) * 100	Resp. 2.2: 0 / 4 * 100 = **0%**	
Questão 2.3: Qual o percentual de situações em que foi necessário alterar as visões conceituais para solucionar as inconsistências?	Métrica 2.3: Percentagem de casos de foi necessário alterar a definição das visões para solucionar a inconsistência (quantidade de alterações nas visões / quantidade de inconsistências existentes) * 100	Resp. 2.3: 1 / 4 * 100 = **25%**	
Questão 2.4: Qual o percentual de situações em que foi necessário alterar os estereótipos para solucionar as inconsistências?	Métrica 2.3: Percentagem de casos de foi necessário alterar os estereótipos para solucionar a inconsistência (quantidade de alterações nos estereótipos / quantidade de inconsistências existentes) * 100	Resp. 2.3: 4 / 4 * 100 = **100%**	

Analisando a Tabela 2 pode-se perceber que mesmo tendo realizado consideráveis evoluções no modelo de negócio, inclusive nos componentes e tecnologias usadas na arquitetura, a resolução das inconsistências se deu de forma simples e sem grandes impactos. Isto se deve ao fato de que os conceitos existentes no modelo conceitual não são definidos diretamente usando a estrutura do modelo de negócio, mas fazem referências aos conceitos implícitos que existem nele fazendo uso dos estereótipos. Todos os elementos do modelo de negócio são marcados com estereótipos que criam uma representação conceitual mais abstrata e mais fácil de manter e estender. Como as visões conceituais são definidas com base nos estereótipos especificados estas são, conseqüentemente, mais flexíveis e também mais fáceis de manter. Por exemplo, no estudo de caso mesmo tendo sido adicionado uma nova tecnologia no projeto (JSF) e novos elementos na arquitetura, como as classes que realizam tarefas temporárias, não foi

preciso redefinir nenhuma visão ou remodelar o modelo de negócios. A adequação do novo cenário ao modelo conceitual consistiu somente na alteração dos estereótipos aplicados aos modelos conceituais e de negócio.

Observamos ainda que as restrições existentes no modelo conceitual foram suficientemente eficientes para detectar todas as inconsistências decorrentes devido à evolução do software e, podemos ainda constatar, que em nenhum caso foi necessário alterar as definições dos *pointcuts* ou ainda o modelo de negócios, embora a aplicação dos estereótipos no modelo de negócio tenha sofrido alterações. Ainda, em apenas 25% dos casos foi necessário realizar alterações nas visões do modelo conceitual. No trabalho de (Kellens et al.2006b), o estudo de caso envolve o mesmo problema da adição de uma nova hierarquia de controladores. Nele, para solucionar esse problema foi necessário realizar alterações no código base, adicionando os elementos a uma mesma hierarquia de entidades e também foi necessário refazer a definição das visões para considerar a nova raiz da hierarquia criada.

É importante enfatizar que, como o processo de desenvolvimento é totalmente apoiado pelo CrossMDA2, inclusive a atividade de validação dos modelos, não é possível prosseguir com o processo caso seja adicionada alguma inconsistência ao modelo. No trabalho de [Kellens et al.2006b], a validação é invocada de uma ferramenta externa ao processo de desenvolvimento, e esta validação pode simplesmente não ocorrer, ou ainda, os desenvolvedores podem ignorar as inconsistências apresentadas durante a validação.

4. Trabalhos relacionados

Existem diversas abordagens que tratam do problema da fragilidade de *pointcuts*. Porém, nenhuma das soluções encontradas resolve tal problema de forma satisfatória e ainda não endereçam a questão da consistência das regras de design imposta diante da evolução do software. [Suzuki e Yamamoto 1999], [Kiczales e Mezini 2005] e [Stoerzer e Graf 2005] propõem formas de tornar a definição dos *pointcuts* mais robusta e flexível reduzindo a ocorrência do problema da fragilidade de poincuts, porém esses trabalhos não propõem formas de verificação automática da ocorrência do problema, fazendo com que ele continue existindo. Outra abordagem que também realiza a definição dos *pointcuts* com base em uma abstração dos elementos do modelo base foi proposta por [Batista e Vieira 2007] utilizando a linguagem RE-AspectLua. A linguagem RE-AspectLua permite que as definições dos *pointcuts* sejam feitas com base em interfaces que serão definidas em tempo de execução para aplicação, e a ligação do aspecto com o código base se dá na carga em tempo de execução por meio da implementação destas interfaces até então desconhecidas. Porém, a definição das interfaces são feitas em termos do código base e não é fornecido nenhuma maneira de verificar a consistência destas definições caso haja alguma evolução do software, tornando necessário que todas as implementações das interfaces sejam revistas.

[Kellens et al. 2006b] propõem que definições de *pointcuts* sejam feitas com base em modelos abstratos e, a partir dessas definições, é possível verificar automaticamente, por meio de uma ferramenta, a ocorrência de inconsistências entre o modelo abstrato e código, evitando assim a propagação do problema de fragilidade de *pointcuts* ao código da aplicação. Este trabalho diferencia-se do trabalho de Kellens pelo fato dos conceitos abstratos criados no modelo conceitual não fazerem uso diretamente do modelo de negócios, mas sim de conceitos de mais alto nível de abstração definidos no modelo de negócio por meio de estereótipos, o que garante uma evolução menos impactante e ao mesmo tempo facilita a resolução de inconsistências, como visto no estudo de caso.

5. Conclusão

O problema da fragilidade de pointcuts pode ser visto como um dos fatores que inibe a adoção de soluções orientadas a aspectos em sistemas reais pelo fato de ocasionar um grande risco de adicionar defeitos durante a evolução no software. Este trabalho propõe uma solução para o problema, baseada no trabalho de [Kellens et al. 2006b], porém aplicado a um ambiente de desenvolvimento MDA. Assim como no trabalho de Kellens, a solução utiliza a técnica de pointcuts baseados em modelos, porém propomos uma nova técnica que faz uso de visões conceituais definidas usando UML e OCL. Ao contrário da proposta de Kellens, as visões conceituais não fazem referência direta à estrutura do modelo de negócio, mas sim a abstrações definidas por meio dos estereótipos. A abordagem proposta produz consideráveis benefícios no momento de solucionar as inconsistências identificadas durante a evolução de sistemas orientados a aspectos. O estudo de caso conduzido corroborou as vantagens da proposta aqui apresentada.

Referências

Alves, M. P., et al. (2006) CrossMDA - Arcabouço para Integração de Interesses Transversais no Desenvolvimento Orientado a Modelos. Dissertação de Mestrado, UFRJ, Brasil.

Alur D., et al (2003) Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall.

Basili, V. et al. (1994). The Goal Question Metric Approach. Encyclopedia of Software Engineering.

Batista, T. and Vieira, M. (2007). RE-AspectLua - Achieving Reuse in AspectLua. Journal of Universal Computer Science, 13(6):786–805.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.

Kellens, A., et al. (2006a). A Model-Driven Pointcut Language for More Robust Pointcuts. Software Engineering Properties of Languages For Aspect Technology (SPLAT).

Kellens, A., et al. (2006b). Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts. (pp. 501-525). Spring-Verlag.

Khatchadourian, R. et al (2008). On the Assessment of Pointcut Design in Evolving Aspect-Oriented Software. Proceedings of the 2nd International Workshop on Assessment of Contemporary Modularization Techniques (ACoM '08).

Kiczales, G., & Mezini, M. (2005). Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. ECOOP 2005

Koppen, C., & Stoerzer, M. (2004). PCDiff: Attacking the Fragile Pointcut Problem. European Interactive Workshop on Aspects in Software, 2004.

Miller, J., & Mukerji, J. (2003). MDA Guide Version 1.0.1. Object Management Group (OMG).

OCL (2008) http://www.omg.org/technology/documents/formal/ocl.htm. Acesso em maio de 2009.

OMG (2008) Meta-Object Facility (MOF™), version 1.4. Disponível em: http://www.omg.org/technology/documents/formal/mof.htm Acesso em out. de 2008

Ostermann, K., Mezini, M., & Bockisch, C. (2005). Expressive Pointcuts for Increased Modularity. ECOOP 2005.

SIGAA (2009) http://www.sigaa.ufrn.br. Acesso em junho de 2009.

Stoerzer, M., & Graf, J. (2005). Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. (pp. 653-656). IEEE Computer Society.

Sullivan, K., Griswold, W., Song, Y., Chai, Y., Shonle, M., Tewari, N., et al. (2005). On the Criteria to be Used in Decomposing Systems into Aspects. FSE'2006, ACM Press.

Suzuki, J., & Yamamoto, Y. (1999). Extending UML with Aspects: Aspect Support in the Design Phase. (pp. 299-300). Springer-Verlag.

UML2 (2008) UML2 metamodel. http://www.omg.org/cgi-bin/doc?ptc/2004-10-05. Acesso em out. de 2008.

Wiley, P., et al. (2001). Aspect-oriented programming. Comm. of the ACM.