INSTITUTO MILITAR DE ENGENHARIA

Cap MARCELO LUZ SANDE E OLIVEIRA

MODELAGEM DE ARQUITETURA DE SOFTWARE ORIENTADA A ASPECTOS COM UML 2.0

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para obtenção do título de Mestre em Ciências em Sistemas e Computação.

Orientadores: Prof[a]. Christina F. G. Chavez - D.Sc.
Prof. Ricardo Choren Noya - D.Sc

Rio de Janeiro
2007

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80-Praia Vermelha
Rio de Janeiro-RJ CEP 22290-270

O48m Oliveira, Marcelo Luz Sande e
Modelagem de Arquitetura de Software Orientada a Aspectos com UML 2.0 / Marcelo Luz Sande e Oliveira. - Rio de Janeiro: Instituto Militar de Engenharia, 2007.
114 p.: il.

Dissertação de mestrado do curso de Sistemas e Computação - Instituto Militar de Engenharia- Rio de Janeiro, 2007

1. Engenharia de Software 2. Arquitetura de Software. 3. Orientação a Aspectos. I. Oliveira, Marcelo Luz Sande e II. Instituto Militar de Engenharia. III. Modelagem de Arquitetura de Software Orientada a Aspectos com UML 2.0.

CDD 005.2

INSTITUTO MILITAR DE ENGENHARIA

CAP MARCELO LUZ SANDE E OLIVEIRA

MODELAGEM DE ARQUITETURA DE SOFTWARE ORIENTADA A ASPECTOS COM UML 2.0

Dissertação de Mestrado apresentada ao Curso de Mestrado em Sistemas e Computação do Instituto Militar de Engenharia, como requisito parcial para obtenção do título de Mestre em Ciências em Sistemas e Computação.

Orientador: Prof. Ricardo Choren Noya - D.Sc

Orientadora:Prof^a. Christina F. G. Chavez - D.Sc.

Aprovada em 11 de janeiro de 2007 pela seguinte Banca Examinadora:

Prof. Ricardo Choren Noya - D.Sc do IME - Presidente

Prof^a. Christina F. G. Chavez - D.Sc. da UFBA

Prof^a. Thais Vasconcelos Batista - D.Sc. da UFRN

Prof^a. Maria Claudia Reis Cavalcanti - D.Sc. do IME

Rio de Janeiro
2007

AGRADECIMENTOS

A Deus em primeiro lugar, pela saúde e pela oportunidade de realizar esse trabalho.

A minha mãe, pelo esforço e dedicação com que me educou e ajudou na formação de meu caráter.

Ao meu pai, pelo incentivo e aconselhamento sempre que precisos.

A minha pequena, pelo carinho e pela felicidade que me proporciona.

A minha irmã pela paciência e pela ajuda na confecção das figuras.

A tia Lígia pelo apoio incondicional no momento em que foi necessário.

Em especial, aos orientadores Ricardo e Christina. Pela forma que o trabalho foi conduzido, pela disponibilidade e, principalmente, pela amizade. Podem contar comigo para o que for preciso.

Agradeço ainda a todos os professores que, ao longo do curso, contribuíram com seus ensinamentos para realização do presente trabalho.

Por fim, agradeço a todos aqueles que contribuíram de alguma forma para o êxito desse trabalho.

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

AD-AUML	*Architecture Description in Aspectual UML*
ADL	*Architecture Description Language* (Linguagem de Descrição Arquitetural)
AO	*Aspect-Oriented* (Orientado a aspectos)
DSOA	Desenvolvimento de Software Orientado a Aspectos
IME	Instituto Militar de Engenharia
MOF	*Meta-Object Facility*
OCL	*Object Constraint Language*
POA	Programação Orientada a Aspectos
SQL	*Structured Query Language*
UFBA	Universidade Federal da Bahia
UFPE	Universidade Federal de Pernambuco
UFRN	Universidade Federal do Rio Grande do Norte
UML	*Unified Modeling Language*

RESUMO

Atualmente, a definição da arquitetura de software possui um papel fundamental no processo de desenvolvimento. A arquitetura de um software é a descrição dos subsistemas e componentes e suas relações, fornecendo uma visão geral do software e facilitando a comunicação com os interessados. O propósito básico da definição de arquitetura é projetar uma estrutura para o software para servir de ponto de partida para a fase detalhada do projeto. As linguagens de descrição arquitetural (ADLs) permitem uma padronização e alguma formalização na descrição da arquitetura.

No entanto, as ADLs não são amplamente empregadas devido a falta de ferramentas que dêem suporte ao seu uso e a falta de regras de rastreabilidade para as outras fases. Uma forma de contornar este problema seria utilizar a UML 2.0, considerada um padrão de facto para modelagem de software. Embora a UML 2.0 tenha apresentado uma evolução em relação à UML 1.4, principalmente no que diz respeito à modelagem arquitetura, ela ainda encontra-se distante da representação proporcionada pelas ADLs.

Outro ponto que vem ganhando destaque é o desenvolvimento de software orientado a aspectos (DSOA). O objetivo do DSOA é modularizar os interesses transversais que encontram-se espalhados e entrelaçados pelo sistema. Inicialmente concebido focando a fase de implementação, atualmente o DSOA está presente em todas as fases do desenvolvimento, inclusive na arquitetura. Embora tenham surgido diversas soluções para a representação de aspectos em nível arquitetural (AO-ADLs), essas propostas sofrem da mesma crítica das ADLs regulares.

Neste sentido, este trabalho propõe uma extensão da UML 2.0 para que seu pacote arquitetural incorpore os conceitos das ADLs textuais e represente relacionamentos transversais entre componentes. Além disso, este trabalho apresenta um conjunto de regras de transformação para uma linguagem de projeto detalhado orientada a aspectos. Desta forma, busca-se oferecer ao arquiteto modelar a arquitetura de software com UML 2.0, incluindo interações transversais e permitir que o projeto de arquitetura do sistema seja reforçado durante a implementação.

ABSTRACT

In the present, the software architecture modeling is an essential phase on the software development process. The software architecture is defined as the subsystems or components descriptions and their relationship, providing a gross vision of the system and making easier the communications between the stakeholders. The basic purpose of the architecture modeling is to give an initial structure to the software design. The architecture description languages (ADLs) provide a standard and some formalization to the architecture modeling.

Nevertheless, the ADLs are not frequently used by the developers because the lack of tools and traceability rules to the others phases of the development process. To address this problem we should use the UML, a standard language (in fact) to the software modeling. Although the UML 2.0 has improvements to this 1.4 version, mainly in the architecture modeling, it's still away from the representation of the ADLs.

Another question that become important is the aspect-oriented software development (AOSD). The purpose of AOSD is to modularize the crosscutting concerns that are scattering and tangled over the system. At the beginning, the aspect concept was presented only in the implementation phase, but actually its present in all phases of the development process, including the architecture modeling. Although has emerged a lot of solutions to the aspects representations on architectural level (Aspect-Oriented ADLs), these proposals suffers the same criticals of the regular ADLs.

In this context, this work presents an UML 2.0 extention to this architectural package, to assemble the concepts of the textual ADLs and to represent the crosscutting relationship between the components. Moreover, this work presents rules to map the solution to an aspect-oriented language of the design phase. Therefore, we attempt to offer to the architect the architecture modeling with UML 2.0, with the crosscutting interactions and enable the architecture design to be reinforced during the implementation.

1 INTRODUÇÃO

O princípio de separação de interesses permeia todo o processo de desenvolvimento de software. Ele cuida das limitações humanas em tratar a complexidade, permitindo que o desenvolvedor se concentre em um assunto ou interesse do sistema por vez (DIJKSTRA, 1976), e desta forma melhore sua compreensão sobre o sistema e aumente a capacidade para evolução e reuso. Em Engenharia de Software, o princípio de separação de interesses está relacionado à decomposição e modularização (PARNAS, 1972); sistemas de software complexos devem ser decompostos em unidades modulares menores, cada uma tratando um único interesse.

Para mitigar a crescente complexidade no desenvolvimento de software, algumas técnicas têm se tornado essenciais. A arquitetura de software (SHAW, 1996) é considerada atualmente, uma fase indispensável para o processo de desenvolvimento. Permite observar o software no seu mais alto nível, através da representação de sua estrutura, fornecendo elementos que permitem ao desenvolvedor enxergar as partes decompostas e a forma como elas interagem.

Apesar dessa importância no processo de desenvolvimento, as arquiteturas eram inicialmente modeladas de forma *ad hoc*, sem uma linguagem específica e sem um conjunto de elementos básicos, onde cada arquiteto utilizava a representação e a descrição que lhe era conveniente. Isso dificultava o entendimento entre todos os envolvidos no processo de desenvolvimento, o que muitas vezes levava a uma corrosão da arquitetura durante esse processo.

As linguagens de descrição arquitetural (ADLs) surgiram com o objetivo de tentar padronizar a descrição da arquitetura. No entanto, por falta de uma padronização, diversas ADLs surgiram, cada uma com o seu conjunto de elementos e domínio específicos.

Se por um lado é preciso separar os interesses, por outro é preciso compô-los. Muitas vezes estes interesses são fortemente relacionados ou entrelaçados, influenciando ou restringindo uns aos outros, sendo chamadas de interesses transversais (KICZALES, 1996). Isto torna difícil a separação e análise das partes do sistema, bem como a análise do impacto que umas exercem sobre as outras.

Para melhorar a separação e composição de interesses transversais, nos últimos anos, foi proposto o paradigma de orientação a aspectos (KICZALES, 1996). Esse paradigma surgiu com o objetivo de propor uma forma de modularizar os interesses transversais,

utilizando uma nova abstração denominada aspecto.

Inicialmente, a programação orientada a aspectos (POA), estava restrita à fase de implementação, mas logo seus conceitos foram aplicados a outras atividades e artefatos do ciclo de vida do software, dando origem a nova área da engenharia de software denominada de desenvolvimento de software orientado a aspectos (DSOA) (FILMAN, 2005).

Novas pesquisas estão sendo propostas (GARCIA (2006), PÉREZ (2003), PESSEMIER (2006), PINTO (2005)) procurando unir os conceitos de arquitetura de software e os princípios de orientação a aspectos. Algumas soluções de linguagens de descrição arquitetural orientadas a aspectos surgiram, tentando promover a representação arquitetural dos aspectos. No entanto, essas soluções estão sujeitas às mesmas críticas das ADLs regulares: falta de ferramentas e distanciamento da fase de projeto detalhado de software.

Esta dissertação tem por objetivo prover uma extensão para a UML 2.0 a fim de apresentar uma nova linguagem de descrição arquitetural que contemple a representação de interesses transversais.

Na Seção 1.1 é explicado o problema a ser tratado nesta dissertação. Na Seção 1.2 são apresentados os objetivos deste trabalho e o resumo da solução proposta. Já na Seção 1.3 são citadas as contribuições oferecidas pela dissertação e, finalmente, na Seção 1.4 descrevemos como o restante desta dissertação está organizado.

1.1 DESCRIÇÃO DO PROBLEMA

As ADLs auxiliaram na padronização e formalização das descrições arquiteturais. No entanto, sofrem com as críticas de falta de ferramentas de suporte e falta de rastreabilidade para outras fases do desenvolvimento, tornando a modelagem da arquitetura uma fase dissociada do restante do processo de desenvolvimento.

Embora tenha trazido novos elementos para a modelagem arquitetural, como como componentes, portas, interfaces (providas e requeridas), e a decomposição hierárquica de componentes, a UML 2.0, que é uma liguagem de desenvolvimento de software utilizada como padrão de fato, ainda encontra-se distante da representatividade provida pelas linguagens de descrição arquitetural.

O primeiro problema a ser enfrentado nessa dissertação é a adaptação da UML 2.0 para que possa ter a mesma representatividade das ADLs, proporcionando aos arquitetos uma linguagem de descrição da arquitetura alinhada com a indústria.

A segunda questão diz respeito à representação de interesses transversais na descrição

da arquitetura. A classificação e separação de interesses são úteis para facilitar a leitura do modelo utilizado e identificar nele as partes importantes em cada momento do processo. Na maioria das vezes, estes interesses são classificados e separados (decompostos) de maneiras diferentes, baseadas, por exemplo: em se são requisitos funcionais, não funcionais ou inversos; nas pessoas que estão interessadas no serviço, que o requisitaram, ou que o implementam; em opiniões dadas pelos requerentes; ou no tipo de serviço que oferecem; dentre outras. Entretanto, muitas vezes estes interesses estão transversais, percebendo-se o espalhamento e o entrelaçamento entre eles.

Considerando o processo de definição arquitetural, os problemas de espalhamento e entrelaçamento trazem sérias dificuldades para representar esses interesses, já que os elementos aquiteturais existentes atualmente não conseguem modularizá-los. Além disso, a forma como os interesses transversais interagem com o restante do sistema é diferente dos outros módulos.

Existem algumas soluções que propõem a identificação e modelagem dos interesses transversais em nível de arquitetura (GARCIA, 2006), (PÉREZ, 2003), (PESSEMIER, 2006), (PINTO, 2005). Essas abordagens são conhecidas como linguagens de descrição arquitetural orientadas a aspectos (AO-ADLs). No entanto, embora tenham contribuído com a inserção dos conceitos de orientação a aspectos em arquitetura de software, as AO-ADLs estão sujeitas às mesmas críticas enfrentadas pelas ADLs regulares, principalmente pela falta de rastreabilidade para as abordagens em nível de projeto e a falta de ferramentas de suporte. Esses problemas comprometem a utilização, adoção e sobrevivência dessas linguagens.

Assim sendo, o segundo problema abordado nesta dissertação é o da modelagem, com UML 2.0, de arquitetura de software levando-se em consideração a existência de interesses transversais. Consideramos que este é um problema natural, decorrente da complexidade dos sistemas e, portanto, que é necessário saber modelar arquitetura de software com interesses transversais.

Por fim, enfrentamos o problema de falta de rastreabilidade da fase de arquitetura para outras fases do processo de desenvolvimento.

1.2 OBJETIVOS DA DISSERTAÇÃO

Para abordar o problema de falta de representatividade na descrição da arquitetura da UML 2.0 (comparada às ADLs) e o problema de espalhamento e entrelaçamento de interesses durante a definição da arquitetura de software, este trabalho tem por objetivo,

propor uma extensão da UML 2.0, no que diz respeito a modelagem de arquitetura de software, que permita a representação de elementos aspectuais.

Para concretizar esse objetivo, este trabalho propõe a AD-AUML (Architecture Description in Aspectual UML), uma extensão da UML 2.0 que define elementos arquiteturais mais próximos das ADLs e que inclui elementos para a modelagem aspectual.

Além desse objetivo principal, são propostas no trabalho, regras de transformação entre os modelos arquiteturais que utilizam a extensão proposta e os modelos de projeto detalhado de software. Nesse contexto foi utilizada a linguagem aSideML (CHAVEZ, 2004), mas poderiam ser definidas regras para outras linguagens. Além disso, são propostos dois estudos de caso para validarem tanto a extensão do metamodelo quanto as regras que garantem a rastreabilidade entre as fases de desenvolvimento.

1.3 LISTA DE CONTRIBUIÇÕES

Ao atingir os objetivos enunciados na seção anterior, espera-se que haja relevantes contribuições para a área de modelagem de interesses transversais em arquitetura de software, bem como áreas correlatas. É possível levantar preliminarmente algumas destas prováveis contribuições e, ao fim do trabalho, atestar se elas foram, de fato, concretizadas. Assim, segue uma lista das contribuições esperadas.

a) Estender a UML 2.0 a fim alinhar suas definições às das linguagens de descrição arquitetural, melhorando sua representatividade e dando suporte aos elementos mínimos requeridos para descrição em ADLs (MEDVIDOVIC, 2000).

b) Prover uma linguagem de modelagem de arquitetura alinhada aos conceitos de DSOA, que seja capaz de representar os interesses transversais de um sistema;

c) Promover as modificações necessárias na UML 2.0, alterando minimamente o meta-modelo da UML 2.0, incentivando sua utilização por arquitetos e facilitando a extensão das ferramentas de modelagem já existentes.

d) Relacionar elementos de descrição arquitetural na linguagem proposta com elementos de alguma linguagem de descrição em projeto detalhado, promovendo a rastreabilidade entre as duas fases.

e) Utilização de dois estudos de casos, baseados em arquiteturas conhecidas e publicadas, para validar a nova linguagem.

1.4 ORGANIZAÇÃO

A presente dissertação está organizada de forma que o **Capítulo 2 - Arquitetura de Software e Modelagem Arquitetural** apresenta e explica os principais conceitos envolvidos na modelagem de arquiteturas de software.

O **Capítulo 3 - Arquitetura de Software e Aspectos Arquiteturais** apresenta os principais conceitos envolvidos no paradigma de orientação a aspectos. Além disso, apresenta algumas questões relacionadas com a representação de interesses transversais em arquitetura de software. Por fim, apresenta algumas propostas de linguagem de descrição arquitetural orientadas a aspectos e discute as vantagens e desvantagens dessas soluções.

O **Capítulo 4 - Trabalhos Relacionados** apresenta alguns trabalhos relacionados a essa dissertação. Para tanto, são considerados trabalhos relacionados aqueles que relacionem ADLs (regulares e orientadas a aspectos) e a UML 2.0.

O **Capítulo 5 - A AD-AUML** apresenta a linguagem de modelagem arquitetural denominada AD-AUML. A proposta é fornecer uma linguagem de modelagem que permita a representação dos interesses transversais em nível de arquitetura. Além disso, são descritas nesse capítulo algumas regras para transformação de arquiteturas representadas com a linguagem proposta para modelos da fase de projeto detalhado de software.

O **Capítulo 6 - Estudos de Caso** apresenta duas arquiteturas de sistemas conhecidos e publicados, modelados com a nova linguagem. Um dos estudos de caso utilizados, o sistema Health Watcher (SOARES, 2002), é *testbed* para o grupo AOSD-Europe (GROUP, 2006). Além disso, este capítulo apresenta a transformação dessas arquiteturas para um modelo de projeto detalhado de software.

Por fim, o **Capítulo 7 - Conclusões e Trabalhos Futuros** examina os resultados obtidos e aponta como foram concretizadas as contribuições enunciadas no Capítulo 1. Também são feitos comentários sobre trabalhos futuros e horizontes abertos a partir do trabalho realizado.

No **Apêndice 1** estão descritos os novos elementos e os elementos que foram alterados, seguindo o padrão de descrição da UML.

2 ARQUITETURA DE SOFTWARE E MODELAGEM ARQUITETURAL

Neste capítulo serão apresentados os principais conceitos envolvidos no trabalho. Na seção 2.1 alguns conceitos básicos sobre Arquitetura de Software são introduzidos. Na seção 2.2 é feita uma breve consideração sobre Linguagens de Descrição Arquitetural (ADLs)[1]. Na seção 2.3 será apresentada a ACME, uma ADL de propósito geral que foi utilizada em nosso trabalho. Na seção 2.4 serão apresentados os elementos da UML 2.0 usados para modelagem arquitetural.

2.1 ARQUITETURA DE SOFTWARE

O desenvolvimento de software tem se tornado cada vez mais complexo. Uma questão essencial no projeto e construção de um software complexo é a sua arquitetura, ou seja, sua organização de mais alto nível, representada como uma coleção de componentes que interagem. Uma boa arquitetura pode ajudar a garantir que o sistema irá satisfazer os principais requisitos de áreas como desempenho, confiabilidade, portabilidade, escalabilidade e interoperabilidade (NAVASA, 2005).

Existem diversas definições para arquitetura de software (BASS, 2003), (GARLAN, 1994b), (SHAW, 1996), mas todas apresentam como característica principal o fato de a arquitetura de um sistema descrever a sua estrutura como um todo. Essa estrutura permite a tomada de decisões de alto nível, como por exemplo, a forma como o sistema será composto de partes que interagem, onde estão os principais caminhos para essas interações e quais as principais propriedades dessas partes.

A arquitetura de software faz o papel de ligação entre a fase de requisitos e a fase de projeto. Por ser uma descrição abstrata do sistema, a arquitetura expõe algumas propriedades, enquanto esconde outras. Idealmente, essas representações fornecem um guia para o sistema e sugerem um plano para a construção e composição do sistema.

Para gerenciar a complexidade na descrição da arquitetura, separou-se a descrição da arquitetura de software em visões (CLEMENTS, 2002). Cada visão descreve uma determinada parte da arquitetura e permite reduzir a quantidade de informações que o arquiteto trata em um dado momento. Outras visões podem ser criadas conforme a

[1]Neste trabalho, utilizamos o acrônimo ADL (do original em inglês, Architecture Description Language) para designar Linguagem de Descrição Arquitetural.

necessidade, mas para prover uma descrição completa do sistema, CLEMENTS (2002) propõe cinco visões: de módulo, de execução, de implementação, de implantação e de dados.

a) Visão de módulo - mostra a estrutura do sistema em termos de unidades de código. É essencial porque representa a planta (*blueprints*) para a construção do software. Normalmente é representada por diagramas UML;

b) Visão de execução - mostra um "raio x" do sistema em execução, ao contrário da visão de módulo, que mostram a estrutura do código fonte. É útil para entender o funcionamento do sistema e analisar propriedades que se manifestam em tempo de execução, como desempenho;

c) Visão de implementação - mostra a estrutura do software em termos de arquivos organizados em diretórios, tanto para o ambiente de desenvolvimento quanto para o ambiente de produção;

d) Visão de implantação - mostra a estrutura de hardware (tipicamente uma rede) na qual o sistema é executado. É particularmente útil em sistemas distribuídos;

e) Modelo de dados - normalmente usada quando o sistema possui uma base de dados cuja estrutura precisa ser modelada. O modelo de dados inicia como um modelo conceitual/lógico que vai sendo refinado até conter toda informação necessária para a criação da base de dados física.

Dessa forma, o projeto da arquitetura de um sistema tornou-se uma disciplina fundamental para a engenharia de software, aumentando a importância de ferramentas e ambientes que permitem sua descrição e análise. Entre essas ferramentas, destacam-se os estilos arquiteturais e as linguagens de descrição arquitetural (ADLs). Os estilos arquiteturais consistem de um vocabulário de elementos de projeto, um conjunto de regras de configuração, uma interpretação semântica da composição dos elementos e um conjunto de análises que podem ser executadas sobre um sistema construído em um determinado estilo (GARLAN, 1994a). Exemplos de estilos arquiteturais incluem camadas, *pipe and filter*, *BlackBoard*, entre outros.As linguagens de descrição arquitetural serão abordadas na seção a seguir.

2.2 LINGUAGENS DE DESCRIÇÃO ARQUITETURAL

A modelagem da arquitetura de um sistema ocupou um lugar de grande importância no processo de desenvolvimento de software. Tornou-se tão importante que a escolha certa da arquitetura pode levar o produto de software a atingir os seus requisitos e facilitar a inserção ou modificação de alguns deles, já a escolha errada pode levar o desenvolvimento e o produto ao fracasso.

No entanto, apesar dessa importância no processo de desenvolvimento, as arquiteturas eram inicialmente modeladas de forma *ad hoc*, sem uma linguagem específica e sem um conjunto de elementos básicos, onde cada arquiteto utilizava a representação e a descrição que lhe era conveniente. Isso dificultava o entendimento entre todos os envolvidos no processo de desenvolvimento, o que muitas vezes levava a uma corrosão da arquitetura durante esse processo.

Para solucionar esses problemas, novas notações de modelagem e de análise, assim como novas ferramentas de desenvolvimento, que atuam no nível de arquitetura, foram desenvolvidas. Nesse conjunto, encontram-se as ADLs e suas ferramentas de suporte. Uma ADL é, portanto, uma notação composta por um framework conceitual e uma sintaxe para caracterizar a arquitetura de um software.

Uma grande quantidade de ADLs foi proposta como, por exemplo, Adage (COGLIANESE, 1993), Meta-H (BINNS, 1993), Aesop (GARLAN, 1994a), Wright (ALLEN, 1994), Rapide (LUCKAHAM, 1995), UniCon (SHAW, 1995) e C2 (MEDVIDOVIC, 1996). No entanto, existia pouco consenso sobre o que era realmente uma ADL e quais aspectos de uma arquitetura deveriam ser modelados por essa linguagem. Outro problema era o nível de suporte que uma ADL deveria prover aos desenvolvedores. Uma corrente achava que o papel principal da descrição arquitetural era ajudar no entendimento e na forma como se comunicava o sistema como um todo. A outra se preocupava com a sintaxe formal e a semântica das ADLs, como as poderosas ferramentas de análise, os verificadores de modelos, compiladores, ferramentas de síntese de código, entre outras (MEDVIDOVIC, 2000). Normalmente, os pesquisadores escolhem uma linha ou outra.

Foi baseado nessa diversidade de opiniões e implementações que MEDVIDOVIC (2000) propôs um framework para avaliação e comparação entre ADLs. O resultado do trabalho é um framework (FIG 2.1) para construção de uma ADL em que os elementos principais são componente, interface, conector e configuração arquitetural.

- Componente - é definido como um elemento computacional ou de armazenamento de dados. Pode representar apenas um procedimento ou até mesmo

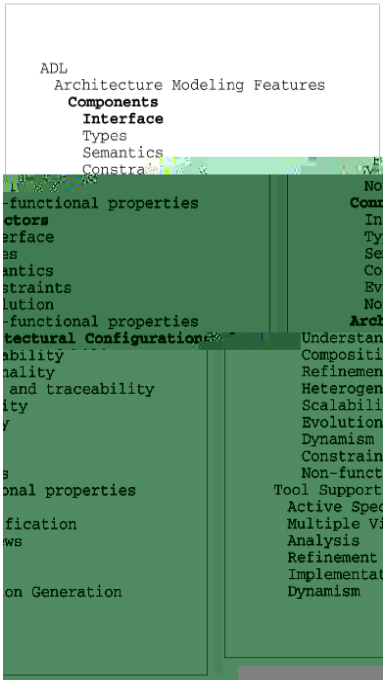


FIG. 2.1: O Framework de MEDVIDOVIC (2000)

um sistema completo. Cada componente pode ter os seus próprios dados e fluxo de execução ou pode compartilhá-los com outros componentes. Os serviços providos ou requeridos por um componente são expressos através de suas interfaces. Componentes são intuitivamente representados por caixas nos diagramas de arquitetura. Exemplos típicos de componentes são clientes, servidores, filtros e banco de dados.

- Interface - define o ponto de interação entre os elementos internos de um componente e o restante do sistema. A interface especifica os serviços (mensagens, operações e variáveis) que um componente provê e, para o completo entendimento sobre o componente e a arquitetura que o suporta, também provê meios para especificar as necessidades de um componente, ou seja, os serviços que são requeridos de outros componentes presentes na mesma arquitetura. As interfaces de um componente, portanto, definem um contrato, explicitando os serviços que são providos ou requeridos e as restrições para sua utilização.

- Conector - utilizado para modelar as interações entre os componentes e as regras acerca dessas interações. Assim como os componentes, os conectores

conector e os componentes ligados a ele ou, até mesmo, entre dois conectores. Em algumas ADLs as interfaces são chamadas de papel. Por outro lado, diferenciando-se dos componentes, os conectores não correspondem a unidades compiladas em um sistema implementado. Os conectores se manifestam no sistema, por exemplo, através de variáveis compartilhadas, entradas em tabelas, estruturas de dados dinâmicas, ligações entre banco de dados e a aplicação, entre outras. Exemplos de conectores incluem dutos (*pipes*) e chamadas de procedimentos. Em uma arquitetura mais elaborada, os conectores podem representar interações mais complexas, como um protocolo cliente-servidor ou um link SQL entre a aplicação e o banco de dados.

- Configuração Arquitetural - Descreve a estrutura arquitetural (topologia) da conexão entre componentes e conectores. Essas informações são necessárias para determinar se os componentes apropriados estão conectados, se suas interfaces são compatíveis, se os conectores estão provendo a comunicação necessária e se essas combinações estão produzindo o comportamento desejado.

Uma descrição arquitetural com uma ADL representa uma descrição formal da arquitetura de um sistema. Procura descrever todos os elementos envolvidos, a semântica envolvida nesses elementos e a forma como os diversos elementos interagem. Por serem linguagens de descrição formal, as ADLs trouxeram enormes vantagens na descrição das arquiteturas, entre elas: permitiram que desenvolvedores entendessem com mais clareza as arquiteturas do sistema (linguagem); permitiram que as arquiteturas pudessem ser avaliadas com relação a sua consistência e completude; as decisões e restrições arquiteturais assumidas no início do desenvolvimento passaram a permear todo o desenvolvimento (com a descrição informal, essas variáveis se deterioravam durante o ciclo de desenvolvimento); por fim, permitiram que os arquitetos utilizassem ferramentas que, realmente, auxiliaram o desenvolvimento do seu trabalho (sem uma descrição formal, qualquer ferramenta poderia ser utilizada e não havia garantia de que o resultado auxiliaria no desenvolvimento).

A grande dificuldade na adoção das ADLs continua sendo a falta de padronização das soluções propostas (COGLIANESE (1993), BINNS (1993), GARLAN (1994a), ALLEN (1994), LUCKAHAM (1995), SHAW (1995) MEDVIDOVIC (1996)). Cada domínio específico pode exigir a adoção de uma solução com estruturas, elementos e ferramentas diferentes. Mais ainda, dentro de cada domínio, autores definem soluções específicas, causando uma enorme quantidade de soluções e dificultando o intercâmbio de informações

entre elas.

2.3 ACME

A proliferação de ADLs, cada uma com o seu objetivo próprio e suas características individuais, favorecendo uma ou mais atividades (comunicação, verificação, etc.) em diferentes domínios, proporcionou o estudo de diferentes aspectos relacionados à modelagem da arquitetura. Com a exposição de diferentes características de modelagem e a forma de explorar essas características, todas as propostas ajudaram a aprofundar o entendimento dos papéis que a modelagem arquitetural pode assumir no desenvolvimento de software. Por outro lado, essa forma de trabalhar de uma maneira independente (cada ADL se preocupa com um domínio específico), dificulta a combinação de funcionalidades de uma ADL com outra. Essa característica é extremamente indesejável para os arquitetos, já que a adoção de uma nova ADL implica em um gasto de recursos para instalação de ferramentas de suporte e tempo para aprender a usá-la de forma efetiva, além de ficar restrito às características e funcionalidades da ADL selecionada.

Para tanto, a ADL ACME (GARLAN, 1997) foi proposta com o principal objetivo de fornecer um formato de intercâmbio para ferramentas e ambientes de desenvolvimento arquiteturais. Dessa forma, a linguagem proporciona a integração das diversas ferramentas desenvolvidas para dar suporte as diversas ADLs, fornecendo um mecanismo para a troca de informações arquiteturais. Além disso, ACME apresenta quatro objetivos secundários:

a) Fornecer um esquema de representação que permita o desenvolvimento de novas ferramentas para análise e visualização das estruturas de uma arquitetura;

b) Fornecer uma base para o desenvolvimento de novas ADLs, possivelmente específicas de domínio;

c) Servir como um veículo para a criação de convenções e padrões para as informações arquiteturais;

d) Fornecer descrições expressivas que possam ser escritas e lidas pelos interessados no sistema (*stakeholders*).

Procurando atender a todos esses objetivos, ACME apresenta sete elementos para descrição arquitetural: componente, porta, conector, papel, sistema, representação e mapa de representação (GARLAN, 1997).

Os *componentes* representam os elementos computacionais básicos e os armazéns de dados do sistema. Um componente pode ter múltiplas interfaces, cada uma representada por uma porta, onde cada porta identifica um ponto de interação entre o componente e o seu ambiente. Os *conectores* representam as interações entre os componentes. Assim como os componentes, os conectores também têm interfaces, que são chamadas de papéis. Cada *papel* define um participante da interação representada pelo conector. Os *sistemas* são definidos como grafos, onde um nó representa o componente e um arco representa o conector. Isso é feito identificando a porta de um componente que está ligada a um papel de um conector.

A FIG 2.2 apresenta os principais elementos de ACME usando uma notação gráfica. A FIG 2.3 apresenta a descrição em ACME de um sistema onde um cliente e um servidor são conectados por um conector RPC.

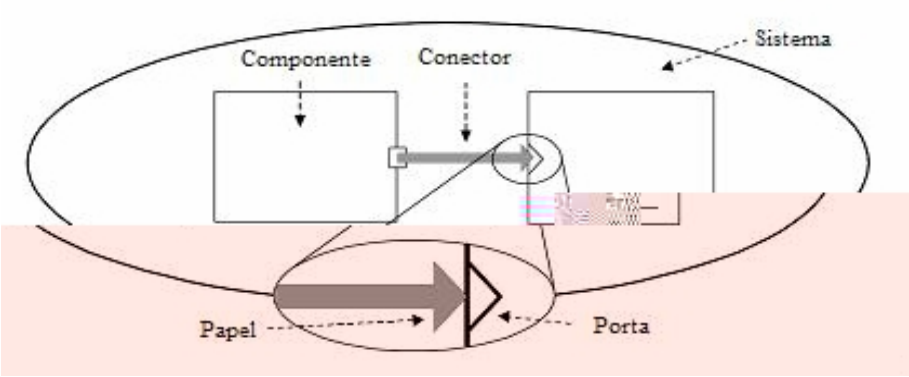


FIG. 2.2: Principais elementos de ACME (GARLAN, 1997)

```
System simple_cs = {
  Component client = { Port
                       Component server = { Port receive-
request }              Connector rpc  = { Roles {caller,
callee} }              Attachments : {
                           client.send-request to rpc.cal
ler ;                      server.receive-request to rpc.
callee }             }
```

FIG. 2.3: Cliente-Servidor em ACME (GARLAN, 1997)

Na FIG 2.3, o componente *client* apresenta apenas a porta *send-request* e o componente *server* a porta *receive-request*. O conector RPC tem dois papéis denominados de *caller* e *callee*. No campo de interconexão (*attachments*) são feitas as conexões entre o conector e o componente. No exemplo, a porta *send-request* do componente *client* está ligada ao papel *caller* do conector RPC. Essa configuração é chamada de topologia do sistema.

Outra característica importante de ACME é o suporte à descrição hierárquica do sistema, onde cada elemento da arquitetura pode ser descrito por um conjunto de elementos de nível mais baixo. A esse tipo de descrição dá-se o nome de *representação* (GARLAN, 1997). É o mesmo que dizer que um componente pode ser representado por um conjunto de componentes internos. Quando um componente ou um conector tem uma representação, deve haver uma forma de indicar a correspondência entre a representação interna do elemento e a sua interface externa. Ao elemento que faz essa correspondência dá-se o nome de *mapa de representação* (GARLAN, 1997). Em um caso simples, um mapa de representação pode apenas definir associações entre portas internas e portas externas do componente (ou entre papéis internos e externos de um conector). Em um caso mais complexo, o mapa de representação pode ser representado por um marcador que será interpretado por uma ferramenta.

Esses sete elementos constituem a estrutura da linguagem. No entanto, em uma descrição arquitetural é necessário um pouco mais do que estrutura. Como, no início do desenvolvimento das ADLs, não havia sido desenvolvida uma padronização para confecção de uma ADL, cada proposta apresentava o seu conjunto de informações auxiliares, embora o conjunto de elementos centrais fosse basicamente o mesmo. Para absorver essas informações, ACME utiliza propriedades. Cada um dos sete elementos estruturais definidos em ACME pode ter o seu conjunto de propriedades. Uma propriedade é definida por um nome, um tipo (opcional) e um valor, e é vista por ACME como um valor não-interpretável. Propriedades podem ser utilizadas por ferramentas de análise, de tradução (para outra linguagem), etc. A FIG 2.4 mostra a representação do sistema cliente-servidor com suas representações e propriedades. A FIG 2.5 ilustra a descrição do mesmo sistema com suas propriedades.

2.4 MODELAGEM ARQUITETURAL NA UML 2.0

A UML é uma linguagem visual universalmente utilizada para o desenvolvimento de software. Atualmente a especificação da linguagem UML encontra-se em sua versão 2.0 e se divide em três documentos básicos: (i) infra-estrutura (OMG, 2006b); (ii) superestrutura (OMG, 2006b); e (iii) OCL (OMG, 2006a). O primeiro documento define as construções fundamentais utilizadas na definição da linguagem, o segundo documento define as construções a serem utilizadas pelo usuário e o terceiro define uma linguagem formal utilizada para complementar os diagramas da UML por meio da definição de condições que são mantidas durante toda a modelagem do sistema. Normalmente as expressões

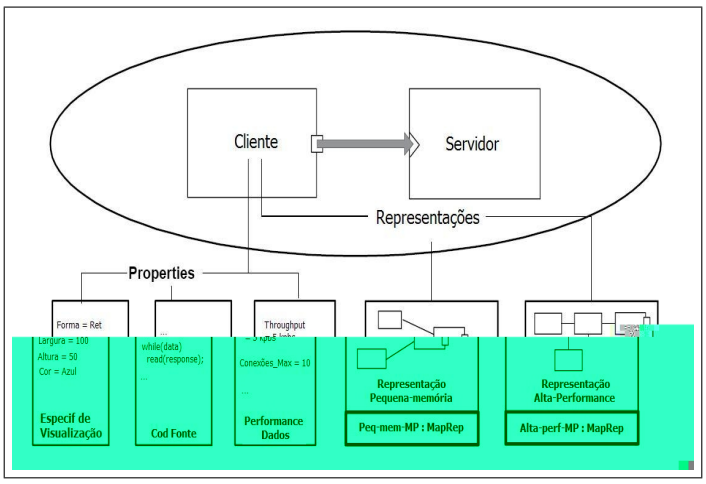


FIG. 2.4: Ilustração do sitema Cliente-Servidor em ACME com propriedades (GARLAN, 1997)

```
System simple_cs = {
  Component client = {
        Port send-request;
        Properties { Aesop-style : style-id = client-server;
                     UniCon-style : style-id = cs;
                     source-code : external = "CODE-LIB/client.c" }}

  Component server = {
        Port receive-request;
        Properties { idempotence : boolean = true;
                     max-concurrent-clients : integer = 1;
                     source-code : external = "CODE-LIB/server.c" }}

  Connector rpc  = {
        Roles {caller, callee}
        Properties { synchronous : boolean = true;
                     max-roles : integer = 2;
                        protocol : Wright = "..." }}

     Attachments {
        client.send-request to rpc.caller ;
        server.receive-request to rpc.callee }
   }
```

FIG. 2.5: Descrição Cliente-Servidor em ACME com propriedades (GARLAN, 1997)

definidas em OCL restringem de alguma forma o modelo que está sendo desenvolvido. A principal mudança em relação à versão anterior, versão 1.4 (OMG, 2003), é a reorganização do meta-modelo (MOF), já que a maioria das construções que existiam conservaram o seu significado ou tiveram alterações superficiais.

Outra mudança importante ocorreu na modelagem arquitetural do sistema, onde a UML 1.4 sofria as críticas mais pesadas. A maioria dessas alterações está concentrada no pacote *Composite Structures*, onde é fundamentado o conceito de estruturas compostas. É nesse pacote que estão definidas as seguintes metaclasses:

- Structured Classifier (from InternalStructures) - define um classificador que pode ser completamente ou parcialmente descrito pela colaboração de instâncias;
- Encapsulated Classifier (from Ports) - estende o classificador com a capacidade de ter portas como ponto de interação com o ambiente;
- Class (from InternalStructures) - estende a classe (Class - from Kernel), com a capacidade de ter estruturas internas e portas;
- Connectable Element (from Internal Structures) - representam um conjunto de instâncias de um classificador que podem ser ligadas por um conector;
- Connector (from InternalStructures) - estabelece uma ligação que permite a comunicação entre duas ou mais instâncias;
- Connector End (from InternalStructures, Ports) - estabelece o ponto final de um conector (terminação), onde se junta a um Connectable Element.
- Port (from Ports) - é uma propriedade de um classificador que especifica um ponto de interação entre o classificador e o ambiente ou entre o classificador e suas estruturas internas;
- Property (from InternalStructures) - estende o conceito de propriedades definido no pacote kernel (Property - from Kernel, AssociationClasses), para os elementos definidos nesse novo pacote. Trata-se, portanto, de características estruturais definidas para estes elementos.

Todos esses elementos participam de forma direta ou indireta na definição dos elementos-base da modelagem arquitetural (componentes e conectores). Componentes e

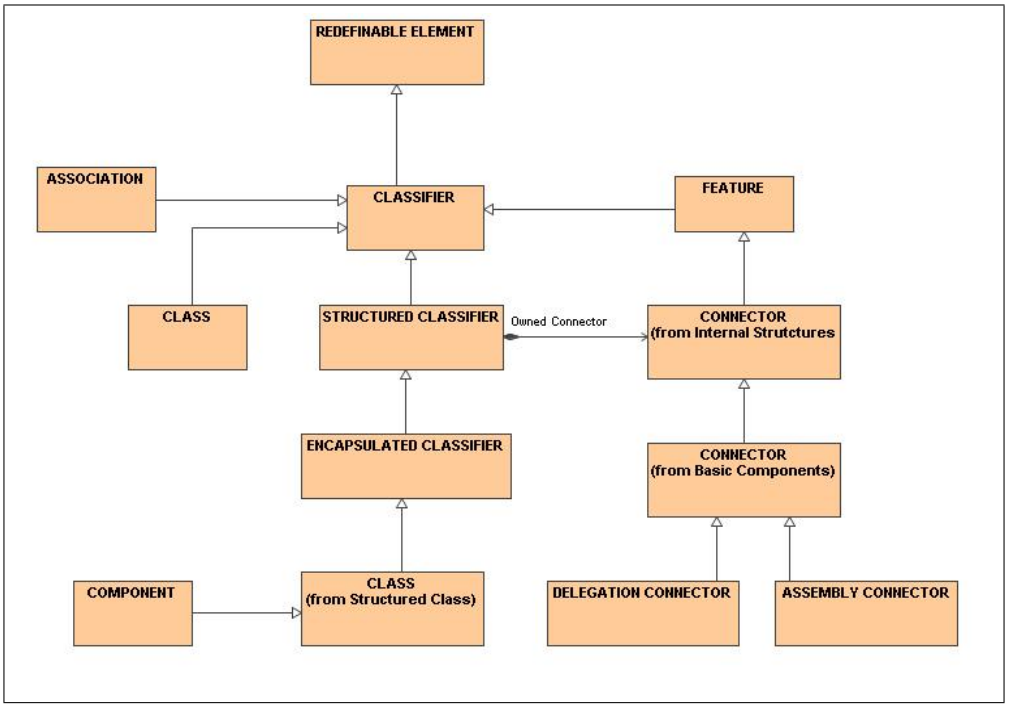


FIG. 2.6: Metamodelo parcial da UML 2.0 (OMG, 2006b)

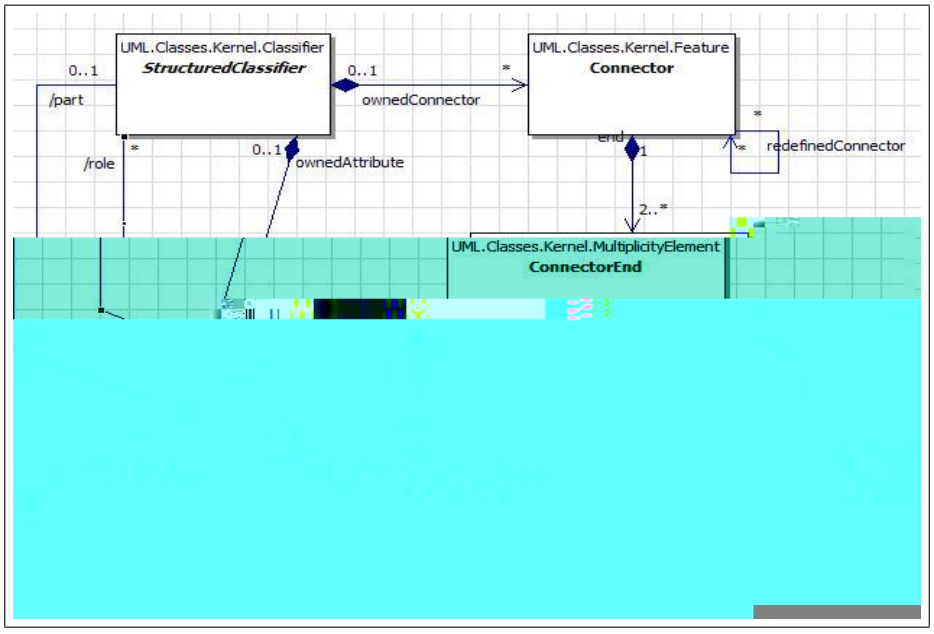


FIG. 2.7: Estrutura do elemento Structured Classifier (OMG, 2006b)

conectores foram definidos dentro do pacote BasicComponents. Os componentes (Components - from BasicComponents, PackagingComponents) apresentam como definição "uma parte modular do sistema que encapsula seu conteúdo e cuja manifestação pode ser substituída no ambiente a que está submetido" (OMG, 2006b). Se observarmos no MOF (FIG 2.6) a árvore da qual o componente é derivado, temos as seguintes meta classes (a partir de Classifier): Structured Classifier, Encapsulated Classifier e Class. Pode-se concluir, portanto, que o componente em UML é um classificador estruturado que possui portas e que estende o comportamento de uma classe. Essa definição de componente é muito parecida com as que encontramos nas ADLs. Há, portanto, uma tendência comum a todas as tentativas de aproximação das ADLs e UML em conservarem a definição de componente existente na UML.

Outro elemento-base da modelagem arquitetural proposta pela UML é o conector. Na UML encontram-se duas definições de conector, localizadas em pacotes diferentes. A primeira definição está presente no pacote InternalStructures. A segunda definição encontra-se no pacote BasicComponents e estende a primeira definição. A definição de conector do pacote BasicComponents existe basicamente para que o conector que será

(from Internal Structures)(FIG 2.7). Essa hierarquia é perfeita quando se trata da conexão entre o componente e seus elementos internos, no entanto, é falha para a conexão entre dois componentes. Falta ao conector de ligação uma representatividade maior, já que, nas ADLs o conector é um elemento de primeira ordem (independente do componente) e na UML está definido como uma característica estrutural de um componente.

A FIG 2.8 apresenta um pequeno exemplo cliente-servidor utilizando os elementos arquiteturais e a notação fornecidos pela UML 2.0.

3 ARQUITETURA DE SOFTWARE E ASPECTOS ARQUITETURAIS

Neste capítulo será apresentada a relação entre o conceito de orientação a aspectos e a modelagem arquitetural de um sistema. Inicialmente, será feita uma breve introdução sobre o paradigma de orientação a aspectos e sua relação com arquitetura de software. Posteriormente, serão discutidas propostas que buscam enriquecer ADLs existentes para dar suporte a conceitos do paradigma de orientação a aspectos, resultando em AO-ADLs (*Aspect-Oriented Architecture Description Languages*). A seção 3.1 introduz os conceitos do Desenvolvimento de Software Orientado a Aspectos (DSOA). A seção 3.2 apresenta uma discussão sobre os elementos que devem estar contidos em uma AO ADL. Na seção 3.3, são apresentadas algumas AO-ADLs relevantes no contexto deste trabalho, com destaque AspectualACME (GARCIA, 2006), uma extensão da ADL ACME. Por último, na seção 3.4, são discutidas as vantagens e desvantagens de utilização de cada uma das AO ADLs apresentadas.

3.1 DESENVOLVIMENTO DE SOFTWARE ORIENTADO A ASPECTOS (DSOA)

O princípio de separação de interesses pode ser apontado como suporte fundamental para o desenvolvimento de software nas últimas décadas (DIJKSTRA, 1976). Esse princípio prega que, para contornar a complexidade de um sistema, deve-se resolver uma questão importante (interesse) por vez. O princípio de separação de interesses está relacionado à decomposição e modularização(PARNAS, 1972) de sistemas de software complexos em unidades modulares menores, cada uma tratando um único interesse ou característica. Um módulo é uma unidade compilável e integrável para formar o programa, e idealmente, deve possuir as seguintes propriedades (STAA, 2000): encapsulamento, acoplamento fraco e coesão alta.

- encapsulamento - o módulo deve tornar invisível para os módulos clientes sua implementação interna, disponibilizando seus serviços apenas por sua interface;
- fraco acoplamento - o módulo deve ter menor acoplamento com os demais, i.e., menor dependência; e
- coesão alta - os elementos que constituem um módulo devem ter um forte

inter-relacionamento, devendo relacionar-se a um único conceito ou interesse.

Módulos e suas propriedades são essenciais para gerenciar a complexidade de partes do sistema e dele como um todo. Entretanto, cada paradigma de desenvolvimento impõe uma maneira dominante de decomposição e modularização (por exemplo, orientada a objetos), explicitando alguns interesses e ocultando outros, igualmente importantes (TARR, 1999). Desse modo, o paradigma de desenvolvimento pode interferir na forma como os conceitos, idealmente separados em altos níveis de abstração, se manifestam e são incorporados por módulos ao longo do processo de desenvolvimento, desde a especificação de requisitos até sua implementação em alguma linguagem de programação.

Nesse contexto, alguns interesses podem perder comportamento modular e apresentar algumas anomalias, em função do paradigma de decomposição adotado. Essas anomalias se manifestam através de sintomas como entrelaçamento e espalhamento. Sem meios apropriados para sua separação em um sistema OO, esses interesses tendem a ficar espalhados e entrelaçados a outros interesses. Um interesse é dito espalhado quando este afeta vários componentes do sistema e entrelaçado quando se mistura com outros interesses dentro de um módulo. Essas anomalias são indesejáveis porque causam maior dificuldade de entendimento, evolução e reuso dos artefatos de software. Esses interesses são conhecidos como *interesses transversais*.[2].

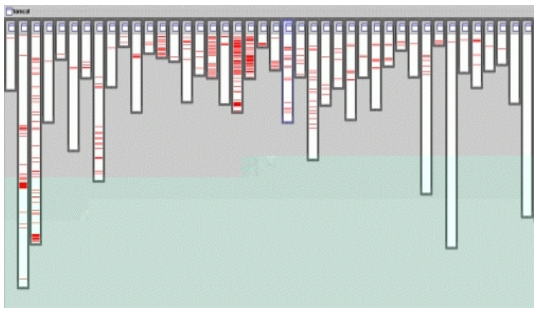


FIG. 3.1: Características transversais. (FOUNDATION, 2006)

A existência de interesses transversais, dado um tipo de decomposição dominante, torna difícil a separação e análise das partes do sistema bem como a análise do impacto que umas exercem sobre as outras. Os exemplos mais intuitivos desses tipos de interesses são os requisitos não-funcionais do sistema como, por exemplo, segurança, *logging*, tratamento de exceções, distribuição e controle de transação, que se manifestam em sistemas orientados a objetos.

[2]Crosscutting Concerns (KICZALES, 1996)

Para melhorar a separação e composição de interesses transversais, nos últimos anos, foi proposto o paradigma de orientação a aspectos (KICZALES, 1996). A Programação Orientada a Aspectos (POA) surgiu com o objetivo de propor uma forma de modularizar os interesses transversais, utilizando uma nova abstração denominada aspecto.

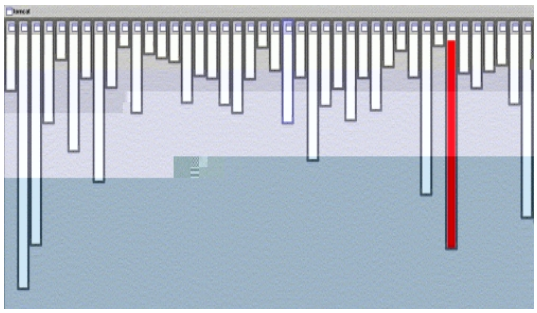


FIG. 3.2: Características transversais modularizadas. (FOUNDATION, 2006)

Juntamente com o conceito de aspectos, a POA trouxe o conceito de pontos de junção (join points), pontos de corte (pointcuts), adendos (advices) e mecanismo de composição (weaving) (FILMAN, 2005). O aspecto é, portanto, uma unidade modular utilizada para modularizar os interesses transversais. O ponto de junção é uma localização bem definida na estrutura ou no fluxo de execução de um sistema, onde um comportamento adicional será inserido. Os pontos de corte são conjuntos de pontos de junção. O mecanismo de composição é responsável pelo processo de composição entre os módulos funcionais do sistema e os aspectos, resultando em um sistema completo.

Originalmente, aspectos foram propostos com duas propriedades básicas que os diferenciavam dos conceitos de classes, objetos e mecanismos de composição da programação orientada a objetos: transparência e quantificação (FILMAN, 2005). Quantificação é a capacidade de escrever declarações unitárias e separadas que afetam muitos pontos de um sistema (ELRAD, 2001). Pela propriedade de quantificação, é possível fazer declarações do tipo "em programas P, sempre que surgir a condição C, faça a ação A". Transparência é a capacidade que o aspecto apresenta de não ser notado pelo restante do sistema (FILMAN, 2005). Em outras palavras, significa dizer que o sistema não tem conhecimento e não precisa ser preparado para receber o aspecto. Atualmente, a propriedade de transparência vem sendo questionada pela comunidade de aspectos, que discute que os benefícios trazidos em relação à flexibilidade não superam possíveis problemas relacionados à confiabilidade dos sistemas (KEVIN, 2005).

Linguagens de programação orientadas a aspectos e ferramentas associadas dão suporte à composição de interesses modularizados em aspectos, tornando a tarefa de composição

transparente para o programador. A programação orientada a aspectos permite a criação de programas que são mais fáceis de compreender, modificar e reutilizar.

Inicialmente, o paradigma de orientação a aspectos, estava restrito à fase de implementação, mas logo seus conceitos foram aplicados a outras atividades e artefatos do ciclo de vida do software, dando origem a nova área da engenharia de software denominada de desenvolvimento de software orientado a aspectos (DSOA) (FILMAN, 2005).

O DSOA surgiu principalmente porque alguns aspectos potenciais podiam ser descobertos facilmente em níveis mais altos do processo de desenvolvimento e, caso não fossem prontamente identificados, poderiam ficar sem solução até a fase de implementação. Diversas soluções surgiram para os níveis de requisitos, arquitetura e projeto de software. Especialmente no nível de arquitetura, as abstrações existentes não eram capazes de capturar e modelar as interações aspectuais entre os elementos.

Foi dessa constatação que surgiu o conceito de arquitetura de software orientada a aspectos. O objetivo é descrever passos para identificar os aspectos arquiteturais e os componentes que são afetados por eles. As abordagens que tratam desse novo conceito, normalmente procuram estender as ADLs convencionais, e são chamadas de AO ADLs (Aspect-oriented ADLs). As seções a seguir discutirão os aspectos relacionados à orientação a aspectos e a arquitetura de software.

3.2 QUESTÕES RELACIONADAS A ASPECTOS E ARQUITETURA DE SOFTWARE

3.2.1 ARQUITETURA DE SOFTWARE ORIENTADA A ASPECTOS: UMA ABORDAGEM ESTRUTURAL

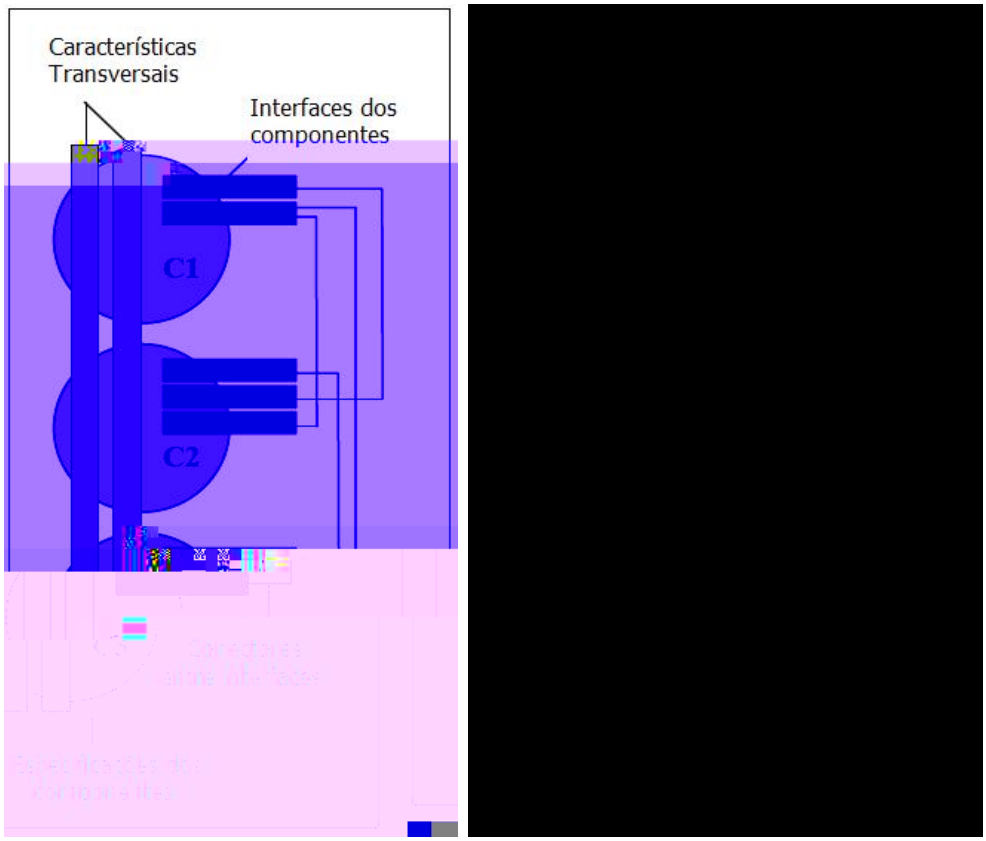
NAVASA (2002) define um conjunto de requisitos que as propostas de AO ADL devem respeitar, para permitir o gerenciamento de interesses transversais em nível arquitetural. O trabalho considera os estilos arquiteturais e as ADLs como as duas mais importantes ferramentas para a descrição da arquitetura. A primeira fornece regras para construir famílias de sistemas com a mesma característica, enquanto a segunda fornece primitivas para especificar componentes e conectores.

O trabalho apresenta, como principal dificuldade para a introdução dos conceitos de orientação a aspectos na modelagem da arquitetura de um sistema, a necessidade de se adaptar essas duas ferramentas. No caso dos estilos arquiteturais existem dois problemas: o primeiro é que a natureza com que um aspecto atua em um sistema é diferente, o que dificulta a modelagem de uma forma simples e uniforme, utilizando como base um único

estilo arquitetural; o segundo é que os mesmos aspectos aplicados em sistemas diferentes podem exigir um tratamento diferenciado, novamente dificultando uma única solução.

No caso das ADLs, o trabalho aponta, como problema principal, o fato das linguagens atualmente desenvolvidas não fornecerem primitivas para promover a separação de aspectos e componentes. Além disso, seria necessário estabelecer uma nova forma de conexão entre aspectos e componentes e entre um aspecto e outro.

Para solucionar essa gama de problemas, é feita uma análise baseada nos problemas expostos nas FIG 3.3(a) e FIG 3.3(b).



(a) Arquitetura sem aspectos (b) Interesses transversais modularizados

FIG. 3.3: Figuras utilizadas por NAVASA (2002)

Suponha que as figuras especificam a arquitetura de um mesmo sistema. A primeira figura apresenta a arquitetura sem a modularização de interesses transversais. Na segunda, esses interesses estão modularizados. Para que a segunda figura produza os mesmos efeitos que a primeira, os arquitetos de software têm que observar as seguintes tarefas: (i) especificar nos componentes os pontos onde as especificações dos interesses transversais foram retirados. Em outras palavras, significa que tem que especificar os pontos de junção nos componentes; e (ii) definir como serão as conexões entre aspectos e os pontos de junção. Essas novas conexões têm que especificar, por exemplo, quando e como cada

aspecto deverá ser tratado no contexto do sistema. A partir dessas observações, são definidos os requisitos que uma AO ADL deve conter: (i) definição de novas primitivas que permitam especificar pontos de junção em componentes funcionais; (ii) definição do aspecto como um tipo especial de componente; e (iii) especificação de conectores entre os pontos de junção e os aspectos. No trabalho é sugerida a utilização de modelos de coordenação existentes para especificar os conectores entre os aspectos e os componentes funcionais.

3.2.2 REFLEXÕES NA CONEXÃO ARQUITETURAL: SETE QUESTÕES ENVOLVENDO ASPECTOS E ADLS

No trabalho apresentado por BATISTA (2006), são discutidas sete questões que envolvem a representação de aspectos em nível de arquitetura de software. A discussão é baseada na comparação de ADLs e AO ADLs, tendo como resultado uma nova proposta de representação dos conceitos de orientação a aspectos no nível de arquitetura.

A primeira questão diz respeito aos elementos de uma ADL que podem conter interesses transversais. Baseado no exemplo de um sistema descrito em ACME em um conjunto de AO ADLs, é verificado que a maioria das soluções divergem com relação a esse assunto. A falta de convergência traz como conclusão que apenas um subconjunto de características é considerado pelas ADLs para a descrição arquitetural. Por exemplo, umas não consideram conectores e configurações como blocos de construção que contém conceitos transversais, enquanto outras dão suporte a componentes e configurações, mas não a conectores.

A segunda questão é com relação à composição entre os elementos arquiteturais. Algumas AO ADLs avaliadas utilizam o conector para representar a composição entre componentes com comportamento regular e componentes com comportamento aspectual, no entanto, propõem a definição de primitivas para especificar os pontos de junção em componentes funcionais, ao invés de utilizar a parte de configuração. Outras AO ADLs, não utilizam esse tipo de abordagem e, normalmente, adicionam complexidade à descrição arquitetural através da inserção de novos elementos que sejam responsáveis pela composição. A proposta do trabalho é utilizar o conector como o elemento responsável por modelar a composição. Para isso, são propostas extensões na estrutura do conector e nas configurações (que definem a conexão entre componentes e conectores).

A terceira questão diz respeito ao mecanismo de quantificação necessário no paradigma orientado a aspectos. Em algumas AO ADLs os mecanismos de quantificação podem ser

definidos como nome de componentes, nomes de interfaces e nomes de métodos. Outras defendem que o mecanismo de quantificação depende do modelo de coordenação adotado pela ADL e, em outras, o mecanismo simplesmente não existe. A conclusão do trabalho é utilizar a parte de configuração para definir o mecanismo de quantificação.

A quarta questão é com relação a interface aspectual. A principal discussão é se há a necessidade de se definir uma nova interface para um componente que tenha comportamento aspectual ou se a interface já existente provê os elementos necessários para expressar o contrato entre o componente "aspectual" e os componentes afetados por ele. Algumas AO ADLs observadas apresentam um conceito diferente de interface. A conclusão do trabalho é que o conceito de interface não precisa ser mudado para expressar a fronteira (os limites) de um componente "aspectual", já que o protocolo de interação está definido pelo conector.

A quinta questão é com relação à exposição do ponto de junção. A discussão é se a interface do componente afetado por outro componente, de comportamento aspectual, precisa expor informações extras para permitir essa conexão e, como conseqüência, se o conceito de interface precisa ser estendido. Todas AO ADLs avaliadas concordam que o ponto de junção tem que estar exposto na interface, no entanto, cada uma implementa essa característica de forma diferente. A proposta do trabalho é que a interface definida por MEDVIDOVIC (2000) em seu framework, já provê uma forma de representar os pontos de junção.

A sexta questão é com relação ao aprimoramento de interfaces. Algumas abordagens de implementação orientadas a aspectos, permitem ao aspecto a habilidade de mudar o tipo e a interface dos módulos através da declarações inter-tipos. Em uma perspectiva arquitetural, significa dizer que um componente com comportamento aspectual, pode aprimorar (melhorar) a interface do componente incluindo novos elementos, como serviços e atributos. Nas AO ADLs que apresentam o aspecto como elemento de primeira ordem, esse tipo de aprimoramento de interfaces é permitido. Como o trabalho utiliza uma abordagem em que o relacionamento de crosscutting é descrito pelo conector, acredita-se que essa habilidade não é necessária para capturar os interesses transversais nas especificações de arquitetura.

A última questão é com relação a forma de representar os aspectos. A discussão é se há a necessidade de introduzir uma nova abstração na descrição arquitetural para representar os aspectos. A maioria das AO ADLs propõem essa inclusão. No entanto, o trabalho argumenta que a principal diferença entre os componentes regulares e os componentes aspectuais é a forma como um e outro interage com o resto do sistema. Como essa

composição já foi definida pelo conector, o trabalho defende que não há a necessidade de se criar uma nova abstração.

3.2.3 ASPECTOS ARQUITETURAIS DE ASPECTOS ARQUITETURAIS

CUESTA (2005) faz uma comparação entre os conceitos do paradigma da orientação a aspectos e os conceitos de arquitetura de software. Dessa comparação estrutural, surgem modelos de combinação entre aspectos e arquiteturas. A classificação desses modelos tem como objetivo cobrir a totalidade de propostas existentes, permitindo identificar pontos em comum entre essas abordagens e relacionamentos entre soluções parecidas. Os modelos classificados são descritos a seguir:

Não exitem aspectos. O modelo sem aspectos utiliza apenas elementos arquiteturais convencionais, sugerindo que não é necessária a inclusão de nenhuma entidade aspectual adicional nesse nível. Como razão para esse tipo de abordagem, o trabalho enumera duas: a primeira simplesmente sugere que o aspecto não é importante, portanto, não deve ser considerado nesse nível; a outra, considera que não é necessária a noção de aspecto porque suas características são providas por algum mecanismo de composição padrão.

Aspectos Arquiteturais. Esse modelo considera que os aspectos são importantes no nível arquitetural e define entidades explícitas para representá-los.

a) Componentes como aspectos - Ao invés de criar uma nova abstração, essa abordagem utiliza os componentes para fazerem papéis de aspectos;

b) Conectores como aspectos - Essa abordagem complementa a anterior; utiliza o conector ao invés de componentes para representar aspectos. Isso faz sentido, uma vez que os aspectos normalmente são introduzidos por uma interação de interceptação;

c) Componentes derivados - Esse modelo fornece um tipo de definição de segunda classe da entidade aspectual, que é concebida como uma variação do componente. Nesse tipo de abordagem continua existindo apenas um tipo de componente, mas alguns deles apresentam características adicionais;

d) Componentes aspectuais - Esse modelo é muito parecido com o anterior, pois continua havendo apenas um tipo de componente. No entanto, o modelo de componentes é que é definido para incluir características aspectuais. Dessa forma, até os componentes regulares têm interesses transversais;

e) Aspectos arquiteturais de primeira classe - Define um elemento parecido com o componente para representar o aspecto em nível arquitetural.

Ligações aspectuais. Esse modelo considera que não há a necessidade de uma entidade aspectual explícita. A característica diferente de um elemento aspectual é o tipo de interação com os outros componentes do sistema. Portanto, o modelo arquitetural deve apenas fornecer um novo tipo de mecanismo de ligação entre os componentes.

a) Interação aspectual - Esse modelo considera a interação como o nível mais baixo de abstração e fornece algumas formas explícitas de interceptá-la em nível arquitetural, fornecendo a orientação aspectual;

b) Composições aspectuais - Tenta definir uma estrutura de nível mais alto que encapsule esse tipo de interação aspectual. Portanto, utiliza uma abordagem arquitetural para esse problema;

c) Ligações aspectuais de primeira classe - Esse modelo é uma generalização dos modelos anteriores. A ligação é uma abstração arquitetural e esse novo tipo de ligação merece um conceito de primeira classe.

Modelos de características. É um modelo simétrico; assume que há uma ADL com características de composição padrões, mas com algum modelo de características interno.

a) Características internas - Nessa abordagem, as características estão explicitamente consideradas na definição interna das abstrações arquiteturais, mas essa definição é fixa. Comportamentos relacionados a essas características podem ser especificadas como parte da especificação;

b) Características de primeira classe - Essa abordagem inclui as definições de características como entidades explícitas no nível arquitetural. Um componente é descrito como um conjunto de aspectos, um explícito modelo de composição e uma interface comum.

Múltiplas dimensões. Muito parecido com a anterior, no entanto, o modelo de características não é interno, mas explícito. As características são consideradas entidades de larga escala e podem ser manipuladas.

a) Visões de características - Essa abordagem define a arquitetura de toda característica em uma visão independente e, posteriormente, essas visões são relacionadas através de algum mecanismo.Obviamente, esse modelo é

bastante parecido com as visões arquiteturais e, portanto, é bastante consistente;

b) Dimensões de primeira classe - Nesse modelo, todo componente tem sua própria dimensão e a estrutura arquitetural somente é criada quando essas dimensões são fundidas. Mesmo sendo muito expressiva, a conveniência da utilização desse tipo de abordagem ainda não pode ser determinada.

3.3 MODELAGEM DE ARQUITETURAS DE SOFTWARE ORIENTADAS A ASPECTOS COM AO-ADLS

Uma diversidade de soluções de modularização dos interesses transversais em nível de arquitetura foram propostas, como por exemplo, GARCIA (2006), PÉREZ (2003), PESSEMIER (2006), PINTO (2005), algumas estendendo conhecidas ADLs e outras desenvolvidas incluindo as abstrações conhecidas de frameworks ou linguagens de programação orientadas a aspecto como aspectos, pontos de junção, pontos de corte, adendos e declarações intertipo. A esse novo conjunto de soluções dá-se o nome de ADLs Orientadas a Aspectos (*Aspect-Oriented Architecture Description Languages - AO ADLs*). Uma AO ADL é *simétrica* quando conserva os elementos das ADLs, apenas estendendo os seus conceitos. Ou seja, nenhum novo elemento é incluído na modelagem arquitetural. Por outro lado, chama-se de AO ADL *assimétrica* aquela que introduz novas abstrações na descrição arquitetural, como por exemplo, separando o conceito de componente e de aspecto. A seguir serão apresentadas e discutidas três propostas de AO ADL (duas assimétricas e uma simétrica).

3.3.1 FUSEJ

SUVEE (2005) define uma nova AO ADL chamada FuseJ. O trabalho defende que, em nível de arquitetura, não é necessária a separação de aspectos e componentes, já que o comportamento apresentado pelos aspectos de nível arquitetural, não difere do comportamento do componente. A única diferença se dá na forma como um e outro interagem com o restante do sistema. Ao invés de introduzir uma nova abstração para representar o interesse transversal (como é defendido pelas AO ADLs assimétricas), propõe apenas aplicar um novo mecanismo de composição entre as construções já existentes.

Para viabilizar essa nova composição, os autores introduzem um novo conceito de interface, chamada de *gate*. O *gate* é responsável por expor a implementação interna do

componente e fornecer um ponto de acesso para permitir interações regulares e aspectuais com outros componentes. Portanto, um *gate* define um conjunto de possíveis pontos de junção para o componente, agindo como um canal de comunicação duplo (entrada e saída). Quando age como canal de entrada o *gate* apresenta a seguinte semântica: "execute o serviço ao qual fornece acesso". Já quando age como canal de saída, apresenta a semântica: "quando estiver executando um serviço ao qual o *gate* de entrada dá acesso, dê início a alguns comportamentos adicionais que são necessários". Esses comportamentos definem a forma como os componentes vão interagir são os conectores.

Já o conector "é responsável por descrever as interações regulares e orientadas a aspectos entre um (ou mais) *gate(s)*". Os conectores regulares têm o mesmo comportamento que os conectores descritos pela maioria dos modelos de componentes. Os conectores que expressam as conexões aspectuais são uma extensão dos conectores regulares que permitem a especificação da forma como um *gate* afeta o comportamento de outro *gate*. Os autores destacam que a utilização dessa proposta tem a vantagem de aumentar a reusabilidade dos componentes, já que o arquiteto não precisa decidir se determinado componente será regular ou aspectual.

A ferramenta para suporte à FuseJ ainda encontra-se em desenvolvimento. No site do grupo (FUSEJ, 2006) há uma versão disponível para download, no entanto, é advertido que não está completa. Ainda não há um mecanismo de rastreabilidade entre FuseJ e o projeto detalhado de software.

3.3.2 FRACTAL ASPECT COMPONENT

O Fractal Aspect Component (FAC) (PESSEMIER, 2006) (CONSORTIUM, 2006) é um modelo que reúne os conceitos da Engenharia de Software Baseada em Componentes (CBSE) e os conceitos da Programação Orientada a Aspectos (AOP). FAC propõe a decomposição de um sistema de software em componentes regulares e componentes aspectuais. Utiliza como base, um modelo orientado a componentes denominado Fractal (BRUNETON, 2004), que utiliza uma ADL chamada Fractal-ADL para representar as descrições arquiteturais.

O modelo FAC propõe três novas abstrações na Fractal-ADL para dar suporte aos interesses transversais: componente aspectual (AC), interação aspectual e domínio aspectual.

Um componente aspectual, modulariza um interesse transversal. A diferença entre o componente aspectual e o componente regular é que, no componente aspectual, o serviço

oferecido é um *advice*. Esse *advice* representa o comportamento que irá afetar um conjunto de componentes regulares. Os autores garantem que a solução é simétrica, já que o componente aspectual tem a mesma natureza do componente regular. Como conseqüência, um aspecto representado por um componente adquire a capacidade de reuso e a composição entre componentes aspectuais e regulares é feita da mesma forma, utilizando as mesmas regras.

São definidos dois tipos de interação: regular e aspectual. A interação regular é um canal de comunicação entre as interfaces funcionais de dois componentes regulares; a interação aspectual é também um canal de comunicação, mas entre componentes aspectuais e componentes regulares, definindo uma interação transversal. Há, portanto, dois níveis de composição a serem considerados: os interesses regulares são compostos através de interação regular e os interesses transversais são compostos utilizando-se a interação aspectual. Por último, o domínio aspectual é um domínio definido entre o componente aspectual e os componentes regulares afetados por ele. O objetivo do domínio aspectual é permitir uma visão geral de todos os componentes que são afetados por um determinado aspecto. A seguir (FIG 3.4) é apresentada a noção do domínio aspectual em uma aplicação baseada em componentes genérica. A definição do sistema é estendida com a inclusão de três novos interesses transversais: logging, persistência e controle de transação. Os domínios aspectuais estão representados na figura pelos retângulos pontilhados e as interações aspectuais foram omitidas para facilitar o entendimento.

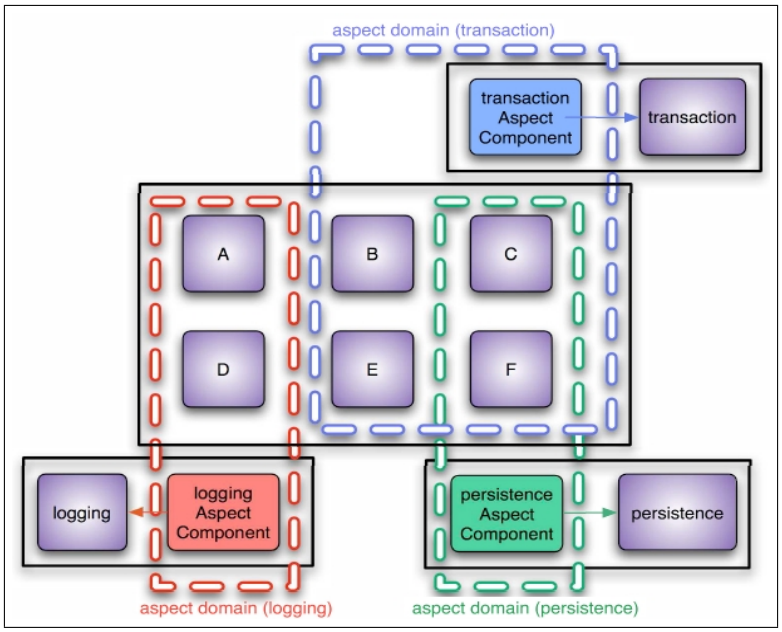


FIG. 3.4: Representação de domínios aspectuais. (PESSEMIER, 2006)

O modelo Fractal apresenta um conjunto de ferramentas para suporte (CONSOR-

TIUM, 2006), no entanto, assim como FuseJ, ainda não tem um mecanismo para rastreabilidade entre a arquitetura e o projeto detalhado de software.

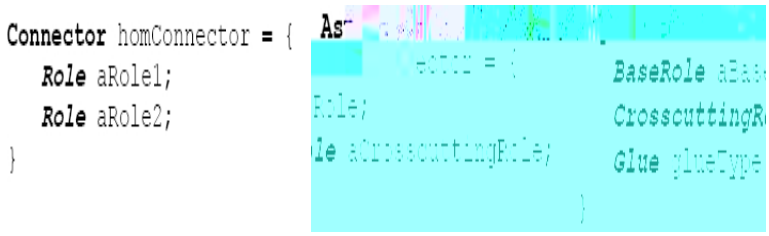
3.3.3 ASPECTUALACME

AspectualACME (GARCIA, 2006) é uma adaptação da ADL ACME, desenvolvida para suportar a modelagem orientada a aspectos de arquiteturas de software. Além das vantagens de ACME (independente de domínio, conjunto central de elementos simplificados), AspectualACME apresenta o conector aspectual como única abstração necessária para o suporte de interesses transversais no nível de arquitetura. É uma enorme vantagem com relação a outras AO ADLs, já que apenas uma pequena adaptação no conector é necessária, simplificando a solução e facilitando a adoção pelos arquitetos.

O conector tradicional não é suficiente para modelar *interações transversais*, isto é, interações entre componentes regulares e componentes que representam interesses transversais (chamados de *componentes aspectuais*) porque a forma como um componente aspectual interage com um componente regular é diferente da interação entre dois componentes regulares. Aqui cabe a ressalva de que AspectualACME não faz distinção entre componentes, o termo aspectual só é utilizado para diferenciar o comportamento dos componentes na interação. Como exemplo dessa diferença, uma interação transversal relaciona um serviço provido por um componente aspectual a serviços providos e/ou requeridos de um componente regular, sendo diferente da interação entre componentes regulares, onde serviços providos podem apenas aparecer relacionados a serviços requeridos de outros componentes.

O conector aspectual distingue-se do conector regular por apresentar uma nova interface. A necessidade de uma nova interface se faz por dois motivos principais: (i) fazer a distinção entre os papéis dos elementos que participam da interação transversal, ou seja, componentes aspectuais e componentes regulares e (ii) expressar a forma como esses dois tipos de componentes interagem. Dessa forma, a interface do conector aspectual apresenta os seguintes elementos em sua estrutura: (i) papel base (*base role*), (ii) papel transversal (*crosscutting role*) e (iii) cláusulas *glue*. A FIG 3.5 a seguir ilustra as diferenças entre o conector regular (definido por ACME) e o conector aspectual de AspectualACME.

O papel transversal do conector deve estar sempre conectado à porta fornecida de um componente com comportamento aspectual. Ele define o componente participante da interação que exercerá o papel aspectual, ou seja, irá afetar o outro componente. O papel base deve estar conectado às portas fornecidas ou requeridas de um ou mais



(a) Conector Regular (b) Conector Aspectual

FIG. 3.5: Conectores em AspectualACME. (GARCIA, 2006)

componentes de comportamento regular. Ele define o(s) participante(s) da interação que exercerá(ão) o(s) papel(is) de componente(s) regular(es), ou seja, que será(ão) afetado(s) pelo componente aspectual.

Um conector aspectual deve conter, no mínimo, um papel base e um papel transversal. A composição entre os dois componentes é fornecida pela cláusula *glue*. Ela é responsável por definir detalhes da composição entre os componentes, como por exemplo, o lugar onde o componente base será afetado pelo componente aspectual. Em uma composição simples, a cláusula glue pode ser apenas uma declaração como *after*, *around* e *before* (onde a semântica é a mesma dos advices em AspectJ (FOUNDATION, 2006)). Em composições mais complexas a *glue* pode ser representada por um protocolo ou procedimento, como por exemplo, no caso de conectores heterogêneos (GARCIA, 2006).

Outra extensão à ADL ACME que se faz necessária é a inclusão de mecanismos de quantificação. É uma extensão necessária para que cada ponto de junção não precise ser explicitamente referenciado na descrição arquitetural. Assim, estendeu-se o campo de interconexão do sistema (*attachments*) para incluir um mecanismo de quantificação. Dessa forma, foi possibilitada a inclusão de *wildcards* (representados pelo símbolo *) para denotarem nomes ou parte de nomes de componentes e suas portas. A quantificação deve ser utilizada na conexão entre papel base de um conector aspectual e um ou mais componente(s) regular(es). A pequena descrição de um sistema exemplo na FIG 3.6 ilustra a utilização do mecanismo de quantificação.

AspectualACME também permite a especificação entre interações aspectuais em nível arquitetural entre dois ou mais conectores aspectuais que tenham pontos de junção de operadores de composição: (i) precedência(*precedence*) e (ii) exclusão mútua (*xor*). O relacionamento de precedência indica qual relação aspectual tem precedência em relação a outra (é definida em ordem de precedência) e a exclusão mútua indica qual relação deve prevalecer. Ambas são definidas quando os conectores têm o mesmo ponto de junção. A relação de precedência só faz sentido, se o tipo da cláusula *glue* dos conectores envolvidos

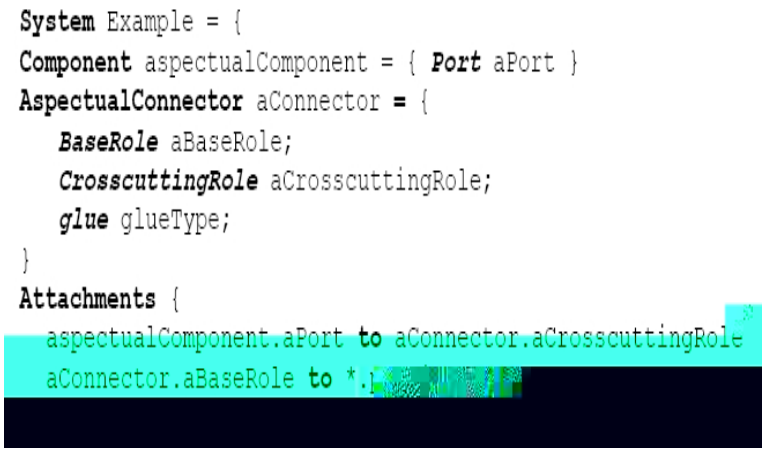


FIG. 3.6: Exemplo de sistema em AspectualACME utilizando mecanismo de quantificação.(GARCIA, 2006)

for o mesmo. O arquiteto pode especificar ainda, se a precedência é válida em toda a arquitetura ou apenas no ponto de junção especificado.

No exemplo a seguir (FIG 3.8 e 3.7), é definido um pequeno sistema com quatro componentes, dois conectores regulares e dois conectores aspectuais. As conexões entre conectores regulares e aspectuais, e os componentes são definidas no campo de interconexão (*attachments*). Note que não há diferença entre as conexões de ACME, exceto pela inclusão dos novos papéis (papel base e papel transversal). Pode-se observar ainda, a relação de precedência existente na porta onde dois diferentes conectores aspectuais se interceptam (note na descrição textual, que o tipo da cláusula *glue* dos dois conectores é igual) . No caso, o conector AC2 tem precedência sobre o AC1. Por último, exemplifica a utilização do *wildcard* como mecanismo de quantificação, fazendo com que o conector AC2 intercepte todas as portas que comecem com o nome *response*.

3.4 VANTAGENS E DESVANTAGENS DAS SOLUÇÕES APRESENTADAS

Uma das questões que envolvem modelagem de arquitetura de software orientada a aspectos, refere-se ao tratamento dado a aspectos no nível arquitetural. NAVASA (2002) sugere que todas as AO ADLs propostas devem apresentar o aspecto como elemento de primeira ordem. É uma linha defendida por alguns autores que será analisada com mais profundidade. O trabalho proposto por PESSEMIER (2006) segue essa linha, definindo uma solução assimétrica.

A vantagem desse tipo de solução é que os interesses transversais ficam modularizados em uma abstração separada do restante do sistema e os elementos utilizados se aproximam

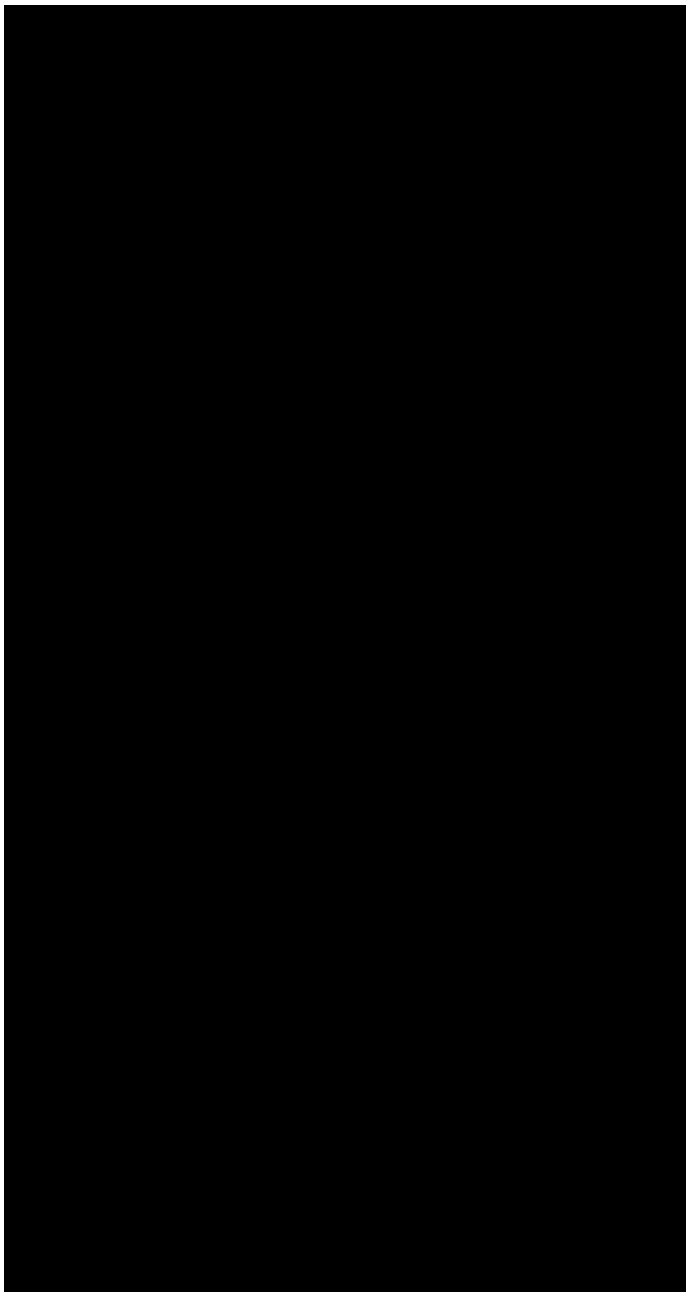


FIG. 3.7: Descrição textual do Exemplo de sistema em AspectualACME

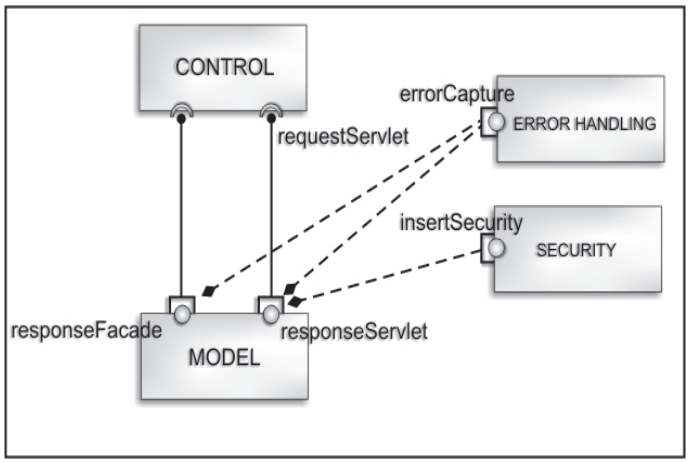


FIG. 3.8: Exemplo de sistema em AspectualACME

bastante do projeto detalhado e da implementação. No entanto, traz uma grande desvantagem: quando se define o aspecto como elemento de primeira ordem, são trazidos para a modelagem arquitetural, novos elementos como pontos de junção, adendos, interfaces de crosscutting, novas formas de interação entre componentes aspectuais e componentes regulares e entre componentes aspectuais. Essas novas definições trazem consigo uma dificuldade de adaptação das ferramentas já em uso nas ADLs convencionais, além de exigir um esforço grande de adaptação dos arquitetos que utilizam essas ADLs. Por ter-se, no nível de arquitetura, uma modelagem abstrata do sistema, cujo objetivo é poder observar o funcionamento do sistema como um todo, talvez não seja necessária essa quantidade de informações acerca da interação aspectual.

FuseJ (SUVEE, 2005) e AspectualACME (GARCIA, 2006) têm propostas semelhantes, no entanto, FuseJ define um novo tipo de interface chamado de *gate*. A inclusão desse novo conceito de interface é aparentemente desnecessária, já que, pela característica de abstração da descrição arquitetural, pode-se aproveitar o conceito tradicional de interface para a representação dos pontos de junção. Outro aspecto a se considerar, é que FuseJ não trabalha com o conceito de configuração - que é utilizado pela maioria das ADLs convencionais e que está definido por MEDVIDOVIC (2000) em seu framework para classificação e comparação de ADLs. Há, portanto, um contraste com a forma de trabalho da maioria das ADLs que declaram as instâncias de conexão dentro da seção de configuração.

AspectualACME apresenta as mesmas vantagens de ACME (conjunto central de elementos simplificado e independência de domínio). Além disso, utiliza o conceito chamado de modelo de interação aspectual (*aspectual binding model*) (CUESTA, 2005), que define que não há a necessidade de ter uma entidade explícita para representar o interesse transversal, cabendo ao modelo arquitetural apenas fornecer um novo tipo de ligação entre

componentes. Isso a torna uma solução mais leve e facilita tanto a adaptação dos arquitetos, quanto das ferramentas de suporte. Por último, por ser uma extensão de ACME, traz consigo os conceitos do framework de comparação (MEDVIDOVIC, 2000) (componente, conector e configuração arquitetural). A TAB 3.1 apresenta uma comparação entre as três AO-ADLs apresentadas e discutidas no contexto desse trabalho.

AO-ADLs	**Tipo**	**Aspecto**	**Interface do componente**	**Conector**	**Atende ao framework de Medvidovic?**
FuseJ	Assimétrica	Definido na interação	Gate	Regular e aspectual	Não
Fractal	Assimétrica	Componente aspectual	Interface + advice	Regular e aspectual	Não
AspectualACME	Simétrica	Definido na interação	Porta + comportamento	Regular e aspectual	Sim

TAB. 3.1: Comparação entre as AO-ADLs apresentadas

Embora tenham contribuído com a inserção dos conceitos de orientação a aspectos em arquitetura de software, as AO ADLs estão sujeitas às mesmas críticas enfrentadas pelas ADLs regulares, principalmente pela falta de rastreabilidade para as abordagens em nível de projeto e a falta de ferramentas de suporte. Esses problemas comprometem a utilização, adoção e sobrevivência dessas linguagens.

A UML 2.0 trouxe novos elementos para a modelagem arquitetural, como componentes, portas, interfaces (providas e requeridas), e a decomposição hierárquica de componentes, trazendo a comunidade de desenvolvedores mais próxima à prática de modelagem de arquitetura de software. No entanto, ainda há uma grande diferença entre as representatividades providas pela UML 2.0 e pelas ADLs, além de uma carência de elementos para dar suporte aos interesses transversais em nível de arquitetura.

É nesse ponto que o trabalho que está sendo proposto se encaixa. A solução proposta busca estender a UML 2.0 a fim de que esta passe a ter a mesma expressividade semântica das ADLs. Essa extensão facilita a utilização das linguagens de descrição de arquitetura pelos arquitetos, já que está alinhada com a indústria.

Além disso, a extensão permite a inclusão de interesses transversais na modelagem da arquitetura. Para isso, tomamos como base a AO-ADL AspectualACME, já que tem um conjunto de elementos bastante simplificado (herança de ACME) e representa a transversalidade na conexão, evitando a inclusão de novos elementos na modelagem da arquitetura.

Essa abordagem facilita a adaptação de arquitetos e de ferramentas.

Por fim, com uma solução para descrição da arquitetura baseada na UML, propomos um conjunto de regras para o mapeamento entre a descrição arquitetural e o projeto de software.

Alguns trabalhos relacionados à nossa abordagem serão discutidos no capítulo 4.

4 TRABALHOS RELACIONADOS

Neste capítulo serão discutidos alguns trabalhos relacionados com a nossa proposta. Consideramos trabalhos relacionados, aqueles que lidam com modelagem arquitetural utilizando UML 2.0. A seção 4.1 apresenta a tentativa de representar em UML 2.0, os conceitos unificados de algumas AO ADLs. A seção 4.2 apresenta uma proposta de representação dos elementos arquiteturais de ACME, em UML 2.0. Na seção 4.3 é apresentada uma discussão envolvendo a representação de interesses tipicamente transversais, utilizando os conectores fornecidos por UML 2.0. Na seção 4.4, os trabalhos apresentados são discutidos.

4.1 PROPOSTA INTEGRADA PARA MODELAGEM DE ARQUITETURAS ORIENTADAS A ASPECTOS

KRECHETOV (2006) apresenta um trabalho com foco na representação modular de aspectos em nível de arquitetura e faz uma tentativa de integrar as melhores práticas de algumas abordagens arquiteturais orientadas a aspectos em uma única linguagem de modelagem de propósito geral, baseada em UML (OMG, 2006b). O trabalho avalia quatro AO ADLs (PCS Framework (KANDÉ, 2003), AOGA (KULESZA, 2004), TranSAT (BARAIS, 2004) e DAOP-ADL (PINTO, 2005)), com o objetivo de extrair as melhores características de cada uma delas e determinar uma nova linguagem de modelagem, com notação compatível com UML2.0. Para avaliação das três abordagens, foi utilizado um framework de comparação que inclui alguns conceitos de DSOA:

- Aspectos;
- Componentes;
- Pontos de corte(*Point-cut*);
- Adendos(*Advice*);
- Crosscutting estático e dinâmico;
- Relação aspecto-componente;
- Relação aspecto-aspecto.

O conceito de componente é considerado porque é um importante elemento de construção (mesmo sem a presença de aspectos). Aspecto, pontos de corte e adendos são incluídos por serem as abstrações primárias utilizadas para capturar os interesses transversais. A inclusão do mecanismo de transversalidade (estático e dinâmico) dá ênfase ao relacionamento transversal em si, deixando fora de foco o mecanismo de composição (normalmente transparente). As duas categorias de relacionamento (aspecto-componente e aspecto-aspecto) são importantes porque definem como os elementos de construção arquitetural estão colaborando para atingir os requisitos do sistema.

Analisando a forma como cada AO ADL implementa os elementos presentes no *framework* de comparação, os autores apresentam uma proposta onde os conceitos de DSOA, em nível de arquitetura, são representados da seguinte forma: (i) aspectos são modelados como componentes UML estereotipados; (ii) componentes são modelados como componentes UML; (iii) interfaces de crosscutting (pertencentes ao aspecto) são modeladas como interfaces UML estereotipadas, onde os nomes dos métodos representam os pointcuts; (iv) os pointcuts e advices são modelados como sub-componentes que utilizam o conector de delegação de UML para realizar as regras de composição e os adendos; (v) o crosscutting estático é modelado como uma associação unidirecional entre uma interface de crosscutting e um componente; (vi) a relação entre o aspecto e o componente é modelada através de um conector UML ligando a interface de crosscutting do aspecto a um componente ou a interface de um componente; (vii) a relação entre aspectos é modelada através de uma relação de herança (generalização da UML) ou precedência. A FIG 4.1 mostra como fica a representação de cada elemento.

4.2 ABORDAGEM DE GOULÃO E ABREU PARA MAPEAMENTO ENTRE ACME E UML 2.0

GOULAO (2003) mapeia as abstrações de modelagem propostas por ACME e para a UML 2.0, com o objetivo de diminuir a distância existente entre as ADLs e a UML, utilizando os novos elementos arquiteturais propostos pela UML 2.0, aliados a algumas restrições em OCL 2.0 (OMG, 2006a). Em ACME, um componente tem portas, propriedades, representações e um conjunto de regras de projeto. O conceito mais próximo, em UML, é o componente. No entanto, para permitir a convivência do componente ACME com outros conceitos que também serão representados por componentes UML, a solução utiliza o estereótipo «AcmeComponent». Além disso, define a restrição OCL abaixo garantindo que as interfaces do componente ACME se manifestam apenas através

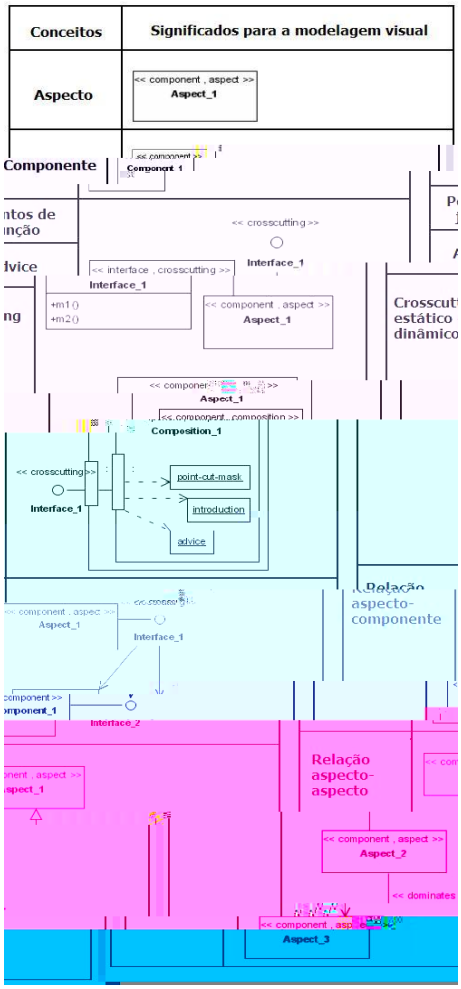


FIG. 4.1: O modelo proposto por KRECHETOV (2006).

de portas ou propriedades ACME.

```
context Component inv: -- Invariant 1
   self.IsAcmeComponent() implies
      self.ownedPort->forAll(ap|
         ap.IsAcmePort() or
         ap.IsAcmeProperty()) and
      self.HasNoOtherInterfaces()
```

FIG. 4.2: Restrições OCL para componentes. (GOULAO, 2003)

A porta definida em ACME identifica pontos de interação entre o componente e o ambiente. Elas podem ser simples, como assinaturas operação ou complexas, como um conjunto de chamadas a procedimentos com restrições dizendo a ordem que devem ser chamadas. As portas em UML são propriedades de classificadores que especificam pontos de interação entre um classificador (no caso, o componente) e o ambiente (no caso, o restante do sistema). As portas UML apresentam interfaces requeridas e providas, que podem conter ainda, pré e pós-condições. A solução define a porta como uma combinação

da porta UML com suas respectivas interfaces providas e recebidas. Além disso, define uma restrição que a porta ACME só pode ser usada em um componente ACME e só pode ter uma interface provida e uma requerida.

```
context Port inv: -- Invariant 2
   self.IsAcmePort() implies
      self.owner.IsAcmeComponent() and
     (self.required->size()=1) and
     (self.provided
```

FIG. 4.3: Restrições OCL para portas. (GOULAO, 2003)

Como UML 2.0 não apresenta os conectores como elementos de primeira ordem, a proposta é utilizar componentes com estereótipo «AcmeConnector» para representar os conectores ACME. Segundo os autores, isso garante a representatividade necessária ao conector. Além disso, é definida uma restrição garantindo que as interfaces do novo conector só podem ser papéis ou propriedades.

```
context Component inv: -- Invariant 3
   self.IsAcmeConnector() implies
      self.ownedPort->forAll(ap|
         ap.IsAcmeRole() or
         ap.IsAcmeProperty()) and
      self.HasNoOtherInterfaces()
```

FIG. 4.4: Restrições OCL para conectores. (GOULAO, 2003)

A FIG 4.5 é um exemplo da utilização dos novos componentes e conectores definidos na solução. No exemplo, são definidos os componentes *client* e *server*, com interfaces que não se complementam, mas o conector RPC implementa um determinado protocolo para que elas possam se conectar. Além disso, foram incluídas interfaces providas e requeridas nos conectores, para ilustrar que há a capacidade de uma comunicação bidirecional.

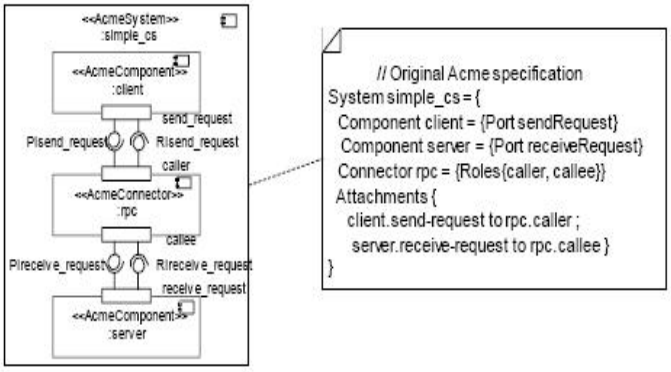


FIG. 4.5: Exemplo de utilização dos novos elementos. (GOULAO, 2003)

Como em ACME os papéis estão para os conectores assim como as portas estão para

os componentes, a idéia é utilizar portas como papéis, observadas as restrições descritas a seguir.

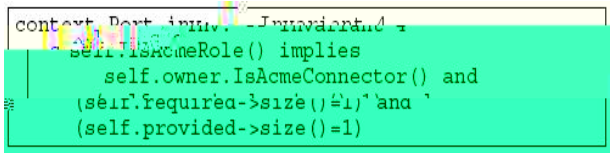


FIG. 4.6: Restrições OCL para conectores. (GOULAO, 2003)

Os sistemas em ACME representam um grafo onde os componentes interagem. O conceito de pacote em UML (com o estereótipo «subsistema») representa um conjunto de elementos, no entanto, não representam a estrutura em que estão contidos e não servem para definir as propriedades de certo nível do sistema. Para resolver esses problemas, são representados por pacotes com o estereótipo «AcmeSystem» e as seguintes restrições:

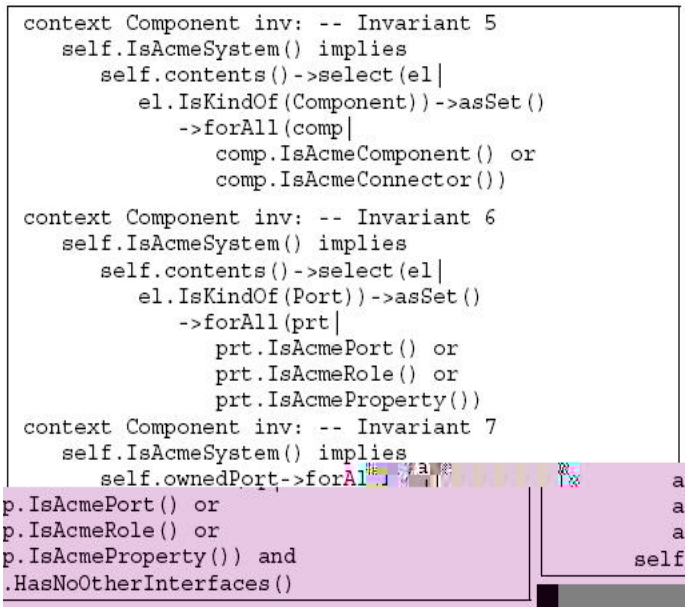


FIG. 4.7: Restrições OCL para sistemas. (GOULAO, 2003)

As representações em ACME fornecem um mecanismo para adicionar detalhes a componentes e conectores. Os mapas de representação são utilizados para mostrar como os níveis mais altos de representação se relacionam com os níveis mais baixos. Em UML, o conector delegation tem a mesma função dos mapas de representação de ACME.

As propriedades representam uma informação semântica acerca do sistema e de seus elementos arquiteturais. A idéia é fazer com que essas propriedades sejam vistas de fora do componente, utilizando restrições OCL. Para isso, é definida uma porta UML com uma interface provida para que o usuário do componente possa acessar suas propriedades. Por-

tanto, é utilizada uma porta com o estereótipo «AcmeProperty» e uma interface provida que define operações de *get* e *set* para que os valores das propriedades possam ser acessados.

```
context Port inv: -- invariant 9
   self.IsAcmeProvided() implies
     (self.required->IsEmpty()) and
     (self.provided->size()=1)
```

FIG. 4.8: Restrições OCL para propriedades. (GOULAO, 2003)

4.3 DESCRIÇÃO DE ASPECTOS MEDIANTE CONECTORES UML 2.0

Outro trabalho interessante envolvendo ADLs e UML é apresentado por RODRÍGUEZ (2004). No trabalho são feitas considerações sobre a utilização de aspectos de segurança e persistência, em um sistema hipotético, e a tentativa de representá-los através de conectores, utilizando os novos elementos arquiteturais apresentados na UML 2.0. Essa tentativa é feita, partindo do princípio de que não há necessidade de estender os conceitos de cada elemento. A primeira tentativa é feita utilizando um sistema (FIG 4.9) onde há uma comunicação entre dois componentes (TOrigen e TDestino) representada por um conector denominado TEnlace. Trata-se, portanto, de uma comunicação normal, definida por um conector assembly de UML 2.0. Para inserir nesse sistema um aspecto de segurança, basta substituir o conector por outro, onde a segurança é implementada. No exemplo, optou-se por um mecanismo de tunelamento, que se fornece segurança ao canal, criptografando as informações. Toda a mensagem emitida pela origem é criptografada pelo componente TCrypt, enviada pelo mesmo canal antigo (TEnlace) e decriptografada no destino.

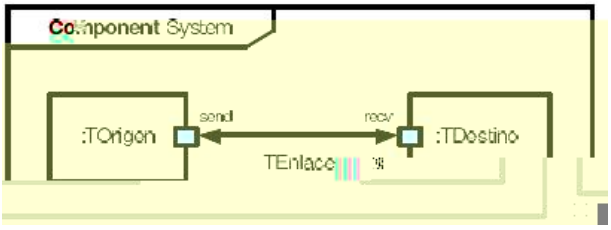


FIG. 4.9: Transferência de informação sem segurança. (RODRÍGUEZ, 2004)

Essa mudança pode ser feita, em UML, substituindo-se o conector por todo o sistema descrito, mostrando que dessa forma pode-se incluir aspectos em um sistema. No entanto, seria interessante colocar tudo em um único componente (FIG 4.10) para que a modularidade do sistema não fosse perdida. Na verdade, o que se pretende, não é substituir

um conector por um componente e sim estender o conector para um elemento composto. Dessa forma, todo o processo se resumiria na substituição de um conector por outro.

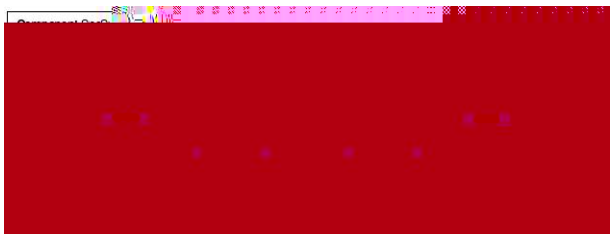


FIG. 4.10: Transferência de informação com segurança. (RODRÍGUEZ, 2004)

Outro exemplo parecido apresentado pelo trabalho, tenta introduzir um novo interesse de persistência, que é menos ligado ao conceito de comunicação. Como no exemplo anterior, há um sistema de origem (FIG 4.11), representado por dois componentes (TFuente e TCaptura) e um conector (TFlujo).

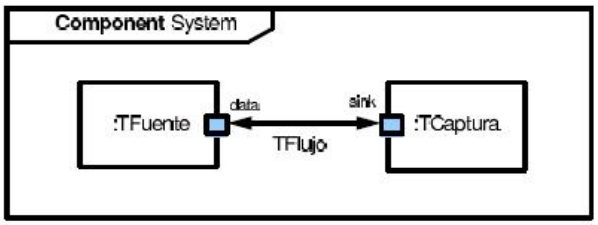


FIG. 4.11: Transferência de informação sem persistência. (RODRÍGUEZ, 2004)

Para se introduzir a persistência (FIG 4.12), é inserido um novo componente (TStore) que representa um banco de dados e uma nova conexão (Tee). Dessa forma, a cada informação transmitida de TOrigen para TCaptura, é gravada uma cópia dos dados no componente TStore. Novamente, substitui-se a conexão original por uma conexão composta, que permite agrupar o aspecto em um único módulo.

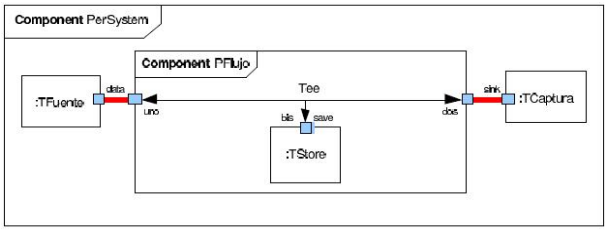


FIG. 4.12: Transferência de informação com persistência. (RODRÍGUEZ, 2004)

No trabalho, fica evidenciado que a UML 2.0 é muito mais flexível e precisa em relação a sua versão anterior, no que diz respeito à especificação arquitetural. No entanto, sua utilização como linguagem de descrição arquitetural ainda está muito distante de se tornar

realidade. Para tentar adaptar os conceitos de UML 2.0 às AO ADLs, os autores propõem extensões simples, como a definição de conectores como elementos de primeira ordem, podendo, dessa forma, ter uma estrutura composta.

4.4 ANÁLISE DOS TRABALHOS RELACIONADOS

KRECHETOV (2006) apresenta uma solução baseada em UML 2.0 para incluir, na representação arquitetural, os elementos da orientação a aspectos. Em sua abordagem, avalia quatro AO ADLs com o objetivo de integrar as propostas em uma única solução. Embora utilize apenas as extensões fornecidas pela UML 2.0, insere novos elementos na descrição da arquitetura (solução assimétrica). KRECHETOV (2006) incorre em alguns erros semânticos como, por exemplo, a representação de adendos, pontos de corte e declarações inter-tipo (introdução) através de sub-componentes. Além disso, permite conexões entre interfaces providas e conexões de interfaces diretamente para o interior do componente (caso da relação entre aspectos), o que não é permitido pela UML 2.0.

GOULAO (2003) tenta fazer um mapeamento da ADL ACME para UML 2.0, utilizando apenas as extensões propostas pela própria linguagem (sem alterar o metamodelo). Embora seja uma solução considerada "leve", a tentativa incorre em alguns erros conceituais graves. A representação de um conector por um componente estereotipado, faz com que o conector tenha uma estrutura diferente do que é proposto em ADLs. Isso traz distorções como: um conector pode estar conectado a outro conector através de um outro conector, já que um componente pode ligar-se a outro componente através de um conector. Além disso, por se tratar a UML de uma linguagem visual, a representação de um conector através do desenho de um componente pode acarretar em um problema semântico em primeira instância, já que o mesmo desenho representa dois elementos completamente distintos dentro do papel de descrição arquitetural. Analogamente, tal discussão envolvendo conectores e componentes pode ainda ser estendida para portas e papéis. O trabalho propõe a representação dos papéis do conector através de interfaces ou portas do componente. Mais uma vez, pode nos levar a conclusão que o conector tem interfaces ou portas, providas e requeridas, o que não é realidade na ADL ACME. Além disso, a representação visual de portas e papéis também poderia ser confundida.

O trabalho proposto por RODRÍGUEZ (2004) é de extrema relevância já que, depois de feitas algumas avaliações com candidatos clássicos a aspectos, como persistência e segurança, chegou-se a conclusão que, para representar aspectos através de conectores, eram necessárias extensões mais profundas do que as proporcionadas pela linguagem UML.

Sugere, portanto, algumas alterações no metamodelo UML 2.0 para a representação de aspectos através de conectores.

A nossa abordagem permite a representação dos aspectos em nível arquitetural estendendo apenas um dos elementos arquiteturais de UML, o conector. No entanto, diferente das soluções encontradas, pois promove algumas alterações no metamodelo para não incorrer em falhas semânticas. O trabalho proposto é apresentado no capítulo 5.

5 SOLUÇÃO PROPOSTA: A LINGUAGEM DE MODELAGEM AD-AUML

Este capítulo apresenta a AD-AUML (*Architectural Description in Aspectual UML*), uma extensão de UML 2.0 para modelagem orientada a aspectos de arquiteturas de software, tendo como base os elementos de descrição arquitetural da linguagem AspectualACME (GARCIA, 2006). A escolha de AspectualACME se deu basicamente por dois motivos: o primeiro é que, por se tratar de uma extensão da ADL ACME, apresenta um conjunto de elementos simplificados e é independente de domínio; o segundo é que AspectualACME é uma solução simétrica, ou seja, apenas estende os elementos já conhecidos das ADLs. Essas duas características fazem de AspectualACME uma solução leve, onde o esforço de adaptação de arquitetos e de ferramentas é reduzido. Além disso, o conjunto de modificações no meta-modelo também é simplificado, sendo esse um de nossos objetivos (alterar o mínimo possível do meta-modelo de UML 2.0)

O trabalho evoluiu de uma proposta inicial de uso dos mecanismos de extensão oferecidos pela UML. Desse modo, na seção 5.1 apresentamos um *profile* UML 2.0 para descrever os elementos de AspectualACME. O *profile* para modelagem OA é discutido em termos de seus benefícios e limitações. Na seção 5.2 descrevemos uma proposta baseada na alteração do metamodelo da UML 2.0, que contorna as limitações da solução baseada em *profile*, resultando na linguagem AD-AUML. Na seção 5.3 apresentamos uma estratégia de transformação de modelos arquiteturais expressos em AD-AUML para modelos de projeto detalhado de software, expressos em AsideML (CHAVEZ, 2004), uma linguagem para modelagem de projetos orientada a aspectos.

5.1 UM *PROFILE* UML 2.0 PARA MODELAGEM ARQUITETURAL ORIENTADA A ASPECTOS

A UML 2.0 utiliza dois elementos básicos para modelagem arquitetural: o componente e o conector. Embora tenha apresentado novas contribuições para a modelagem da arquitetura de um sistema, a versão 2.0 da UML (OMG, 2006b) ainda não dá suporte aos conceitos arquiteturais apresentados pela maioria das ADLs. Além disso, não há, por parte da UML, nenhum suporte para a modelagem orientada a aspectos no nível arquitetural.

A primeira tentativa de adaptação da UML 2.0 para incorporar os novos conceitos de

DSOA provenientes da linguagem AspectualACME, foi a definição de um *profile* (SANDE, 2006) que estende a UML utilizando apenas os mecanismos de extensão providos pela própria linguagem. A definição através de um *profile* tem como principais vantagens, permitir uma rápida adaptação de arquitetos (os elementos a serem utilizados já são conhecidos) e a reutilização das ferramentas que dão suporte a UML (as ferramentas já dão suporte às extensões fornecidas pela linguagem). No *profile* proposto, foram analisados os elementos para descrição arquitetural de AspectualACME e a forma como estes poderiam ser representados em UML 2.0.

Componentes. Há uma correspondência entre o componente definido em AspectualACME e o componente proposto pela UML. O componente UML permite a definição de elementos internos (representações de AspectualACME) e o conector de delegação (*delegation*), que pertence ao componente UML, é responsável pelo mapeamento entre a porta do componente e os elementos internos que implementam os serviços providos pela porta (mapas de representação de AspectualACME). Portanto, foi definido que nenhuma alteração seria necessária para a representação de componentes de AspectualACME em UML.

Conectores. Em AspectualACME, um conector define papéis, que é um conceito inexistente em UML. Por outro lado, em UML, um conector liga interfaces, que é um conceito que não está presente em AspectualACME (utiliza porta como interface do componente). Para que uma representação do conector definido em AspectualACME fosse possível na UML 2.0, o conector de ligação (*assembly*) de UML passou a usar a notação de interface para denotar os papéis do conector. A conexão passou a ser representada como uma associação direcionada partindo da interface requerida para a interface provida dos componentes (FIG 5.1).

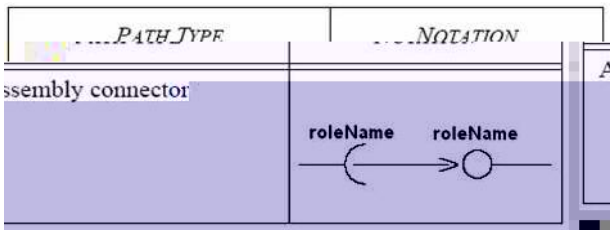


FIG. 5.1: Notação do conector de ligação *(profile)*. (SANDE, 2006)

Conectores aspectuais. Para permitir a modelagem de interações transversais através de conectores aspectuais, foi definida uma extensão do conector, proposto acima, utilizando um estereótipo «crosscuts». Além de indicar que esse conector estendido implementa uma relação aspectual, é preciso redefinir a regra de que toda conexão parte de uma

interface requerida de um componente para uma interface provida de outro componente (que não é o caso em interações aspectuais). Logo, a regra para a extensão estereotipada do conector diz que ele pode fazer conexões entre interfaces providas de componentes com interfaces providas e requeridas de outro componente. A cláusula *glue* é expressa através de uma *tagged value* (outra extensão provida pela linguagem), indicando o tipo de protocolo que está sendo utilizado (FIG 5.2).

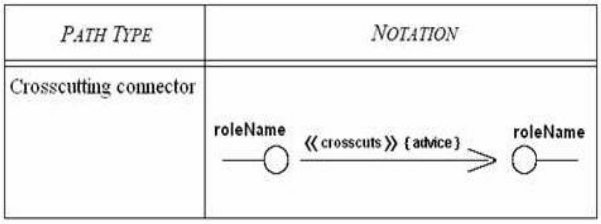


FIG. 5.2: Notação do conector aspectual *(profile)*. (SANDE, 2006)

Na FIG 5.3, apresentamos a arquitetura parcial de um sistema, utilizando os conceitos definidos no *profile*. Podemos observar três componentes, sendo dois com comportamentos regulares (Distribution e Business) e um com comportamento aspectual (Error Handling). Pode-se observar ainda, as extensões propostas para os dois conectores (regular e aspectual).

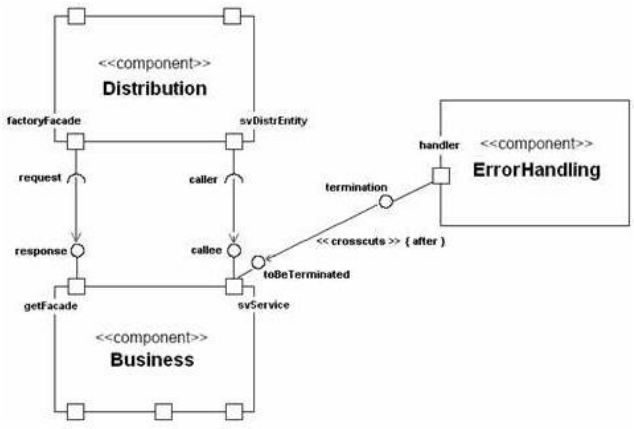


FIG. 5.3: Modelagem parcial de um sistema com o *profile* proposto. (SANDE, 2006)

5.2 A AD-AUML

A AD-AUML (Architectual Description in Aspectual UML) é uma extensão da linguagem de modelagem UML 2.0, que possibilita a inclusão de interesses transversais na

modelagem da arquitetura de um sistema, com o intuito principal de dar ao conector uma estrutura de primeira ordem.

Assim como na UML 2.0, as regras de boa formação necessárias para o modelo, exceto as que já são definidas no próprio metamodelo (através das multiplicidades), estão descritas em OCL. Esse conjunto de regras pode ser encontrado no APÊNDICE 1 dessa dissertação, onde os elementos do metamodelo que foram modificados estão formalmente descritos, no formato proposto pela UML. Alguns desses elementos tiveram apenas os seus nomes alterados, para melhorar a semântica e facilitar o entendimento.

Para desenvolvermos a discussão sobre o que está sendo proposto, dividiremos essa seção em três, onde em cada uma será comentado um elemento arquitetural. A seção 5.2.1 apresenta a discussão sobre as adaptações necessárias no componente e a seção 5.2.2 apresenta as novidades acerca do conector regular e, por último, a seção 5.2.3 apresenta o novo conector aspectual. A proposta de extensão apresentada busca contornar limitações da UML 2.0 para criação de modelos que descrevem a visão arquitetural conhecida como visão de componente-conector (também conhecida como visão de runtime) que é aquela tipicamente contemplada por ADLs.

5.2.1 COMPONENTES

A primeira análise feita no metamodelo da UML 2.0 (FIG 5.4 e FIG 5.5) foi na

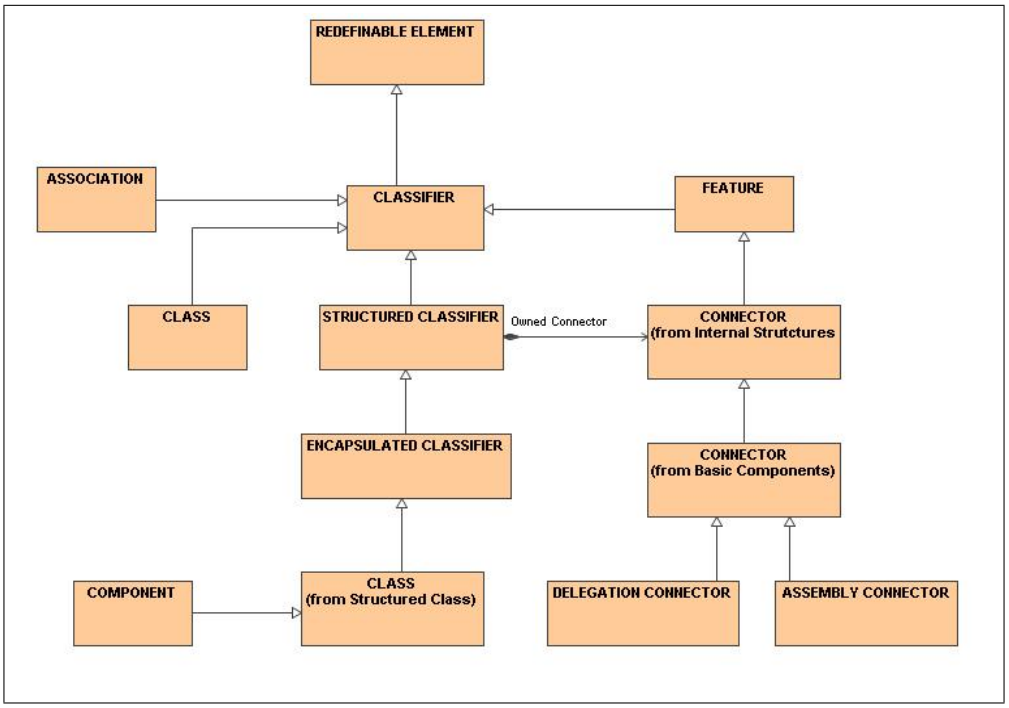


FIG. 5.4: Metamodelo parcial de UML 2.0 (elementos arquiteturais)

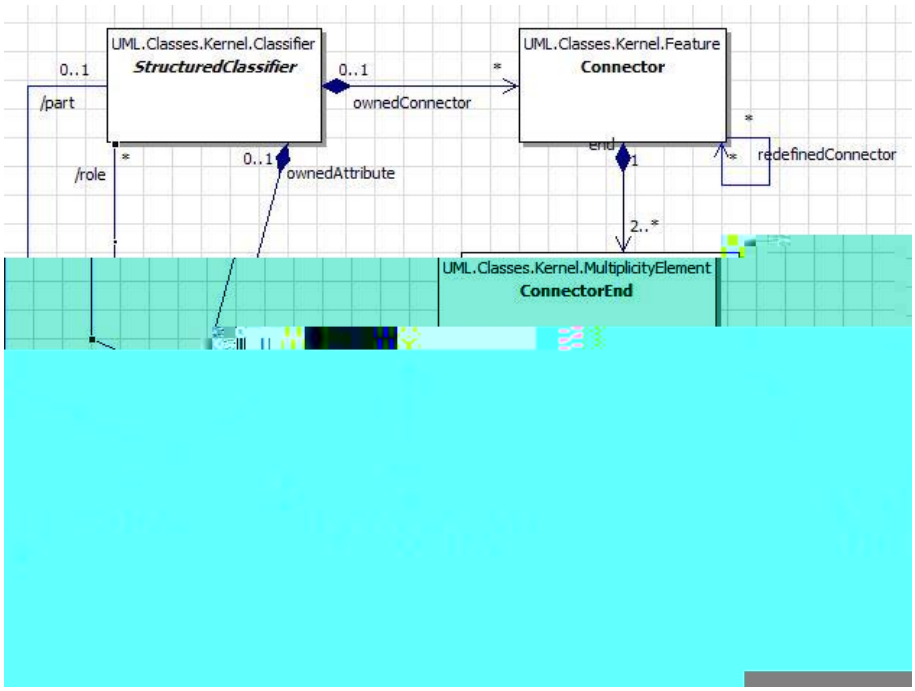


FIG. 5.5: Estrutura do elemento Structured Classifier

primeira classe, ie. a definição de um não implica a definição de outro. Para se manter a analogia entre UML e AspectualACME, foi preciso retirar a definição de conector de ligação da estrutura de definição de componente, retirando-se a classe CONNECTOR da hierarquia de FEATURE. Além disso, o conector de delegação de UML é representado em AspectualACME nas definições de mapas de representação, ou seja, o conector de delegação de UML não é um conector em AspectualACME. Também no intuito de manter a analogia neste ponto, foi criada a meta-classe BINDING. O BINDING será o elemento responsável por representar a descrição hierárquica de componentes, ou seja, indicará o elemento interno que irá implementar o serviço oferecido pelo componente. Dessa forma, a meta-classe DELEGATION CONNECTOR é extinta (FIG 5.6). Com essas alterações (retirada do CONNECTOR, eliminação de CONNECTOR END e criação de BINDING), a estrutura de componente em UML se torna semelhante à definição de componente em AspectualACME.

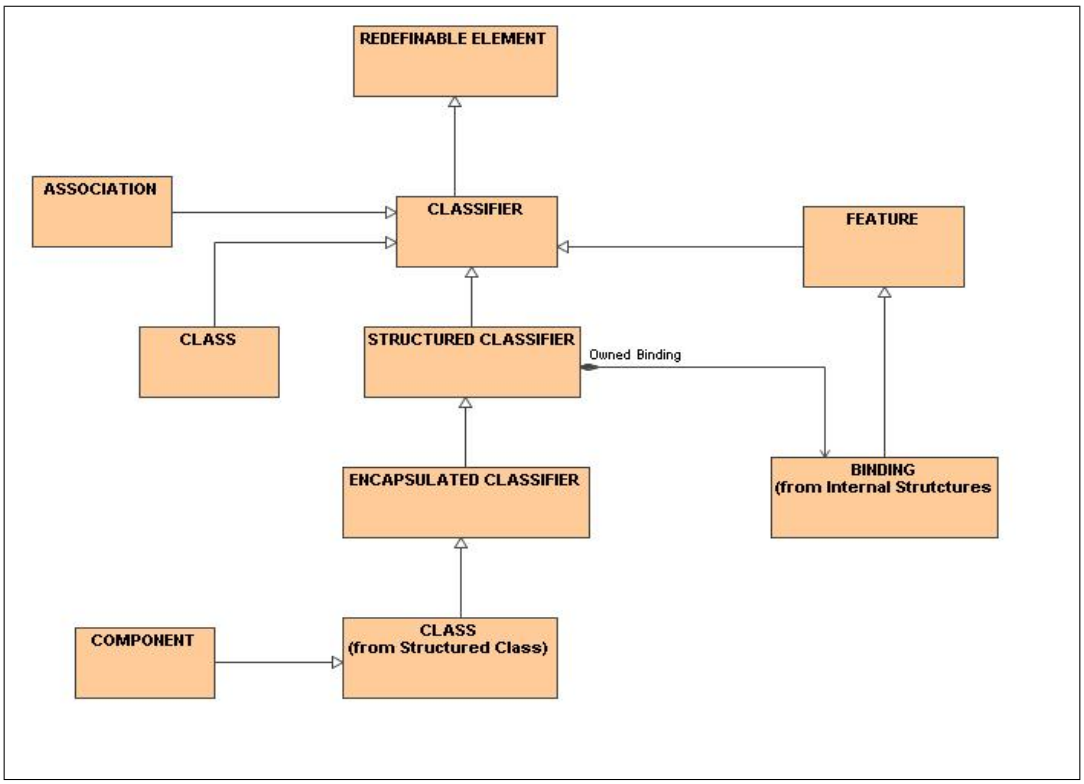


FIG. 5.6: Kernel - Alterações no componente

Modificação 2. Alteração da estrutura de STRUCTURED CLASSIFIER. A estrutura de STRUCTURED CLASSIFIER ficará bastante parecida com a anterior. A diferença é que, onde havia a meta-classe CONNECTOR, será incluída a meta-classe BINDING. Além disso, foi necessária a criação de uma nova meta-classe chamada BINDING END para substituir a meta-classe CONNECTOR END nessa estru-

tura(FIG 5.7). Essa nova meta-classe define as terminações do BINDING, que podem ser apenas no tipo porta provida.

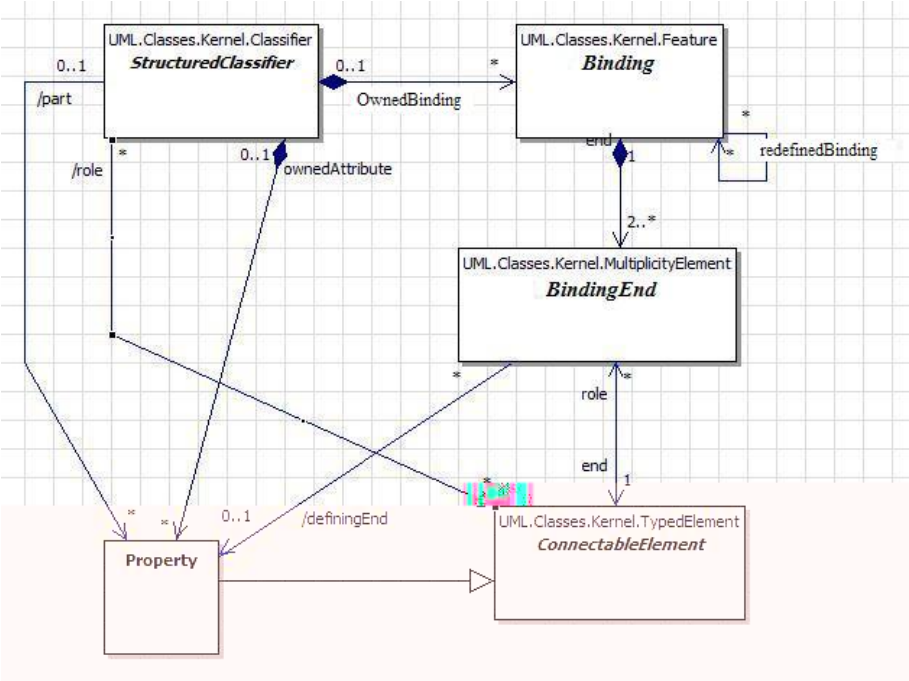


FIG. 5.7: Nova estrutura do STRUCTURED CLASSIFIER

Modificação 3. As conexões dos componentes serão feitas através de portas. Outro ponto importante acerca dos componentes é a questão da interface. Em UML, há uma pequena confusão nas definições de porta e interface. Em geral, as definições contidas no metamodelo, se referem a porta ou interface, não fazendo diferença entre as duas. Para que esse problema seja contornado, em nossa notação, as conexões serão através de porta, já que em AspectualACME os elementos arquiteturais associados a componentes são as portas e não há elemento que materialize o conceito de interface. Lembramos que a interface continua existindo em UML, apenas não estará explicitamente expressa. Pode ser que em algum momento o arquiteto tenha necessidade de expressá-la e o metamodelo proposto não restringe essa opção.

5.2.2 CONECTORES REGULARES

A primeira tarefa a ser discutida na nova definição dos conectores UML é onde colocar a meta-classe CONNECTOR, que foi excluída da definição de componentes. Para que o conector seja um elemento de primeira ordem, não pode ser definido como uma característica (FEATURE). A opção encontrada foi defini-lo como um tipo de classificador, para que possa ter uma nova estrutura, elementos internos e uma representatividade melhor.

Modificação 4. Transformação da meta-classe CONNECTOR em um classificador. A modificação 1 retirou a definição de conector da definição de componente, pois ambos são elementos de primeira ordem em AspectualACME. Logo, por serem elementos de mesmo nível na ADL, a meta-classe CONNECTOR passa a ser hierarquicamente subordinada a CLASSIFIER (como o componente) (FIG 5.8). De fato, o conector em AspectualACME tem características de um classificador, ie. é um elemento que tem seus próprios features e que pode ser especializado e redefinido. CONNECTOR, da mesma forma que CLASSIFIER, é uma meta-classe abstrata cuja semântica é conectar componentes. A forma como essa interação é concebida será determinada pelas instâncias dessa meta-classe. O próximo passo então, é definir a estrutura do conector, procurando aproveitar ao máximo (estendo o mínimo possível) os elementos do meta-modelo. Para que esteja de acordo com a definição de AspectualACME, o conector tem que estabelecer uma ligação entre, pelo menos, dois componentes e apresentar como parte de sua estrutura o conceito de papéis. Entretanto, não existe o conceito de papéis no metamodelo.

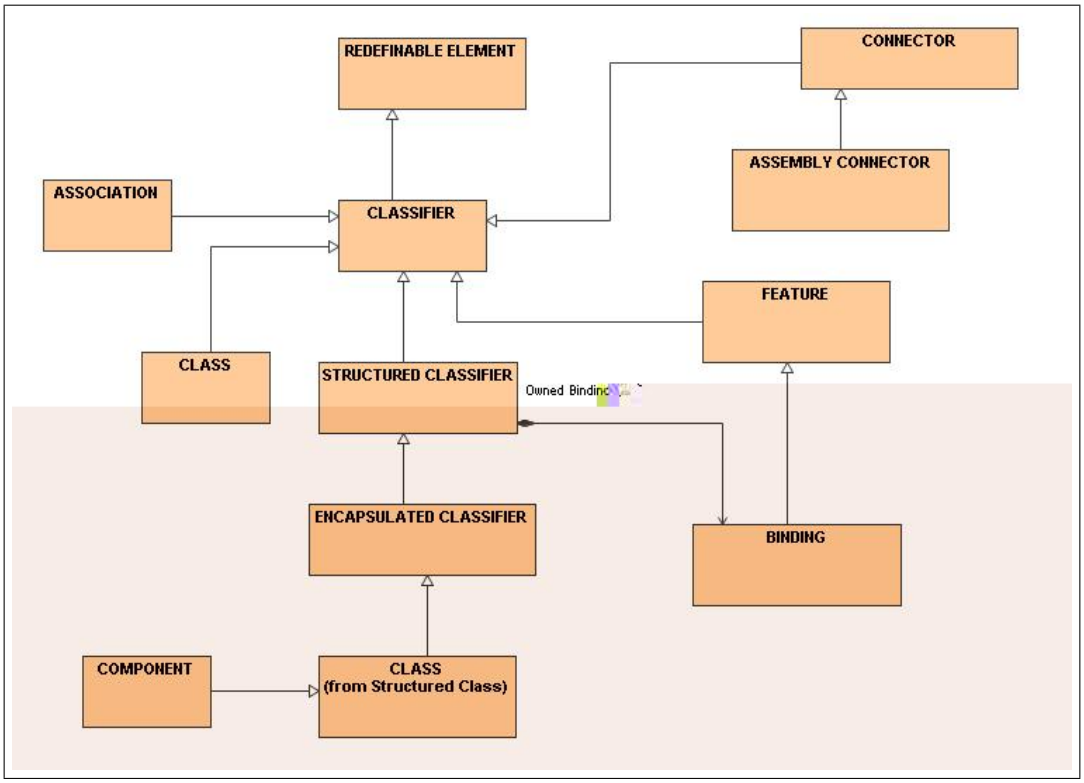


FIG. 5.8: Kernel - Conector de ligação (*Assembly*)

Modificação 5. Definição da estrutura do CONNECTOR e do ASSEMBLY CONNECTOR. Há uma analogia natural entre porta e papel. A porta é definida em AspectualACME, como a interface de um componente, assim como o papel é definido como a "interface" de um conector. Em UML, a porta é uma propriedade que faz parte

de um componente. Decidimos, portanto, criar uma nova metaclasse em UML, denominada ROLE, com a mesma hierarquia de porta, ou seja, sendo um tipo de PROPERTY. Através de uma agregação, inserimos o papel como parte da estrutura do conector de ligação.

Para representar a ligação entre componentes e conectores resolvemos reutilizar o conceito definido pela meta-classe CONNECTOR (original de UML 2.0) com o nome alterado para CONNECTION (FIG 5.11). A estrutura de CONNECTION define uma conexão como uma ligação que permite a comunicação entre duas ou mais instâncias. No entanto, queremos definir essa nova ligação entre um componente e um conector (porta e papel). Cada ligação desse tipo (CONNECTION) é composta por elementos de junção (CONNECTION END) que, por sua vez, relacionam elementos conectáveis (CONNECTABLE ELEMENTS). Utilizamos então o mesmo conceito, sendo que componentes e conectores representam os elementos conectáveis e as portas e papéis (que são do mesmo tipo - PROPERTY) representam os CONNECTION ENDs. O CONNECTION END é tão somente o antigo CONNECTOR END (original de UML 2.0) com o nome alterado.

Com esse conjunto de alterações no metamodelo, temos um novo conector de ligação, de primeira ordem, que apresenta papéis como parte de sua estrutura e que está de acordo com o conceito de conector apresentado por ACME e AspectualACME. O próximo passo é definir uma notação para o novo conector.

Modificação 6. Nova notação para o ASSEMBLY CONNECTOR. Definimos, então, que o novo conector será representado por meio de um retângulo com duas associações: uma associação sem direção partindo do componente que requer o serviço e chegando ao conector e, do outro lado, uma associação direcionada partindo do conector e chegando ao componente que provê o serviço (FIG 5.9). Os papéis são representados através de círculos preenchidos com os respectivos nomes em cima e são inseridos sobrepostos ao retângulo do conector. A representação do nome do conector, dos papéis e dos nomes dos papéis é opcional.

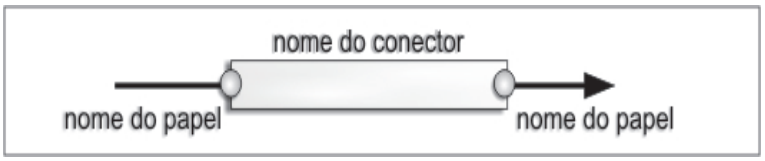


FIG. 5.9: Notação AD-AUML do conector regular.

5.2.3 CONECTOR ASPECTUAL

Para entender o que significa a inclusão do conector aspectual, fizemos uma comparação entre as características do conector regular de ACME e do conector aspectual, definido em AspectualACME, de modo a tornar explícitas suas diferenças (TAB 5.1).

Características	**Conector Regular**	**Conector Aspectual**
Interface	Papel	Papel transversal e Papel base
Protocolo de interação	Opcional (através de propriedades)	Obrigatório (cláusula Glue)
Regras de composição entre conectores	Não há	Precedence e Xor
Restrição de conexão	Não deve haver papel definido sem que apareça no campo de interconexão	(i) Não deve haver papel transversal e papel base definidos no conector que não apareçam na cláusula Glue e no campo de interconexão. (ii) Papéis transversais só podem estar ligados a portas providas.

TAB. 5.1: Comparação entre conector regular e aspectual

Da tabela acima podemos observar que o conector regular e o conector aspectual têm algumas diferenças estruturais. A primeira delas é que o conector aspectual apresenta dois tipos de papéis (papel transversal - *crosscutting role* - e papel base - *base role*) com

definidos através de um atributo *kind* que é do tipo roleKind. Para o protocolo de interação, definimos uma nova metaclasse chamada de *GLUE*, que é uma generalização de PROPERTY. A *GLUE*, por sua vez, pode ser de três tipos (after, around e before), definidos através do atributo *kind* que é do tipo glueKind. As definições das regras de composição entre os conectores estão no APÊNDICE 1.

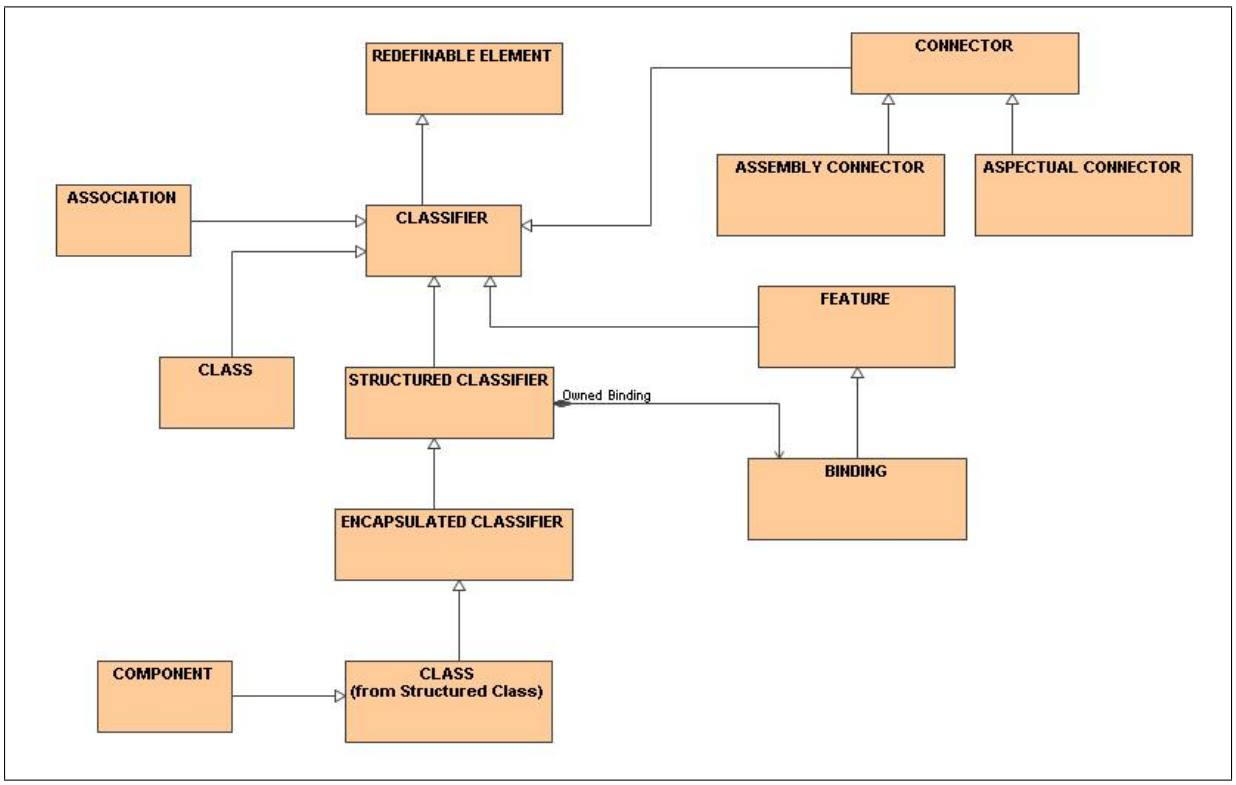


FIG. 5.10: Kernel de elementos arquiteturais proposto

Modificação 8 Nova notação para ASPECTUAL CONNECTOR. Como notação gráfica para o conector aspectual, definimos um losango com duas associações: uma associação sem direção partindo do componente que tem comportamento aspectual (irá afetar o outro componente) e chegando ao conector e, do outro lado, uma associação direcionada partindo do conector e chegando ao componente de comportamento regular (será afetado pelo componente aspectual) (FIG 5.12). Os papéis são representados por hexágonos preenchidos e são inseridos sobrepostos ao losango do conector. Note que não há, visualmente, distinção entre os tipos de papéis, já que podem ser identificados pela direção da conexão (o papel ligado ao componente de origem é o papel transversal e o papel ligado ao componente de destino é o papel base). Os nomes dos papéis são inseridos acima dos desenhos. O tipo da cláusula Glue é indicado por um nome dentro do losango. A representação do nome do conector, dos papéis, e dos nomes dos papéis é opcional.

Uma observação importante a ser feita é que, para manter a definição dos conectores

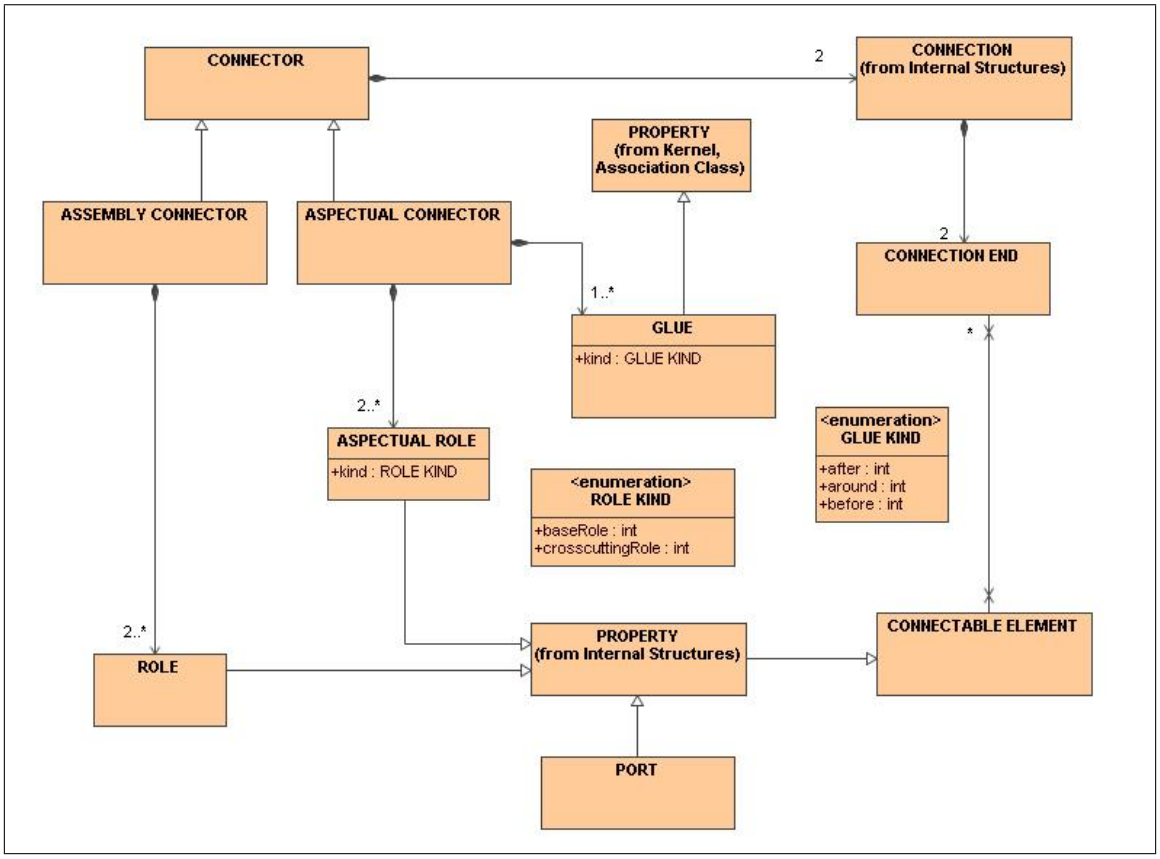


FIG. 5.11: Estrutura completa do Connection Classifier

FIG. 5.12: Notação do conector aspectual

em AspectualACME, não foi restrito no metamodelo o número máximo de papéis e de cláusulas *glue*. No entanto, por ser a UML uma linguagem visual, uma representação desse tipo (com vários papéis e várias cláusulas *glue* - N:N) tenderia a tornar o diagrama arquitetural incompreensível. Para evitar este problema de compreensão, a AD-AUML sugere que a representação visual de um conector N:N seja feita por N conectores 1:1. Isto não incorre em uma modificação semântica, mas somente em uma modificação na notação para facilitar a visualização do diagrama. O exemplo da FIG 5.13 apresenta uma descrição arquitetural em AspectualACME para o sistema ExemploConexão. Na FIG 5.14, mostramos a descrição do sistema ExemploConexão em AD-AUML.

```
System Exemplo_Conexão {
        Component A {Port A1}
    Component B {Port B1}
    Component C {Port C1}
    Aspectual Connector AB {
            Crosscutting Role source;
            Base Role target1, target2;
            Glue {source after target1,
            source before target2}
ttachments {
A.A1 to AB.source,
AB.target1 to B.B1,
AB.target2 to C.C1
}
                                    }
```

FIG. 5.13: Descrição do sistema Exemplo em AspectualACME

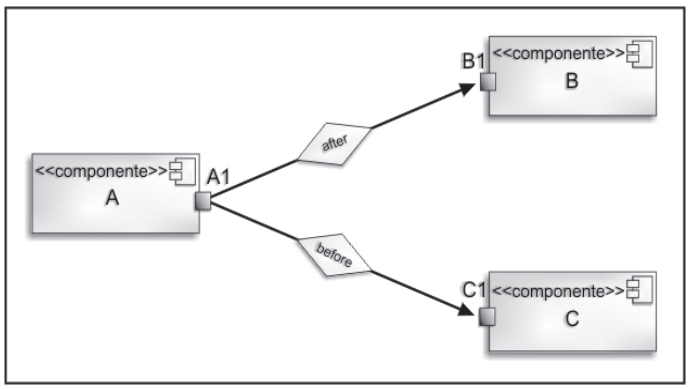


FIG. 5.14: Sistema Exemplo em AD-AUML

Podemos observar que, na descrição AD-AUML (FIG 5.14), há duas instâncias do mesmo conector, cada uma partindo de um componente e chegando a outro, caracteri-

zando uma conexão 1:1.

Modificação 9. Inclusão do relacionamento entre conectores. Em AspectualACME, pode-se definir relacionamentos entre conectores. Um relacionamento de precedência pode ser definido tendo como contexto todo o sistema ou para apenas um ponto de junção. Essa definição é necessária porque pode ser definido mais de um ponto de junção para cada conector. Como esse problema foi contornado substituindo-se uma conexão N:N em N conexões 1:1, podemos dizer que a precedência só estará presente em um ponto de junção específico (par de conectores). Além disso, cada conexão (metaclasse Connector) é representada por uma instância de associação. Podemos então definir a precedência como uma restrição, que é um elemento de extensão fornecido pela própria linguagem UML. Representa-se, portanto, com chaves, o nome PRECEDENCE, dois pontos e os nomes dos crosscutting roles envolvidos na composição (FIG 5.15).

Uma analogia pode ser feita para a exclusão mútua. Em AspectualACME ela já é definida em pares, portanto em UML, a representação seria da mesma forma: o nome XOR entre chaves (restrição). Note que nesse tipo de representação não é possível identificar quais dos dois conectores será utilizado em determinado momento. Isso será definido nas fases mais adiantadas (projeto detalhado e implementação). Essa limitação é uma herança de AspectualACME.

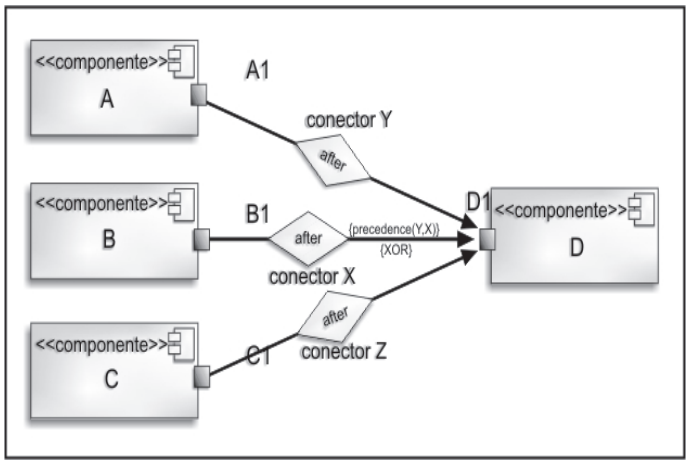


FIG. 5.15: Notação AD-AUML para *precedence* e *xor*

5.3 RASTREABILIDADE ENTRE ARQUITETURA E PROJETO DETALHADO DE SOFTWARE

Uma das maiores críticas imposta às linguagens de descrição arquitetural é a falta de mapeamento entre os modelos definidos a partir dessa linguagem e os modelos que serão

utilizados na fase de projeto detalhado de software. Para resolvermos esse problema, propomos um conjunto de recomendações para transformação entre modelos de AD-UML e modelos aSideML (CHAVEZ, 2004). A primeira subseção apresenta a linguagem aSideML, utilizada para a modelagem de interesses transversais em nível de projeto detalhado de software. Na segunda subseção será discutida a transformação propriamente dita.

5.3.1 A LINGUAGEM DE MODELAGEM ASIDEML

A linguagem de modelagem aSideML (CHAVEZ, 2004) é usada para especificar e comunicar projetos orientados a aspectos. Ela oferece semântica, notação e regras que permitem que o projetista construa modelos cujo foco sejam os principais conceitos, mecanismos e propriedades de sistemas orientados a aspectos, nos quais os aspectos sejam explicitamente tratados como elementos de primeira classe. Esses modelos ajudam a lidar com a complexidade de sistemas orientados a aspectos, ao fornecer visões essenciais da estrutura e do comportamento que enfatizam o papel dos aspectos e seus relacionamentos com outros elementos. Esses modelos também servem como planos preliminares (*blueprints*) que devem ser desenvolvidos na direção dos modelos de implementação de ferramentas e linguagens de programação orientadas a aspectos.

Os principais elementos estruturais de modelagem de aSideML são aspectos, os elementos base (de UML) que aspectos afetam e seus relacionamentos. Os aspectos são definidos como elementos parametrizados que abstraem em relação à identidade das classes que irão afetar, declarando parâmetros de templates a fim de manter os nomes reais de classes e dos métodos. O relacionamento de *crosscutting* representa um relacionamento entre um aspecto parametrizado e um elemento base; ele também realiza uma associação que define as operações e os elementos base que substituem os parâmetros de templates do aspecto. Os principais elementos comportamentais de modelagem são instâncias de aspecto, interações aspectuais e colaborações aspectuais. No contexto desse trabalho só trataremos de elementos estruturais da linguagem.

Modelagem Estrutural Em aSideML, a modelagem estrutural oferece a visão estática de um sistema na presença dos aspectos. Os principais constituintes dos modelos estruturais são as classes, aspectos e seus relacionamentos. A visão estática descreve os interesses transversais de aspectos como elementos de modelagem discretos, organizados em interfaces transversais.

- Aspectos

Um aspecto em aSideML é um elemento de primeira ordem que encapsula um conjunto de interesses transversais organizadas em interfaces transversais (FIG 5.16). Os aspectos podem afetar as classes de forma heterogênea, ou seja, duas ou mais classes diferentes podem ser afetadas por diferentes subconjuntos de interesses transversais localizadas dentro do mesmo aspecto.

Os aspectos também podem ter impactos variados sobre diferentes tipos de classes. Portanto, um aspecto é definido como um elemento parametrizado que prove abstração em relação à identidade das classes que irá afetar, declarando parâmetros formais a fim de conter os nomes reais de classes e métodos.

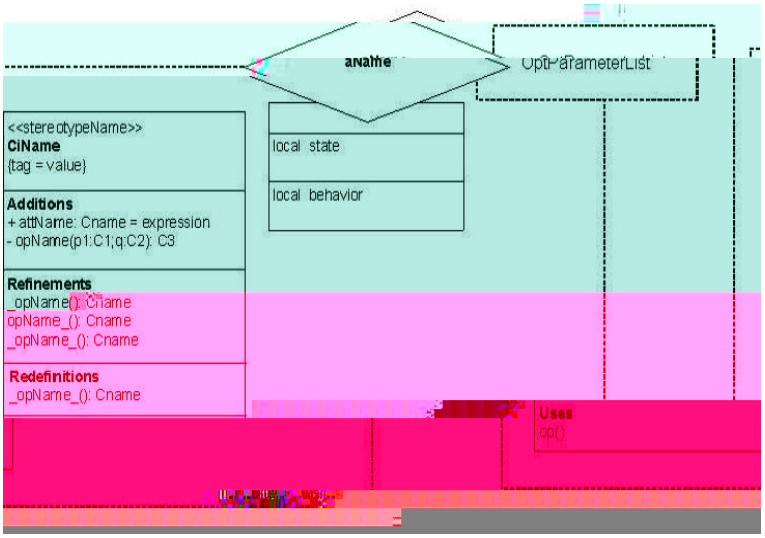


FIG. 5.16: Declaração de aspecto completa. (CHAVEZ, 2004)

Um aspecto é representado como um retângulo tracejado, com um losango contendo o nome do aspecto, que toca um retângulo usado para descrever suas características locais (atributos e métodos). Um aspecto pode conter vários elementos internos, como interfaces transversais, classes, interfaces e relacionamentos.

Um aspecto é apresentado com um retângulo tracejado pequeno, sobreposto no canto superior direito do retângulo do aspecto. Esse retângulo é chamado de caixa de parâmetros de templates e contém listas de parâmetros formais, sendo uma lista para cada interface transversal de aspecto. O primeiro parâmetro de cada lista é o nome da interface transversal correspondente.

Como opção, pode-se utilizar a visão condensada de um aspecto, que omite todas as informações sobre seus elementos internos, exceto os nomes das interfaces transversais; cada interface transversal é exibida como um pequeno círculo com seu nome colocado ao lado (FIG 5.17). O círculo esta ligado por uma linha sólida ao losango que representa o

aspecto.

Se o aspecto é parametrizado e for usada uma visão condensada, a(s) lista(s) de parâmetros formais deve(m) ser explicitamente exposta(s).

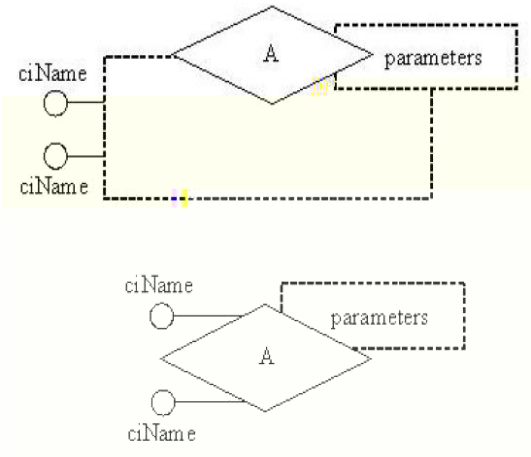


FIG. 5.17: Declaração de aspecto condensada. (CHAVEZ, 2004)

- Interfaces Transversais

As interfaces transversais são conjuntos de características transversais com nome associado, que caracterizam o comportamento crosscutting de aspectos. Elas são declaradas dentro de aspectos.

Uma interface transversal declara: (i) um conjunto de características comportamentais e estruturais que o aspecto adiciona a uma classe base; (ii) um conjunto de características transversais comportamentais que refinam ou redefinem algumas características comportamentais existentes na classe base. O pronome base é usado para denotar um objeto base; (iii) um conjunto de assinaturas para características requisitadas (*required features*) usadas pelo aspecto.

As interfaces transversais podem participar de relacionamentos com outros elementos que pertencem ao espaço de nomes do aspecto, incluindo outras interfaces transversais.

Uma interface transversal é representada por um retângulo com linhas sólidas com compartimentos separados por linhas horizontais. O nome que aparece no topo do compartimento representa o nome da interface transversal. O segundo compartimento contém as características transversais que oferecem suporte a introduções (FOUNDATION, 2006) (*additions*), e o terceiro e quarto compartimentos contêm as características que oferecem suporte a *advice* (FOUNDATION, 2006) (*refinements* e *redefinitions*). Um compartimento opcional pode ser fornecido para definir *placeholders* para operações requisitadas (*uses*).

Os nomes de algumas características transversais comportamentais (*advices*) são decorados com o símbolo _. A FIG 5.18 mostra uma interface transversal com adições, refinamentos, redefinições e usos. Como opção, adornos textuais - *before, after, around* e *use* - podem substituir o símbolo _. Por exemplo, podemos escrever *before op* no lugar de _op.

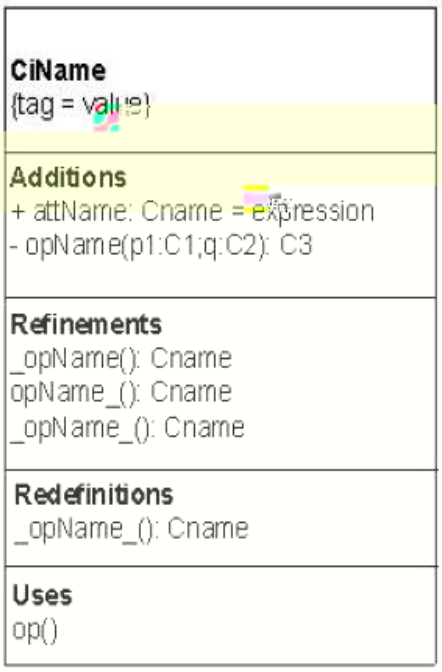


FIG. 5.18: Interface transversal com adições, refinamentos, redefinições e usos. (CHAVEZ, 2004)

- Relacionamento de *crosscutting*

Crosscutting é um relacionamento entre um elemento transversal (por exemplo, um aspecto) e um elemento base. Ele especifica que o aspecto deve atravessar os limites do elemento base em pontos de combinação bem definidos e modificar incrementalmente a base nesses pontos.

Em aSideML, o relacionamento de *crosscutting* classifica um relacionamento entre um aspecto parametrizado e um elemento base; ele também realiza uma associação que define as operações e os elementos base que substituem os parâmetros de templates do aspecto.

Os parâmetros de templates são agrupados por interface transversal. Para cada interface transversal especificada no aspecto, um conjunto de casamento de templates (*template matches*) que definem substituições para essa interface transversal são colocados entre *brackets* <>. A restrição *xor* pode ser aplicada a um conjunto de relacionamentos

crosscutting que compartilham uma conexão a um elemento base, especificando que, nesse conjunto, exatamente um aspecto afetará tal elemento base.

Notação

Um relacionamento de *crosscutting* é representado como uma seta tracejada com a parte final no elemento transversal e a ponta no elemento base, e a palavra-chave «crosscut». Pode incluir uma lista de associações. A FIG 5.19 apresenta a representação gráfica do relacionamento *crosscutting* .

Cada associação relaciona um parâmetro de template definido na interface transversal do aspecto (nome da interface ou nome da operação) a um nome (ou seqüência de nomes) definido na interface de uma classe (nome de classe ou um nome de método). Essa informação transversal é exibida como uma lista de correspondências de parâmetros de *templates* separadas por vírgulas, <templateMatch*>.

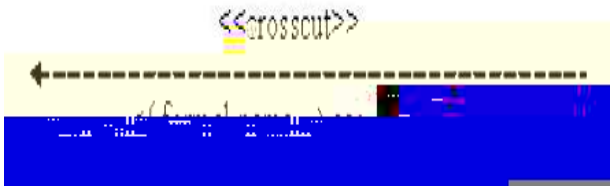


FIG. 5.19: Relacionamento de *crosscutting*. (CHAVEZ, 2004)

- Precedência e xor

- Componentes Regulares

O primeiro elemento a ser analisado é o componente regular. Um componente na arquitetura é basicamente representado por um conjunto de portas que interagem com outros componentes. Cada porta tem um tipo (provida e requerida) que define a forma como o componente irá participar da interação. Em projeto detalhado de software, há o conceito de classe, cada uma com seus atributos e métodos. A interface de uma classe é definida pelos métodos públicos que implementa.

Para que possamos garantir que cada conexão entre componentes tenha a sua representatividade em projeto detalhado, optamos por transformar cada porta de componente com comportamento regular, em uma classe ou uma interface a ser implementada por uma classe, já que o que nos interessa é a mensagem que está sendo transmitida. Não há, ainda, que se diferenciar o tipo de porta, já que o relacionamento entre classes se dá de uma forma unificada.

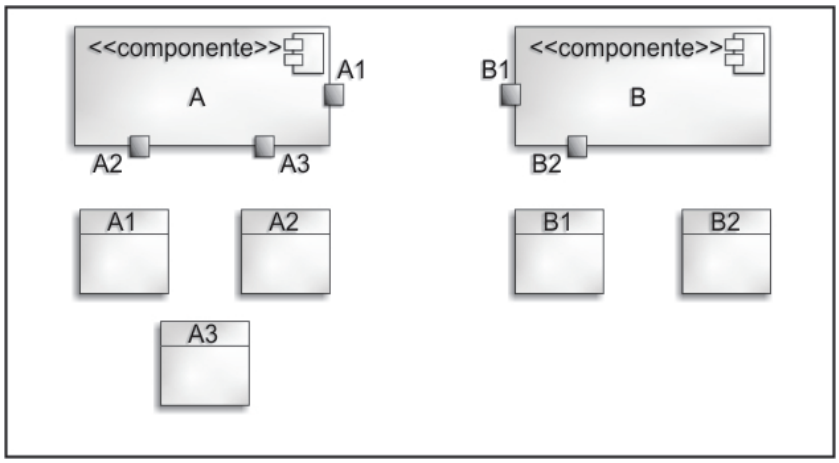


FIG. 5.20: Mapeamento de componentes regulares

- Componentes Aspectuais

O próximo elemento a ser analisado é o componente com comportamento transversal. (lembramos que AD-AUML não faz distinção entre componentes, portanto o nome aspectual apenas é utilizado para distinguir os dois comportamentos). Esse componente também apresenta um conjunto de portas que definem os serviços que são prestados pelo componente. As portas que representam a interação transversal são sempre do tipo provida. Cada componente que tenha, pelo menos uma porta com comportamento transversal, na descrição da arquitetura, será representado por um aspecto, cujo nome é o nome do componente. Cada porta do componente aspectual dará origem a uma interface transversal

do aspecto, que terá o nome da porta. Nesse contexto, o conceito de interface transversal de aSideML é bastante útil.

O componente aspectual pode, ainda, apresentar um conjunto de portas com comportamento regular. Para mapear essas estruturas, é utilizada a regra dos componentes regulares.

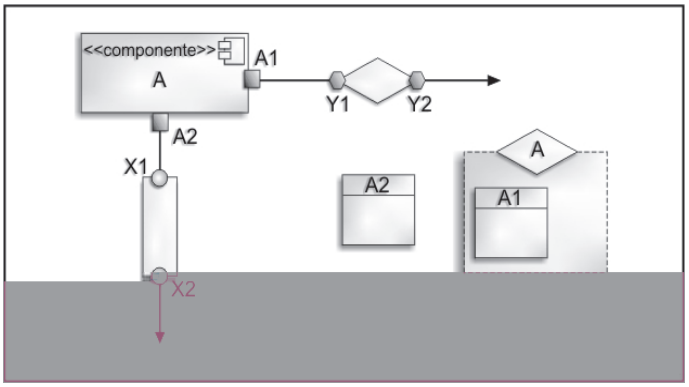


FIG. 5.21: Mapeamento de componentes aspectuais

- Conector regular

Cada instância de conector regular em arquitetura de software dará origem a uma associação em projeto detalhado. A porta de origem (requerida) será a classe de origem da associação e a porta de destino (provida) será a classe de destino. Os papéis definidos no conector darão origem aos nomes dos comportamentos das classes envolvidas. A multiplicidade da associação será definida pela semântica da associação e o nome da associação pode ou não (opcional) ter o nome do conector regular.

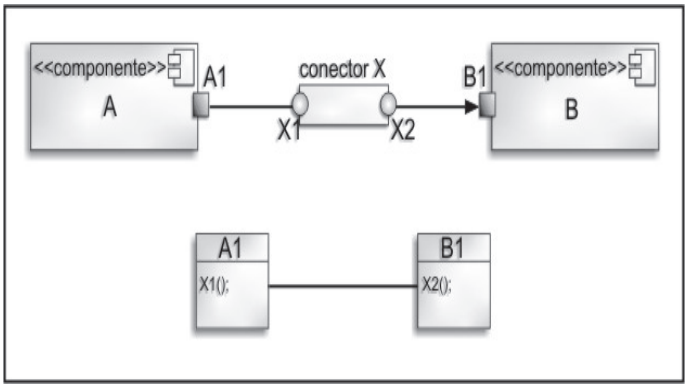


FIG. 5.22: Mapeamento de conector regular

- Conector aspectual

Cada instância de conector aspectual, em arquitetura de software, será mapeada para um relacionamento de *croscutting* em projeto detalhado de sofware. Os papéis transversais do conector aspectual ligados a um componente darão origem às características comportamentais descritas na interface (compartimento refinements) do aspecto que representa a porta do componente que o conector está ligado e a cláusula *glue* do conector aspectual dará origem ao adorno que é utilizado nos nomes dos comportamentos descritos na interface transversal. Os papéis base do conector aspectual deverão ser mapeados para o comportamento das classes que representam a porta do componente que é afetado pelo conector aspectual.

Dessa forma, a leitura da conexão (*crosscuting role - glue - base role*) que é possível ser feita na descrição arquitetural é transformada em um relacionamento de *crosscutting* descrito em aSideML. O casamento de templates (*template matches*) é definido pelo comportamento do aspecto correspondente ao papel transversal da conexão, juntamente com o comportamento da classe correspondente ao papel base (<comportamento aspecto —>comportamento da classe>).

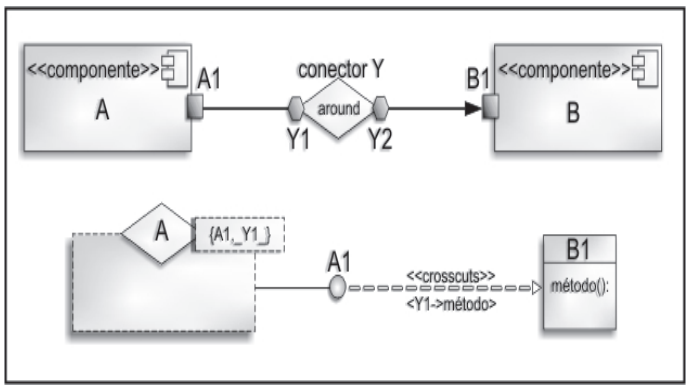


FIG. 5.23: Mapeamento de conector aspectual

- Precedência e Exclusão

Por último, as restrições de precedência definidas na arquitetura serão transformadas no relacionamento «precede» definido entre aspectos. A exclusão mútua continuará sendo uma restrição(xor) aplicada no par de relacionamento de crosscutting que compartilham uma conexão a um elemento base. A FIG 5.24 mostra um exemplo de *precendence* e *xor* em AD-AUML e a FIG 5.25 mostra o exemplo mapeado para aSideML.

A TAB 5.2 apresenta um resumo das transformações entre AD-AUML e aSideML propostas:

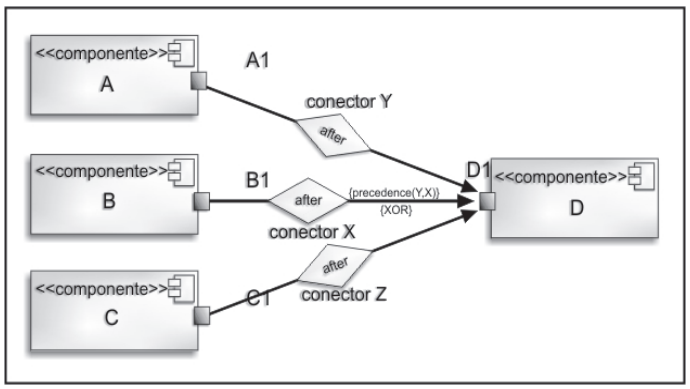


FIG. 5.24: Exemplo de *precedence* e *xor* em AD-AUML

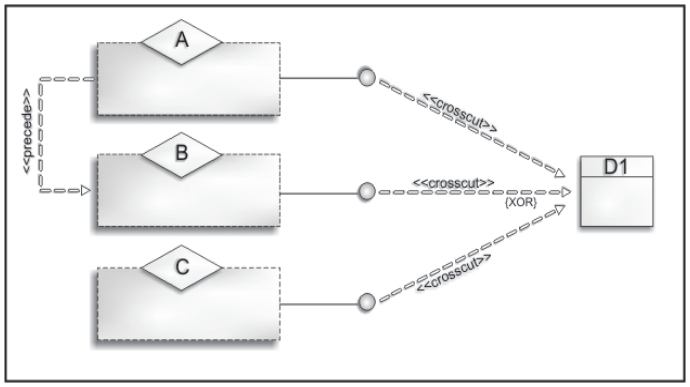


FIG. 5.25: Mapeamento de *precedence* e *xor*

AD-AUML	**aSideML**
Componente Regular (CR)	——
Componente Aspectual (CA)	Aspecto
Porta do CR	Classe
Porta do CA	Interface transversal
Papéis	Comportamento das classes
Papéis Transversais	Características transversais comportamentais
Papéis Base	——
Cláusula Glue	Adorno nas características transversais comportamentais
Conector Regular	Relacionamento entre classes
Conector Aspectual	Relacionamento de *crosscutting*
Precedence	Relacionamento «precede» entre aspectos
Xor	Restrição entre relacionamentos de *crosscutting*

TAB. 5.2: Resumo do mapeamento entre AD-AUML e aSideML

6 ESTUDOS DE CASO

Nesse capítulo serão apresentados os conceitos discutidos no capítulo 5, no contexto da arquitetura de dois sistemas, Mobigrid (seção 6.1) e Health Watcher (seção 6.2). Em cada seção, serão apresentadas e discutidas as arquiteturas dos referidos sistemas e a inclusão de relações aspectuais nessas arquiteturas. Depois disso, será apresentada a representação das arquiteturas utilizando a linguagem de modelagem AD-AUML. Por último, será discutida a transformação da arquitetura para projeto detalhado, utilizando a linguagem AsideML.

6.1 MOBIGRID

6.1.1 DESCRIÇÃO DO SISTEMA

O primeiro estudo de caso é baseado no framework MobiGrid (BARBOSA, 2004). Nesse sistema, agentes móveis são utilizados para encapsular e executar longas tarefas de processamento, utilizando ciclos ociosos de uma rede de computadores pessoais. Os agentes podem migrar de uma máquina para outra no momento que uma máquina local é requisitada por seu usuário, já que têm capacidade de migração automática. A arquitetura parcial do sistema é dividida basicamente em quatro componentes:

a) MOBIGRID - modulariza as características básicas de uma aplicação baseada em agentes;

b) MOBILITY PROTOCOL - modulariza a execução do protocolo de mobilidade, ou seja, a instanciação, migração, inicialização remota e destruição dos agentes;

c) MOBILITY MANAGEMENT - fornece uma integração flexível entre o MobiGrid e as diferentes plataformas de mobilidade;

d) MOBILITY PLATAFORM - representa uma plataforma de mobilidade específica que está sendo utilizada.

O objetivo principal do componente MOBILITY PROTOCOL é fornecer a separação explícita entre a característica de mobilidade e o componente Mobigrid. Já o componente MOBILITY MANAGEMENT conecta o MobiGrid com o componente MOBILITY

PLATAFORM, que modulariza e externaliza os serviços da plataforma de mobilidade escolhida. SANTANNA (2006), em seu trabalho, propõe uma adaptação da arquitetura parcial do MobiGrid para o conceito de orientação a aspectos. Em outras palavras, foi invertida a forma de acesso aos serviços de mobilidade que tipicamente estão presentes em sistemas de agentes móveis. Na arquitetura orientada a aspectos, o componente MOBILITY PROTOCOL intercepta as chamadas à interface iAplicationAgent, introduzindo a característica de mobilidade. O componente MOBILITY MANAGEMENT intercepta o componente MOBILITY PLATAFORM através de sua interface iReferenceObserver, para manter a consistência entre o tempo de execução da plataforma e o Mobigrid. Isso é possível porque a interface iPlataformEvents na qual a iReferenceObserver atua, também é afetada pelos eventos do componente MOBILITY PROTOCOL responsáveis pela instanciação, partida, chegada e destruição dos agentes. Na FIG 6.1, temos uma descrição em AspectualACME da arquitetura parcial orientada a aspectos do sistema MobiGrid.

6.1.2 REPRESENTAÇÃO DA ARQUITETURA DO MOBIGRID UTILIZANDO AD-AUML

Com base na descrição arquitetural em AspectualACME (FIG 6.1), foi possível a representação da arquitetura do sistema MobiGrid utilizando a nova linguagem de descrição arquitetural AD-AUML.

Na FIG 6.2, podemos observar que há um conector regular e três conectores aspectuais. Nesse estudo de caso, utilizamos a representação completa do sistema. Podemos observar que o conector regular está com os seus papéis e os conectores aspectuais apresentam os seus papéis transversais, papéis base e a cláusula glue. O componente MobilityProtocol utiliza os serviços do componente MobilityManagement (representado pelo conector regular) e corta os outros dois componentes (MobiGrid e MobilityPlataform), utilizando os conectores aspectuais. Já o componente MobilityManagement corta o componente MobilityPlataform (representado pelo conector aspectual).

6.1.3 TRANSFORMAÇÃO DO MOBIGRID DE AD-AUML PARA ASIDEML

A FIG 6.3 representa a transformação do sistema MobiGrid para aSideML, utilizando as regras descritas no CAP 5. Podemos observar que a interação regular é transformada para uma associação de duas classes (iReferenceManagement e iMobileAgentProtocol). As interações aspectuais dão origem a dois aspectos (Mobility Management e Mobility Protocol). O aspecto Mobility Manegement tem como interface transversal iReferenceObserver

```
System MobiGrid {
    Component CompMobiGrid = {
        Port iAplicationAgent;
    }
    Component MobilityProtocol = {
        Port iMobileElement;
        Port iMobileAgentProtocol;
    }
    Component MobilityManagement = {
        Port iReferenceMobilityAgent;
        Port iReferenceObserver;
    }
    Component MobilityPlataform = {
        Port iPlataformEvents;
    }

    Connector C1 = {Roles {mobility, protocol}}

    AspectualConnector AC1 = {
        Crosscutting Role insertMobility;
        Base Role mob;
        Glue insertMobility around mob;
    }
    AspectualConnector AC2 = {
        Crosscutting Role {insertEvents};
        Base Role {even1};
        Glue {insertEvents around even1}
    }
    AspectualConnector AC3 = {
        Crosscutting Role {observer};
        Base Role {even2};
        Glue {observer around even2}
    }
```

FIG. 6.1: Descrição do MobiGrid em AspectualACME

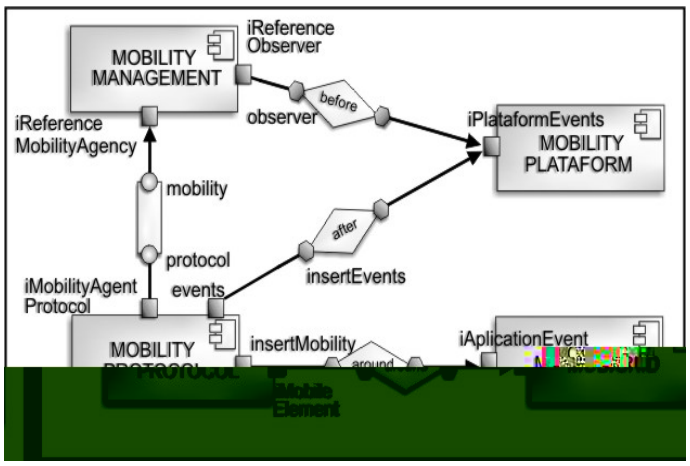


FIG. 6.2: Modelagem do MobiGrid em AD-AUML

e como característica transversal observer. Podemos observar que a característica tem o adorno before. A classe iPlataformEvents é cortada pelo aspecto Mobility Management, no entanto, por ser uma representação parcial do sistema, não é possível determinar qual o método que será cortado (esse nome surgiria de uma chamada à interface que está sendo cortada). Cabe lembrar que não faz sentido a inserção de um comportamento aspectual sem uma chamada ao método afetado. As demais interações podem ser compreendidas estendendo o que foi discutido para o restante dos conectores.

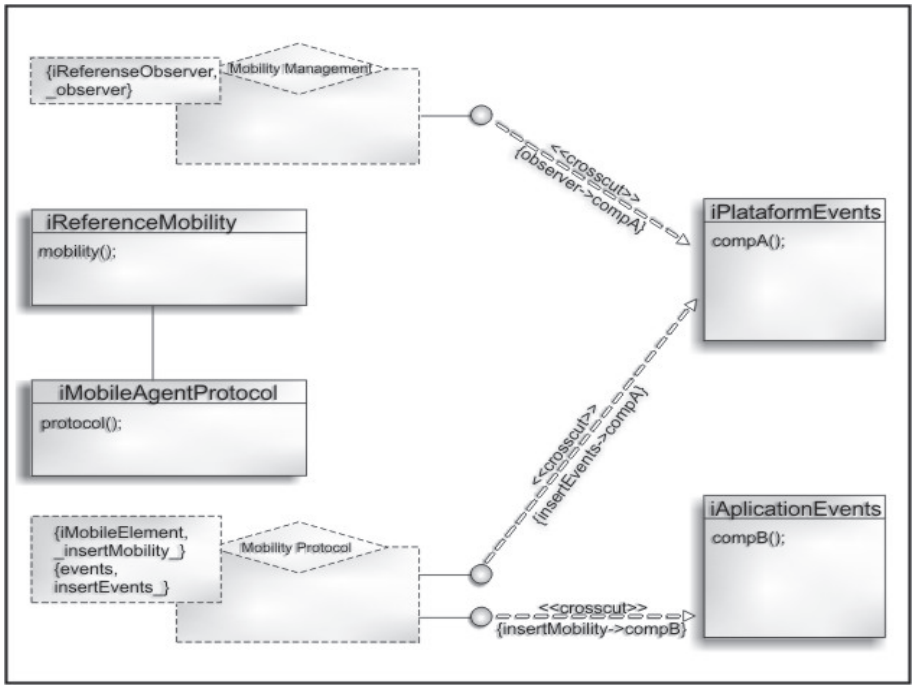


FIG. 6.3: Modelagem do MobiGrid em aSideML

6.2 HEALTH WATCHER

6.2.1 DESCRIÇÃO DO SISTEMA

O segundo estudo de caso é um sistema desenvolvido para a Web, pelo grupo de pesquisa em produtividade de software da Universidade Federal de Pernambuco (UFPE), chamado Health Watcher (SOARES, 2002). O principal objetivo do sistema é cadastrar as reclamações de usuários ao sistema público de saúde. O sistema permite que sejam feitos diversos tipos de reclamação como, por exemplo, reclamações contra restaurantes e fast-foods, permitindo que as autoridades de saúde tomem providências. A arquitetura do sistema utiliza o padrão arquitetural MVC, bastante conhecido e utilizado para sistemas desenvolvidos para web. A arquitetura do sistema é dividida basicamente em quatro

componentes:

a) VIEW - é responsável pela interface com o usuário e pela solicitação dos serviços aos servlets. É composto, basicamente, pelas páginas JSP.

b) CONTROL - nessa camada estão presentes todos os servlets utilizados pelo sistema. Os servlets fazem a comunicação da camada de apresentação com a camada de negócio;

c) BUSINESS - a camada de negócio reúne todas as classes de negócio. Na implementação utiliza o padrão de projeto FACADE. Essa camada reúne ainda as classes responsáveis pela distribuição do sistema, utilizando RMI. A distribuição é uma funcionalidade opcional e deve ser selecionada em um arquivo de configuração;

d) PERSISTENCE - a última camada é responsável pelo armazenamento dos dados do sistema. Estão presentes nesse componente ainda, o controle de transação e de concorrência. No HW a persistência pode ser implementada de duas maneiras: (i) utilizando um banco de dados relacional (RDBMS); ou, (ii) armazenando os dados em memória, utilizando arrays. A forma de persistência que será utilizada também é definida em arquivo de configuração.

A implementação orientada a objetos do Health Watcher modulariza grande parte das responsabilidades do sistema, no entanto, duas dessas responsabilidades chamam a atenção por encontrarem-se espalhadas e entrelaçadas na implementação dos componentes do sistema: a distribuição e o controle de erros. A distribuição do sistema é uma funcionalidade opcional. Já o controle de erros é encontrado por todo o sistema, normalmente nas chamadas entre componentes.

Para resolver esses problemas, uma nova implementação para o Health Watcher foi proposta (SOARES, 2002). Nessa implementação foram utilizados os conceitos de orientação a aspectos e a linguagem AspectJ (FOUNDATION, 2006). Na arquitetura do sistema, podemos perceber a separação das características de distribuição e controle de erros e o comportamento transversal dessas características com relação aos outros componentes.

Aos novos componentes foram dados os nomes de Distribution e Error Handling. Uma descrição da arquitetura orientada a aspectos do Health Watcher pode ser observada na FIG 6.4.

```
System HealthWatcher {
    Component View = {
        Port jspRequest;
    }
    Component Control = {
        Port servletFacade;
        Port servletLogin;
        Port servletRequest;
    }
    Component Business = {
        Port iFacade;
        Port hwFacade;
        Port initPersistence;
        Port requestData;
    }
    Component Persistence = {
        Port iPersistenceMechanism;
        Port iDataRepository;
    }
    Component Error Handling = {
        Port termination;
        Port retrial;
    }
    Component Distribution = {
        Port iRemoteDistribution;
    }

    Connector C1 = {Roles {jspCall, servletResponse}}
    Connector C2 = {Roles {facadeRequest, facadeResponse}}
    Connector C3 = {Roles {servletRequestCall, hwResponse}}
    Connector C4 = {Roles {persistenceRequest, mechanismResponse}}
    Connector C5 = {Roles {dataCall, dataResponse}}

    AspectualConnector AC1 = {
        Crosscutting Role rmiSourceAdapter;
        Base Role facadeStarts;
        Glue rmiSourceAdapter before facadeStarts;
    }
```

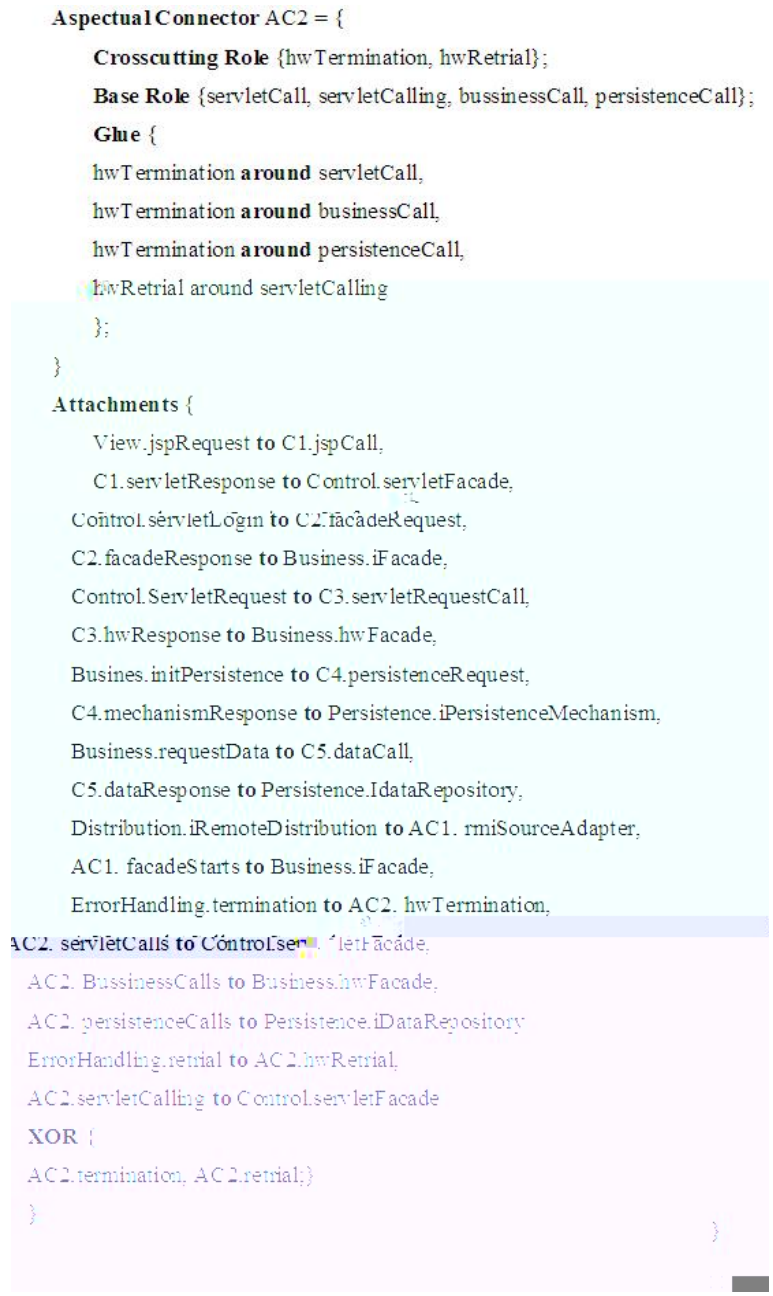


FIG. 6.4: Descrição do Health Watcher em AspectualACME

6.2.2 REPRESENTAÇÃO DA ARQUITETURA DO HW UTILIZANDO AD-AUML

Com base na descrição arquitetural em AspectualACME (FIG 6.4), foi possível a representação da arquitetura do sistema Health Watcher utilizando a nova linguagem de descrição arquitetural AD-AUML.

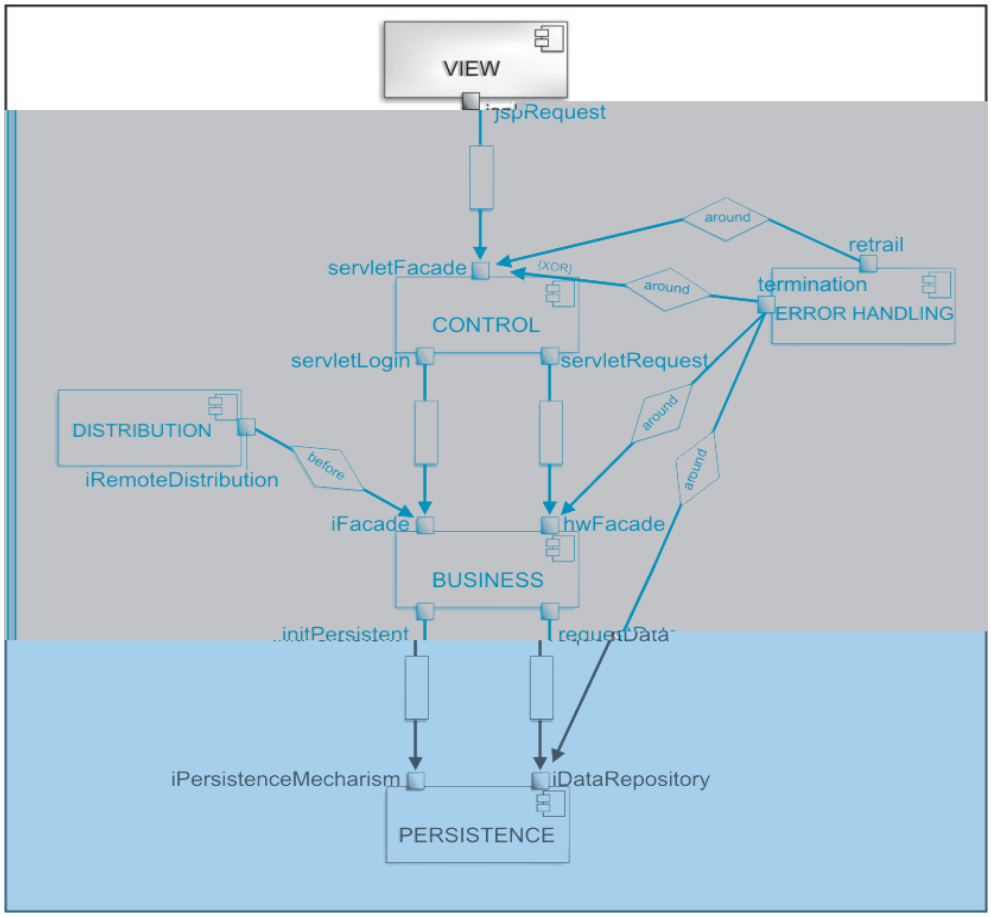


FIG. 6.5: Arquitetura do Health Watcher em AD-AUML

Na FIG 6.5, podemos observar que as chamadas regulares entre os componentes estão representadas pelo conector regular e que a interações aspectuais estão representadas por conectores aspectuais. Na figura foram omitidos os nomes dos papéis dos conectores para facilitar o entendimento. Para exemplificar, comentaremos uma conexão de cada tipo (regular e aspectual). Como exemplo de interação regular, utilizaremos a conexão entre o componente VIEW e o componente CONTROL. Podemos observar que há uma chamada através da porta requerida jspRequest do componente VIEW (o tipo da porta é determinado pela direção do conector) a um ou mais serviços providos pela porta servletFacade do componente CONTROL. Como exemplo de interação aspectual, utilizaremos a conexão entre o componente ERROR HANDLING e o componente CONTROL. Podemos observar que há uma inserção de comportamento na chamada que o componente VIEW

faz (através da porta jspRequest) aos serviços prestados pelo componente CONTROL (através da porta servletFacade). O componente ERROR HANDLING fará uma inserção ou modificação de um comportamento durante o andamento da chamada (determinado pelo tipo da cláusula glue - around - expressa no conector aspectual). Note que apenas uma das inserções será executada, determinada pela restrição XOR. Essas semânticas envolvendo os componentes exemplificados podem ser estendidas para toda a arquitetura, fornecendo um completo entendimento.

6.2.3 TRANSFORMAÇÃO DO HW DE AD-AUML PARA ASIDEML

Para apresentarmos a transformação da arquitetura aspectual do Health Watcher, utilizaremos apenas três componentes: VIEW, CONTROL e ERROR HANDLING. Na FIG 6.6, pode-se observar a parte da arquitetura considerada e que foram inseridos detalhes dos papéis necessários à transformação (anteriormente omitidos). Aplicaremos as regras de transformação descritas no cap 05 para as conexões entre VIEW e CONTROL e entre ERROR HANDLING e CONTROL. Essa transformação pode ser estendida para todos os componentes pertencentes a arquitetura.

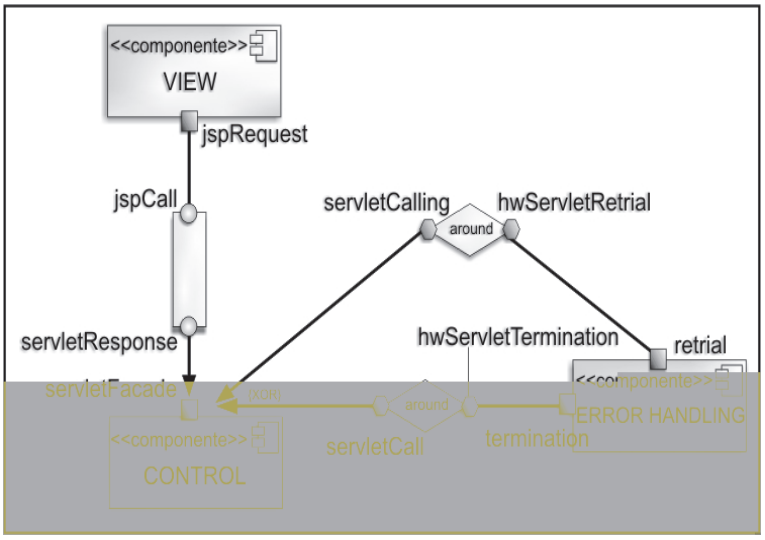


FIG. 6.6: Parte da arquitetura do HW considerada para a transformação

Na FIG 6.7 podemos observar o resultado da transformação. As portas envolvidas na conexão regular foram transformadas para classes (JSP REQUEST e SERVLET FACADE). Os papéis dos conectores regulares foram transformados em procedimentos (jspCall() e servletResponse()). Por fim o conector foi transformado em uma associação entre as duas classes. Já o componente responsável pela interação aspectual, foi transformado em um aspecto (Error Handling) e sua porta em interface transversal (handlerSearch).

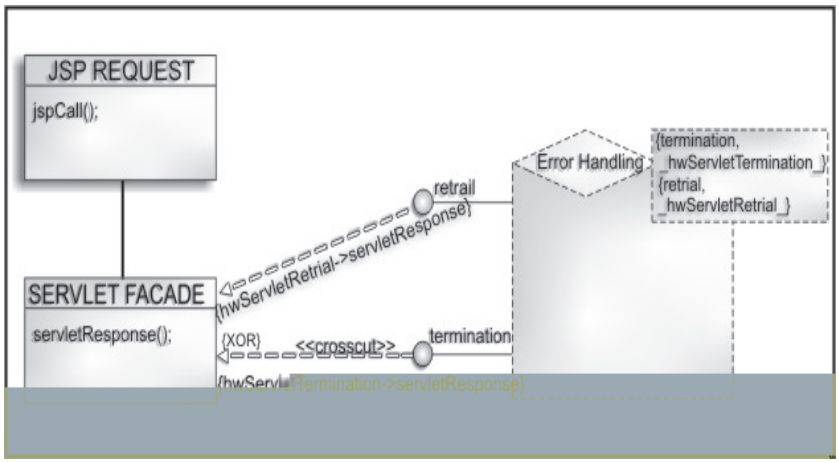


FIG. 6.7: Arquitetura parcial do HW em aSideML

O papel transversal e o papel base foram mapeados para o parâmetro de template (<hwServletExHand —>servletResponse>). Por último, a cláusula glue foi transformada para o adorno, observado nos parâmetros do aspecto (_hwServletExHand_).

7 CONCLUSÕES E TRABALHOS FUTUROS

A construção de sistemas de software de boa qualidade é o objetivo que tem guiado os esforços dos engenheiros de software durante as últimas décadas. A arquitetura de software tem ganho um espaço importante no processo de desenvolvimento uma vez que permite aos desenvolvedores enxergar as partes decompostas do sistema e a forma como essas partes interagem.

Atributos de qualidade importantes, como manutenibilidade e reusabilidade, podem ser afetados negativamente pela modularização inadequada durante a definição da arquitetura. Sendo assim, à medida que o desenvolvimento de software orientado a aspectos ganha mais popularidade, torna-se necessário a realização de pesquisas significativas no sentido de permitir a correta modularização e representação de interesses transversais no nível de arquitetura.

Neste sentido, este trabalho apresentou uma nova linguagem para modelagem de arquitetura de software (AD-AUML), baseada na UML 2.0, que contempla a representação de interesses transversais. Foram propostas alterações no metamodelo da UML, principalmente nos conectores de ligação e de delegação. O conector de ligação deixou de ser uma característica do componente e passou a ser um elemento de primeira ordem, tendo seus elementos internos e sua representação. Já o conector de delegação, foi separado do conector de ligação e passou a ter uma metaclasse que o representa. Além disso, foram propostas a criação de algumas novas metaclasses para introduzir o conceito de conector aspectual e sua estrutura interna.

Além da nova linguagem, foi proposto um conjunto de regras de transformação de modelos arquiteturais gerados com a nova linguagem para modelos de projeto de software, utilizando a linguagem aSideML. Esse conjunto de regras garante a rastreabilidade entre a fase de definição da arquitetura e a fase do projeto detalhado de software.

Cabe ressaltar que o trabalho baseou-se apenas na visão arquitetural de execução (Seção 2.1) e na modelagem estrutural da arquitetura. A modelagem de outras visões arquiteturais e a necessidade de eventuais extensões do metamodelo de UML associadas não são contempladas neste trabalho.

Até agora não foi desenvolvida uma ferramenta para apoiar o uso da notação proposta. Mas, pelo fato da notação proposta ser uma extensão de UML, é possível contar com o apoio inicial de alguma ferramenta já existente e aperfeiçoá-la para representar a parte

orientada a aspectos de uma arquitetura. Seria interessante ter uma integração desta ferramenta com outra que conseguisse representar o projeto orientado a aspectos do sistema, permitindo a aplicação das regras de transformação.

A realização dos estudos de caso serviu como uma primeira avaliação da utilidade da notação e do modelo de transformação. Em ambos os estudos, o processo de desenvolvimento foi apoiado pelo trabalho proposto e gerou resultados que ajudaram muito para o entendimento das diferenças que a orientação a aspectos introduz no desenvolvimento da arquitetura de um sistema. Mesmo utilizando o sistema Health Watcher (SOARES, 2002), que é o sistema atualmente utilizado pelo grupo AOSD-Europe (GROUP, 2006) como *testbed*, os resultados dos dois estudos não podem ser generalizados, mas indicam que a notação proposta gerou soluções que tendem a ter melhor modularização no nível arquitetural. Pode-se concluir, então, que os estudos serviram como indicação de que a extensão proposta merece ser aplicada em outros contextos de forma a poder ser refinada e amadurecida.

7.1 LISTA DE CONTRIBUIÇÕES

As contribuições inicialmente listadas como metas para o presente trabalho foram concretizadas da seguinte forma:

a) **Estender a UML 2.0 a fim alinhar suas definições às das linguagens de descrição arquitetural, melhorando sua representatividade e dando suporte aos elementos mínimos requeridos para descrição em ADLs (MEDVIDOVIC, 2000)**

   O framework de MEDVIDOVIC (2000) define componente, conector e configuração arquitetural como os elementos básicos para a modelagem de uma arquitetura. A definição de componente na UML é bastante parecida com o que encontramos nas ADLs. A configuração arquitetural é definida pelos próprios diagramas UML. Havia uma deficiência, em UML, na definição do conector. Com a separação do conector de ligação da estrutura do componente e a alteração do nome do conector de delegação (que deixa de ser um conector), o conector UML passou a ser um elemento de primeira ordem e ter uma estrutura e notação próprias. Dessa forma, acreditamos que a AO-AUML seja uma linguagem de descrição arquitetural que está de acordo com o framework definido por MEDVIDOVIC (2000).

b) **Prover uma linguagem de modelagem de arquitetura alinhada aos conceitos de DSOA, que seja capaz de representar os interesses transversais de um sistema**

Diante das soluções apresentadas pelas AO ADLs para modularizar os interesses transversais, as chamadas simétricas são a que exigem menor esforço para adaptação por parte dos arquitetos. Como base dessa dissertação, foi escolhida a linguagem AspectualACME (GARCIA, 2006), que apresenta apenas a extensão do conector como alteração necessária para representação dos interesses transversais. Com a inclusão do conector aspectual e sua estrutura no metamodelo da UML 2.0, é possível representar os interesses transversais, assim como a sua interação com o restante do sistema.

c) **Promover as modificações necessárias na UML 2.0, alterando minimamente o meta-modelo da UML 2.0, incentivando sua utilização por arquitetos e facilitando a extensão das ferramentas de modelagem já existentes**

Todas as alterações propostas no meta-modelo da UML 2.0 foram feitas pensando em modificar o mínimo possível da estrutura já existente. Essas mudanças pontuais estimulam a utilização da AD-AUML, pois a curva de aprendizado de um arquiteto que já esteja acostumado a utilizar a UML tende a ser pequena. Além disso, uma vez que as modificações têm pouco impacto, permitem a extensão de ferramentas já existentes para a modelagem de arquitetura com UML 2.0.

d) **Relacionar elementos de descrição arquitetural na linguagem proposta com elementos de alguma linguagem de descrição em projeto detalhado, promovendo a rastreabilidade entre as duas fases.**

Foi proposto um conjunto de regras de transformação da linguagem AD-AUML para a linguagem aSideML. A utilização das regras permite uma rastreabilidade entre a fase de definição de arquitetura (modelada com AD-AUML) e a fase de projeto detalhado (modelada com aSideML).

e) **Utilização de dois estudos de casos, baseados em arquiteturas conhecidas e publicadas, para validar a nova linguagem**

No trabalho são apresentados dois estudos de caso com arquiteturas publicadas e conhecidas. A utilização desses estudos de caso, permite observar a

forma como a AD-AUML se comporta na modelagem de arquiteturas, permitindo avaliar sua representatividade. O fato de utilizarmos um sistema (Health Watcher) que é *testbed* para um importante grupo de pesquisa em desenvolvimento de software orientado a aspectos (AOSD-EUROPE), corrobora ainda mais essa contribuição.

7.2 TRABALHOS FUTUROS

Algumas questões referentes ao trabalho desenvolvido ainda ficaram em aberto. Assim, é possível propor algumas orientações para pesquisas e desenvolvimentos futuros, a saber:

a) **Desenvolvimento de ferramenta computacional que permita a modelagem de arquiteturas utilizando a linguagem AD-AUML**

Até agora não foi desenvolvida uma ferramenta para apoiar o uso da notação proposta. Mas, pelo fato da notação proposta ser uma extensão de UML, é possível contar com o apoio inicial de alguma ferramenta já existente e aperfeiçoá-la para representar a parte orientada a aspectos de uma arquitetura. Seria interessante ter uma integração desta ferramenta com outra que conseguisse representar o projeto orientado a aspectos do sistema, permitindo a aplicação das regras de transformação.

b) **Novos estudos de caso.**

Embora tenhamos utilizado dois estudos de caso, sendo que um é utilizado como *testbed* pelo AOSD-Europe (GROUP, 2006), consideramos que novos estudos devem ser feitos para dar maturidade à AD-AUML.

c) **Extensão da cláusula Glue.**

A forma como a cláusula Glue está definida no metamodelo, restringe sua utilização a apenas três tipos: *after*, *around* e *before*. Seria interessante propor uma extensão da Glue para que futuramente pudessem ser incluídas interações mais complexas entre elementos aspectuais.

d) **Regras de transformação da linguagem AD-AUML para outras linguagens de projeto detalhado.**

Embora tenha sido proposto um conjunto de regras de mapeamento entre AD-AUML e aSideML, pode ser feito um mapeamento para outras linguagens, como por exemplo ThemeUML (CLARKE, 2002).

e) **Investigar a forma como o conceito de interesses transversais se manifesta em outra visões arquiteturais.**

O presente trabalho tratou apenas da visão arquitetural de execução e na modelagem estrutural da arquitetura, seria interessante investigar o comportamento de interesses transversais em outras visões da arquitetura.

7.3 PALAVRAS FINAIS

Esta dissertação propôs uma nova linguagem de modelagem de arquitetura denominada AD-AUML. Essa linguagem, além de diminuir a distância entre os arquitetos (que basicamente utilizam ADLs) e o restante dos envolvidos no desenvolvimento, permite a representação de interesses transversais em nível de arquitetura. Por estar baseada na UML 2.0, a linguagem facilita a rastreabilidade entre o nível de arquitetura e o nível de projeto detalhado de software, já que a maioria das linguagens desenvolvidas são baseadas em UML. Dessa forma, acreditamos ter contribuído para o avanço nas pesquisas no campo da arquitetura, bem como no campo da orientação a aspectos.

8 REFERÊNCIAS BIBLIOGRÁFICAS

ALLEN, R. e GARLAN, D. **Formalizing Architectural Connection**. Em Proceedings of the 16th international conference on Software engineering (ICSE 1994), págs. 71–80, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

BARAIS, O., DUCHIEN, L., CARIOU, E., PESSEMIER, N. e SEINTURIER, L. **TranSAT: A Framework for the Specifcation of Software Architecture Evolution Workshop: Coordination and Adaptation Techniques for Software Entities**. Em European Conference on Object-Oriented Programming (ECOOP), págs. 14–18, Oslo, Norway, 2004.

BARBOSA, R. e GOLDMAN, A. **Mobigrid-Framework for Mobile Agents on Computer Grid Environments**. Em Mobility Aware Technologies and Applications (MATA), págs. 147–157, Florianópolis(SC), Brasil, 2004.

BASS, L., KAZMAN, R. e CLEMENTS, P. **Software Architecture in Practice**. Addison-Wesley Professional, 2003.

BATISTA, T., CHAVEZ, C., GARCIA, A., SANTANNA, C., KULESZA, U., RASHID, A. e FILHO, F. **Reflections on Architectural Connection: Seven Issues on Aspects and ADLs**. Em 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, 2006.

BINNS, P. e VESTAL, S. **Formal Real-Time Architecture Specification and Analysis**. Em Proceedings of the 10th IEEE workshop on Real-time operating systems and software table of contents, págs. 104–108, New York, NY, USA, 1993. IEEE Computer Society Washington, DC, USA.

BRUNETON, E., COUPAYE, T., LECLERCQ, M., QUEMA, V. e STEFANI, J. **An Open Component Model and its Support in Java**. Em International Symposium on Component-based Software Engineering (CBSE), Edinburg, Scotland, 2004. Springer.

CHAVEZ, C. **Um Enfoque Baseado em Modelos para o Design Orientado a Aspectos**. Tese de Doutorado, PUC-RIO, 2004.

CLARKE, S. e WALKER, R. **Towards a Standard Design Language for AOSD**. Em Proceedings of the 1st international conference on Aspect-oriented software development, págs. 113–119, Enschede, The Netherlands, 2002. ACM Press New York, NY, USA.

CLEMENTS, P., BACHMANN, F., BASS, L., GARLAN, D., IVERS, J., LITTLE, R., NORD, R. e STAFFORD, J. **Documenting Software Architectures: Views and Beyond**. Addison-Wesley Professional, 2002.

COGLIANESE, L. e SZYMANSKI, R. **DSSA-ADAGE: An Environment for Architecture-Based Avionics Development**. AGARD, Aerospace Software Engineering for Advanced Systems Architectures 8 p(SEE N 94-29315 08-61), 1993.

CONSORTIUM, O. **The Fractal Project**. http://fractal.objectweb.org/, 2006. Capturado em Dez 2006.

CUESTA, C., ROMAY, M., DE LA FUENTE, P. e BARRIO-SOLÓRZANO, M. **Architectural Aspects of Architectural Aspects**. Em 2nd European Workshop on Software Architecture (EWSA), LNCS, volume 3527, págs. 247–262, Pisa, Italy, 2005. Springer.

DIJKSTRA, E. **A Discipline of Programming**. Prentice-Hall, Englewood Cliffs, NJ, 1976.

ELRAD, T., AKSIT, M., KICZALES, G., LIEBERHERR, K. e OSSHER, H. **Discussing Aspects of AOP**. Communications of the ACM, 44(10):33–38, 2001.

FILMAN, R., ELRAD, T., CLARKE, S. e AKSIT, M. **Aspect-oriented software development**. Addison-Wesley, 2005.

FOUNDATION, T. E. **The AspectJ Project**. http://www.eclipse.org/aspectj/, 2006. Capturado em Dez 2006.

FUSEJ. **Unifying Aspects and Components**. http://ssel.vub.ac.be/fusej/, 2006. Capturado em Dez 2006.

GARCIA, A., CHAVEZ, C., BATISTA, T., SANTANNA, C., KULESZA, U., RASHID, A. e LUCENA, C. **On the Modular Representation of Architectural Aspects**. Em European Workshop on Software Architecture (EWSA 2006), Nantes, France, 2006.

GARLAN, D., ALLEN, R. e OCKERBLOOM, J. **Exploiting Style in Architectural Design Environments**. Em Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering, págs. 175–188, New Orleans, Louisiania, USA, 1994a. ACM Press New York, NY, USA.

GARLAN, D., MONROE, R. e WILE, D. **Acme: An Architecture Description Interchange Language**. Em Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research, Toronto, Ontario, Canadá, 1997. IBM Press.

GARLAN, D. e SHAW, M. **An Introduction to Software Architecture**. School of Computer Science, Carnegie Mellon University, 1994b.

GOULAO, M. e ABREU, F. **Bridging the Gap between Acme and UML 2.0 for CBD**. Em Workshop at European Software Engineering Conference (ESEC/FSE), págs. 1–2, Helsink, Finland, 2003.

GROUP, A.-E. **The AOSD-Europe Project**. http://www.aosd-europe.net/, 2006. Capturado em Dez 2006.

KANDÉ, M. **A Concern-Oriented Approach to Software Architecture**. Tese de Doutorado, École Polytechnique Fédérale de Lausanne, 2003.

KEVIN, S., GRISWOLD, W., SONG, Y., CAI, Y., SHONLE, M., TEWARI, N. e RAJAN, H. **Information Hiding Interfaces for Aspect-Oriented Design**. Em Proceedings of the 10th European software engineering conference (ESEC/FSE), págs. 166–175, Lisboa, Portugal, 2005. ACM Press.

KICZALES, G. **Aspect-Oriented Programming**. ACM Computing Surveys (CSUR), 28(4es), 1996.

KRECHETOV, I., TEKINERDOGAN, B., GARCIA, A., CHAVEZ, C. e KULESZA, U. **Towards an Integrated Aspect-Oriented Modeling Approach for Software Architecture Design**. Em 8th Workshop on Aspect-Oriented Modelling (AOM 06), Bonn, Germany, 2006.

KULESZA, U. e LUCENA, A. **Towards a Method for the Development of Aspect-Oriented Generative Approaches**. Em Workshop on Early Aspects (OOPSLA), volume 4, Vancouver, Canadá, 2004.

LUCKAHAM, D., AUGUSTIN, L., KENNEY, J., VEERA, J., BRYAN, D. e MANN, W. **Specifications and Analysis of System Architecture using Rapide**. IEEE Transactions on Software Engineering, 1995.

MEDVIDOVIC, N., OREIZY, P., ROBBINS, J. e TAYLOR, R. **Using Object-Oriented Typing to Support Architectural Design in the C2 Style**. Em Proceedings of the Fourth ACM Symposium on the Foundations of Software Engineering. ACM Press, 1996.

MEDVIDOVIC, N. e TAYLOR, R. **A Classification and Comparison Framework for Softwares ADLs**. IEEE Transactions on Software Engineering, 26:70–93, 2000.

NAVASA, A., PÉREZ, M. e MURILLO, J. **Aspect Modelling at Architecture Design**. Em 2nd European Workshop on Software Architecture (EWSA 2005), volume 3527, Pisa, Italy, 2005. Springer.

NAVASA, A., PÉREZ, M., MURILLO, J. e HERNÁNDEZ, J. **Aspect Oriented Software Architecture: a Structural Perspective**. Em Proceedings of the Aspect-Oriented Software Development (AOSD 2002), Enschede, The Netherlands, 2002.

OMG. **Object Management Group - UML 1.4**. http://www.uml.org/, 2003. Capturado em Dez 2006.

OMG. **Object Constraint Language - version 2.0**. http://www.uml.org/, 2006a. Capturado em Dez 2006.

OMG. **Unified Modeling Language - version 2.0**. http://www.uml.org/, 2006b. Captturado em Dez 2006.

PARNAS, D. **On the Criteria to be Used in Decomposing Systems into Modules**. Communications of the ACM, 15(12):1053–1058, 1972.

PÉREZ, J., RAMOS, I., JAÉN, J., LETELIER, P. e NAVARRO, E. **PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures**. Em Proceedings of 3rd IEEE International Conference on Quality Software (QSIC), Dallas, Texas, USA, 2003.

PESSEMIER, N., SEINTURIER, L., COUPAYE, T. e DUCHIEN, L. **A Model for Developing Component-based and Aspect-oriented Systems**. Em 5th International Symposium on Software Composition, Vienna, Austria, 2006.

PINTO, M., FUENTES, L. e TROYA, J. **A Dynamic Component and Aspect Platform**. Computer Journal, 2005.

RODRÍGUEZ, M., CUESTA, C., DE LA FUENTE, P. e SOLÓRZANO, M. **Descripción de Aspectos Mediante Conectores Uml 2.0**. Em Iberian workshop on Aspect Oriented Software Development (DSOA 2004), Malaga, Spain, 2004.

SANDE, M., CHOREN, R. e CHAVEZ, C. **Mapping AspectualACME into UML 2.0**. Em Proceedings of the 9th Workshop on Aspect-Oriented Modeling (AOM 2006), Genoa, Italy, 2006.

SANTANNA, C., LOBATO, C., KULESZA, U., GARCIA, A., CHAVEZ, C. e LUCENA, C. **On the Quantitative Assessment of Modular Multi-Agent System Architecture**. Em Multiagent Systems and Software Architecture (MASSA), Erfurt, Germany, 2006.

SHAW, M., DELINE, R., KLEIN, D., ROSS, T., YOUNG, D. e ZELESNIK, G. **Abstractions for Software Architecture and Tools to Support Them**. IEEE Transactions on Software Engineering, 21(4):314–335, 1995.

SHAW, M. e GARLAN, D. **Software Architecture: Perspectives on an Emerging Discipline**. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1996.

SOARES, S., LAUREANO, E. e BORBA, P. **Implementing Distribution and Persistence Aspects with AspectJ**. ACM SIGPLAN Notices, 37(11):174–190, 2002.

STAA, A. **Programação Modular: Desenvolvendo Programas Complexos de Forma Organizada e Segura**. Campus, 2000.

SUVEE, D., DE FRAINE, B. e VANDERPERREN, W. **FuseJ: An architectural Description Language for Unifying Aspects and Components**. Em Software Engineering Properties of Languages and Aspect Technologies, Chicago, Illinois, USA, 2005.

TARR, P., OSSHER, H., HARRISON, W. e SUTTON JR, S. **N Degrees of Separation: Multi-Dimensional Separation of Concerns**. Em Proceedings of the 21th International Conference on Software Engineering (ICSE 1999), págs. 107–119, Los Angeles, CA, USA, 1999.

9 APÊNDICES

9.1 APÊNDICE 1: DEFINIÇÕES E ALTERAÇÕES DOS ELEMENTOS DO METAMODELO

9.1.1 BINDING (FROM BASIC COMPONENTS)

O *binding* liga o contrato externo de um componente (especificado por suas portas) à realização interna desses comportamentos, representados pelas partes que compõem o componente. Ele representa a passagem de sinal (requisições de operações e eventos): um sinal que chega a uma porta do componente tem sua ligação para uma parte ou para outra porta, que será passada até o destino para que possa ser cumprida.

Generalizações

- "Feature (from Kernel)"

Descrição

No meta-modelo, um *binding* é utilizado para ligar a porta de um componente à sua implementação interna.

Atributos

Não há atributo adicional.

Associação

- redefinedBinding: Binding [0..*] - Um *binding* deve ser redefinido quando o classificador que o contém é especializado. O *binding* redefinido deve ser de um tipo que especialize seu tipo, assim como suas terminações. As propriedades das terminações devem ser substituídas.

Restrições

[1] Um *binding* só deve ser definido entre portas do mesmo tipo, ou seja, entre portas providas ou entre portas requeridas.

[2] Se um *binding* é definido entre uma porta e uma parte interna do classificador, então o classificador interno deve implementar a porta.

[3] Se o *binding* é definido entre a porta externa do componente e uma porta interna de uma parte dele, então a porta da parte interna deve dar suporte ao um conjunto de assinaturas de operações definidas na porta externa.

[4] Em um modelo completo, se uma porta externa tem um conjunto de *bindings* para um conjunto de portas internas, a união de todas as portas internas deve conter todas as assinaturas de operações presentes na porta externa.

Semântica

Um *binding* é uma declaração que os comportamentos disponibilizados pela instância de um componente não é realizado por ele próprio e sim por outra(s) instância(s) de elementos compatíveis. Pode ser outro componente ou mesmo uma classe. Caso seja uma classe, esse relacionamento é modelado partindo de uma porta do componente diretamente para a classe interna. Nesse caso, a classe deve implementar a porta do componente.

Os *bindings* são utilizados para modelar a decomposição hierárquica do comportamento, onde serviços que são providos pelos componentes podem ser ultimamente realizados por um classificador que esteja há alguns níveis hierárquicos dentro do mesmo componente. A palavra *binding* sugere que a mensagem ou o sinal que chegam à porta externa, podem passar por diversas portas internas, possivelmente através de múltiplos níveis.

Uma porta pode delegar a um conjunto de porta em componentes internos. Nesse caso, essas portas internas devem, juntamente, oferecer as funcionalidades da porta externa. Em tempo de execução, os sinais serão entregues para a porta apropriada. Nos casos onde múltiplas portas internas dão suporte aos serviços contidos no mesmo sinal, o sinal será entregue para todas essas portas.

Notação

O *binding* tem sua notação (FIG 9.1) como a de uma associação, partindo de uma porta de origem até a porta interna que implementa a funcionalidade, e vice-versa para portas requeridas.

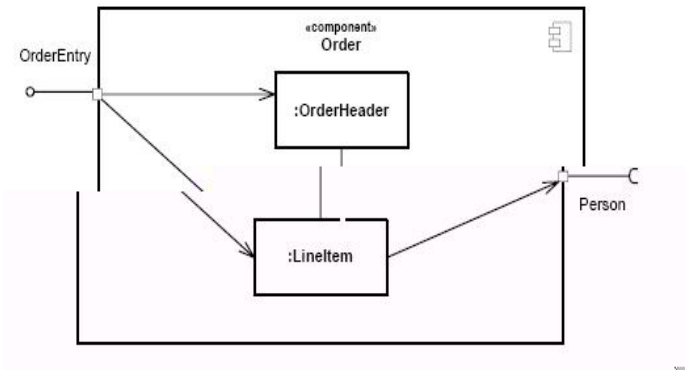


FIG. 9.1: Notação do binding

Mudanças em relação à versão anterior

Essa metaclasse foi adicionada à UML.

9.1.2 BINDING END (FROM INTERNAL STRUCTURES)

Generalizações

- "Multiplicity Element (from Kernel)"

Descrição

Um *binding end* é uma terminação de um *binding*, que liga o *binding* ao elemento conectável. Cada *binding end* é parte de um *binding*.

Atributos

Não há atributo adicional.

Associação

- role:ConnectableElement[1] - O elemento conectável ligado a esse *binding end*.

Restrições

Não há restrição adicional.

Semântica

Um *binding end* descreve que elementos conectáveis estão ligados ao *binding* a que pertence a terminação. Sua multiplicidade indica o número de instâncias que podem ser ligadas a cada instância de propriedade ligadas a outra terminação.

Notação

Podem ser utilizados os mesmo adornos de Association End.

Mudanças em relação à versão anterior

Essa metaclasse foi adicionada à UML.

9.1.3 ASSEMBLY CONNECTOR (FROM BASIC COMPONENTS)

Um conector de ligação (*assembly conector*) é um conector que liga dois componentes e que define que um componente fornece serviços que outro componente requer. Um conector de ligação é definido a partir de uma porta requerida de um componente para uma porta provida de outro componente.

Generalizações

- "Connector (from Kernel)"

Descrição

O conector de ligação define uma interação entre dois componentes, onde o primeiro utiliza os serviços providos pelo outro. A interação é sempre da porta requerida de um componente para a porta provida de outro componente.

Atributos

Não há atributo adicional.

Associação

- ownedRole: Role [2..*] - Cada conector de ligação tem que ter pelo menos dois papéis. Os papéis são interfaces do conector que estarão conectados às interfaces dos componentes.

Restrições

[1] Um conector de ligação só pode ser definido, se partir de uma porta requirida e chegar a uma porta provida.

```
context Assembly Connector inv:
        self.Connector1.ConnectorEnd.isAttached() = required port implies
        self.Connector2..ConnectorEnd.isAttached() = provided port;
```

Semântica

A semântica de tempo de execução para um conector de ligação é que sinais passam através da instância de um conector, originando em uma porta requerida e chegando a uma porta provida de componentes. Múltiplos conectores direcionados de uma única porta requerida para uma porta provida de diferentes componentes indicam que a instância que irá carregar o sinal será definida em tempo de execução. Similarmente, múltiplas portas requeridas que são ligadas a uma única porta provida, indica que a solicitação deve ser originada de instâncias de diferentes tipos de componentes.

A compatibilidade entre portas providas e requeridas que estão conectadas permitem que um determinado componente seja trocado por outro que (minimamente) ofereça o mesmo conjunto de serviços. Também, no contexto onde componentes são utilizados para estender o sistema, oferecendo serviços existentes, mas também adicionando novas funcionalidades, os conectores de ligação podem ser utilizados para ligar essa nova definição do componente.

Notação

O conector de ligação é representado por meio de um retângulo com duas associações: uma associação sem direção partindo do componente que requer o serviço e chegando ao conector e, do outro lado, uma associação direcionada partindo do conector e chegando ao componente que provê o serviço. Os papéis são representados através de círculos preenchidos com os respectivos nomes em cima e são inseridos sobrepostos ao retângulo

do conector. A representação do nome do conector, dos papéis e dos nomes dos papéis é opcional (FIG 9.2).

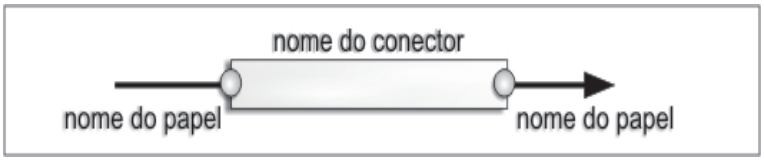


FIG. 9.2: Nova notação do conector de ligação

Mudanças em relação à versão anterior

Essa metaclasse foi adicionada à UML.

9.1.4 ASPECTUAL CONNECTOR (FROM BASIC COMPONENTS)

Um conector aspectual é um conector entre dois componentes que define que um componente com comportamento aspectual afeta um outro componente com comportamento regular. O conector aspectual é definido a partir de uma porta provida para uma porta provida ou requerida.

Generalizações

"Connector (from Kernel)"

Descrição

O conector aspectual define uma interação entre dois componentes, indicando que o componente de origem afeta o componente de destino. A forma de iteração é definida pela cláusula Glue. O Conector Aspectual tem dois tipos de papéis: o papel transversal e o papel base. O papel transversal está sempre ligado ao componente que exerce o papel aspectual, ou em outras palavras, que afeta o componente de destino. Já o papel base está sempre conectado ao componente que é afetado pelo comportamento do componente aspectual, ou como chamamos, componente regular.

Atributos

Não há atributo adicional.

Associação

- OwnedAspectualRole: Aspectual Role [2..*] - Cada conector aspectual tem, no mínimo, dois papéis aspectuais.
- OwnedGlue: Glue [1..*] - A Glue define a forma de interação entre dois componentes que estão conectados por um conector aspectual. Cada conector aspectual tem pelo menos uma cláusula Glue.

Restrições

[1] Não deve haver nenhum aspectual role definido no conector aspectual que não apareça na cláusula Glue.

```
context AspectualConnector inv:
        self.AspectualRole.isDefined(name) = true implies
        self.Glue.isDefined(name) = true;
```

Semântica

Dizemos que um dois componentes conectados por um conector aspectual interagem de forma que o componente de origem afeta o componente de destino, modificando ou inserindo comportamentos. A forma como se dará essa interação é definida pela cláusula Glue. Cada conector ainda apresenta dois papéis denominados papel transversal e papel base. O papel transversal é conectado ao componente que afeta (componente aspectual) e o papel base conectado ao componente que é afetado (componente regular).

Notação

Como notação gráfica para o conector aspectual (FIG 9.3), definimos um losango com duas associações: uma associação sem direção partindo do componente que tem comportamento aspectual (irá afetar o outro componente) e chegando ao conector e, do outro lado, uma associação direcionada partindo do conector e chegando ao componente de comportamento regular (será afetado pelo componente aspectual). Os papéis são representados por hexágonos preenchidos inseridos sobrepostos ao losango do conector. O tipo de papel é percebido pela direção da conexão. Os nomes dos papéis são inseridos acima dos desenhos. O tipo da cláusula Glue é indicado por um nome dentro do losango. A representação do nome do conector, dos papéis, e dos nomes dos papéis é opcional.

FIG. 9.3: Nova notação do conector aspectual

Mudanças em relação à versão anterior

Essa metaclasse foi adicionada à UML.

9.1.5 CONNECTOR (FROM KERNEL)

Generalizações

- "Classifier (from Kernel, Dependencies, PowerTypes)"

Descrição

Connector é uma metaclasse abstrata que representa um classificador cuja responsabilidade é ligar dois componentes.

Atributos

Não há atributo adicional.

Associação

- OwnedConnection: Connection [2] - Cada instância do Connectortem que ter exatamente dois Connection, que representam as associações (ligações) feitas pela instância;

Restrições

Não há restrição adicional.

Semântica

Uma instância de um Connector deve conectar um componente a outro componente. A semântica é que o componente de origem está interagindo de alguma forma com o componente de destino. A forma como se dará essa interação é definida pelo tipo da instância.

Notação Não é necessária notação adicional. As subclasses concretas irão definir sua notação específica.

Mudanças em relação à versão anterior

Essa metaclasse foi adicionada à UML.

9.1.6 CONNECTION (FROM INTERNAL STRUCTURES)

Generalizações

- "Feature (from Kernel)"

Descrição

Cada conector pode ser atado a dois ou mais elementos conectáveis, cada um representando um conjunto de instâncias. Cada terminação do conector é distinta no sentido de que representa um papel diferente na comunicação realizada por um conector. As conexões realizadas pelo conector podem ser restringidas por várias restrições que são aplicadas aos elementos que estão conectados.

Atributos

Não há atributo adicional.

Associação

- end: ConnectionEnd [2] - Um Connection tem duas terminações chamadas de Connection End. Cada terminação representa a participação de uma instância de classificadores, que são elementos conectáveis.

Restrições

[1] Os elementos que são atados às terminações de um conector devem ser compatíveis.

[2] Quando a terminação (connection end) do conector estiver ligada a um papel transversal de um conector aspectual, a outra terminação deve, obrigatoriamente, estar ligada a uma porta provida de um componente.

```
context Connector inv:
        self.ConnectorEnd1.isAttached() = CrosscuttingRole implies
        self.ConnectorEnd2.isAttached() = provided port;
```

[3] Quando a terminação (connection end) do conector estiver ligada a um papel base, a outra terminação pode estar conectada a uma porta provida ou requerida de um componente.

```
context Connector inv:
        self.ConnectorEnd1.isAttached() = BaseRole implies
        self.ConnectorEnd2.isAttached() = provided port or required port;
```

Semântica

Se o conector entre dois papéis de um classificador é uma característica de um classificador instanciável, ele declara que uma ligação deve existir entre as instâncias dos classificadores. Se o conector entre dois papéis de um classificador é uma característica de um classificador não-instanciável, ele declara que uma ligação deve existir entre as instâncias que realizam o classificador. Essas ligações vão conectar as instâncias que correspondem às partes ligadas pelo conector.

A ligação correspondente aos conectores devem ser criadas juntamente com a criação da instância do classificador que as contém. O conjunto de ligações é um subconjunto do conjunto total de ligações especificado pela associação definida pelo conector. Todas as ligações são destruídas juntamente à instância do classificador.

Notação

Um connection é desenhado utilizando a notação de associação.

Mudanças em relação à versão anterior

Essa metaclasse foi adicionada à UML.

9.1.7 ROLE (FROM INTERNAL STRUCTURES)

Um papel é uma propriedade de um conector de ligação que define os participantes da interação representada pelo conector.

Generalizações

- "Property (from InternalStructures)"

Descrição

As interfaces de um conector são chamadas de papel. Cada papel define um participante na interação representada pelo conector. Cada participante tem o seu papel na interação. Cada conector tem, no mínimo, dois papéis.

Atributos

Não há atributo adicional.

Associação

Não há associação adicional.

Restrições

Não há restrição adicional.

Semântica

Não há semântica adicional.

Notação

Não há notação.

Mudanças em relação à versão anterior

Essa metaclasse foi adicionada à UML.

9.1.8 ASPECTUAL ROLE (FROM INTERNAL STRUCTURES)

O papel aspectual (aspectual role) é uma propriedade de um conector aspectual que define os papéis (interfaces do conector) que esse conector pode ter.

Generalizações

- "Property (from InternalStructures)"

Descrição

No metamodelo, o atributo RoleKind é adicionado à metaclasse aspectual role. Seu valor é um tipo enumerado com os valores baseRole (papel base) e crosscuttingRole (papel transversal).

Atributos

+kind : RoleKind indica o tipo do Aspectual Role.

Associação

Não há associação adicional

Restrições

Não há restrição adicional

Semântica

O papel aspectual de um conector aspectual é a sua interface. O papel transversal e o papel base são semanticamente diferentes. O primeiro indica que o componente a que o conector está ligado tem um comportamento aspectual, ou seja, irá afetar o outro componente. Já o segundo, indica que o componente a que está conectado tem um comportamento regular, ou seja, é afetado pelo componente "aspectual".

Notação

Não há notação.

Mudanças em relação à versão anterior

Essa metaclasse foi adicionada à UML.

9.1.9 GLUE (FROM INTERNAL STRUCTURES)

A cláusula Glue é responsável por definir a forma de composição entre dois componentes em uma interação aspectual. A Glue é propriedade do conector aspectual responsável pela interação.

Generalizações

- "Property (from Kernel, Association Classes)"

Descrição

A cláusula Glue descreve o protocolo de composição em uma relação de crosscutting entre dois componentes. O atributo GlueKind é adicionado à metaclasse Glue. Seu valor é um tipo enumerado com os valores after, around e before.

Atributos

+kind : GlueKind indica o tipo da cláusula Glue.

Associação

Não há associação adicional

Restrições

[1] Os tipos de papéis relacionados na cláusula Glue têm que ser diferentes.

```
context Glue inv:
        self.roleType1() != self.roleType2();
```

Semântica

A Glue define a forma de composição dos elementos que participam de uma interação aspectual.

Notação

Não há notação adicional.

Mudanças em relação à versão anterior

Essa metaclasse foi adicionada à UML.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo